JAVA 2020 – AND BEYOND

Dossier für Java-Entwickler:

Project Loom, Spring Boot,
Serverless Java, Domain-driven Design
& Elasticsearch

jaxcon jax.konferenz videosjaxenter jax

jax.de

Inhalt

Project Loom: Besser skalieren durch virtuelle Threads

Blick in die fernere Zukunft

Das Project Loom ist eine experimentelle Version des JDKs. Es erweitert Java um virtuelle Threads, die leichtgewichtige Nebenläufigkeit erlauben. Preview Releases sind schon jetzt verfügbar und zeigen, was möglich ist.

von Lutz Hühnken

Serverseitige Java-Applikationen sollten in der Lage sein, viele Requests parallel abzuarbeiten. Das Modell, das in der serverseitigen Java-Programmierung bis heute am verbreitetsten ist, ist Thread-per-Request. Hierbei wird einem eingehenden Request ein Thread zugeordnet und alles, was getan werden muss, um mit einer passenden Response zu antworten, wird auf diesem Thread abgearbeitet. Allerdings schränkt das die maximale Anzahl nebenläufig bearbeiteter Requests stark ein. Denn die Java-Threads, die jeweils einen nativen Betriebssystemthread beanspruchen, sind keine Leichtgewichte. Zum einen wegen des Speicherbedarfs: Jeder einzelne Thread beansprucht standardmäßig mehr als ein Megabyte Speicher. Zum anderen wegen der Kosten für den Wechsel zwischen den Threads, dem Context Switch.

Als Reaktion auf diese Nachteile sind in den letzten Jahren viele asynchrone Libraries, die zum Beispiel CompletableFutures verwenden, aber auch ganze „reactive“ Frameworks, wie z. B. RxJava [1], Reactor [2] oder Akka Streams [3] entstanden. Zwar nutzen sie allesamt die Ressourcen weit effektiver, fordern den Entwickler aber auf, sich auf ein deutlich anderes Programmiermodell umzustellen. Hierbei stöhnen einige über den kognitiven Ballast, da sie lieber wie gewohnt sequenziell auflisten, was das Programm tun soll, statt mit Callbacks, Observables oder Flows zu hantieren. Hier stellt sich die Frage, ob es nicht möglich wäre, die Vorteile beider Welten zu vereinen: So effektiv wie die asynchrone Programmierung und dennoch in gewohnter, sequenzieller Befehlsabfolge programmieren können. Das Project Loom [4] von Oracle will mit einem modifizierten JDK genau diese Option erkunden. Es bringt zum einen ein neues, leichtes Konstrukt für die Nebenläufigkeit mit, nämlich die virtuellen Threads, und zum anderen eine angepasste Standardbibliothek, die auf diesen aufsetzt.

Virtual Threads

Erinnert sich noch jemand an Multithreading in Java 1.1? Damals kannte Java nur sogenannte Green Threads. Die Möglichkeit, mehrere Betriebssystemthreads zu verwenden, wurde gar nicht genutzt. Threads wurden lediglich innerhalb der JVM emuliert. Ab Java 1.2 wurde schließlich für jeden Java-Thread auch wirklich ein nativer Betriebssystemthread gestartet.

Jetzt feiern die Green Threads ein Revival. Zwar in deutlich veränderter, modernisierter Form, aber im Grunde sind virtuelle Threads nichts anderes: Es sind Threads, die innerhalb der JVM verwaltet werden. Allerdings sind sie nun nicht mehr Ersatz für die nativen Threads, sondern Ergänzung. Eine relativ geringe Menge nativer Threads wird als Carrier eingesetzt, um eine nahezu beliebig große Menge virtueller Threads abzuarbeiten. Der Overhead der virtuellen Threads ist so gering, dass der Programmierer sich keine Gedanken darüber machen muss, wie viele davon er startet.

Ein nativer Thread reserviert in einer 64 Bit JVM mit Defaulteinstellungen schon einmal ein Megabyte für den Stack (der Thread Stack Size, der mit der Option *-Xss* auch explizit gesetzt werden kann). Dazu kommen noch ein paar weitere Metadaten. Und wenn nicht der Speicher die Grenze bildet, wird das Betriebssystem bei ein paar tausend Schluss machen.

Der Versuch in Listing 1, 10 000 Threads zu starten, wird die meisten Rechner in die Knie (bzw. die JVM zum Absturz) zwingen. Achtung: eventuell erreicht das Programm das Threadlimit Ihres Betriebssystems und folglich könnte Ihr Rechner einfrieren.

Mit virtuellen Threads ist es hingegen kein Problem, eine ganze Million an Threads zu starten. Listing 2 wird auf der Project Loom JVM ohne Probleme durchlaufen!

Listing 1

```
// Achtung, kann Rechner einfrieren...
void excessiveThreads(){
  ThreadFactory factory = Thread.builder().factory();
  ExecutorService executor = Executors.newFixedThreadPool(10000,
factory);
  IntStream.range(0, 10000).forEach((num) -> {
    executor.submit(() -> {
      try {
        out.println(num);
        // Wir warten ein bisschen, damit die Threads wirklich alle
gleichzeitig laufen
        Thread.sleep(10000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    });
  });
  executor.shutdown();
  executor.awaitTermination(Integer.MAX_VALUE, TimeUnit.SECONDS);
}
```

Listing 2

```
void virtualThreads(){
  // Factory für virtuelle Threads
  ThreadFactory factory = Thread.builder().virtual().factory();
  ExecutorService executor = Executors.newFixedThreadPool(1000000,
factory);
  IntStream.range(0, 1000000).forEach((num) -> {
    executor.submit(() -> {
      try {
        out.println(num);
        // Thread.sleep schickt hier nur den virtuellen Thread schlafen
        Thread.sleep(10000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    });
  });
  executor.shutdown();
  executor.awaitTermination(Integer.MAX_VALUE, TimeUnit.SECONDS);
}
```

JDK APIs

Wir könnten nun also eine Million Threads gleichzeitig starten. Das ist vielleicht ein schöner Effekt, aber der Nutzen ist dennoch begrenzt. Wirklich interessant wird es, wenn all diese virtuellen Threads die CPU nur kurzzeitig beanspruchen. Die meisten serverseitigen Anwendungen sind weniger CPU-lastig als vielmehr I/O-lastig. Es wird vielleicht ein bisschen Input validiert, aber dann werden vor allem Daten über das Netz geholt (oder geschrieben), zum Beispiel aus der Datenbank, oder über HTTP von einem anderen Service.

Im Thread-per-Request-Modell mit synchroner I/O führt das dazu, dass der Thread für die Dauer der I/O-Operation geblockt ist. Das Betriebssystem erkennt, dass der Thread auf I/O wartet, worauf der Scheduler direkt zum nächsten schaltet. Das scheint erst einmal nicht weiter schlimm, da der geblockte Thread die CPU nicht belegt. Jeder Wechsel zwischen Threads bringt allerdings einen Overhead mit sich. Dieser Effekt hat sich übrigens durch die modernen, komplexen CPU-Architekturen mit mehreren Cache-Layern (non-uniform Memory Access, NUMA) noch deutlich verschärft.

Die Anzahl der Context Switches sollte minimiert werden, um die CPU tatsächlich effektiv auszulasten. Aus CPU-Sicht wäre es perfekt, wenn auf jedem Core genau ein Thread dauerhaft laufen und nie ausgewechselt würde. Diesen Zustand werden wir in der Regel nicht erreichen können, schließlich laufen neben der JVM auch noch andere Prozesse auf dem Server. Aber es bleibt festzuhalten: „Viel hilft viel“ gilt zumindest bei nativen Threads nicht – man kann es hier durchaus übertreiben.

Um viele parallele Requests mit wenigen nativen Threads ausführen zu können, gibt der in Project Loom eingeführte virtuelle Thread freiwillig die Kontrolle ab, wenn er auf I/O wartet und pausiert. Den darunterliegenden nativen Thread, der als Worker den virtuellen Thread ausführt, blockiert er aber dadurch nicht. Der virtuelle Thread signalisiert vielmehr, dass er gerade nichts machen kann, während sich der native Thread den nächsten virtuellen Thread greifen kann, ohne CPU-Kontextwechsel. Wie aber geht das vonstatten, ohne asynchrone I/O APIs zu verwenden? Denn Project Loom hat sich schließlich auf die Fahnen geschrieben, die Programmierer vor der Callback-Wüste zu bewahren.

Hier zeigt sich der Vorteil dessen, die neue Funktionalität in Form einer neuen JDK-Version bereitzustellen. Eine Third-Party Library für ein derzeit aktuelles JDK ist darauf angewiesen, ein asynchrones Programmiermodell zu verwenden. Project Loom hat stattdessen eine angepasste Standardbibliothek im Gepäck. Viele I/O Libraries sind so umgeschrieben worden, dass sie nun intern virtuelle Threads verwenden (Kasten: „Geänderte Standardbibliotheken“). Ein gewohnter Netzwerkaufruf ist – ohne jegliche Veränderung des Programmcodes – plötzlich keine blocking I/O mehr. Lediglich der virtuelle Thread pausiert. Durch diesen Kniff profitieren auch bestehende Programme von den virtuellen Threads, ohne dass Anpassungen notwendig sind.

Continuations

Das Konzept, das die Basis für die Implementierung der virtuellen Threads bildet, nennt sich Delimited Continuations. Die meisten werden wohl schon einmal einen Debugger benutzt haben. Dazu setzt man einen Breakpoint im Code. Wenn dieser Punkt erreicht ist, wird die Ausführung angehalten und der aktuelle Zustand des Programms im Debugger dargestellt. Es wäre nun doch vorstellbar, diesen Zustand einzufrieren. Das ist die Grundidee der Continuation: halte an einer Stelle im Ablauf an, nimm den Zustand (des aktuellen Threads, also den Call Stack, die aktuelle Position im Code etc.) und wandle diesen in eine Funktion um, die „Mach-dort-weiter-wo-du-aufgehört-hast-Funktion“. Diese kann dann zu einem späteren Zeitpunkt aufgerufen und der begonnene Ablauf wieder aufgenommen werden. Genau das also, was für die virtuellen Threads benötigt wird: Die Möglichkeit, ein Programm zu einem belie-

Geänderte Standardbibliotheken

Die folgenden Klassen wurden angepasst, sodass blockierende Aufrufe darin nicht mehr den nativen Thread blockieren, sondern lediglich den virtuellen Thread.

- java.net.Socket
- java.net.ServerSocket
- java.net.DatagramSocket/MulticastSocket
- java.nio.channels.SocketChannel
- java.nio.channels.ServerSocketChannel
- java.nio.channels.DatagramChannel
- java.nio.channels.Pipe.SourceChannel
- java.nio.channels.Pipe.SinkChannel
- java.net.InetAddress

bigen Zeitpunkt anzuhalten und später im gemerkten Zustand fortzusetzen.

Continuations haben auch abseits von virtuellen Threads ihre Berechtigung und sind ein mächtiges Konstrukt, um den Fluss eines Programms beliebig zu beeinflussen. Project Loom stellt ein API für die Arbeit mit Continuations zur Verfügung. Für die Anwendungsentwicklung sollte es nicht notwendig sein, damit direkt zu arbeiten. Es ist in erster Linie das Low-Level-Konstrukt, das die virtuellen Threads möglich macht. Wer damit experimentieren möchte, hat aber die Möglichkeit dazu (Listing 3).

Virtuelle Threads sind übrigens eine Form des kooperativen Multitaskings. Nativen Threads wird vom Betriebssystem die CPU entzogen, unabhängig davon, was sie gerade tun (präemptives Multitasking). Auch eine Endlosschleife wird den CPU-Kern so nicht blockieren, es werden trotzdem andere drankommen. Auf der Ebene der virtuellen Threads gibt es aber keinen solchen Scheduler – der virtuelle Thread muss selbst die Kontrolle an den nativen Thread zurückgeben.

Noch nicht im Paket: Tail-Call Elimination

Der Vollständigkeit halber sei erwähnt, dass zu den Features, die im Project Loom implementiert werden sollen, auch die Optimierung von Endrekursion gehört. Wenn eine rekursive Funktion als letzte Aktion sich selbst aufruft, kann sie vom Compiler in eine nichtrekursive Schleife umgewandelt werden. Das geschieht bereits in einigen anderen Sprachen als Java. Auf der JVM unterstützen etwa Scala, Kotlin (mit *tailrec*) oder Clojure (mit *recur*) diese Tail-Call Elimination.

Auch Project Loom will eine Direktive einführen, die den Compiler zu dieser Optimierung, welche die Verwendung vieler rekursiver Algorithmen erst ermöglicht, anweist. In den derzeitigen Previews ist das aber noch nicht enthalten. Es ist sogar bisher noch gar nicht spezifiziert, wie es aussehen soll.

Listing 3

```
void continuationDemo() {
  // Der scope ist ein Hilfsmittel, um geschachtelte Continuations zu
  // ermöglichen.
  ContinuationScope scope = new ContinuationScope("demo");
  Continuation a = new Continuation(scope, () -> {
    out.print("To be");
    // hier wird die Funktion eingefroren und gibt die Kontrolle an den
    // Aufrufer.
    Continuation.yield(scope);
    out.println("continued!");
  });
  a.run();
  out.print(" ... ");
  // die Continuation kann von dort, wo sie angehalten wurde, fortgesetzt
  // werden.
  a.run();
  // ...
  }
```

Jenseits virtueller Threads

Die eingangs geschilderten Probleme mit Threads beziehen sich allein auf die Effizienz. Noch gar nicht betrachtet wurde dabei eine ganz andere Herausforderung: die Kommunikation zwischen Threads. Die Programmierung mit den aktuellen Java-Mechanismen ist nicht ganz einfach und folglich fehleranfällig. Threads kommunizieren über geteilte Variablen (shared mutable State). Um dabei Race Conditions zu vermeiden, müssen diese durch *synchronized* oder explizite Locks geschützt werden. Treten hier Fehler auf, sind sie durch den Nichtdeterminismus zur Laufzeit besonders schwer zu finden. Und selbst wenn alles richtig gemacht wurde, stellen diese Locks oft einen Point of Contention dar, einen Flaschenhals in der Ausführung. Denn potenziell müssen dann viele auf genau den einen warten, der gerade das Lock in Anspruch nimmt.

Es gibt durchaus alternative Modelle. Im Kontext von virtuellen Threads sind hier insbesondere Channels zu nennen. Kotlin und Clojure (Kasten: „Was machen

Was machen die anderen?

Die virtuellen Threads mögen neu für Java sein, neu auf der JVM sind sie nicht. Wer Clojure oder Kotlin kennt, fühlt sich wahrscheinlich an Coroutines erinnert. In der Tat sind sie diesen technisch sehr ähnlich und lösen das gleiche Problem. Mindestens einen kleinen, aber interessanten Unterschied gibt es jedoch aus Entwicklerperspektive: Für Coroutines gibt es spezielle Schlüsselwörter in den jeweiligen Sprachen. In Clojure ein Makro für einen Go-Block, in Kotlin das *suspend*-Schlüsselwort.
Die virtuellen Threads in Loom kommen ohne zusätzliche Syntax aus. Dieselbe Methode kann unverändert von einem virtuellen, oder direkt von einem nativen Thread ausgeführt werden.

War da nicht was mit Fibers?

Wer früher schon einmal von Project Loom gehört hat, kennt den Begriff der Fibers (Faser). Zusammen mit dem Faden (Thread) führte das wohl auch zum Projektnamen, denn Loom bedeutet Webstuhl. In den ersten Versionen von Project Loom war Fiber der Name für den virtuellen Thread. Er geht zurück auf ein vorheriges Projekt des jetzigen Loom-Projektleiters Ron Pressler, die Quasar Fibers [6]. Der Name Fiber wurde aber, ebenso wie die Alternative Coroutine, Ende 2019 verworfen, durchgesetzt hat sich Virtual Thread.

die anderen?“) bieten diese als bevorzugtes Kommunikationsmodell für ihre Coroutines an. Statt auf einen geteilten, veränderbaren Zustand setzen sie auf unveränderliche Nachrichten, die (bevorzugt asynchron) in einen Kanal geschrieben und von dort vom Empfänger aufgenommen werden. Ob Channels Teil von Project Loom werden, ist allerdings noch offen.

Es ist vielleicht aber auch gar nicht nötig, dass Project Loom alle Probleme löst – eventuelle Lücken werden sicher von neuen Third-Party Libraries gefüllt werden, die auf den virtuellen Threads basierend Lösungen auf einer höheren Abstraktionsebene bieten. Das Experiment Fibry [5] ist zum Beispiel eine Aktoren-Library für Loom (Kasten: „War da nicht was mit Fibers?“).

Das Alleinstellungsmerkmal von Project Loom

Für das Problem, das Project Loom löst, gibt es im Grunde schon eine etablierte Lösung: Asynchrone I/O, entweder durch Callbacks oder durch „reactive“ Frameworks. Diese zu verwenden heißt aber, sich auf ein anderes Programmiermodell einzulassen. Nicht allen Entwicklern fällt es leicht, auf eine asynchrone Denkweise umzusteigen. Es mangelt teilweise auch an Unterstützung in gängigen Libraries – alles, was Daten in ThreadLocal speichert, ist plötzlich unbrauchbar. Und im Tooling: Das Debuggen asynchronen Codes hat oft mehrere Aha-Erlebnisse zur Folge. Auch in dem Sinne, dass der Code, der untersucht werden soll, nicht auf dem Thread ausgeführt wird, den man gerade den Debugger in Einzelschritten durchgehen lässt.

Der besondere Reiz von Project Loom liegt darin, dass es die Änderungen auf JDK-Ebene vornimmt, sodass der Programmcode unverändert bleiben kann. Ein heute ineffizientes Programm, das für jede HTTP Connection einen nativen Thread verbraucht, könnte unverändert auf dem Project Loom JDK laufen und wäre plötzlich effizient und skalierbar (Kasten: „Wann kommen virtuelle Threads für alle?“). Der veränderten, jetzt auf virtuellen Threads basierenden java.net Library sei Dank.

Wann kommen virtuelle Threads für alle?

Project Loom hält sich sehr bedeckt, wenn es um die Frage geht, in welches Java-Release die Features einfließen sollen. Im Moment ist alles noch experimentell und APIs können sich noch ändern. In JDK 15 sollte man wohl noch nicht damit rechnen. Wer es ausprobieren möchte, kann aber entweder den Source Code von GitHub [7] auschecken und sich das JDK selbst bauen, oder fertige Preview Releases herunterladen [8].

Lutz Hühnken ist Chief Solutions Architect bei der Reederei Hamburg Süd. Aktuell beschäftigt er sich vorrangig mit Events-First Microservices, Domain-driven Design, Event Storming und Reactive Systems.

@lutzhuehnken

Links & Literatur

[1] https://github.com/ReactiveX/RxJava

[2] https://projectreactor.io

[3] https://doc.akka.io/docs/akka/current/stream/index.html

[4] https://wiki.openjdk.java.net/display/loom

[5] https://github.com/lucav76/Fibry

[6] http://docs.paralleluniverse.co/quasar/

[7] https://github.com/openjdk/loom

[8] http://jdk.java.net/loom/

Kolumne: EnterpriseTales

Müssen es immer Microservices sein? Die Wahl der richtigen Architektur

von Arne Limburg

Microservices waren in den letzten Jahren ein echtes Hypethema. Doch vielerorts setzt nach der ersten Euphorie nun langsam Ernüchterung ein. Zeit für mich, einmal einen unvoreingenommenen Blick auf die Wahl der richtigen Architektur zu werfen.

Architekturtrends scheinen sich wie so vieles in Wellenbewegungen zu entwickeln. Um die Jahrtausendwende war SOA (Service-oriented Architecture) der Architekturtrend schlechthin. Bereits ein paar Jahre später merkte man, dass der gewählte Ansatz große Nachteile hatte: Systeme, die zur Entwicklung entkoppelt waren, mussten zur Laufzeit dennoch zusammenspielen. Schnittstellenänderungen mussten immer an zwei Systemen (Aufrufer und Aufgerufener) implementiert werden. Der Versuch, das Problem mit einem Enterprise Service Bus zu entschärfen, scheiterte, weil weiterhin zwei Systeme angefasst werden mussten (diesmal Aufgerufener und ESB).

Da war es doch deutlich leichter, gleich alles in eine Deployment-Einheit zu packen – es entstanden monolithische Applikationen. Irgendwann wurden diese allerdings so groß, dass sie für die Teams nicht mehr beherrschbar waren. Die Folge war, dass scheinbar kleine Änderungen unverhältnismäßig lange brauchten, bis sie umgesetzt waren. Getrieben durch Netflix, Amazon und Co. kam der Trend der Microservices auf, der kleine Änderungszyklen versprach.

Schwergewichtige Monolithen

Aber was genau hindert die Entwickler eigentlich daran, auch in den alten Monolithen neue Features schnell in Produktion zu bringen? Meistens sind es die so viel zitierten „historisch gewachsenen“ Strukturen. Diese manifestieren sich in unleserlichem Spaghetticode mit nicht nachvollziehbaren Aufrufketten. Dazu kommt der Mangel an automatisierten Tests.

Bei Monolithen, die im ersten Jahrzehnt dieses Jahrtausends entstanden sind, wurde häufig ein großer Fokus auf die Schichtenarchitektur gelegt. Die Einhaltung dieser Architektur war in der Regel wichtiger, als auf den fachlichen Schnitt zu achten.

Wenn es überhaupt fachliche Module gab, bezogen sich diese häufig auf Domänenobjekte und nicht auf Use Cases. Das hat(te) zur Folge, dass an einzelnen Use Cases (häufig sogar an einzelnen Requests) mehrere Module beteiligt waren (nämlich alle Module der beteiligten Domänenobjekte). Das Ergebnis war ein unübersichtlicher Codefluss durch die Module und hohe Abhängigkeiten zwischen den Modulen.

Ein Entwickler konnte dann nicht mehr genau abschätzen, was seine Codeänderung tatsächlich für Auswirkungen auf die beteiligten Module hat. Fehlende automatisierte Tests taten ihr Übriges. Nach jeder noch so kleinen Änderung musste eigentlich die gesamte Anwendung auf unerkannte Seiteneffekte überprüft werden. Mehrwöchige Testphasen vor einem Release waren die Folge, was im Ergebnis zu maximal zwei bis vier

Releases pro Jahr führte. Viele der so aufgebauten Monolithen existieren heute noch. Allein die Tatsache der seltenen Releases führt dazu, dass ein Feature verhältnismäßig lange braucht, um in Produktion zu gelangen. Dazu kommen der höhere Entwicklungs- und Testaufwand.

Darüber hinaus erfordert der ungünstige Modulschnitt bei jeder Änderung einen hohen Abstimmungsaufwand zwischen verschiedenen Teams. Das erhöht natürlich die Entwicklungszeit eines Features weiter.

Kleinere Änderungszyklen

Aber was ist an Microservices anders? Warum ist es mit ihnen schneller möglich, Features in Produktion zu bringen?

Da ist zunächst einmal der Schnitt. Microservices sind grundsätzlich vertikal nach Use Cases geschnitten. Die Realisierung eines „normalen" Features betrifft dann in der Regel nur einen Microservice. Allein dieser Unterschied zu den Monolithen sorgt schon dafür, dass die meisten der oben genannten Probleme der Monolithen nicht mehr vorhanden sind: Da ein Microservice verhältnismäßig klein ist, ist der Code überschaubar und die Seiteneffekte eines Features sind abschätzbar. Zudem ist es bei einer so kleinen Codebasis einfacher, die Testabdeckung hochzuhalten. Abstimmung zwischen verschiedenen Teams fällt häufig ganz weg.

Sollten sich zwei Microservices doch einmal gegenseitig aufrufen müssen, so ist diese Kommunikation expliziter. Eine Schnittstellenänderung muss aktiv kommuniziert und so umgesetzt werden, dass die aufrufenden Microservices nicht kaputtgehen. Ein Entwickler kommt nicht „mal eben" auf die Idee, eine Schnittstelle zu ändern und die aufrufende Seite gleich mit zu ändern, so wie es vielleicht im Monolithen der Fall wäre. Das hat zur Folge, dass derjenige, der den Code ändert, sich im betroffenen Code auch gut auskennt. Im Monolithen gilt das ggf. nicht für beide Seiten der Schnittstelle.

Dadurch, dass die Änderungen überschaubar sind, ist natürlich auch der Testaufwand überschaubar, der betrieben werden muss, bis der Code produktiv gehen kann. Im Idealfall läuft der komplette Prozess bis zum Deployment in die Produktionsumgebung automatisch ab. Dadurch sind viel häufigere Releases möglich, als es beim Monolithen der Fall war. Allein deshalb ist ein Feature deutlich schneller in Produktion als früher.

Ein weiterer Punkt von Microservices, der die Durchlaufzeit von Features verringert, ist die Unabhängigkeit der einzelnen Teams. Selbst wenn zwei Microservices voneinander abhängen, sieht es der Architekturansatz vor, dass sie unabhängig entwickelt, deployt und betrieben werden können.

Der Preis der Unabhängigkeit

Diese Unabhängigkeit bekommt man natürlich nicht geschenkt. Wenn es Beziehungen zwischen Microservices gibt, muss natürlich ein besonderes Augenmerk auf das Schnittstellendesign gelegt werden. Clients müssen als Tolerant Reader implementiert und serverseitig müssen Schnittstellenänderungen abwärtskompatibel realisiert werden, um die Entwicklungs- und Laufzeitunabhängigkeit zu bewahren. Das alles ist selbstverständlich mit einem höheren Aufwand verbunden. Zudem werden Themen wie Logging und Tracing in einer Microservices-Landschaft deutlich komplexer als beim Monolithen. Es werden Architekturvorgaben benötigt, die über die Codestruktur innerhalb der einzelnen Services hinausgehen. Statt sich auf den inneren Aufbau der einzelnen Services zu beziehen, regeln diese Vorgaben die Kommunikation der Services untereinander.

Es gibt nun mal gewisse Entscheidungen, die für alle Services gemeinsam getroffen werden müssen. Diese können sowohl technologischer als auch architektonischer Natur sein. Zu behandeln sind Themen wie Service Discovery, Authentication, Tracing, ggf. ein gemeinsames Logformat, um das Logging zu zentralisieren, Health-Checks, Testing-Strategien (z. B. über Consumer-driven Contract Testing) oder auch eine gemeinsame Dokumentation des API (z. B. über OpenAPI). Zu guter Letzt ist es selbstverständlich auch deutlich aufwendiger, einen Zoo an Microservices zu betreiben als einen einzelnen Monolithen.

Der Vorteil der Unabhängigkeit einzelner Microservices muss also gegen den Aufwand aufgewogen werden, der entsteht, wenn eine Microservices-Architektur betrieben und weiterentwickelt wird. Nicht immer ist es dabei für die Gesamtapplikation sinnvoll, sie auf Microservices umzustellen.

Modulare Monolithen

Der große, bereits erwähnte Vorteil von Microservices ist, dass sie unabhängig voneinander entwickelt,

deployt und betrieben werden können. Das Hauptproblem, das ich oben bei den „alten“ Monolithen beschrieben habe, besteht aber häufig gar nicht aus den drei Aspekten Entwicklung, Deployment und Betrieb. Der größte Schmerzpunkt liegt häufig im Aspekt Entwicklung. Deployment und Betrieb an sich stellen meistens kein Problem dar. Wenn in solchen Monolithen also die Phasen Entwicklung und ggf. noch Abnahmetest beschleunigt werden könnten, würde nichts dagegensprechen, die Applikation als Monolith zu deployen und zu betreiben.

Auch Monolithen lassen sich nämlich gut strukturieren: Zunächst einmal ist die Aufteilung in fachliche Module wichtig. Dabei sollte der Fokus nicht daraufgelegt werden, welche Domänenobjekte ein Modul enthält, sondern vielmehr, welche Use Cases es abbilden soll. Dabei kann es durchaus vorkommen, dass ein Domänenobjekt in unterschiedlichen Ausprägungen in unterschiedlichen Modulen vorkommt. Durch den Fokus auf Use Cases schafft man es im Idealfall, dass die Module komplett unabhängig voneinander sind. Sollten sie doch Abhängigkeiten aufweisen, dann sollten diese lose gekoppelt werden. Fachliche Events, die in Java z. B. als CDI Events oder Spring Application Events realisiert werden können, sollten zur Kommunikation zwischen Modulen verwendet werden. Die Menge der gemeinsam genutzten fachlichen Klassen sollte möglichst gering sein, den fachlichen Kern abbilden und einer möglichst geringen Änderungsfrequenz unterliegen. Eric Evans beschreibt ein solches Modul in seinem Buch „Domain Driven Design“ [4] als „Abstract Core“.

Unabhängige Teams

Wenn es mehrere Teams gibt, die an einem Monolithen entwickeln, ist die Abstimmung der Teams untereinander eine sehr große Herausforderung, die zeitaufwendig sein und kurze Releasezyklen verhindern kann. Dann muss sichergestellt werden, dass jedes Team an einem eigenen Modul arbeitet und dass die Module wie beschrieben entkoppelt sind, der Code also komplett getrennt ist (im Idealfall in unterschiedlichen Source Code Repositories liegt) und es separate Build-Artefakte gibt. Damit die Teams tatsächlich unabhängig arbeiten können, muss die Trennung aber noch weiter gehen: Es darf keine gemeinsam genutzten Datenbanktabellen geben und natürlich auch keine direkten Aufrufe der Module untereinander.

Wenn das alles gewährleistet ist, steht der Entwicklung eines Monolithen (auch mit mehreren Teams) nichts im Weg.

Weitere Gründe für Microservices

Wenn es also möglich ist, modulare Monolithen zu bauen und dadurch an die Entwicklungsgeschwindigkeit von Microservices heranzukommen, ohne sich die Komplexität des Betriebs von Microservices einzuhandeln – welchen Grund gibt es dann noch, Microservices zu bauen?

Natürlich gibt es ein paar weitere Gründe, die ich an dieser Stelle nicht alle aufzählen möchte. Ich greife nur einen exemplarisch heraus: unabhängige Skalierbarkeit. Bereits bevor der Architekturtrend der Microservices aufkam, hatten wir als open knowledge GmbH Kunden, die für spezielle Aufgaben, die besonders viel Rechenpower benötigten, wie z. B. die Verarbeitung von Bildern, spezielle Server aufsetzten, die dann unabhängig von der eigentlichen Applikation skaliert werden konnten. Mit Microservices ist ein solches Szenario noch flexibler zu handhaben. Jeder Microservice kann genau so viel CPU, Arbeitsspeicher und Storage erhalten, wie er benötigt.

Gute Monolithen fallen nicht vom Himmel

Meine Erfahrung zeigt mir, dass die meisten Monolithen (insbesondere die historisch gewachsenen) nicht den Anforderungen entsprechen, die ich oben beschrieben habe. Andere Projekte, die wir in den vergangenen Jahren durchgeführt haben, zeigen aber auch, dass es möglich ist, solche Monolithen zu entwickeln. Was soll man also tun, wenn der eigene Monolith eher zu der Fraktion „historisch gewachsen“ gehört?

Wenn andere Gründe nicht gegen die Entwicklung der Applikation als Monolith sprechen, spricht allein die Tatsache, dass er schlecht strukturiert ist, auch nicht grundsätzlich gegen die monolithische Architektur. Die Frage ist nur: Wie überführe ich den schlecht strukturierten, historisch gewachsenen Monolithen in einen sauber modularisierten Monolithen? Je nachdem, wie der Zustand des Codes ist, ist das natürlich ein mehr oder weniger umfangreiches Unterfangen, das vordergründig der Applikation erst einmal keinen Mehrwert bietet. Wenn

man zu seinem Vorgesetzten (oder dem Product Owner) geht und sagt: „Wir brauchen jetzt ein Jahr, um unseren Code neu zu strukturieren. Danach können wir auch wieder schneller Features umsetzen, in der Zwischenzeit gibt es aber keine neuen Features" – dann wird das selten auf Zustimmung stoßen. Da ist es doch deutlich attraktiver, sagen zu können: Es gibt da dieses neue Architekturparadigma „Microservices", mit dem kann man besser und schneller entwickeln. Darauf müssen wir umstellen, um wieder effizienter arbeiten zu können.

Natürlich ist es schade, dass vielerorts Budget für eine Umstellung auf Microservices vorhanden ist, nicht aber Budget für ein Refactoring eines bestehenden Monolithen. Man muss diese Gegebenheiten aber annehmen und ggf. z. B. in eine Migrationsstrategie einfließen lassen. Übrigens sollte man dabei auch eine Einschätzung des Budgets einfließen lassen, das für eine Umstellung auf Microservices bereitsteht, auf halbem Wege ausgehen oder gekürzt werden kann.

Es gibt verschiedene Strategien, von einem Monolithen zu Microservices zu migrieren [5]. Ein paar Aspekte gilt es dabei aber unabhängig von der gewählten Strategie zu beachten:

1. Das grundsätzliche Ziel ist immer das Herauslösen unabhängiger Module …
2. … und jeder Schritt dorthin sollte immer in sich abgeschlossen und atomar sein, d. h. sollte das Projekt nach einem Schritt abgebrochen werden, sollte das Gesamtsystem auf jeden Fall besser (weil modularisierter) dastehen als vor dem Schritt.

Beachtet man diese beiden Punkte, kann man eine Migration ruhigen Gewissens angehen. Dann ist es auch fast egal, ob am Ende eine Menge an Microservices oder doch „nur" ein modularer Monolith herauskommt.

Fazit

Damit Softwaresysteme langfristig wartbar sind, sollten sie in (möglichst) unabhängige Module unterteilt werden. Beim Architekturansatz der Microservices ist diese Trennung so strikt, dass sie sich automatisch von der Datenbank über die Codebasis bis zu den Deployment-Einheiten und Laufzeitumgebungen erstreckt. Die strikte Einhaltung der Modulgrenzen ist so auf jeden Fall sichergestellt. Damit einher geht aber eine höhere Komplexität in Laufzeit und Betrieb der Services. Daher sollte man überprüfen, ob in der eigenen Situation nicht andere, weniger strikte Aufteilungen in Module möglich sind. Wenn man seinen Monolithen in kleine, möglichst unabhängige Module unterteilt, ist der Code auch gut wartbar, und kleine Features innerhalb eines Moduls sind schnell zu realisieren.

Arbeitet man mit mehreren Entwicklungsteams an einer Applikation, sollte auf jeden Fall sichergestellt werden, dass diese Teams unabhängig voneinander arbeiten können, um den Abstimmungsaufwand gering zu halten. Auch hier bieten Microservices eine sehr strikte Variante. Mit Microservices sind Teams in der Lage, unabhängig zu entwickeln und auch zu deployen. Eine unabhängige Deploybarkeit bietet häufig gar keinen Mehrwert.

Ziel sollte es immer sein, die Anwendung modular aufzubauen. Und das ist ja auch eines der Ziele von „Microservices"-Migrationen. Aber unabhängig davon, ob das Schlagwort „Microservices" im Budgetantrag auftaucht oder nicht: Wichtig ist, die Migration so zu planen, dass jeder Schritt auch dann einen Mehrwert bietet, wenn die Migration nach ihm abgebrochen wird.

Wenn man am Ende einen modularen Monolithen statt einer Microservices-Architektur hat, kann das häufig die beste Lösung sein. Um das zu erreichen, muss allerdings die richtige Migrationsstrategie gewählt werden. Aber das ist dann eine eigene Kolumne wert. In diesem Sinne – stay tuned.

Arne Limburg ist Enterprise Architect bei der open knowledge GmbH in Oldenburg. Er verfügt über mehrjährige Erfahrung als Entwickler, Architekt und Trainer im Enterprise- und Microservices-Umfeld. Zu diesen Bereichen spricht er regelmäßig auf Konferenzen und führt Workshops durch. Darüber hinaus ist er im Open-Source-Bereich tätig, unter anderem als PMC Member von Apache OpenWebBeans und Apache DeltaSpike sowie als Urheber und Projektleiter von JPA Security.

Links & Literatur

[1] https://chelseatroy.com/2016/07/02/when-not-to-use-microservices/

[2] https://containerjournal.com/topics/container-ecosystems/when-to-use-and-not-to-use-microservices/

[3] https://medium.com/digitalfrontiers/microservices-and-when-not-to-use-them-801c724d8fb0

[4] https://domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf

[5] https://martinfowler.com/articles/break-monolith-into-microservices.html

Serverless Java: Infrastrukturoverhead reduzieren

Leicht und schnell …

Nach wie vor ist Java die erste Wahl, wenn es um Softwareentwicklung für den Unternehmenseinsatz geht [1]. Allein mit der Entwicklung einer Java-Software ist es allerdings nicht getan: Für den Produktiveinsatz werden Maschinen, Betriebssysteme, JREs, Application Server etc. benötigt – und als Grundlage für die eigentliche Funktionalität im Code auch große Frameworks und Bibliotheken. Dieser Overhead schmerzt umso mehr, je einfacher die eigentlich benötigte Funktionalität ist, denn er erschwert Entwicklung, Tests und Betrieb. Der Gegenentwurf: Serverless.

von Thomas Darimont

Bei diesem Gegenentwurf wird die Software statt als eigenständig lauffähiges Binary als aufrufbare Funktion in ein passendes Umfeld deployt, das sich anstelle eines Application Servers und aller Infrastruktur darunter um die Verbindung zur Außenwelt kümmert. Bisher werden diese Funktionen lieber in JavaScript, Python und Go verfasst, u. a. wegen langer Startzeiten und hoher Speicheranforderungen JVM-basierter Anwendungen.

Dabei kann man auch sowohl von vorhandener Java-Erfahrung und bestehendem Code als auch vom reduzierten Aufwand für Deployment und Betrieb profitieren: Verbesserungen der JVM, neue schlanke Web-Frameworks wie Quarkus und Micronaut und Techniken wie Native-Image-Kompilierung mit GraalVM machen Java wieder interessant für den Einsatz im Serverless-Bereich. Dieser Artikel stellt einige Ansätze zur Verwendung von Java in Serverless-Umgebungen vor.

Anwendungsszenarien

Allgemein eignet sich ein Serverless-Ansatz gut für Aufgaben, die zu folgenden Nutzungsprofilen passen:

- Asynchron, leicht parallelisierbar in unabhängige Ausführungseinheiten
- Unregelmäßige oder sporadische Nachfrage mit großer, nicht vorhersehbarer Varianz des Nutzungsaufkommens
- Zustandslos, kurzlebig, toleriert hohe Latenz
- Schnell ändernde Geschäftsanforderungen mit Bedarf an hoher Entwicklungsgeschwindigkeit

Einige konkrete Beispiele sind:

- Dateiverarbeitung nach Upload
- WebHooks, um Logik mit HTTP-Aufrufen zu verbinden
- Backgroundjobs – zeitgesteuerte Verarbeitungen
- einfaches Backend für Web-Frontends

Grundlagen

Zwei Sichtweisen auf Serverless möchte dieser Artikel beleuchten: Entwicklung und Betrieb. Für den Entwickler bedeutet das, die Anwendungslogik in Form einfacher Funktionen zu implementieren. Für den Admin ist es eine einfache Art, Software zu deployen und zu betreiben, nämlich in entsprechenden lokalen oder Cloud-Umgebungen, die sich um das betriebsnotwendige Drumherum kümmern.

Einmal deployt, kann eine Funktion dann z. B. über HTTP, Messaging oder sonstige Ereignisse aufgerufen werden. Ein Beispiel ist die Erzeugung einer Vorschau für hochgeladene Fotos: Hat ein Nutzer ein neues Bild hochgeladen, so löst dieses Ereignis die Ausführung einer Funktion zur Erstellung einer Vorschau aus. Die Funktion kann selbst wieder andere Funktionen anstoßen.

Wie eine Serverless-Plattform intern arbeitet, ist letztendlich unerheblich für den Entwickler, solange sie ihm nur Boilerplate-Code abnimmt, z. B. den HTTP-Layer.

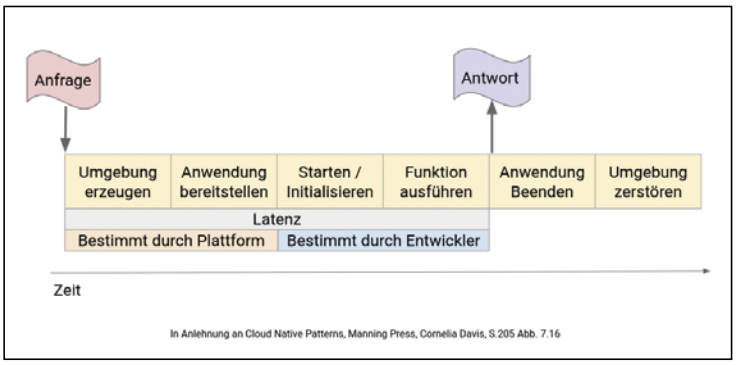

Abb. 1: Idealtypischer Lebenszyklus der Anfrageverarbeitung einer Serverless-Funktion

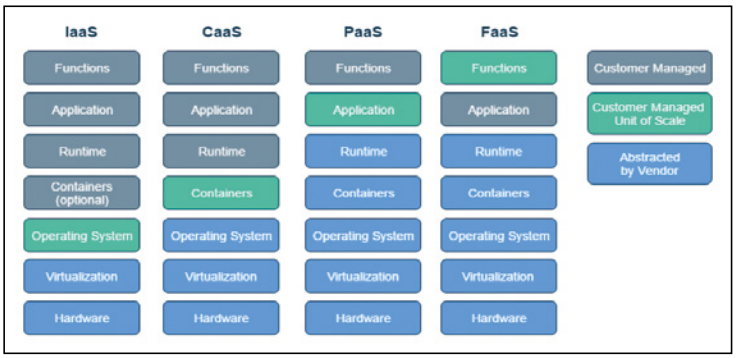

Abb. 2: Einordnung von FaaS in den Kontext von Cloud-Modellen [3]

Hat er seine Software hochgeladen, kümmert sich die Plattform um Betrieb und Skalierung der Funktion. Cloud-basierte Angebote berechnen Kosten nur für tatsächlich anfallende Ressourcennutzung, z. B. die verwendete CPU-Zeit, Arbeitsspeicher oder Anzahl der Aufrufe. Neben einem effizienten und ökonomischen Betrieb liegt somit der Fokus auf der Produktivität der Entwickler. Eine gute Zusammenfassung zum Thema bietet das Serverless Whitepaper der CNCF Serverless Working Group [2].

Geht eine Anfrage ein, die die Software bedienen soll, wird sie zunächst in einer Ausführungsumgebung gestartet und ein Endpunkt eingerichtet, der die Funktion aufruft, Argumente übergibt und das Ergebnis zurücksendet. Maßgeblich für die Performance ist dabei die Latenzzeit von der Anfrage bis zur Antwort, die der Entwickler zum einen über die Wahl der Plattform, zum anderen über geeignete Laufzeitumgebungen sowie Programmoptimierungen beeinflussen kann (**Abb. 1**). Anschließend kann die Plattform die Umgebung zerstören oder ggf. wiederverwenden. Plattformen begrenzen in der Regel die maximale Ausführungszeit einer Funktion z. B. auf 15 Minuten, danach wird die Ausführung abgebrochen.

Funktionen müssen zustandslos sein, da die Laufzeitumgebung jederzeit beendet oder neu gestartet werden kann. Auch mehrere Umgebungen sind möglich, sodass dieselbe Funktion mehrmals parallel laufen kann. Innerhalb einer Umgebung können z. B. statische Felder als Caches verwendet werden, damit zeitaufwendige Initialisierungen sich über mehrere Aufrufe hinweg amortisieren können, solange die Umgebung existiert.

Bisher hat Java im Serverless-Umfeld keine große Rolle gespielt, da allein der Start einer JVM in Cloud-Umgebungen mehrere Sekunden dauern kann, die sich auf die Latenz addieren. Das ist insbesondere ungünstig, wenn die eigentliche Funktionsausführung nur wenige Millisekunden braucht und häufig angefragt wird. Darüber hinaus verlängern die verwendeten Bibliotheken und Frameworks mit ihrer Initialisierung unter Umständen die Startdauer. Mittlerweile kann man sich aber einerseits auf Abhängigkeiten beschränken, die zur Ausführung benötigt werden, andererseits gibt es Alternativen, die JVM-Start-up-Latenz praktisch komplett zu eliminieren.

Darüber hinaus können Serverless-Plattformen die Bereitstellungslatenz minimieren, indem sie Ausführungsumgebungen für mehrere Anfragen wiederverwenden. Kurz aufeinanderfolgende Anfragen werden so von derselben Ausführungsumgebung behandelt. Man spart sich also die erneute Bereitstellung.

Plattformen

Serverless-Plattformen werden in der Cloud-Nomenklatur auch als FaaS (Function as a Service) bezeichnet und rücken damit die Funktionsausführung in den Mittelpunkt (**Abb. 2**), analog zu virtuellen Maschinen bei IaaS und kompletten Anwendungen bei PaaS. Nicht im Bild ist BaaS

(Backend as a Service), d. h. Infrastrukturdienste wie Datenbanken, Messagingsysteme und Storage, da dort in der Regel kein eigener Anwendungscode eingebracht wird.

Serverless-Plattformen bieten u. a. folgende Eigenschaften: kein Installations- oder Wartungsaufwand für Infrastruktur, flexible Skalierbarkeit, verbrauchsabhängige Kosten (d. h. keine Nutzung = keine Kosten), ereignisgesteuert, selbstskalierend innerhalb gegebener Grenzen, einfache Integration mit anderen Diensten der Plattform.

Die wichtigsten FaaS-Cloud-Anbieter mit nativer Java-Unterstützung sind derzeit Azure Functions, AWS Lambda und IBM Cloud Functions. Google Cloud Functions bietet derzeit keine direkte Unterstützung für Java-basierte Funktionen.

Neben den genannten Cloud-Diensten gibt es auch einige Plattformen, die sich lokal betreiben lassen, z. B. Riff, Fn Project, OpenWhisk und OpenFaaS.

Programmiermodelle

Zur Definition einer Funktion reicht häufig eine einfache Java-Klasse aus. Zur Bereitstellung erforderliche Metadaten werden dabei deklarativ über Annotationen oder externe Konfigurationsdateien angegeben. Manche FaaS-Dienste, z. B. Azure Functions, verlangen jedoch die Verwendung von speziellen APIs, mit denen auch Besonderheiten der Plattformen genutzt werden können. Möchte man Logik über mehrere FaaS-Dienste hinweg verwenden, bietet sich das Spring-Cloud-Function-Projekt [4] an. Damit kann man vom FaaS-Anbieter unabhängige Funktionslogik definieren, die dennoch in verschiedenen FaaS-Umgebungen aufgerufen werden kann. Außerdem gibt es auch die Möglichkeit, Java-Apps mit HTTP-Endpunkten in Container zu packen und auf Serverless-Containerplattformen wie Google Cloud Run oder AWS Fargate auszuführen.

Fn Project

Für die ersten Schritte im Serverless-Umfeld ist das Fn Project [5] gut geeignet, eine in Go geschriebene Serverless-Plattform, die auch auf dem eigenen Rechner läuft und neben Go und JavaScript auch Java als Laufzeitumgebung unterstützt. Fn bietet eine Serverkomponente, die die Clientanfragen entgegennimmt und zur Funktionsausführung dynamisch gestartete Docker-Container verwendet. Zusätzlich gibt es eine UI-Komponente [6], über die weitere Informationen zu den bereitgestellten Anwendungen und Funktionen verfügbar sind. Eine Funktion in Fn lebt im Kontext einer logischen Anwendung.

Eine kurze Anleitung zur Einrichtung des Fn Tooling bietet der Quick Start auf der Website des Projekts [7]. Mit dem Kommandozeilentool *fn* erzeugen wir ein neues Maven-Projekt mit dem Namen *hello-fn*:

```
fn init --runtime java hello-fn
```

Dieses Projekt enthält eine Beispielfunktion in *HelloFunction.java* (Listing 1) und die Konfigurationsdatei *func.yaml* (Listing 2), um die Funktion im Fn-Server zu betreiben.

Das Projekt und die Klasse *HelloFunction* sind einfach und verwenden keine weiteren Abhängigkeiten. Die Logik ist in der Methode *String handleRequest(String)* abgebildet.

Zu den wichtigsten Angaben in *func.yaml* gehören *name*, *runtime* und *cmd*. Das Attribut *name* gibt den logischen Namen der exponierten Funktion an. Mit *runtime: java* weisen wir den Fn-Server an, unsere Funktion in einer Java-Laufzeitumgebung auszuführen. Über das Attribut *cmd* weiß die Fn-Laufzeit, dass eine neue In-

Listing 1: Beispielfunktion in HelloFunction.java

```
package demo;

public class HelloFunction {

  public String handleRequest(String input) {
    String name = (input == null || input.isEmpty()) ? "world" : input;
    return "Hello, " + name;
  }
}
```

stanz der Klasse *HelloFunction* erzeugt und mittels Reflektion die *handleRequest*-Methode aufgerufen werden muss.

Die logische Anwendung, die die Funktion beinhaltet, erzeugen wir über den Befehl *fn create app*:

```
fn create app hello-fn-app
```

Anschließend wird die Funktion mittels des Befehls *fn deploy* über den Fn-Server bereitgestellt:

```
fn deploy --app hello-fn-app --local
```

Die Funktion kann mit dem Befehl *fn invoke* aufgerufen werden:

```
fn invoke hello-fn-app hello-fn
```

Das Fn User Interface kann als Docker-Container gehostet werden:

```
docker run --rm -it --link fnserver:api -p 4000:4000 \
  --name ui -e "FN_API_URL=http://api:8080" fnproject/ui
```

Unter *http://localhost:4000* findet man eine Übersicht zu den bereitgestellten Anwendungen und Funktionen, die auch direkt über die Oberfläche getestet werden können.

Azure Functions

Azure Functions ist ein FaaS-Dienst, der auch Java unterstützt. Dazu muss eine entsprechende Java-Klasse mit dem Azure Functions Java SDK erstellt werden. Eine gute Anleitung zur Erstellung einer Java-basierten Azure Function bietet [8].

Listing 3 zeigt eine Beispielfunktion unter Verwendung von Annotations aus dem Azure Functions Java SDK. Dabei wird über *@FunctionName* der externe Name der Funktion definiert. Mittels *@HttpTrigger* wird festgelegt, wie die Funktion über HTTP aufgerufen werden kann. Dabei kapselt *HttpRequestMessage* Informationen zur Anfrage, und über den *ExecutionContext* kann man auf Informationen zur Ausführungsumgebung, wie z. B. Konfiguration und Logging, zugreifen.

Das Beispiel verwendet das Azure Functions Maven Plugin und benötigt die Installation des azfunc-Tools aus dem Azure Function SDK [9]. Das Projekt kann als Maven-Projekt gebaut und lokal getestet werden. Dazu kann man über *mvn azure-functions:run* die Funktion lokal starten und über *curl -d ‚{„name“:“World“}‘ http://localhost:7071/api/greet* den lokalen Endpunkt aufrufen.

Listing 2: func.yaml

```
schema_version: 20180708
name: hello-fn
version: 0.0.1
runtime: java
cmd: demo.HelloFunction::handleRequest
```

Spring Cloud Function

Spring Cloud Function bietet Unterstützung für die Serverless-Entwicklung mit allen Möglichkeiten von Spring, wie z. B. Dependency Injection, Integrationen, Auto-Configuration in einem einheitlichen Programmiermodell über mehrere Serverless-Provider hinweg. Aktuell bietet Spring Cloud Function Unterstützung für AWS Lambda, Microsoft Azure, Apache OpenWhisk SDKs, Project Riff und Fn Project. Dazu stellt das Framework spezielle Adapterimplementierungen zur Integration mit FaaS-Providern bereit, die die providerspezifischen APIs abstrahieren. Funktionen können entweder als Spring Beans deklariert oder automatisch via Component Scanning entdeckt werden. Nachfolgend eine Bean-Function-Definition:

```
@Bean // function is exposed with name “greet”
public Function<User, Greeting> greet() {
  return user -> new Greeting(String.format(“Hello, %s”, user.getName()));
}
```

Listing 4 zeigt ein Beispiel für die Spring-Adaption von Azure-Function-spezifischen APIs. Über den Aufruf der *handleRequest(..)*-Methode wird der eigentliche Funktionsaufruf an der Funktion *Bean* mit dem Namen *greet* angestoßen.

Listing 3: Beispielfunktion unter Verwendung von Annotations aus dem Azure Functions Java SDK

```
public class Fun {

  @FunctionName("greet")
  public HttpResponseMessage<String> httpHandler(
    @HttpTrigger(name = "req", methods = "post",
                authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<String>> request,
    ExecutionContext context) {

    context.getLogger().info("processed a request.");

    String name = request.getBody().orElse(null);

    if (name == null)
      return request.createResponse(400, "Missing name");

    return request.createResponse(200, "Hello, " + name);
  }
}
```

Um die Funktion auf Azure bereitzustellen, muss man sich in der Konsole mittels *az login* bei Azure anmelden. Danach ruft man das *maven goal mvn azure-functions:deploy* auf. Anschließend ist die App unter dem generierten URL erreichbar.

Quarkus mit GraalVM

Quarkus ermöglicht die Entwicklung von schlanken Java Microservices [10]. In Kombination mit GraalVM und deren Native-Image-Erweiterung lässt sich aus einer klassischen JAX-RS-Anwendung sogar eine ausführbare native Binary erzeugen [11]. Diese Binary beinhaltet die vorab kompilierte Anwendung sowie GraalVMs SubstrateVM als Laufzeitumgebung. SubstrateVM benötigt nicht nur weniger Ressourcen als klassische JVMs, sondern startet auch sehr viel schneller – häufig in wenigen Millisekunden. Das macht die Kombination von Quarkus und GraalVM Native Image ideal für die Verwendung in Serverless-Container-Plattformen wie Google Cloud Run [12]. Die Plattform bietet Pay per Use, flexible Skalierung und kann nicht mehr benötigte Container auch komplett entfernen [13].

Tipps

Ein paar Tipps zur Verwendung von Java in Serverless-Umgebungen:

- Start-up-Zeiten minimieren
- auf effiziente Verarbeitung achten
- Batch-Verarbeitung von Nachrichten nutzen
- minimale Abhängigkeiten verwenden

Listing 4: GreetingHandler zur Abbildung auf Azure Functions

```
public class GreetingHandler extends AzureSpringBootRequestHandler<User, Greeting> {

  @FunctionName("greet")
  public Greeting execute(
    @HttpTrigger(name = "request", methods = HttpMethod.POST,
      authLevel = AuthorizationLevel.ANONYMOUS)
    HttpRequestMessage<Optional<User>> request, ExecutionContext context) {

    String name = request.getBody().map(User::getName).orElse("unknown");
    context.getLogger().info(String.format("Invoking greeting =: %s", name));

    // this invokes the greet function
    return handleRequest(request.getBody().get(), context);
  }
}
```

Fazit

Geeignete Anwendungsfälle können vom Serverless-Paradigma enorm profitieren: Ist die Aufgabe überschaubar und gut parallelisierbar, reduziert sich der Infrastrukturoverhead praktisch auf null. Wo bisher nur JavaScript, Python und Go infrage kamen, kommt durch schnell startende Application Frameworks wie Quarkus und Micronaut (insbesondere in Kombination mit GraalVM Native) neuerdings auch Java häufiger in die engere Wahl für die Serverless-Implementierung [14].

Thomas Darimont arbeitet als Fellow bei der codecentric AG in Saarbrücken, zuvor war er als Software Architect bei der eurodata AG und im Spring Data Team bei Pivotal tätig. Er ist Organisator der Java User Group Saarland und beschäftigt sich in seiner Freizeit mit Open Source sowie Themen rund um die JVM und Security. Thomas ist Sprecher auf Konferenzen und bei User Groups.

@thomasdarimont

Links & Literatur

[1] https://www.tiobe.com/tiobe-index/java/

[2] https://github.com/cncf/wg-serverless/tree/master/whitepapers/serverless-overview

[3] https://serverless.zone/abstracting-the-back-end-with-faas-e5e80e837362

[4] https://spring.io/projects/spring-cloud-function

[5] https://fnproject.io/

[6] https://github.com/fnproject/ui

[7] https://github.com/fnproject/fn#quickstart

[8] https://code.visualstudio.com/docs/java/java-azurefunctions

[9] https://docs.microsoft.com/en-us/azure/azure-functions/functions-run-local

[10] https://quarkus.io/guides/getting-started

[11] https://quarkus.io/guides/building-native-image

[12] https://github.com/cncf/wg-serverless/tree/master/whitepapers/serverless-overview

[13] https://cloud.google.com/run/

[14] https://github.com/thomasdarimont/serverless-javamagazin-article

Neue Features in ECMAScript 2020

Frisches Aroma für JavaScript

JavaScript wird unter dem ECMAScript-Standard stetig weiterentwickelt. Bis 2015 wurde der Standard mit Ganzzahlen versioniert, seit Version 6 ist man jedoch auf Jahreszahlen umgestiegen, sodass im Jahr 2020 die Version 2020 des Standards veröffentlicht wurde, nicht Version 11.

von Sebastian Springer

Der ECMAScript-Standard ist in Form einer PDF-Datei zum freien Download unter [1] verfügbar. Diese Datei weist für die Version 2020 einen Umfang von 860 Seiten auf und enthält sämtliche Sprachelemente und wie sie von den Browserherstellern umgesetzt werden. Welche Features in eine neue Version einfließen, bestimmt das Technical Committee 39 der Ecma International, kurz TC39. Die Vorschläge für die neuen Sprachfeatures werden auf GitHub unter [2] für alle öffentlich einsehbar verwaltet. Der Standardisierungsprozess besteht aus insgesamt fünf Stufen:

- *Stufe 0 (Strawperson):* Ein neuer Vorschlag für ein ECMAScript-Feature beginnt in der Stufe 0. Es gibt keine Einstiegskriterien für einen neuen Vorschlag.
- *Stufe 1 (Proposal):* Diese Stufe dient dazu, das Feature und eine potenzielle Problemstellung zu beschreiben. Außerdem sollen die potenziellen Probleme identifiziert werden. Damit ein Vorschlag in diese Stufe aufgenommen wird, müssen unter anderem folgende Anforderungen erfüllt werden: Es muss eine Person geben, die das Feature vorantreibt, das Problem muss beschrieben sein, es muss eine grobe API-Beschreibung geben und außerdem müssen die Kernalgorithmen beschrieben werden. Für das Feature wird entweder ein Polyfill oder eine Demo erwartet.
- *Stufe 2 (Draft):* In der zweiten Stufe werden Syntax und Semantik detailliert beschrieben. Hierfür muss eine initiale textuelle Beschreibung für das Feature existieren. Es gibt experimentelle Implementierungen im Browser.
- *Stufe 3 (Candidate):* In dieser Stufe wird die Spezifikation weiter verfeinert und das Feedback aus den Implementierungen und das der Benutzer aufgenommen. Damit ein Feature in diese Stufe aufgenommen wird, muss der Text der Spezifikation vollständig sein, außerdem muss er überprüft worden und von allen ECMAScript-Editoren genehmigt worden sein.
- *Stufe 4 (Finished):* Diese Stufe sagt aus, dass das Feature bereit ist, in den offiziellen ECMAScript-Standard integriert zu werden. Damit ein Feature diese Stufe erreichen kann, muss es eine Reihe von Akzeptanztests bestehen. Außerdem müssen zwei konkrete Implementierungen existieren.

Im Folgenden werfen wir einen Blick auf die neuesten Features, die teilweise bereits in den verschiedenen Browsern verfügbar sind oder durch Polyfills oder Transpiler emuliert werden können. Eine populäre Anlaufstelle für einen ersten Überblick stellt die Webseite www.caniuse.com

dar. Im zentralen Suchfeld können Sie sowohl HTML- als auch CSS- und JavaScript-Features eingeben und erhalten dann eine Übersicht über die Browserunterstützung des jeweiligen Features. Eine weitere Ressource sind die Compatibility Tables von Kangax auf GitHub unter [3]. Auf dieser Seite sehen Sie eine tabellarische Auflistung der verschiedenen JavaScript-Features und die jeweilige Browserunterstützung.

Ein kurzer Blick auf die Features von 2019

Die Änderungen der Version 2019 beschränken sich hauptsächlich auf Erweiterungen in den Standardobjekten des Sprachstandards. Das TC39 hat für diese Version die folgenden Methoden aufgenommen:

- Array: *Array.prototype.flat*, *Array.prototype.flatMap*
- Object: *Object.fromEntries*
- String: *String.prototype.trimStart*, *String.prototype.trimEnd*
- Symbol: *Symbol.description*

Eine der bedeutendsten Änderungen in ECMAScript 2019 verbirgt sich hinter der Bezeichnung „Optional Catch Binding“. Mit diesem Feature müssen Sie in einem try-catch-Statement nicht mehr zwingend das Fehlerobjekt beim *catch* angeben. Relevant ist diese Änderung vor allem dann, wenn Sie nicht direkt auf das Fehlerobjekt zugreifen müssen. Ohne dieses Feature definieren Sie eine lokale Variable im *catch*-Block, die im Quellcode nicht verwendet wird. Damit verstoßen Sie eigentlich gegen eine Standardregel zahlreicher Codingstandards, die besagt, dass es keine ungenutzten Variablen geben soll.

Im Gegensatz dazu können Sie von ECMAScript 2020, also der neuesten Version von JavaScript, einiges mehr erwarten.

Die neuen Features in ECMAScript 2020

Am 02.04.2020 hat das TC39 den ES2020 Candidate verabschiedet und damit das finale Featureset für die nächste ECMAScript-Version festgelegt. Dabei handelt es sich um die Features, die sich bis zu diesem Zeitpunkt auf der 4. Stufe des Standardisierungsprozesses befanden. Kurz vor der Finalisierung wurde der Funktionsumfang noch um das *import.meta*-Feature ergänzt. Die Features der neuen Version sind:

- Erweiterung der String-Klasse um die Methode *match-All*
- dynamische Importe mit der *import*-Funktion
- Import von Metadaten über *Import.meta*
- der neue Datentyp *BigInt*
- Ergänzung der *Promise*-Klasse um die *allSettled*-Methode
- Standardisierung von *globalThis*
- Standardisierung der *for-in* Mechanics
- Optional Chaining
- Nullish Coalescing

Diese Features sehen wir uns im Folgenden im Detail und mit passenden Beispielen an.

String.prototype.matchAll

Die *matchAll*-Methode ist auf allen JavaScript-Zeichenketten definiert, akzeptiert einen regulären Ausdruck und gibt ein Iterator-Objekt mit allen Treffern in der Zeichenkette zurück. Die Unterstützung dieser Methode ist über alle modernen Browser gut. Einzig Safari unterstützt die Methode noch nicht. Das Beispiel in Listing 1 sucht alle Wörter, die ein *th* beinhalten.

Das Ergebnis aus diesem Codeblock ist ein Iterator-Objekt, mit dem Sie über jeden Treffer iterieren können. Diese einzelnen Treffer beinhalten das gefundene Wort, den Index innerhalb der Zeichenkette sowie die gesamte ursprüngliche Zeichenkette und gegebenenfalls Gruppen innerhalb des regulären Ausdrucks.

Bei der Arbeit mit dem Iterator müssen Sie beachten, dass Sie diesen nicht wieder auf das erste Element zurücksetzen können. Das bedeutet, dass Sie den Iterator lediglich einmal verwenden können. So ist es möglich,

Listing 1: Wörter mit der String.prototype.match-All-Methode finden

```
const string = 'This Ecma Standard defines the ECMAScript 2019 Language.
It is the tenth
                    edition of the ECMAScript Language Specification';
const pattern = /\w*th\w*/gi;
const iterator = string.matchAll(pattern);
```

mit *Array.from* oder einem Destructuring-Statement *([…iterator])* den Iterator in ein Array umzuwandeln. Alternativ können Sie auch mit einer *for-of*-Schleife über die einzelnen Elemente des Iterators iterieren.

import()

Eines der bedeutendsten Features, das in der Vergangenheit in den Sprachstandard aufgenommen wurde, ist das Modulsystem. Es basiert auf den Schlüsselworten *import* zur Einbindung von Strukturen und *export* für das Exportieren von Strukturen, die an anderer Stelle eingebunden werden können. Das Modulsystem arbeitet auf Dateibasis, sodass die Exporte die Schnittstelle einer Datei zur Außenwelt bilden. Bisher sah der Sprachstandard lediglich statische Importe vor. Bei einem solchen Import müssen Sie die Quelle, von der Sie eine Struktur laden möchten, als Zeichenkette angeben. Im Gegensatz zu normalem JavaScript-Code müssen die Import-Statements bereits vor der Laufzeit feststehen, damit sie von der Engine analysiert werden können. Das bedeutet, dass die Zeichenkette, die die Quelle des Imports angibt, ein einfacher String sein muss. Es sind weder zusammengesetzte Strings noch Variablen oder Funktionsaufrufe erlaubt. Bei der Implementierung einer Applikation bedeutet das, dass Sie alle potenziell benötigten Dateien importieren müssen, egal, ob sie zur Laufzeit wirklich gebraucht werden oder nicht.

Abhilfe für dieses Problem schaffen dynamische Importe in der neuen ECMAScript-Version. Mit der *import*-Funktion haben Sie die Möglichkeit, dynamisch Quellen zu importieren. Der *import*-Funktion übergeben Sie den Pfad der zu ladenden Datei. Das Laden erfolgt asynchron. Dabei arbeitet die Funktion mit dem Promise-API des Browsers. Nach dem erfolgreichen Ladevorgang liegt ein Objekt vor, das alle Exporte der angeforderten Datei beinhaltet. Durch das verwendete Promise-Objekt können Sie dynamische Importe auch mit async/await kombinieren, um Ihren Quellcode besser lesbar zu gestalten. Ein solcher dynamischer Import darf im Quellcode Ihrer Applikation überall dort stehen, wo auch ein reguläres JavaScript-Statement vorkommen darf, also auch in Bedingungen und Schleifen. Sie sollten hier darauf achten, dass Sie keine potenziellen Fehlerquellen schaffen und auch nicht die Performance Ihrer Applikation negativ beeinflussen. In Listing 2 sehen Sie ein konkretes Beispiel für einen solchen dynamischen Import.

Listing 2: Dynamischer Import mit der import-Funktion

```
const fileName = './myFile';

if (fileName) {
  const result = import(fileName);
}
```

Listing 3: BigInts im Einsatz

```
const b1 = 123n;
const b2 = BigInt(321);

const r = b1 + b2;

console.log(r); // 444n
console.log(typeof r); // bigint
console.log(2n === 2); // false
```

import.meta

Bei dem *import.meta*-Objekt handelt es sich um eine Erweiterung des ECMAScript-Modulsystems. Dieses Objekt enthält zusätzliche Daten über das aktuelle Modul. Aktuell ist hier nur die *url*-Eigenschaft definiert. Sie gibt den URL des aktuellen Moduls an.

Führen Sie in Node.js *console.log(import.meta);* aus, wird der Inhalt des *import.meta*-Objekts auf der Konsole ausgegeben. Die *url*-Eigenschaft weist den absoluten Pfad der Moduldatei auf. Der entscheidende Vorteil ist auch an dieser Stelle wieder, dass es sich um ein Feature handelt, das Ihnen sowohl im Browser als auch in Node.js zur Verfügung steht. Damit ist es möglich, Quellcode zu implementieren, der in allen Umgebungen ausgeführt werden kann, ohne dass Sie spezielle Weichen einbauen.

BigInt

Im Umgang mit Zahlen weist JavaScript einige Beschränkungen auf. So ist die Größe einer Ganzzahl vom Typ *Number* in JavaScript auf 2^{53} - 1 limitiert. Die Begrenzung des *Number*-Datentyps können Sie über die *Number.MAX_SAFE_INTEGER* auslesen. Eine Zahl im *BigInt*-Format können Sie auf zwei verschiedene Arten erzeugen: Entweder hängen Sie an eine Zahl ein *n* an oder Sie erzeugen die Zahl über die *BigInt*-Funktion. Beim Einsatz von *BigInt* müssen Sie einige Dinge beachten: Sie können einen *BigInt* nicht mit den Operationen der JavaScript-*Math*-Klasse verwenden. Auch die Verwendung von *BigInt* und *Number* innerhalb einer Operation ist nicht erlaubt und führt zu einem Type Error. In einem solchen Fall müssen Sie die beteiligten Operanden in ein einheitliches Format bringen. Als Operationen werden alle Grundrechenarten sowie der Moduloperator und die Bitoperatoren unterstützt. Gerade bei der Division mit *BigInt* müssen Sie beachten, dass es sich bei *BigInt* um Ganzzahlen handelt. Konkret bedeutet das, dass die Nachkommastellen des Ergebnisses einfach abgeschnitten werden. Das Ergebnis der Operation *8 / 3* ist der *BigInt 2*. Nutzen Sie den *typeof*-Operator mit einem *BigInt*, ist das Ergebnis die Zeichenkette *big-int*. Auch beim Vergleich zwischen den Typen *Number* und *BigInt* sieht der ECMAScript-Standard einen Bruch vor. So liefert ein einfacher Vergleich (mit zwei Gleichheitszeichen)

bei gleichen Werten das Ergebnis *true*, bei einem strikten Vergleich (mit drei Gleichheitszeichen) ergibt *2 === 2n* den Wert *false*. In Listing 3 sehen Sie einige Beispiele zum Einsatz von *BigInt*.

Promise.allSettled

Auch das Promise-API bekommt mit der neuen ECMAScript-Version Zuwachs in Form der *allSettled*-Methode. Bisher sah der ECMAScript-Standard lediglich die beiden Hilfsmethoden *all* und *race* für den Umgang mit mehreren Promises vor. *Promise.all* können Sie verwenden, um die Ergebnisse aus einem Array von Promises zu sammeln und die Ausführung erst fortzusetzen, sobald alle Promises resolved sind. Wird eine Promise des Arrays rejected, gilt auch das Promise-Objekt, das *Promise.all* zurückgibt, als rejected. *Promise.race* akzeptiert ebenfalls ein Array aus Promises, arbeitet allerdings nur mit dem Ergebnis der schnellsten Promise weiter, egal, ob das ein *resolve* oder ein *reject* ist.

In keinem der beiden Fälle ist es ohne Weiteres möglich, mit der Ausführung unabhängig vom Ergebnis der Promises fortzufahren. Dieses Problem löst nun die *Promise.allSettled*-Methode. *Promise.allSettled* akzeptiert, wie *Promise.all* und *Promise.race*, ein Array von Promises und gibt selbst ebenfalls ein Promise-Objekt zurück. Es wird resolved, sobald alle Promises aus dem Übergabearray entweder resolved oder rejected sind. Das resultierende Array, auf das Sie in der *then*-Methode der Promise zugreifen können, unterscheidet sich etwas von dem, das Sie bei *Promise.all* erhalten. Da Sie auf jeden Fall Zugriff auf alle Ergebnisse haben, müssen Sie unterscheiden können, ob es sich um einen Erfolg oder einen Fehlschlag handelt. Zu diesem Zweck weist das Ergebnisarray in *Promise.allSettled* pro Promise im ursprünglichen Array ein Objekt mit der Eigenschaft *status* auf. Diese kann entweder den Wert *rejected* im Fall eines Fehlers oder den Wert *fulfilled* im Erfolgsfall annehmen. Bei einem Fehlschlag enthält die Eigenschaft *reason* den Wert, der beim Reject übergeben wurde. Auf den Wert des Erfolgsfalles greifen Sie über die Eigenschaft *value* zu. Listing 4 enthält ein kurzes Beispiel, um diesen Sachverhalt zu demonstrieren.

Führen Sie den Quellcode aus Listing 4 in Node.js aus, sehen Sie nach einer Sekunde die Struktur des Ergebnisarrays aus *Promise.allSettled*. Die erste Promise aus der *p1*-Funktion wird nach einer Sekunde mit dem Wert *Rejected Promise* rejected und die zweite Promise wird sofort resolved. Das Ergebnisarray enthält die Resultate in der Reihenfolge des Eingabearrays und nicht in der Reihenfolge, in der die Ergebnisse vorliegen.

Listing 4: Die Arbeit mit Promise.allSettled

```
const p1 = () =>
  new Promise((resolve, reject) =>
    setTimeout(() => reject('Rejected Promise'), 1000),
  );
const p2 = Promise.resolve('Resolved Promise');

Promise.allSettled([p1(), p2]).then(result => console.log(result));
```

globalThis

Eine der größten Herausforderungen für Einsteiger in JavaScript ist die Bestimmung des Werts von *this* im Code. In den Methoden einer Klasse ist der Fall relativ klar: *this* ist eine Referenz auf eine konkrete Instanz der Klasse. Doch *this* ist auch außerhalb des Klassenkontexts definiert und weist dort je nach Ausführungsumgebung unterschiedliche Werte auf. Die Spanne reicht hier von null über ein leeres Objekt bis hin zu einer Referenz auf das globale Objekt. Noch schwieriger wird der Fall, wenn Sie innerhalb einer Methode einer Klasse eine Callback-Funktion nutzen möchten. Greifen Sie in dieser Callback-Funktion auf *this* zu, ist das nicht mehr die Referenz auf das Objekt, sondern das globale *this*. Nutzen Sie im Gegensatz dazu eine Arrow-Funktion, ist *this* wieder die Referenz auf das Objekt. In Listing 5 sehen Sie ein Beispiel für diese Fälle.

In manchen Fällen kann es erforderlich sein, auf das globale Objekt zuzugreifen. Bis zur Standardisierung des *globalThis*-Objekts half an dieser Stelle nur ein Hack. Sie mussten überprüfen, in welcher Umgebung Sie sich befanden beziehungsweise welches globale Konstrukt

verfügbar war. Zur Auswahl stehen hier: *self, window* und *global*. Der Grund hierfür ist, dass je nach Umgebung, also Browser oder Node.js, und je nach konkreter Implementierung ein anderes globales Konstrukt zur Verfügung steht. Der *globalThis*-Standard vereinheitlicht die Umgebungen, sodass Sie sich überall darauf verlassen können, dass dieses Konstrukt global existiert.

Trotz der Tatsache, dass der ECMAScript-Standard ein solches globales Objekt vorsieht, sollten Sie es möglichst wenig verwenden. Es sprechen einige Gründe gegen globale Strukturen in JavaScript. So kann es sehr leicht zu Namenskonflikten und Überschreibungen von Werten kommen. Gerade in größeren Applikationen, in denen es schwerfällt den Überblick zu behalten, oder durch die Einbindung von Bibliotheken von Drittanbietern, die ebenfalls globale Werte setzen, kann es vorkommen, dass eine globale Eigenschaft von mehreren Stellen eingesetzt wird, was zu schwer nachvollziehbaren Fehlern führt. Außerdem gegen globale Werte spricht, dass der Garbage Collector der JavaScript Engine diese nie bereinigen kann, da es zur Laufzeit der Applikation mit der globalen Referenz immer einen Verweis auf den Wert gibt und der Speicher somit nicht freigegeben werden kann. Speichern Sie größere Datenstrukturen wie Teilbäume des DOM in einer globalen Variablen, kann das leicht zu einem Memory Leak führen.

For-in Mechanics

Hinter dem Feature mit dem Titel For-in Mechanics verbirgt sich eine Spezifikation über das Verhalten der *for-in*-Schleife in JavaScript. Mit einer solchen Schleife können Sie über die Eigenschaften eines Objekts iterieren, haben bei jeder Iteration Zugriff auf den Namen der jeweiligen Eigenschaft und können damit auf den Wert zugreifen. Im Gegensatz zu einem Array, bei dem die Elemente geordnet sind, ist die Reihenfolge der Eigenschaften eines Objekts nicht spezifiziert. Das bedeutet dann allerdings auch, dass die Durchläufe der *for-in*-Schleife beliebig erfolgen können. Als Folge dieser Spezifikationslücke gibt es unterschiedliche Implementierungen der Browserhersteller, die zu verschiedenen Ergebnissen und Fehlern in der Ausführung einer Applikation führen können. Aus diesem Grund hat das TC39 an dieser Stelle nachgelegt und die Ausführung der *for-in*-Schleife spezifiziert.

Diese Spezifikation wirkt sich jedoch nicht nur auf die Reihenfolge der Eigenschaften in der *for-in*-Schleife aus, sondern noch auf eine Reihe weiterer Konstrukte wie beispielsweise *Object.keys, Object.values* und *Object.entries*. Methoden, die von dieser Spezifikation nicht betroffen sind, sind beispielsweise *Object.assign* und *Object.create*.

Listing 5: this im Kontext einer Klasse

```
class GlobalThisClass {
  defineGlobal() {
    globalThis.someValue = 'My global value';
  }

  logGlobal() {
    console.log(globalThis.someValue);
  }
}

const g = new GlobalThisClass();
g.defineGlobal();
g.logGlobal();
```

Optional Chaining

Falls Sie über Erfahrung in der Entwicklung von JavaScript-Applikationen verfügen, kennen Sie diese Fehlermeldung bestimmt: „TypeError: Cannot read property ‚x' of undefined". Diese Meldung erhalten Sie, wenn Sie versuchen, innerhalb eines verschachtelten Objekts auf eine Eigenschaft einer nicht definierten Eigenschaft zuzugreifen. Die Lösung hierfür ohne das Optional-Chaining-Feature besteht aus einer Reihe von Prüfungen, die sicherstellen, dass die jeweilige Eigenschaft existiert. In Listing 6 sehen Sie ein konkretes Beispiel.

Im Beispiel nutzen Sie das *user*-Objekt und möchten die Eigenschaft *city* der *address*-Eigenschaft auf der Konsole ausgeben. Ohne vorherige Prüfung würde ein solcher Zugriff zum Abbruch der Ausführung der Applikation wegen des zuvor erwähnten Type Errors führen. Also müssen Sie vor der Verwendung der Eigenschaft prüfen, ob der gesamte Pfad im Objekt existiert. In unserem Fall mit einem dreistufigen Pfad und nur einer Eigenschaft, die verwendet wird, ist das wenig problematisch. Bei längeren Pfaden oder vielen unterschiedlichen Konstellationen von Eigenschaften wird der Quellcode Ihrer Applikation schnell unübersichtlich. Um diese häufige Problemstellung zu lösen, hat das TC39 das Optional-Chaining-Feature in den ECMAScript-Standard aufgenommen. In Listing 7 sehen Sie ein Stück Quellcode, das ebenfalls auf *user.address.city* zugreift und den entsprechenden Wert auf der Konsole ausgibt. In diesem Fall jedoch mit Optional Chaining.

Wie Sie hier sehen können, wird mit dem neuen *?.*-Operator gearbeitet. Der sorgt dafür, dass die JavaScript Engine selbst überprüft, ob die *address*-Eigenschaft des

Listing 6: Zugriff auf eine Eigenschaft ohne Optional Chaining

```
const user = {
  name: 'Klaus'
}

if (user && user.address) {
  console.log(user.address.city);
}
```

user-Objekts existiert. Ist das der Fall, wird die *city*-Eigenschaft verwendet. Falls es keine *address*-Eigenschaft im *user*-Objekt gibt, wird mit dem Wert *undefined* für den gesamten Pfad weitergearbeitet. Für die Applikation wirkt es, als würde die *address*-Eigenschaft existieren, die *city*-Eigenschaft aber nicht beziehungsweise, dass diese Eigenschaft den Wert *undefined* beinhaltet. Der wichtigste Unterschied ist, dass die JavaScript Engine beim Zugriff über den Optional-Chaining-Operator keine Exception wirft und die Applikation damit nicht beendet wird.

Nullish Coalescing

Bei dem Nullish Coalescing- oder *??*-Operator handelt es sich, wie auch beim Optional Chaining, um eine Erweiterung des JavaScript-Sprachkerns, die häufig vorkommende Anforderungen im Entwicklungsalltag deutlich vereinfacht und die Applikationen robuster gestalten soll. Im weitesten Sinne ist der *??*-Operator mit einem logischen *OR* vergleichbar. Der Unterschied ist, dass Nullish Coalescing mit Null-ähnlichen Werten, also mit *null* und *undefined*, arbeitet, der logische *OR*-Operator aber mit allen *falsy*-Werten von JavaScript, also auch beispielsweise mit leeren Strings oder der Zahl Null.

Der *??*-Operator ist ein binärer Operator, akzeptiert also zwei Operanden. Einer davon steht links vom Operator, der zweite rechts davon. Wertet die JavaScript Engine den Ausdruck aus, wird der linke Operand zurückgegeben, falls dieser nicht den Wert *null* oder *undefined* aufweist, andernfalls wird der rechte Operand zurückgegeben. Listing 8 zeigt ein konkretes Beispiel für den Nullish-Coalescing-Operator.

Mit dem Nullish-Coalescing-Operator kennen Sie nun alle Features der Version 2020 des ECMAScript-Standards. In der Pipeline für die nächsten Versionen befinden sich allerdings bereits einige weitere sehr interessante Features, auf die wir im Folgenden noch einen kurzen Blick werfen wollen. Einige dieser Features sind bereits in verschiedenen JavaScript Engines implementiert oder können über Transpiler und Polyfills verwendet werden.

Listing 7: Zugriff auf eine Eigenschaft mit Optional Chaining

```
const user = {
  name: 'Klaus',
};

console.log(user.address?.city);
```

Listing 8: Nullish Colescing

```
const value = 0;
const result = value ?? -1;
console.log(result); // 0

const value2 = null;
const result2 = value2 ?? -1;
console.log(result2); // -1
```

Listing 9: Top Level Await

```
const p = Promise.resolve(4);

const result = await p;

console.log(result);
```

Ausblick auf einige zukünftige Features

In den folgenden Abschnitten möchte ich Ihnen noch eine kleine Auswahl von Features vorstellen, die es wahrscheinlich in näherer Zukunft in den Standard schaffen werden. Diese Features sind:

- Top Level Await
- Numeric Separators
- Private Methods und Getter/Setter

Top Level Await

Mit dem *await*-Schlüsselwort können Sie auf eine Promise warten. Das bedeutet, dass Sie nicht mehr mit Callback-Funktionen arbeiten müssen, sondern den Rückgabewert einer Promise-basierten Funktion direkt verwenden können. Die Fehlerbehandlung erfolgt mittels *try-catch*-Statement und ebenfalls ohne Callback-Funktion. Bei der Verwendung von *await* gibt es allerdings eine Einschränkung: Der bisherige ECMAScript-Standard sieht die Verwendung des *await*-Schlüsselwortes immer nur in Verbindung mit *async*-Funktionen vor. Das bedeutet, dass Sie eine solche Operation immer in eine *async*-Funktion wrappen müssen. Gerade im Node.js-Kontext kommt es häufiger vor, dass eine Promise-basierte Funktion direkt im Modulkontext und nicht innerhalb einer Funktion ausgeführt wird. Mit dem neuen Top-Level-Await-Standard wird es nun möglich, das *await*-Schlüsselwort losgelöst von *async*-Funktionen zu verwenden. Listing 9 enthält ein einfaches Beispiel für die Verwendung von Top Level Await.

Numeric Separators

Arbeitet man mit großen Zahlen, kann es schnell passieren, dass der Überblick verloren geht. Aus diesem Grund werden häufig die sogenannten Tausendertrenner eingesetzt. Nachdem der Punkt bereits als Dezimaltrennzeichen verwendet wird und auch das Komma eine semantische Bedeutung in JavaScript hat, fiel die Wahl für den Tausendertrenner auf den Unterstrich. Der Unterstrich ist zwar als Tausendertrenner gedacht, kann allerdings an beliebigen Stellen innerhalb einer Zahl verwendet werden, ohne dass ein Fehler geworfen wird. Sie können den Numeric

Separator sowohl bei Zahlen vom Typ *Number*, also ganz gewöhnlichen Zahlen, als auch bei *BigInt* nutzen. In Listing 10 sehen Sie einige Anwendungsbeispiele.

Private Methods und Getter/Setter

Lange Zeit gab es in JavaScript nur eine Variante des Zugriffs auf Eigenschaften und Methoden eines Objekts: *public*. Das bedeutet, dass aus der kompletten Applikation auf alle Eigenschaften und Methoden eines Objekts zugegriffen werden konnte, solange sich das Objekt selbst im Scope befand. Zugriffsmodifikatoren wie *private* oder *protected*, wie sie aus anderen Programmiersprachen oder Type-Script bekannt sind, existieren in JavaScript nicht. Zwar sind die Schlüsselwörter *public, protected* und *private* schon seit langer Zeit reserviert, aber dennoch geht der ECMAScript-Standard hier einen ganz anderen Weg. Die Definition privater Eigenschaften in einer Klasse ist bereits Bestandteil des Sprachstandards und sieht vor, dass diese Eigenschaften mit einem Hashzeichen (#) eingeleitet werden. Die gleiche Schreibweise wird auch für private Methoden angewendet. Mit dieser Spracherweiterung haben Sie jetzt die Möglichkeit, den Zugriff auf die Eigenschaften und Methoden Ihrer Klassen detaillierter und vor allem auch zur Laufzeit zu steuern. Die Möglichkeiten, die Ihnen bisher im Rahmen von TypeScript zur Verfügung gestanden haben, sind nur auf die Compilezeit beschränkt, sodass Sie während der Ausführung Ihrer Applikation dennoch auf geschützte Inhalte zugreifen können.

Fazit

Niemand konnte die Erfolgsgeschichte von JavaScript vorhersagen. Was als einfache Skriptsprache im Browser für dynamische Effekte in sonst statischen HTML-Seiten begann, hat sich mittlerweile zur am weitesten verbreiteten Programmiersprache weltweit entwickelt. Doch an einigen Stellen merkt man JavaScript seine Vergangenheit mehr oder weniger deutlich an. Gerade bei performancekritischen oder sehr umfangreichen Applikationen kommen Sie als Entwickler immer wieder an die Grenzen der Sprache. Mit der Weiterentwicklung des Sprachstandards werden jedoch viele Probleme aus dem Entwickleralltag adressiert und auf Sprachebene gelöst. Die besten Beispiele hierfür sind die Integration der *class*-Syntax und die native Implementierung von Promises. An einigen Stellen werden bestehende Features der Sprache weiterentwickelt, an anderen komplett neue Konzepte und Strukturen implementiert.

Mit Node.js hat JavaScript den Browser als Laufzeitumgebung verlassen, was zu einer Menge neuer und anders gelagerter Problemstellungen führt. Mit den neuen Sprachfeatures wachsen clientseitiges und serverseitiges JavaScript wieder mehr zusammen, sodass die Wiederverwendung einzelner Codefragmente auf beiden Seiten zunehmend einfacher möglich wird.

Das Erfreuliche am ECMAScript-Standard ist, dass er für jeden einsehbar entwickelt wird, sodass neue Features kein großes Geheimnis sind. Die JavaScript-Community nutzt diese Möglichkeit und stellt teilweise schon, lange bevor alle Browserhersteller ein Feature implementiert haben, Polyfills zur Verfügung, mit denen sich die Features bereits nutzen lassen.

Listing 10: Der Numeric Separator

```
const amount = 1_234_567;
console.log(amount); // 1234567
```

Sebastian Springer ist JavaScript-Entwickler bei MaibornWolff in München und beschäftigt sich vor allem mit der Architektur von client- und serverseitigem JavaScript. Sebastian ist Berater und Dozent für JavaScript und vermittelt sein Wissen regelmäßig auf nationalen und internationalen Konferenzen.

Links & Literatur

[1] https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf

[2] https://github.com/tc39/proposals

[3] https://kangax.github.io/compat-table

Natives Elasticsearch auf Kubernetes

ECK macht es einfach

Vor nicht langer Zeit war es undenkbar, persistente Daten auf Kubernetes zu stellen. Kluge Ops-Teams hätten eher auf einen vergleichbaren Cloud-Service gesetzt und die Funktionalität für Simplizität und ein ruhiges Gewissen geopfert. In den letzten Jahren ist Kubernetes aber gereift und in Bereiche vorgestoßen, die zuvor nicht annähernd als geeignet für die Containerorchestrierung angesehen wurden.

von Dimitri Marx

Kubernetes hat an Stabilität und Funktionsumfang zugenommen und sich von einem Tool zu einer richtigen Plattform entwickelt. Eines der mächtigsten Features der letzten Releases ist die Kombination von Custom Resource Definitions (CRDs) und dem Operator Pattern.

Das Operator Pattern (oder einfach Operators) stellt ein sehr mächtiges Konzept dar, mit dem Softwareanbieter Betriebsmuster in Kubernetes abstrahieren können. Der Einsatz des Operator Patterns kann die notwendige Menge an Tools, Skripten und manueller Arbeit zum Betreiben komplexer Software massiv reduzieren. Operators erlauben es, innerhalb eines Dokuments – in einem sogenannten Kubernetes-Manifest – das gesamte Deployment zu beschreiben, nicht nur die Artefakte, die deployt werden. Auch wenn man eine Datenbank sehr schnell auf Kubernetes deployen kann, wenn man Helm oder ein Kubernetes-Manifest verwendet, ist es doch meist Aufgabe des Endnutzers, für Sicherheit, Back-ups und Nutzerverwaltung zu sorgen.

Im Gegensatz dazu können viele dieser Aufgaben in einem System, das via CRDs und Operators deployt wird, automatisiert und mit der Kubernetes Configuration Language definiert werden. Diese Herangehensweise macht es leichter, bestehende Cluster und deren Zustand bzw. Verhalten zu durchblicken, die Konfiguration zu zentralisieren und Best Practices bestimmter Anbieter anzuwenden. Obwohl das Operator Pattern noch relativ neu ist, wird es bereits bei vielen Anbietern (inkl. Elastic) eingesetzt, um einen Betrieb der eigenen Programme auf Kubernetes zu ermöglichen.

Für Fans von Elasticsearch und dem wachsenden Elastic Stack bietet sich so die Möglichkeit, die Vorteile der Standarddistribution nativ auf Kubernetes zu nutzen. Elastic hat hierfür kürzlich Elastic Cloud on Kubernetes (ECK) veröffentlicht, es soll zum offiziellen Weg werden, Elastic Cloud auf Kubernetes auszuführen. Ein

recht großer Funktionsumfang ist im Operator bereits enthalten (Kasten: „Fünf Vorteile des nativen Betreibens von Elasticsearch auf Kubernetes via ECK“).

Bis vor Kurzem lief die Installation eines produktionsfertigen Elastic Stack ziemlich standardmäßig ab. Man erstellt ein Set von Masters, um den Cluster zu managen, dann werden Nodes hinzugefügt und konfiguriert, sie können verschiedene Rollen einnehmen. Je nach Automatisierung war das Management des Elastic-Clusters sehr umfangreich, und eine gewisse Orchestrierung wurde nötig, um sicherzugehen, dass der Cluster aktiv und bereit für Traffic ist.

Das Erstellen einer produktionsfertigen Elastic-Umgebung ist nicht annähernd so kompliziert wie das anderer Enterprise-Software. Doch es ist komplex genug, dass ein gewisses Maß an Planung, Wissen, Automatisierung und Ressourcen unabdinglich ist, um dessen Bereitschaft zur Nutzung sicherzustellen. Mit ECK existiert eine gute Alternative für das Deployment von Elastic, mit der die Vorteile von Kubernetes und des Operator Patterns genutzt werden können. Auch wenn dies das generelle Clustermanagement nicht komplett obsolet macht, werden so einige Arbeitsschritte obsolet und einige Aufgaben erleichtert. Elemente wie Clusterupgrades, das Absichern des Clusters und dynamische Clusterskalierung sind in ECK out of the box verfügbar. Gleiches gilt für eine vereinfachte Automatisierung.

Als Beispiel einer ECK-Installation wollen wir einen produktionsreifen Elastic Stack auf einem existierenden Kubernetes-Cluster installieren. Dieser kann entweder lokal via Minikube oder K3s erstellt werden, oder man deployt ihn auf einem verwalteten Cluster in der Cloud (etwa GCP oder EKS). Um zu starten, müssen wir die Custom Resource Definitions und die Operators auf dem Cluster installieren: *kubectl apply -f https://download.elastic.co/downloads/eck/1.0.0-beta1/all-in-one.yaml*. Der Befehl wendet alle Ressourcen an, aus denen sich die verschiedenen Komponenten-ECK zusammensetzen.

Jetzt haben wir die ECK-Komponenten installiert und können unsere Elastic-Konfiguration erstellen. In diesem Beispiel werden wir ein robustes Layout mit mehreren Mastern, separaten Daten-Nodes und dedizierten Ingress-Servern erstellen. Wir beginnen mit der Erstellung der Master-Nodes. Erstellen Sie eine neue Datei *elastic.yaml* und fügen den Code aus Listing 1 ein.

Die ersten beiden Zeilen definieren, welches Kubernetes API wir für die Erstellung der neuen Ressourcen verwenden werden; in diesem Fall ein API, das über das Elastic CRD und eine Ressourcenart von Elasticsearch bereitgestellt wird. Dieses Beispiel zeigt die Vorteile von CRDs als System zur Erweiterung von Kubernetes, das es Produkten erlaubt, Funktionalität in einem bestimmten Namespace hinzuzufügen.

Listing 1

```
apiVersion: elasticsearch.k8s.elastic.co/v1beta1
kind: Elasticsearch
metadata:
  name: monitoring
  namespace: monitoring
spec:
  version: 7.4.0
  nodeSets:
  - name: master
    count: 1
    config:
      node.master: true
      node.data: false
      node.ingest: false
      node.store.allow_mmap: false
    resources:
      requests:
        memory: 4Gi
        cpu: 2
      limits:
        memory: 4Gi
        cpu: 2
    env:
    - name: ES_JAVA_OPTS
      value: "-Xms3g -Xmx3g"
    volumeClaimTemplates:
    - metadata:
        name: elasticsearch-data
      spec:
        accessModes:
        - ReadWriteOnce
        resources:
          requests:
            storage: 10Gi
        storageClassName: standard
```

Fünf Vorteile des nativen Betreibens von Elasticsearch auf Kubernetes via ECK

- *Sicherheit von Beginn an*: ECK [1] konfiguriert die Sicherheitseinstellungen, Node to Node TLS, Zertifikate und einen Defaultuser für jeden Cluster automatisch.
- *Kubernetes-native Elasticsearch-Ressourcen*: Elasticsearch kann genau wie jede andere Kubernetes-Ressource genutzt werden. Es gibt keine Notwendigkeit, endlose Kubernetes Pods, Services und Secrets zu konfigurieren.
- *Best Practices von Elastic*: ECK funktioniert im täglichen Elasticsearch-Betrieb nach den etablierten Prinzipien – von der Skalierung bis zum Versionswechsel.
- *Exklusive Elastic-Features*: Zugang zu allen Funktionen von Elastic, inklusive Elastic SIEM [2], Observability [3], Logs [4], Infrastruktur [5] und mehr.
- *Fortschrittliche Topologie*: Nutzer können die Vielseitigkeit der eigenen Kubernetes-Infrastruktur auf das Elasticsearch Deployment anwenden. So können etwa auch Hot-Warm-Cold-Architekturen [6] genutzt werden, um die Kosten zu reduzieren.

Als Nächstes definieren wir einige Metadaten: Den Namen des Elastic Stack und den Namespace, in dem wir ihn erstellen werden. Das Hinzufügen eines Namespace hilft dabei, den ECK-Stack zu organisieren, sodass Sie Ihre Monitoring-Workload einfach von Ihren anderen Workloads trennen können. Das ermöglicht eine granularere Sicherheit innerhalb Ihres Kubernetes-Clusters. Wird der Namespace weggelassen, erstellt das ECK CRD alle Ressourcen im Standard-Namespace.

Wir kommen zum Kern der Konfiguration, der Spezifikation. Das ist im Wesentlichen der einzige Ort, an dem Konfigurationselemente für ECK-Komponenten abgelegt werden. In diesem Beispiel spezifizieren wir *nodeSet:*, eine Menge von Elastic Nodes, seien es Master, Daten oder Ingests. Es ist eine Option, die in die Nodeset-Konfiguration übertragen wird. Die Nodes werden generiert und der zu erstellenden Elastic-Konfiguration hinzugefügt. Für unsere Konfiguration setzen wir dieses spezielle Nodeset als Master Nodes ein und stellen sicher, dass wir drei haben. Das wird im Allgemeinen als Best Practice angesehen, da es (Quorum-basierte) Entscheidungsfindung innerhalb des Stacks ermöglicht. Im Allgemeinen sollten Sie immer eine ungerade Anzahl von Master Nodes haben.

Jetzt übermitteln wir die Konfigurationselemente. ECK unterstützt die meisten Optionen wie Nicht-ECK-Installationen. Besonders hervorzuheben ist in diesem Beispiel die Option *node.store.allow_mmap:*. Dieses Konfigurationselement ist in manchen Fällen für Kubernetes-Nodes notwendig, die eine niedrige Kerneleinstellung für *vm.max_map_count:* besitzen. Diese Einstellung hat jedoch Auswirkungen auf die Leistung bei Stacks mit hohem Durchsatz, sodass sie nicht für stark ausgelastete Cluster empfohlen wird. Stattdessen sollte die Konfiguration aus Listing 2 zum Manifest innerhalb des Nodeset Tree hinzugefügt werden.

Diese Einstellung gibt die Kerneloptionen an den darunterliegenden Node weiter. Sie setzt jedoch voraus, dass Sie in der Lage sind, privilegierte Container auszuführen. Da die Privilegierung keineswegs sicher ist, insbesondere auf hochsicheren Kubernetes-Clustern, lohnt es sich, bei der Planung Ihres Elastic ECK Rollouts zu prüfen, ob sie verfügbar ist.

Wir kehren zu Listing 1 zurück und schauen uns das Festlegen von Ressourcenzuweisungen auf verschiedenen Ebenen innerhalb des Stacks an. Werden diese Einstellungen nicht vorgenommen, verwendet das CRD die Standardeinstellungen, was die Gesamtleistung des Clusters erheblich beeinflussen kann.

Die ersten beiden Elemente in diesem Teil, *resources:* und *env:*, setzen die Limitierungen des Volumens der verschiedenen Ressourcen, die das Nodeset verbrauchen darf. In unserem Beispiel darf jeder unserer drei Master Nodes 4 GB RAM und zwei CPU-Kerne aus dem Kubernetes-Cluster nutzen. Ohne diese Konfiguration können die Elastic Instances ein weitaus größeres Volumen an Ressourcen verbrauchen, als Sie vielleicht wünschen, was das Planen von Arbeitslasten innerhalb des Clusters beeinträchtigen kann. *env* setzt eine Umgebungsvariable, um die Limitierung an die JVM zu übergeben. Ohne diese Einstellung sind die JVM-Optionen auf 1 GB RAM voreingestellt, d. h. unabhängig davon, wie viel Sie im Kubernetes-Cluster zulassen, kann die JVM sie nicht effektiv nutzen.

Schließlich richten wir das *volumeclaimTemplate:* ein. Es ist entscheidend, da es definiert, wie und wo die Daten für Elastic gespeichert werden. Standardmäßig setzt ECK einen Speicheranspruch von 1 GB innerhalb des Clusters. Diese Vorgabe stellt sicher, dass Daten von den Nodes getrennt gehalten werden, sodass die Informationen bei einem Node Replacement oder einer Löschung nicht verloren gehen. Er stellt jedoch nicht sicher, dass auf diesem Volumenanspruch genügend Platz oder Performance für eine Produktionslast vorhanden ist.

Stattdessen setzen wir, wie in diesem Beispiel, ein explizites *volumeclaimTemplate:* und verwenden eine *storageClassName:*-Einstellung, um sicherzustellen,

Listing 2

```
podTemplate:
  spec:
    initContainers:
    - name: sysctl
      securityContext:
        privileged: true
      command: ['sh', '-c', 'sysctl -w vm.max_map_count=262144']
```

Listing 3

```
- name: ingest
  count: 2
  config:
    node.master: false
    node.data: false
    node.ingest: true
    node.store.allow_mmap: false
  resources:
    limits:
      memory: 4Gi
      cpu: 4
  env:
    - name: ES_JAVA_OPTS
      value: "-Xms3g -Xmx3g"
  volumeClaimTemplates:
    - metadata:
        name: elasticsearch-data
      spec:
        accessModes:
        - ReadWriteOnce
        resources:
          requests:
            storage: 10Gi
        storageClassName: ssd
```

Listing 4

```
- name: data
  count: 2
  config:
    node.master: false
    node.data: true
    node.ingest: true
    node.store.allow_mmap: false
  resources:
     limits:
       memory: 3Gi
       cpu: 4
  env:
    - name: ES_JAVA_OPTS
      value: "-Xms3g -Xmx3g"
  volumeClaimTemplates:
  - metadata:
      name: elasticsearch-data
    spec:
      accessModes:
      - ReadWriteOnce
      resources:
        requests:
          storage: 100Gi
      storageClassName: ssd
```

dass der richtige Laufwerktyp verwendet wird. Der Typ und die Größe des Laufwerks hängen von Ihrem Kubernetes-Anbieter ab. Meist wird ein Standard (plattenbasierte Speicherung oder langsame SSD) und eine Form von schnellerem Speicher (wie schnelle SSD oder auch exotischere schnelle Speicherung) angeboten. Auch hier ist die Abstimmung des Speichers auf die Workload einer der wichtigsten Aspekte von Elastic Deployments. ECK macht es leicht, den richtigen Speicher dem richtigen Node zur Verfügung zu stellen.

Nun, da wir unseren Master definiert haben, können wir einige Ingest Nodes definieren, indem wir die Konfiguration aus Listing 3 in unsere *elastic.yaml*-Datei hinzufügen.

Wie Sie erkennen können, sieht dies dem vorherigen Beispiel sehr ähnlich, da die gleiche CRD sie erstellt hat. In diesem Fall ändern wir drei Dinge, um dieses Nodeset für die Aufnahme bereit zu machen: *count:*, Grenzen für die *cpu:*-Limits und *storageclassName:*. Dieser Unterschied in den Einstellungen soll die besondere Rolle der Ingest Nodes berücksichtigen, da sie genug CPU zum Parsen eingehender Anfragen, eine schnelle Festplatte, um sie zu puffern, und kein Quorum benötigen.

Um unsere Daten-Nodes mit der gleichen Vorlage zu erstellen, fügen wir den Code aus Listing 4 in die Datei *elastic.yaml* ein.

Auch hier haben wir einige kleinere Änderungen vorgenommen: die Anzahl auf vier Nodes geändert, die CPU auf vier Cores erhöht und eine große Menge an High-Speed-Disks konfiguriert.

Wir haben jetzt alles, was wir brauchen, um Elastic in unserem Kubernetes-Cluster einzusetzen. Zunächst ist es jedoch sinnvoll, auf das Konzept der Pod Affinity einzugehen. Dieses erlaubt es Kubernetes, die Pods so zu planen, dass sie entweder auf dem gleichen Node eingeplant werden oder, im Fall der Anti-Affinity, getrennt gehalten werden, wo immer es möglich ist. Diese Eigenschaft ist besonders relevant im Fall von Elastic, wo man bestimmte Nodes sowohl aus Performance- als auch aus Zuverlässigkeitsgründen auseinanderhalten möchte.

Das Scheduling ist besonders für die Master relevant, da der Ausfall eines Kubernetes Node mit mehreren Mastern die Funktionsfähigkeit des Elastic Stack gefährden kann. Standardmäßig legt ECK eine Standard-Policy der Pod Anti-Affinity für alle von ihm erstellten Ressourcen fest, um sicherzustellen, dass Kubernetes versucht, sie auseinanderzuhalten. Diese Voreinstellung ist in den meisten Fällen problemlos, kann aber bei Bedarf mit der üblichen Kubernetes-Konfiguration überschrieben werden: *https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity*. Es ist erwähnenswert, dass man, um die Elastic-Komponenten korrekt getrennt zu halten, genügend Kubernetes Nodes benötigt, um dies zu unterstützen. Zum Beispiel braucht man bei drei Master-, zwei Daten- und zwei Ingest Nodes mindestens sieben Nodes, um sie getrennt zu halten, und diese Nodes müssen in der Lage sein, die Elastic Pods

zu planen. Beim Ausführen des Stacks ist es sinnvoll, das Ausmaß der Co-Lokalisierung von Pods zu überwachen. Falls dies geschieht, sollte dem entweder durch das Festlegen von eindeutigen Richtlinien für die Anti-Affinity oder durch das Erhöhen der Anzahl der Kubernetes-Nodes entgegengewirkt werden.

ECK legt auch ein Standard-Pod-Disruption-Budget (pdb) für den Elastic Stack fest. Standardmäßig ist es auf 1 gesetzt, was bedeutet, dass immer nur ein Pod außer Betrieb sein kann. Diese Voreinstellung stellt sicher, dass Kubernetes bei der Umplanung der Elastic Pods versucht, diese auf jeweils einen Pod zu begrenzen. Diese Begrenzung ist bei der Aktualisierung des Stacks relevant, da sie sicherstellt, dass ein rolling Update erzwungen wird. Bei größeren Stapeln, bei denen mehr Pods gleichzeitig sicher außer Betrieb sein können, können Sie diese Grenze explizit höher setzen. Jetzt haben wir einige der Elemente, die ECK zur Gewährleistung der Stabilität hinzufügt, abgedeckt, sodass wir unseren Elastic-Cluster einsetzen können: *kubectl apply -f elastic.yaml*. Das wendet die Konfiguration auf Ihren Cluster an. Innerhalb weniger Minuten sollten Sie einen neuen Cluster eingerichtet haben, was Sie mit dem folgenden Befehl überprüfen können: *kubectl get pods -n elastic-cluster*. Sie sollten Folgendes erhalten:

```
monitor-es-mdi-0                1/1   Running   0      6m21s
monitor-es-mdi-1                1/1   Running   0      6m21s
monitor-es-mdi-2                1/1   Running   0      6m21s
monitor-kb-56bdd68b65-97wnz     1/1   Running   0      6m21s
```

Schließlich müssen wir Kibana zu unserer Installation hinzufügen. Kibana wird mit einem separaten CRD zum Elastic-Cluster definiert. Fügen Sie den Code aus Listing 5 zu Ihrer *elastic.yaml*-Datei hinzu (oder sogar eine separate *yaml*-Datei, wenn Sie es vorziehen, sie getrennt zu halten).

Das sollte Ihnen mittlerweile ziemlich bekannt vorkommen. Wie bei den Elastic-Komponenten beginnen wir damit, dass wir am Anfang des Dokuments erklären, dass wir die ECK CRDs verwenden, und fragen nach einem Ressourcentyp *Kibana*. Wie beim Elastic Stack geben wir ihm dann sowohl einen Namen als auch einen Namespace. Die Spezifikationsversion ist wieder sehr ähnlich. Hier geben wir die Version von Kibana an, die wir installieren möchten, plus die Anzahl der Nodes, die wir haben wollen. Da wir ein Production Deployment anstreben, sind zwei die minimale Anzahl, die wir benötigen, da sie einen kontinuierlichen Service ermöglicht, wenn ein Pod außer Betrieb ist (je nach Umfang und SLAs, unter denen Sie arbeiten, können sogar mehr als zwei gewünscht werden). Die Einführung mehrerer Nodes führt zu einem zusätzlichen manuellen Schritt, da jeder Node im Cluster den gleichen Encryption Key verwenden muss. Um ihn zu ergänzen, fügen Sie mit dem folgenden Terminalbefehl ein neues Secret zu Ihrem Kube-Cluster hinzu: *kubectl create secret generic kibana-enc-key --from-literal=xpack.security.encryptionKey =94d2263b1ead716ae228277049f19975aff8644b4fcfe429c95143c1e90938md*

Der *xpack.security.encryptionKey* kann alles sein, was Sie möchten, solange er mindestens 32 Zeichen lang ist. Weitere Details zu dieser Einstellung finden sich im Bereich der Sicherheitseinstellungen in Kibana [7].

Jetzt haben wir einen Security Key festgelegt, wir können damit den Rest dieses Teils der Konfiguration abschließen. Das nächste Element, *elasticsearchRef:*, ist der Name des Elasticsearch-Clusters, mit dem wir diese Kibana-Instanz verbinden wollen. Dies sollte so eingestellt werden, dass es mit *name:* übereinstimmt, der in den Metadaten des Elastic-Cluster-Dokuments eingestellt ist, in diesem Fall *Monitoring*. Schließlich teilen wir den Kibana-Instanzen mit, wo der Security Key zu finden ist, den wir mit dem vorherigen Terminalbefehl mit dem *secureSettings*-Key eingerichtet haben. Das definiert das Grundlayout unseres Kibana-Clusters. Allerdings sollten wir, ähnlich wie beim Elastic Stack, gewisse Bedingungen für die Ressourcennutzung festlegen. Es kann auch sein, dass wir den Kibana-Dienst über einen Load Balancer verfügbar machen wollen, entweder intern oder extern. Dazu fügen Sie das Code-Snippet aus Listing 6 zu Ihrer *kibana.yaml*-Datei hinzu.

Dieser Code gibt unserem Kibana-Cluster den letzten Schliff. Zu Beginn fügen wir ein Konfigurationselement hinzu, das an Kibana übergeben wird. Diese spezielle Einstellung *console.enabled:* schaltet die Entwicklerkonsole in Kibana aus und kann in sichereren Einstellungen wünschenswert sein. Der nächste Teil des Dokuments (*podTemplate*) ist im Wesentlichen das Gleiche wie das, was wir im Elastic-Cluster einstellen, und definiert, wie viele Ressourcen Kibana mindestens benötigt. Die *http:*-Einstellung ist der Ort, an dem wir den Benutzern unseren Kibana-Cluster präsentieren können, ohne dass sie einen Kube-Proxy oder andere, eher manuelle Methoden verwenden müssen, um in das Kubernetes-Cluster-Netzwerk zu gelangen. Das erlaubt uns, zwei wichtige Elemente zu definieren: die Art und Weise, wie der Kubernetes-Dienst definiert wird, und

Listing 5

```
apiVersion: kibana.k8s.elastic.co/v1beta1
kind: Kibana
metadata:
  name: monitoring
  namespace: monitoring
spec:
  version: 7.4.0
  count: 2
  elasticsearchRef:
    name: monitoring
  secureSettings:
    - secretName: kibana-enc-key
```

zusätzliche DNS-Namen innerhalb des generierten Zertifikats. Standardmäßig erstellt das ECK CRD Dienste als NodePorts. Das stellt sicher, dass der Zugriff nur von innerhalb des Clusters kommen kann, und bedeutet im Allgemeinen, dass Sie sich in den jeweiligen Node einloggen müssen, um Kibana zu sehen. Durch das Überschreiben dieser Vorgabe zu einem *LoadBalancer*-Typ wird der Dienst von außerhalb des Clusters zugänglich gemacht. Die letzte Einstellung *tls:* ermöglicht es, den vom Dienst erstellten Zertifikaten zusätzliche DNS-Namen hinzuzufügen. Das sollte etwaige Zertifikatsfehler bei der externen Verbindung verhindern.

Wenn Sie mit den Einstellungen zufrieden sind, speichern und wenden Sie diese mit dem Befehl *kubectl -f kibana.yaml* an. Dadurch werden die Einstellungen auf Ihren Kubernetes-Cluster angewendet. Wenn Sie keinen Load Balancer verwendet haben, sollten Sie innerhalb weniger Minuten in der Lage sein, sich mittels eines Kube-Proxys mit dem Nodeport zu verbinden. Oder mit der Load-Balancer-Adresse, wenn Sie den Kibana-Log-in-Bildschirm angezeigt bekommen. Um sich einzuloggen, benutzen Sie den Benutzernamen von Elastic und rufen mit folgendem Terminalbefehl das automatisch gesetzte Passwort ab: *kubectl get secret -n monitoring monitoring-es-elastic-user -o=jsonpath='{.data.admin-password}' | base64 --decode; echo*. Beachten Sie besonders *-n*, um sicherzugehen, den richtigen Namespace anzuwählen, der in der Clustererstellung generiert wurde. Das sollte Ihnen ein Passwort liefern, um sich in Kibana einzuloggen.

Hoffentlich haben Sie jetzt einen laufenden, produktionsbereiten Elastic-Cluster mit einer Kibana-Konsole, mit der Sie ihn abfragen können. Dieser Artikel hat die Benutzerfreundlichkeit und Leistungsfähigkeit des ECK-Produkts untersucht und gezeigt, wo es konfiguriert werden kann, um es widerstandsfähig und bereit für Benutzer zu machen. In einem nächsten Artikel werden wir untersuchen, wie wir mit Hilfe von Metricbeat und Filebeat-Daten in unserem neuen Cluster bekommen können.

Listing 6

```
config:
  console.enabled: false
podTemplate:
  spec:
    containers:
    - name: kibana
      resources:
        requests:
          memory: 1Gi
          cpu: 0.5
        limits:
          memory: 2Gi
          cpu: 2
http:
  service:
    spec:
      type: LoadBalancer
  tls:
    selfSignedCertificate:
      subjectAltNames:
      - dns: monitoring.example.com
```

Dimitri Marx ist Solutions Architect bei Elastic. Er hat Informatik studiert und arbeitet seit seinem Abschluss an Open-Source-Projekten. Dimitri unterstützt Kunden dabei, den Wert von Suchtechnologien zu verstehen und wie man diese im Enterprise-Kontext einsetzen kann. Logging, Security und Analysemetriken gehören zu seinen Tätigkeitsfeldern, genau wie die Lösung der größten Herausforderungen auf dem Gebiet der strukturierten und unstrukturierten Daten.

Links & Literatur

[1] https://www.elastic.co/de/blog/announcing-elastic-cloud-on-kubernetes-eck-0-9-0-alpha-2

[2] https://www.elastic.co/de/blog/elastic-siem-7-3-0-released

[3] https://www.elastic.co/de/blog/kubecon-2019-elastic-doubles-down-on-observability-and-orchestration-for-kubernetes

[4] https://www.elastic.co/de/blog/elastic-logs-app-released

[5] https://www.elastic.co/de/blog/elastic-infrastructure-7-4-0-released

[6] https://www.elastic.co/de/blog/implementing-hot-warm-cold-in-elasticsearch-with-index-lifecycle-management

[7] https://www.elastic.co/guide/en/kibana/current/security-settings-kb.html

Tipps und Tricks für einen sauberen Product Backlog

Agiler Frühjahrsputz

Die Feiertage sind längst vorbei, Zeit für einen Frühjahrsputz. Das gilt nicht nur für das heimische Idyll, sondern auch für Projekte im Arbeitsalltag. Oder genauer gesagt: Es wird höchste Zeit, den Product Backlog zu pflegen. Wie man einen außer Kontrolle geratenen Product Backlog wieder in einen geordneten Zustand bringt, wird in diesem Artikel erklärt.

von Daniel Ruckriegel

Stellen wir uns folgendes Szenario vor: Ein Product Backlog wird nicht regelmäßig aufgeräumt und nicht kontinuierlich gewissenhaft gepflegt. Dadurch wurde es über die Zeit ziemlich groß, unstrukturiert und kompliziert. Das hat Konsequenzen: Der Überblick geht verloren, es häufen sich hinfällige und redundante Stories, der Product Backlog ist nicht nach Priorität geordnet, die Informationen in einigen Stories sind veraltet und die Relevanz gehört überprüft. Zudem fehlt dem Product Owner die Einschätzung, wie gut das Team einzelne Stories im Product Backlog bereits versteht und an welcher Stelle noch weiterer Refinement-Bedarf besteht.

Ist erst einmal solch ein Berg von Arbeit entstanden, ist der jeweilige Product Owner nicht zu beneiden. Trotzdem gibt es keinen Grund, die Hoffnung aufzugeben. Folgende Schritte helfen dabei, die Kontrolle zurückzugewinnen:

- Verschaffe dir einen Überblick mit Hilfe von Story Mapping
- Gruppiere konsequent Themen und Epics
- Identifiziere Items, die gelöscht werden können
- Ordne den Product Backlog nach Priorität
- Schalte den Turbo ein und nutze mit der Methode Team Estimation Game schnell und einfach die gesamte Kapazität des Teams für den Backlog-Frühjahrsputz
- Bringe durch die Einführung regelmäßiger Refinement-Termine den Product Backlog langfristig unter Kontrolle

Im Folgenden sehen wir uns diese Tipps genauer an.

Erster Tipp: Überblick verschaffen durch Story Mapping

Sind sehr viele Backlog Items in der Listenansicht des Product Backlogs in Jira oder Excel vorhanden, fällt es schwer, den Überblick zu behalten. Vielen Product Ownern hilft es, ihr Product Backlog zusätzlich als sogenannte Story Map darzustellen. Bei einer Story Map werden die Backlog Items entlang der Customer Journey (auch Backbone oder Narrative Flow genannt) angeordnet und in High-Level-Aktivitäten gruppiert (**Abb. 1**). Das hat gleich mehrere Vorteile:

- Die Darstellung verschafft deutlich bessere Übersichtlichkeit
- Die Darstellung hilft, ein gemeinsames Verständnis über das Produkt zu schaffen
- Die Darstellung hilft, im Gespräch mit Stakeholdern und Anforderungsgebern leichter die Orientierung zu behalten, über welchen Bereich gerade gesprochen wird
- Die Gruppierung nach High-Level-Aktivitäten hilft, leichter Themen und Epics abzuleiten
- Der Fokus liegt auf der Kundensicht sowie den Wirkungen, die für die Kunden erzielt werden sollen

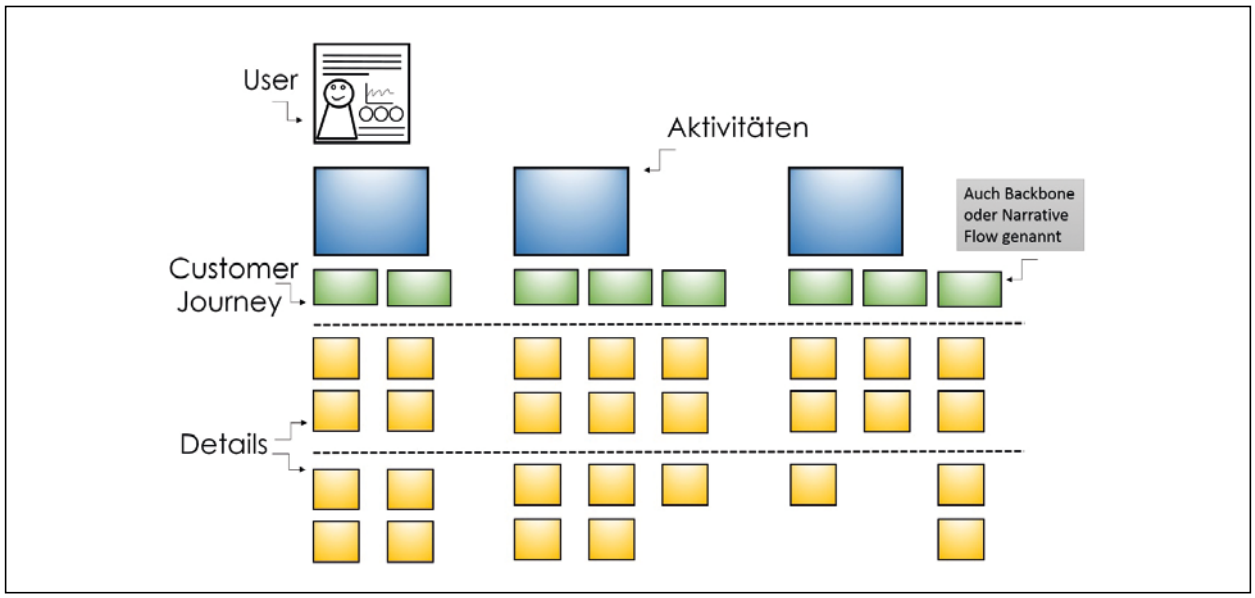

Abb. 1: Darstellung einer Story Map

Zweiter Tipp: Konsequentes Gruppieren in Themen und Epics

Hast du bereits den ersten Tipp umgesetzt, ist es bereits viel leichter, auch den nächsten Tipp umzusetzen. Durch die Story-Map-Darstellung wurden allen Backlog Items High-Level-Aktivitäten des Kunden zugeordnet. Hieraus lassen sich Themen und Epics ableiten, in die sich die Backlog Items gruppieren lassen.

Gehe nun deinen Product Backlog durch und prüfe, ob jedes Item einem Epic beziehungsweise Thema zugeordnet ist. Ein gut strukturierter Backlog ist nicht nur übersichtlicher, sondern hilft auch bei der Prüfung auf Vollständigkeit.

Dritter Tipp: Identifikation von Items, die gelöscht werden können

Viele Product Owner haben leider die Angewohnheit, Anforderungen im Product Backlog hinzuzufügen, aber nie wieder zu entfernen – selbst wenn sie nicht mehr von Interesse sind. Um den Umfang des Product Backlogs zu verringern, ist es hilfreich, die Backlog Items auf Hinfälligkeit beziehungsweise Redundanz zu prüfen. Die als hinfällig oder redundant identifizierten Stories sind anschließend zu löschen.

Vierter Tipp: Ordne den Product Backlog nach Priorität

Ein gut priorisierter Product Backlog ist ein entscheidender Erfolgsfaktor in der agilen Produktentwicklung. Mit Hilfe einer laufenden Priorisierung wird sichergestellt, dass das Team in einem volatilen, schnelllebigen Umfeld die Energie und Kapazität in die wirklich wichtigen Themen investiert. Es gibt sehr viele Priorisierungsmethoden, zum Beispiel MoSCoW, Relative Weight, Theme Screening oder WSJF. Welche dieser Methoden du wählst, ist eher sekundär. Wichtig ist, dass überhaupt eine strukturierte Priorisierung durchgeführt wird. Entscheide dich für eine Methodik und priorisiere ab jetzt regelmäßig.

Fünfter Tipp: Veranstalte einen Workshop für den Frühjahrsputz

Die bisherigen Tipps sind sehr arbeitsintensiv. Nicht in jedem Team kann ein Product Owner diese Aufgaben allein lösen. Glücklicherweise gibt es mit der Dynamic-Variante des Team Estimation Game eine erstaunlich

zeiteffiziente sowie effektive Methode, um einen großen Teil der Arbeit mit einem Mal in einer konzentrierten Teamaktion abzutragen. Das geballte Wissen des Teams und der parallelisierte Arbeitsablauf des Spiels sorgen dabei für eine hohe Effizienz. Wie nun der Frühjahrsputz mit dem Team Estimation Game genau abläuft, stelle ich in den folgenden Abschnitten vor.

Vorbereitung

Der komplette Product Backlog wird ausgedruckt: Bewährt hat sich der Ausdruck jeder Story auf ein DIN-A4-Blatt (solltest du mit Jira arbeiten, hat es sich bewährt, einen Excel-Export des Product Backlog durchzuführen und mit Hilfe eines Word-Serienbriefs die Daten als formatierte Story Cards aufzubereiten). Folgende Felder sollten mindestens ersichtlich sein:

- Titel
- Issue-Typ
- Beschreibung
- Akzeptanzkriterien
- Weitere, je nach Bedarf

Es sollten weitere leere Felder ausgewiesen sein:

- Größe
- Hinfällig/Duplikat
- Refinement – Bedarf

Das Team ist mit Product Owner komplett vertreten, der Scrum Master hat eine Skala nach der Fibonacci-Reihe für die Größe und eine Skala für die Priorität (z. B. niedrig, mittel, hoch) vorbereitet.

Ablauf

In der ersten Runde des Spiels geht es darum, die Stories zu verteilen und abzulegen.

- Der Product Owner gibt die User Stories möglichst gleichmäßig ans Team aus.
- Die Estimation wird in stiller Arbeit durchgespielt.
- Die Teammitglieder lesen ihre User Stories, und jedes Teammitglied legt sie auf der Skala zu der Zahl, die seiner Meinung nach die Größe der User Story darstellt.
- Versteht ein Teammitglied die Story nicht, so kommt diese auf den Stapel „Fragezeichen“.
- Stellt das Teammitglied fest, dass die Story bereits hinfällig ist oder ein Duplikat darstellt, so wird diese auf den Stapel „Hinfällig/Duplikate“ gelegt.
- Haben alle Teammitglieder ihre Stories abgelegt, beginnt die zweite Runde.

In der zweiten Spielrunde werden die abgelegten Stories sortiert und entsprechend verschoben.

- Die Teammitglieder sehen sich die abgelegten Stories der anderen Teammitglieder an und haben die Möglichkeit, Stories zu verschieben.
- Während des Umsortierens einer User Story gibt das Teammitglied eine kurze (!), diskussionslose Erklärung ab und versieht diese Story mit einem Punkt.
- Das Spiel wird vom Teamleiter beendet, wenn keine User Stories mehr umsortiert werden oder nichts Neues dabei herauskommt.

Nach dem Spiel bespricht das Team alle Stories mit mehr als zwei Punkten in der Gruppe im Detail und ordnet das Ergebnis gemeinsam ein. Danach kommt der Stapel mit den Fragezeichen an die Reihe. Entweder lässt sich ein gemeinsames Verständnis während des Termins noch ad hoc innerhalb des Teams finden oder der Product Owner kennzeichnet die Story mit „Refinement-Bedarf“ für später. Somit haben wir in einer konzentrierten Aktion sowohl die Größe der Backlog Items geschätzt als auch Items identifiziert, die hinfällig sind, die ein Duplikat darstellen oder noch weiteren Refinement-Bedarf besitzen.

Die letzte Aufgabe für den Product Owner ist es nun, alle ermittelten Werte der Stories im Tool für das Anforderungsmanagement zu aktualisieren. Alle Product Owner, mit denen ich diese Methode bisher durchspielen durfte, waren erstaunt, wie schnell sie mit dem Team zu Ergebnissen gekommen sind. Das Team Estimation Game lässt sich auch hervorragend für die Priorisierung einsetzen. Hier besteht der Teilnehmerkreis allerdings aus den Stakeholdern und dem Product Owner.

Letzter Tipp

Mit den vorherigen Tipps haben wir den Product Backlog wieder in einen strukturierten und aktuellen Zustand gebracht. Aber nur eine kontinuierliche Pflege des Product Backlog hilft, diesen Zustand auch zu erhalten. Bewährt haben sich regelmäßige Product-Backlog-Refinement-Termine mit dem gesamten Team bzw. in Kleingruppen. Bei diesen Terminen, die einen zeitlichen Aufwand von fünf bis zehn Prozent der Sprint-Kapazität einnehmen dürfen, werden laufend Items geschnitten, Akzeptanzkriterien verfeinert, neue Items geschätzt und bestehende Items erneut geschätzt.

Daniel Ruckriegel ist als Agile Coach und agiler Requirements Engineer bei der PENTASYS AG tätig. Als agiler Enthusiast unterstützt er Kunden und Kollegen dabei, agile Methoden für den Projekterfolg einzusetzen. Dabei greift er auf seine Erfahrungen in agilen Projekten in den Bereichen Financial Services, Automotive und E-Commerce zurück.

Warum Domain-driven Design?

Fachlich sinnvoll schneiden …

Domain-driven Design (DDD) ist eine alte Technik, aber gerade voll im Trend. Worum geht es bei DDD und ist der Hype berechtigt?

von Eberhard Wolff

Domain-driven Design (DDD) geht auf das gleichnamige Buch von Eric Evans zurück [1], das 2004 erschienen ist. Also ist DDD fünfzehn Jahre alt. Die IT-Industrie behauptet von sich, sehr schnelllebig zu sein. Dann sollte ein Buch dieses Alters keine Rolle mehr spielen. Aber das Gegenteil ist der Fall: Das Buch verkauft sich immer noch sehr gut. Das Thema DDD ist im Moment sogar so präsent wie schon lange nicht mehr.

Für diese erstaunliche Entwicklung gibt es verschiedene Gründe. Das Besondere an DDD drückt schon der Begriff aus: Die Domäne soll das Design treiben. Anders gesagt: Architektur und Implementierung orientieren sich konsequent an der Fachlichkeit. Da die meiste Software speziell für die Unterstützung fachlicher Prozesse implementiert wird, ist die Ausrichtung an der Fachlichkeit eigentlich eine offensichtliche Möglichkeit, um fachliche Prozesse noch besser zu unterstützen.

Kern von DDD ist die Ubiquitous Language (allgegenwärtige Sprache). Diese Sprache besteht aus allen Begriffen, die Domänenexperten benutzen, wenn sie über die Domäne sprechen. Tatsächlich zeigt die Erfahrung, dass Projekte ihre ganz eigene Sprache entwickeln. Die Begriffe aus der Sprache sollen dann auch im Code und in der Datenbank genutzt werden, um Felder, Klassen, Spalten oder Tabellen zu benennen. Dadurch wird es einfacher, die Fachlichkeit in der Software umzusetzen: Die fachlichen Begriffe müssen nicht noch in andere Begriffe übersetzt werden, die bei der technischen Umsetzung genutzt werden. Damit verschwimmen die Grenzen zwischen den Modellen, die für Analyse, Implementierung und Architektur genutzt werden. Das vereinfacht Feedback über diese Modelle gerade von Domänenexperten.

DDD gibt sowohl Architekten als auch Entwicklern konkrete Techniken an die Hand, um die Fachlichkeit geschickt umzusetzen. Das ist eine wichtige Besonderheit von Domain-driven Design: Es betrachtet völlig verschiedene Ebenen wie Architektur und Code. Das strategische Design teilt ein System in grobgranulare Bounded Contexts auf. Der Begriff Bounded Context sagt schon, dass es um das Begrenzen geht. Ein Bounded Context ist ein begrenzter Bereich, in dem eine Ubiquitous Language definiert ist. In einer Bibliothek ist der Begriff „Buch“ für den Bounded Context „Suche“ ein Datensatz, wie man ihn früher auf eine Karteikarte geschrieben hat. Zu dem Buch gehören der Autor, der Titel oder die Schlagwörter. Für die Leihe ist das „Buch“ hingegen ein konkretes Exemplar, das ausgeliehen werden kann. Relevant ist dabei beispielsweise, ob das Buch so gut erhalten ist, dass es noch ausgeliehen werden kann. Der Begriff „Buch“ ist dabei sehr unterschiedlich definiert: Für die Suche hat ein Buch beispielsweise mehrere ISBNs, da die Druckausgabe und die E-Books unterschiedliche ISBNs haben, aber unter einem Suchergebnis zusammengefasst werden. Für die Leihe haben mehrere „Bücher“ dieselbe ISBN. Das Buch in der Leihe ist also keineswegs identisch mit dem Buch in der Suche.

Indem die Definition auf einen Bounded Context eingeschränkt wird, kann ein Begriff wie „Buch“, der je nach Kontext oder Ansprechpartner unterschiedliche Bedeutungen hat, eindeutig definiert werden. Das erleichtert die Analyse. Gleichzeitig helfen Bounded Contexts dabei, das Softwaresystem zu strukturieren. Jeder Bounded Context hat ein eigenes Domänenmodell. Also gibt es ein Domänenmodell für die Suche, das neben relevanten Informationen über die Bücher auch die Logik für die Suche und beispielsweise die Suchpräferenzen der Benutzer umfasst. Für die Leihe gibt es ebenso ein Domänenmodell, das Bücher im Sinne der Leihe, Ausleihen, Informationen über die Benutzer und die Logik für die Leihe enthält.

Neben dieser Aufteilung in Bounded Contexts umfasst DDD auch Techniken dafür, wie diese Domänenmodelle umgesetzt werden können (**Abb. 1**). Dieser Bereich des taktischen Designs enthält Regeln, um ein gutes objektorientiertes Design zu erstellen. DDD ist

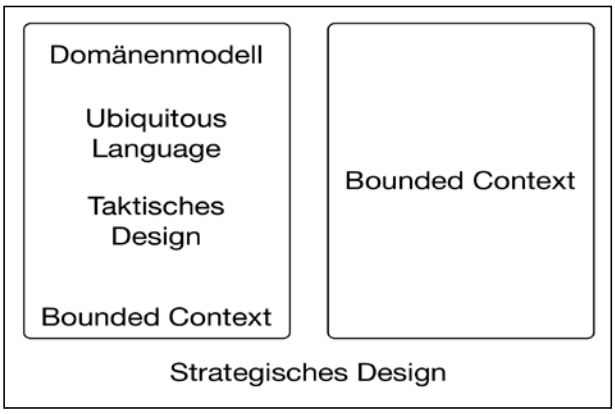

Abb. 1: Strategisches und taktisches Design

also eine vollständige Architekturmethode, die Techniken für alle Granularitätsebenen anbietet. Die Bereiche des strategischen und taktischen Designs sind nahezu unabhängig voneinander: Man muss Bounded Contexts nicht objektorientiert umsetzen, wie es das taktische Design vorschlägt, sondern dazu können auch funktionale Ansätze genutzt werden. Auch und gerade die funktionale Community unterstützt daher DDD, das ursprünglich aber als objektorientiertes Konzept gestartet ist.

Mit DDD zu guter Architektur und Implementierung

Es ist relativ klar, was eine gute Architektur auszeichnet: Lose Kopplung zwischen Modulen, hohe Kohäsion in einem Modul, Änderungen wirken sich auf möglichst wenige Module aus – die Liste lässt sich fortsetzen. Weniger klar ist, wie man zu einer solchen Architektur kommt.

Eine Möglichkeit ist, eine generische Architektur zu entwerfen, die möglichst stark vom eigentlichen System abstrahiert. Da die Architektur generisch ist, sollte sie gegenüber Änderungen relativ stabil sein. DDD predigt das Gegenteil: Die Lösung soll sich möglichst stark an der konkreten Fachlichkeit orientieren. Es scheint so, als wäre eine DDD-Architektur daher schlecht auf zukünftige Änderungen vorbereitet, weil sie nicht abstrahiert.

Das wesentliche Problem mit zukünftigen Änderungen ist aber, dass sie schwer vorherzusagen sind. Ein generischer Ansatz kann nur dann funktionieren, wenn er gemeinsame Abstraktionen über verschiedene Änderungsszenarien umsetzt. Dazu müssen die Änderungsszenarien bekannt sein und gute Abstraktionen entwickelt werden. Das ist aber unmöglich, wenn die Änderungen nicht vorab bekannt sind – und das ist bei den meisten nicht der Fall. Daher sind Architekturen dann oft an Stellen generisch, wo das gar nicht nötig ist. Das macht die Architektur unnötig komplex. Die Generizität fehlt dann an den Stellen, wo sie dann doch benötigt wird. Dieses Problem ist kaum lösbar, weil vorab eben nicht klar ist, wo später Änderungen notwendig sein werden.

Eine konsequente Ausrichtung an der Fachlichkeit wie bei DDD kann durchaus gegen fachliche Änderungen stabil sein. Leihe und Suche werden immer ganz wesentliche Bestandteile einer Bibliothek sein. Es ist unwahrscheinlich, dass diese Aufteilung später zu einem Nachteil wird. Wenn es eine Änderung gibt, kann es sehr gut sein, dass sie auf einen dieser Bounded Contexts begrenzt ist. Wie Suche oder Leihe funktionieren sollen, wird sich eher ändern als das Zusammenspiel. So könnte es neue Suchmöglichkeiten mit neuen Attributen oder Kategorien geben. Und wenn die Fachlichkeit der Suche gut umgesetzt worden ist, sind diese Änderungen wahrscheinlich nicht schwer umzusetzen. Also ist nicht nur die Aufteilung in Bounded Contexts aus dem Strategic Design wichtig für die Änderbarkeit, sondern auch die Implementierung in den Bounded Contexts, die das taktische Design regelt.

Diese fachliche Aufteilung von DDD kann Änderungen recht einfach umsetzbar machen. So sollte eine DDD-Architektur zu einer änderbaren Architektur führen, die beispielsweise auch lose gekoppelt ist. Vor allem beschreibt DDD einen konkreten Ansatz, mit dem Teams solche Architekturen konstruieren können. Das ist wesentlich nützlicher als Techniken, mit denen man später eine Architektur nur bewerten kann. Lose Kopplung ist zwar ein Ziel, aber es ist kaum direkt erreichbar, sondern erst Konstruktionsansätze wie DDD führen zu solchen Zielen.

Vorgehensmodell

Domain-driven Design stellt also einen Architektur- und Implementierungsansatz dar. Eigentlich sollte das Erstellen der Architektur unabhängig vom Vorgehensmodell in dem Projekt sein. Die Abbildung der Fachlichkeit kann aber nur dann erfolgen, wenn es Zugang zu den Fachexperten gibt. Schließlich sind sie es, die genau wissen, wie die Domäne funktioniert. Genau dieses Wissen soll in der Software abgebildet werden. Ein Lastenheft kann dieses Wissen nur unzureichend transportieren.

DDD beschreibt daher einen Prozess des „Knowledge Crunching“, bei dem Entwickler und Domänenexperten gemeinsam an einem Modell arbeiten. Das löst das Problem der unvollständigen Anforderungen, die im Dialog geklärt werden. Aber auch ein solches agiles Vorgehen garantiert noch nicht, dass ein Modell erfolgreich erstellt wird. Dazu müssen die Entwickler nicht einfach Feature um Feature heruntercoden, sondern Wissen in ein gutes Modell verpacken und dieses Modell schrittweise weiterentwickeln. Die Domänenexperten können ein solches Modell nicht allein erstellen. Sie haben zwar das Wissen über die Domäne, aber ihnen fehlen die Modellierungstechniken. Außerdem muss ein Modell für die Implementierung in Software eindeutig sein. Ein Artefakt wie eine Klasse hat nur einen Namen und eine Beschreibung im Code. Leider sind Begriffe in der Sprache oft bei weitem nicht so eindeutig. Durch diesen Zwang zur Eindeutigkeit müssen Domänenexperten zusammen mit dem Softwareteam das Modell so lange weiterentwickeln, bis es

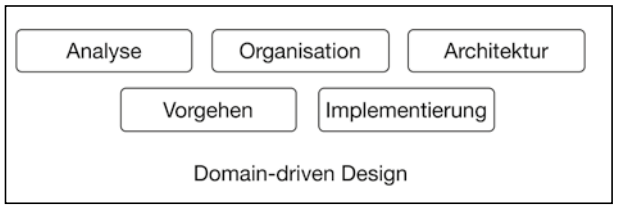

Abb. 2: DDD umfasst viele unterschiedliche Bereiche

eindeutig ist. Dazu müssen Begriffe aus der Ubiquitous Language geklärt werden. So führt die Erstellung der Software zu einem Lernprozess über das Wissen über die Domäne. Das ist natürlich ausgeschlossen, wenn man die fachlichen Anforderungen nur einmal aufschreibt und dann als fest betrachtet. Dann kann das Modell nicht weiterentwickelt werden und bleibt verbesserungswürdig.

Architektur und Organisation

So überschreitet Domain-driven Design die Grenze zwischen Architekturansatz und Vorgehensmodell. Aber auch an einer anderen Stelle geht DDD über klassische Architektur hinaus: Bounded Contexts haben Abhängigkeiten. Wenn in der Bibliothek bei der Suche auch angezeigt werden soll, ob die Bücher aus dem Suchergebnis ausgeliehen werden können, dann muss der Bounded Context „Suche“ die Information, ob ein Buch ausgeliehen werden kann, von dem Bounded Context „Leihe“ erhalten. Eine klassische Architektur würde diese Beziehung als Abhängigkeit zwischen Modulen auffassen. Domain-driven Design stellt aber die Beziehungen zwischen den Teams, die für die jeweiligen Bounded Contexts verantwortlich sind, in den Vordergrund: Gibt das Team „Suche“ die Gestaltung der Schnittstelle vor oder ist es eher das Team „Leihe“? Damit verschwimmen die Grenzen zwischen einem Bounded Context als Gültigkeitsbereich einer Ubiquitous Language und einem Domänenmodell auf der einen Seite und dem Zuständigkeitsbereich eines Teams auf der anderen Seite.

DDD steht in der Tradition des Gesetzes von Conway [2] von 1968. Dieses Gesetz stellt fest, dass die Architektur eines Systems mit den Kommunikationsstrukturen der Organisation übereinstimmt, die das System entwirft. DDD definiert Maßnahmen auf der Organisationsebene, mit denen die Architektur beeinflusst werden soll. So nutzt DDD das Gesetz von Conway aus, um die Architektur mit Hilfe der Organisation zu beeinflussen. Dieser Ansatz, Architektur und Organisation gemeinsam zu betrachten, ist im Rahmen des Microservices-Hypes wieder aktuell geworden. Der Ansatz ist notwendig, weil das Erstellen von Software ein Prozess ist, den der Komplexität wegen meist nur ein Team bewältigen kann. Hinzu kommt, dass Teammitglieder mit Domänenexperten zusammenarbeiten müssen. Diese Kollaboration ist die wichtigste Herausforderung. Ein erfolgreiches Projekt und eine erfolgreiche Architektur können nur entstehen, wenn die Kollaboration erfolgreich ist.

DDD umfasst also neben Architektur auch zahlreiche andere Bereiche (**Abb. 2**).

DDD erfolgreich einsetzen

Aufgrund der vielen Techniken, die DDD umfasst, ist natürlich die Frage, wie man DDD erfolgreich einsetzt. Muss ein Team alle Techniken nutzen? Ist das überhaupt machbar?

Für die Beantwortung der Frage ist eine historische Betrachtung interessant: Kurz nach dem Erscheinen des ursprünglichen Buchs hat sich die Diskussion im Wesentlichen auf taktisches Design und die Umsetzung von objektorientiertem Design mit DDD fokussiert. Damals hat dieser Teil wesentliche Probleme in einigen Projekten gelöst. Vielfach war nicht klar, wie ein gutes fachliches, objektorientiertes Modell aussehen soll. Das strategische Design hingegen hat kaum eine Rolle gespielt. Vielleicht ist auch ein Grund dafür, dass taktisches Design am Anfang des Buchs diskutiert wird, sodass es besonders wichtig erschien und jeder vermutlich zumindest diesen Teil des Buchs gelesen hat. Mittlerweile spielt das strategische Design die entscheidende Rolle, denn es beantwortet die Fragen nach einem grobgranularen fachlichen Modell und ist für die Aufteilung eines Systems in Microservices sehr wichtig.

Vermutlich stehen also je nach Problemstellung unterschiedliche Teile von Domain-driven Design im Fokus. Allerdings bedeutet das auch, dass die Teile von DDD, die gerade nicht so relevant erscheinen, eben auch nicht beachtet werden. Das ist schade. Beispielsweise hätte das strategische Design schon viel früher genutzt werden können, um übermäßig komplexe Domänenmodelle zu vermeiden, die es oft schon lange gibt. Also sollte man sich mit DDD als Ganzem auseinandersetzen, um tatsächlich den vollen Nutzen aus dem Ansatz zu ziehen.

Dazu stellt sich Frage, ob die schon diskutierten Bereiche des taktischen und strategischen Designs alles sind, was DDD zu bieten hat. Diese Frage vermeidet den Fehler, sich nur wieder auf einige Aspekte von DDD zu konzentrieren und andere zu vernachlässigen, die

gegebenenfalls auch Lösungen für Probleme sein können. Dazu lohnt ein Blick in die „Domain-driven Design Referenz“. Sie kommt ebenfalls von Eric Evans, ist aus dem Jahr 2015 und daher aktueller als das ursprüngliche Buch. Außerdem ist sie deutlich kompakter, sodass man sie recht schnell durcharbeiten kann. Teil I der Referenz enthält Patterns für das grundlegende Vorgehen, also Ideen zum Knowledge Crunching. Teil II beschreibt die Patterns für das taktische Design. Die Teile IV und V erläutern strategisches Design. Teil III beschreibt grundlegende Praktiken für einen objektorientierten Entwurf und Teil IV einige Tipps für den Entwurf des Gesamtsystems. Es lohnt sich also, die Referenz durchzulesen und vielleicht gerade die Bereiche wie Teil III und IV zu lesen, die nicht zu den traditionellen Kategorien strategisches und taktisches Design gehören.

Am Ende ist DDD nur ein weiteres Werkzeug, um bessere Software zu schreiben. Es lohnt sich, DDD genauer zu betrachten, um so das Werkzeug möglichst genau kennenzulernen. Das ist vor allem wichtig, weil Softwareentwicklung oft nicht effektiv ist, da Konzepte nur halb verstanden oder umgesetzt werden. DDD ist jedoch keine Religion, wie Eric Evans selbst schon gesagt hat. Es ist also völlig in Ordnung, nur Teile von DDD zu nutzen oder die Konzepte weiterzuentwickeln. Meiner Ansicht nach ist es dafür aber notwendig, die Konzepte zu verstehen. Denn nur wenn man die Konzepte wirklich versteht, kann man sich mit ihnen fundiert auseinandersetzen und sie weiterentwickeln.

Neben den ursprünglichen Techniken aus dem Buch gibt es weitere Ansätze, die ebenfalls zu DDD gezählt werden. So beispielsweise Event Storming [4], bei dem ein Team gemeinsam eine Domäne mit Hilfe der auftretenden Events analysiert. Auch so zeigt sich, dass DDD kein festes Verfahren ist, sondern sich weiterentwickelt, auch wenn das ursprüngliche Buch sich gegenüber der Fassung von 2004 kaum verändert hat.

Ebenso ändert sich die Implementierung von DDD: Während das ursprüngliche Buch objektorientierte Ansätze gezeigt hat, sind funktionale oder „reactive“ Ansätze mittlerweile auch durchaus üblich. Auch das zeigt, dass DDD sich über die Zeit wandelt und so relevant bleibt, selbst wenn sich die technologische Basis ändert.

Fazit

Die meiste Software soll fachliche Prozesse unterstützen. Domain-driven Design unterstützt den Entwurf solcher Software durch eine konsequente Orientierung an der Fachlichkeit. Damit ist es eine sehr wichtige Technik, um gute Softwareentwürfe nicht nur auf der Ebene der Architektur, sondern auch des Designs auf Klassenebene zu erstellen. Dazu kommen Regeln für Organisation und Vorgehen. So stellt DDD eine umfangreiche Lösung für die Kernbereiche der Softwareentwicklung dar. Der Umfang ist aber auch gleichzeitig ein Problem: Es ist alles andere als einfach, DDD vollständig umzusetzen. Das ist aber oft auch gar nicht notwendig. Dennoch sollte man sich mit DDD intensiv auseinandersetzen, um die Konzepte tatsächlich zu verstehen und sie auch wirklich effektiv zu nutzen.

Die Zeit für das Erlernen von DDD ist auf jeden Fall gut investiert: DDD ist eine der wenigen Techniken, die klare Prinzipien aufstellt, um zu guten Entwürfen zu kommen. Und gute Entwürfe sind sehr wichtig für den Erfolg eines Projekts.

Eberhard Wolff ist Fellow bei INNOQ und arbeitet seit mehr als fünfzehn Jahren als Architekt und Berater. Er ist Autor zahlreicher Artikel und Bücher und trägt als Sprecher auf internationalen Konferenzen vor. Sein technologischer Schwerpunkt sind moderne Architektur- und Entwicklungsansätze wie Continuous Delivery, DevOps und Microservices.

Links & Literatur

[1] Evans, Eric: „Domain-Driven Design. Tackling Complexity in the Heart of Software“, Addison-Wesley, 2004.

[2] http://www.melconway.com/Home/Conways_Law.html

[3] Evans, Eric: „Domain-Driven Design Referenz“, kostenlos unter: https://ddd-referenz.de/

[4] https://www.eventstorming.com

Spring Boot 2.3

Der zehnte Geburtstag

Mit Spring Boot 2.3 erschien im Mai 2020 die nunmehr zehnte große Spring-Boot-Version, rund sechs Jahre nach Version 1.0. Ich habe das Projekt in diesem Zeitraum sowohl als Bug-Reporter als auch als Contributor, Benutzer und Autor begleitet.

von Michael Simons

In dieser Zeit hat mich eine Sache immer wieder begeistert: Das Spring-Boot-Team hatte immer ein offenes Ohr für Trends und Bedürfnisse der Benutzerinnen. Oftmals wurde abgewartet und abgewogen, bis eine optimierte Lösung angeboten werden konnte. Für Spring Boot 2.3 ist das in meinen Augen eine erhebliche Verbesserung im Build-Prozess. War das von Spring Boot 2014 favorisierte Fat-Jar-Modell damals im Vergleich zu hochkomplexen WAR und EAR Deployments ein erstrebenswertes Ziel, so hat sich die Welt inzwischen weitergedreht. Spring Boot 2.3 trägt dieser Tatsache Rechnung und startet mit Buildpack-Support und Layered Jars in eine neue Ära.

Bereits in meinem Leitartikel zu Spring Boot 2.2 auf JAXenter im vergangenen Jahr [1] habe ich die neuen Herausforderer im Microservices-Umfeld erwähnt: Micronaut und Quarkus. Diese Frameworks und die Art und Weise, wie mit ihnen extrem schnell startende Services gebaut, nativ kompiliert und komfortabel in Docker Images paketiert werden können, haben Spring Boot 2.3 und das Spring Framework im vergangenen Jahr maßgeblich beeinflusst. Auf die native Kompilierung von Spring-basierten Anwendungen werde ich am Ende des Artikels eingehen. Quarkus hat seit Version 1 ein Dockerfile – sowohl für JVM als auch für Native Images – mitgeliefert. Das Ziel-Deployement war schnell erkennbar. Spring Boot zieht nun nach.

Optimierte Docker Images

Prominentestes 2.3-Feature ist wohl eine Erweiterung der Spring-Boot-Maven- und Gradle-Plug-ins: Beide können jetzt auf Basis von Cloud Native Buildpacks [2] Docker Images erzeugen. Buildpacks sind ein Konzept,

das ursprünglich von Heroku [3] stammt. Heroku ist eine „Cloud Plattform as a Service“, die seit jeher unterschiedlichste Programmiersprachen und Plattformen unterstützt und großen Bedarf hat, Anwendungen mit optimierten Images möglichst schnell und optimal zu starten. Buildpacks sollen die operative Last von Entwicklern nehmen und gleichzeitig Compliance- und Securityrichtlinien sicherstellen.

Für Spring Boot 2.3 können Buildpacks – eine Docker Installation vorausgesetzt – mit *./mvnw spring-boot:build-image* genutzt werden. Es entsteht ein auf Ubuntu 18 LTS basierendes Image inklusive korrekter Java-Heap-Settings (für einen Betrieb im Container), dem JVMKill Agent (der die VM nach einem *OutOfMemoryError* beendet) und einiger Dinge mehr. Der Agent läuft außerhalb der JVM und verhindert, dass die JVM in einem undefinierten Zustand nach *OutOfMemoryError* weiterläuft.

Kodifizierung von Best Practices

Es gibt gute Gründe, keine Buildpacks zu benutzen (zum Beispiel aufgrund des bestehenden Toolings oder dem Wunsch, eigene Base Images zu nutzen). Dabei existiert mit dem Default-Deployment von Spring-Boot-Anwendungen, dem „Fat Jar“-Model, jedoch ein Problem: Die gesamte Anwendung, alle Libraries und Ressourcen liegen als ein Artefakt vor. Was in einem Szenario von Vorteil ist, ist ein riesiger Nachteil, wenn dieses Artefakt in einem Container-Image weiterverteilt und nicht per *java -jar* auf einem beliebigen Server gestartet wird.

Spring Boots Fat Jars sind eine komfortable Art, komplette Anwendungen inklusive aller Abhängigkeiten und der Runtime zu paketieren, aber gerade in Szenarien, in denen Services wegen kleiner Änderungen neu verteilt werden, nicht optimal: Es muss jedes Mal ein großes Jar mitsamt allen nicht geänderten Abhängigkeiten neu verteilt werden und dementsprechend ein neuer, großer Layer im Container-Build.

So wie Spring Boot seit über fünf Jahren versucht, Spring Best Practices mit Startern und automatischer Konfiguration zu kodifizieren, so geht es diesen Weg nun auch bezüglich Docker Images, um das oben beschriebene Problem der Fat Jars zu lösen. Spring Boot nutzt dazu ein neues Fat-Jar-Layout sowie einen eigenen Boot Loader, dessen Optionen ich bereits 2018 im Spring-Boot-Buch [4] beschrieben habe. Dieser Mechanismus wird nun um Layered Jars ergänzt. Die Defaultlayer sehen aus wie in Listing 1 gezeigt:

Listing 1: Default Layer

```
- "dependencies":
  - "BOOT-INF/lib/"
- "spring-boot-loader":
  - "org/"
- "snapshot-dependencies":
- "application":
  - "BOOT-INF/classes/"
  - "BOOT-INF/classpath.idx"
  - "BOOT-INF/layers.idx"
  - "META-INF/"
```

- *dependencies* für alle Nicht-Snapshot-Dependencies
- *snapshot-dependencies* für Snapshot-Dependencies
- *resources* für statische Ressourcen wie *META-INF/resources/, resources/, static/, public/*.
- *application* für die eigentliche Anwendung

Die Reihenfolge hier ist wichtig: Dependencies ohne Snapshot werden selten geändert. Sie liegen unten, darüber Dependencies, die sich öfter ändern können, dann die Ressourcen und schlussendlich die Anwendung.

Ein Layered Jar wird über die Konfiguration des Maven- bzw. Gradle-Plug-ins erstellt. Listing 2 zeigt dies für Maven, das nachfolgende Beispiel (Layered Jars mit dem Spring-Boot-Gradle-Plug-in) für Gradle.

Listing 2: Layered Jars mit dem Spring-Boot-Maven-Plug-in

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <version>2.3.0.RC1</version>
        <configuration>
          <layers>
            <enabled>true</enabled>
          </layers>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
bootJar {
  layered()
}
```

Die Layer einer so paketierten Anwendung können sogar mit einem eingebauten Befehl angezeigt werden:

```
java -Djarmode=layertools -jar target/demo-0.0.1-SNAPSHOT.jar list
dependencies
spring-boot-loader
application
```

Mit *extract* statt *list* können die Layer entpackt werden. Im Vergleich zum generischen Google-Containertool jib [5] kennt das Jar-File seine eigenen Layer und ermöglicht dieses Tooling. Darauf aufbauend kann relativ schnell ein Docker Image mit korrespondierenden Image-Layern erzeugt werden. Der Layer der Abhängigkeiten ändert sich dann entsprechend selten, Deployments sind schneller, da nur noch ein kleiner Teil des

Image neu erzeugt werden muss, und die neuen Images sind entsprechend kleiner.

Welche Java-Version?

Die gleiche Frage wie letztes Jahr: Welche Java-Version wird unterstützt? Während andere Frameworks es sich recht leicht erlauben können, den Support für Java 8 abzuschalten (Quarkus wird dies mit 1.6 tun und fortan Java 11 erfordern), ist das für Spring Boot und insbesondere das zugrunde liegende Spring Framework schwierig, ist doch die Liste von Nutzern groß, die nicht ohne Weiteres auf Java 11+ wechseln können. Spring Boot 2.3 setzt wie gehabt auf Spring Framework 5.2, das bis Ende 2021 unterstützt werden wird, und ist aktuell Java-8- bis Java-15-kompatibel. Dementsprechend kann Spring Boot 2.3 bereits mit Java 14 genutzt werden. Mit den neuesten Spring-Data-Modulen – allen voran Spring Data Neo4j⚡RX [6] – können sogar JDK 14 Records (Preview-Feature von unveränderlichen Data Classes) genutzt werden, um Datenbankzugriff zu realisieren.

Erwähnenswerte Features und Änderungen

R2DBC-Support: Wenn R2DBC (Reactive Relational Database Connectivity) auf dem Klassenpfad ist, wird automatisch eine Connection Factory – ähnlich wie eine JDBC DataSource – konfiguriert. Gleiches gilt für Spring Data R2DBC.

Verbesserungen des Health Actuator, bessere Health-Indikatoren für Datenbanken: Für SQL-Datenbanken wird nun *java.sql.Connection#isValid* anstelle einer Abfrage genutzt – es sei denn, eine solche ist definiert. Ähnliches wird für Neo4j getan. Der Neo4j Health Endpoint zeigt nun zusätzlich die Version des Servers und die Edition an.

Lebendig und bereit? Spring Boot kann nun unterscheiden, ob eine Anwendung lediglich „alive“ oder auch in der Lage ist, Anfragen zu beantworten: Das Setzen von *management.health.probes.enabled* auf *true* konfiguriert den Health Endpoint so, dass er unter */actuator/health/liveness* Auskunft über generelle Verfügbarkeit gibt und unter */actuator/health/readiness* darüber, ob Anfragen beantwortet werden können.

Web Starter ohne Validation: Beide Web Starter – imperativ und reaktiv – kommen nun ohne die Abhängigkeit zum Validation Starter aus. Dies reduziert die Größe von einfachen Webanwendungen. Wird Java Bean Validation benötigt, muss nun *spring-boot-starter-validation* als separate Abhängigkeit aufgeführt werden.

Graceful Shutdown: Für alle unterstützten Webserver sowie für die reaktiven Konnektoren wird nun ein Graceful Shutdown unterstützt. Durch *server.shutdown.grace-period* kann ein Zeitraum definiert werden, in dem die Anwendung keine neuen Anfragen mehr entgegennimmt und nur noch offene Anfragen abarbeitet.

Der Spring Cloud Connectors Starter wurde entfernt: Nach einer Deprecation in Spring Boot 2.2 wurde nun der Spring Cloud Connectors Starter entfernt. Er wird langfristig durch *java-cfenv* ersetzt, einer neuen Library zum Zugriff auf Cloud Foundry Services.

DynamicPropertySource: Die Annotation *DynasmicPropertySource* wurde zwar bereits mit Spring Boot 2.2.6 eingeführt, ich halte sie dennoch für erwähnenswert. Sie vereinfacht die Arbeit mit Testdatenbanken und anderen dynamischen Quellen für Konfiguration ungemein:

```
@DynamicPropertySource
static void neo4jProperties(DynamicPropertyRegistry registry) {
  registry.add("org.neo4j.driver.uri", embeddedDatabaseServer::boltURI);
  registry.add("org.neo4j.driver.authentication.password", () -> "");
}
```

Mehr ist nicht mehr nötig, um die Umgebung eines *@SpringBootTest* aus einer statischen Init-Methode zu beeinflussen.

Bean Overriding während Tests: Im Applikationskontext werden Beans mit ihrem Namen identifiziert. Wird eine zweite Bean gleichen Namens definiert, überschreibt sie die erste. Das führt in vielen Fällen zu Problemen und unerwartetem Verhalten und wurde daher bereits in Spring Boot 2.1 für den Kontext der *SpringApplication* abgeschaltet. 2.3 tut dies nun auch für den *ApplicationContextRunner*, der während

Spring-Boot-Tests genutzt wird. Um das vorherige Verhalten in Tests wiederherzustellen, kann die Builder-Methode *withAllowBeanDefinitionOverriding* genutzt werden.

Open Session In View (OSIV): Das Open-Session-In-View-Pattern (OSIV) wird von vielen Experten der Datenbankentwicklung als kritisch angesehen. Spring Boot loggt seit geraumer Zeit eine Warnung, wenn es für JPA/Hibernate aktiviert ist, für Neo4j-OGM wurde es nun in Absprache mit dem Neo4j-OGM-Team per Default deaktiviert.

Konfiguration

Die Eigenschaften, die Spring Boot über Properties, Environment-Variablen und andere Mechanismen zur Konfiguration zur Verfügung stellt, hängen natürlich direkt von den unterstützten Libraries ab und sind daher sehr umfangreich. Einige davon wurden von Spring Boot 2.2 zu Spring Boot 2.3 deprecated und einige ersetzt. Das Spring Boot 2.3.0 RC1 Configuration Changelog [7] bietet eine gute Übersicht. Erwähnenswert ist hier keine spezielle Eigenschaft, sondern ein neuer Wert *iso*, der für verschiedene Datums- und Datum-Zeit-Angaben verwendet werden kann und das entsprechende ISO-8601-Format des jeweiligen Typens konfiguriert. Verwendet werden kann *iso* unter anderem mit:

- *spring.mvc.format.date*
- *spring.mvc.format.date-time*
- *spring.mvc.format.time*
- *spring.webflux.format.date*
- *spring.webflux.format.date-time*
- *spring.webflux.format.time*

Libraries

Auch mit Spring Boot 2.3 ist die Liste der verwalteten Abhängigkeiten und die dazugehörige Auto Configuration nicht kleiner geworden. Es ist nachvollziehbar, dass viele Entwickler bei einem ersten Blick auf Spring Boot denken: „Das ist aber komplex, die Fülle an Optionen ist einschüchternd.“ Man sollte aber nicht vergessen, dass Spring Boot enorm erfolgreich ist, weil es viele Abhängigkeiten im Kontext des Spring-Ökosystems konfiguriert und umkehrt viele Hersteller möchten, dass ihr Produkt möglichst gut in der Plattform funktioniert und sie ihre Libraries integriert wissen wollen. Oft genutzte und aktualisierte Abhängigkeiten sind unter anderem:

- Couchbase Client
- Elasticsearch
- Flyway
- JUnit
- Kotlin
- Lettuce
- MongoDB
- Micrometer
- Neo4j-OGM

Die Liste aktualisierter Spring-eigener Module ist ähnlich groß:

- Spring Data Neumann
- Spring Framework
- Spring HATEOAS
- Spring Integration
- Spring Kafka
- Spring Security
- Spring Session

In den meisten Fällen hängt Boot von diesen Projekten hinsichtlich Auto Configuration und Ähnlichem ab, teilweise bestehen aber auch umgekehrte Abhängigkeiten, meist über das Spring Framework selbst.

Erwähnenswert ist an dieser Stelle die Aktualisierung vieler Datenbankclients. In den Treibern für Neo4j, MongoDB und Cassandra gibt es viele Änderungen, die teilweise Breaking Changes sind. Eine Abstraktion in Form von Spring Data schützt die meisten Projekte massiv vor diesen Auswirkungen. Der Nutzen von Spring Data geht weit über die reine Repository- beziehungsweise Mapping-Abstraktion hinaus.

Übersicht der verfügbaren Changelogs

Grundlage für den Artikel sind – abgesehen von eigener Arbeit – die öffentlich verfügbaren Changelogs. Ich kann nur empfehlen, die entsprechenden Wiki Pages auf GitHub zu verfolgen, um für ein Upgrade gewappnet zu sein. Ebenso möchte ich die Empfehlung des Spring-Teams wiederholen: Ein Upgrade sollte in Phasen erfolgen. Falls die Anwendung noch auf Spring Boot 2.0 basiert, sollte schrittweise nach 2.3 migriert werden. Den größten Aufwand sehe ich im ersten Schritt (von 2.1 auf 2.3).

- Milestone 1 [8]
- Milestone 2 [9]
- Milestone 3 [10]
- Milestone 4 [11]
- RC1 [12]

Ausblick: Going Native

2019 war es in der Java-Welt fast unmöglich, um das Thema native Kompilierung mit der GraalVM herumzukommen. Die eingangs erwähnten Frameworks Quarkus und Micronaut waren Vorreiter in Bezug auf dieses Thema. Der Autor selbst war damit beschäftigt, den Neo4j-Datenbanktreiber dahingehend zu ertüchtigen, dass er nativ mit der GraalVM kompilierbar ist. Mit dem experimentellen „Spring Graal Native“-Projekt [13] arbeitet das Spring-Team daran, native Kompilierung von Spring-Boot-Anwendungen zu ermöglichen. Dabei müssen unter anderem folgende Herausforderungen gelöst werden:

- Dynamische Konfiguration von Anwendungen – der extrem flexible Konfigurationsmechanismus von Spring Boot ist an dieser Stelle Fluch und Segen zugleich.

- Dynamisches Laden von Klassen und Codepfaden.
- GraalVM Substitutions für Teile von Spring und Spring Boot, die aktuell nicht nativ kompilierbar sind.

Aktuell ist noch viel Handarbeit nötig, um selbst kleine Spring-Boot-Anwendungen nativ zu kompilieren, wie Jonas Hecht in seinem Post [14] Anfang Mai eindrucksvoll beschrieben hat. Es ist aber zu erwarten, dass das Spring-Team auch diese Hürden nehmen wird.

Michael Simons ist Vater, Ehemann und Athlet (Letzteres vermutlich nur in seiner Vorstellung). Er ist Java Champion, Mitgründer und aktueller Leiter der Euregio JUG (http://euregjug.eu). Michael ist sowohl in Deutschland durch sein deutschsprachiges Spring-Boot-Buch, aber auch international sehr stark mit dem Spring-Ökosystem verbunden. Spring ist ein immer wiederkehrendes Thema auf seinem Blog. Daran wird sich auch so schnell nichts ändern, denn Michael arbeitet als Software Engineer bei Neo4j und beschäftigt sich dort mit dem Spring Data Modul für die gleichnamige Graphdatenbank Neo4j.

https://info.michael-simons.eu

Links & Literatur

[1] https://jaxenter.de/spring-boot-2-2-release-88156

[2] https://buildpacks.io

[3] https://www.heroku.com

[4] http://springbootbuch.de

[5] https://github.com/GoogleContainerTools/jib

[6] https://github.com/neo4j/sdn-rx

[7] https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.3.0-RC1-Configuration-Changelog

[8] https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.3.0-M1-Release-Notes

[9] https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.3.0-M2-Release-Notes

[10] https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.3.0-M3-Release-Notes

[11] https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.3.0-M4-Release-Notes

[12] https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.3.0-RC1-Release-Notes

[13] https://github.com/spring-projects-experimental/spring-graalvm-native

[14] https://blog.codecentric.de/en/2020/05/spring-boot-graalvm/