02-2017 | Sommer | www.ijug.eu

Java aktuell

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java im Mittelpunkt

Big Data
Predictive Analytics mit Apache Spark

Microservices
Lagom, das neue Framework

Achtung, Audit
Nutzung und Verteilung von Java SE

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

4 191978 304903 02

iJUG Verbund

Lagom: Einmal Microservices mit allem, bitte!

Lutz Hühnken, Freiberufler

Microservices sind das Architektur-Muster der Stunde und an entsprechenden Frameworks herrscht kein Mangel. Mit Lagom betritt ein weiterer Player die Arena, in der sich schon Spring Boot, Dropwizard und andere tummeln. Wodurch unterscheidet es sich von den anderen?

Man kann davon ausgehen, dass den Programmierern von Lightbend (manchen noch bekannt unter dem vormaligen Namen „Typesafe") auch bewusst war, dass es schon andere Microservice-Frameworks gibt – zumal sie selbst schon zwei Open-Source-Frameworks unter ihren Fittichen haben, nämlich Play! *(siehe „http://www.playframework.com")* und Akka HTTP *(siehe „http://github.com/akka/akka-http")*, die durchaus geeignet sind, um Java-Microservices zu entwickeln. Trotzdem hielten sie es für eine gute Idee, mit Lagom (schwedisch für „gerade richtig", „nicht zu viel, nicht zu wenig") noch ein weiteres auf den Markt zu bringen. Die wesentlichen Design-Entscheidungen dabei waren:

- Bei Microservices geht es nie um einen einzelnen Service, daher sollte auch das Framework sich nicht auf den einzelnen Service fokussieren, sondern auf das (aus Microservices bestehende) System. Kommunikation zwischen den Services und ein Entwicklungs-Setup, das es erlaubt, effektiv an mehreren Services gleichzeitig zu arbeiten, sollten zum Framework gehören.
- Lagom will den Entwicklern Entscheidungen abnehmen. Statt nur eine HTTP/REST-Library vorzugeben und es den Entwicklern zu überlassen, sich für alles Weitere aus dem Pool von Netflix-OSS und anderen zu bedienen, gibt es ein Standard-Setup für den gesamten Stack (HTTP, JSON, Datenbank, Message-Bus).
- Entwickler sollen an die Hand genommen werden, Microservices skalierbar und resilient (also ausfallsicher, fehlertolerant) zu entwickeln. Jeder einzelne Service soll als Cluster mit variabler Anzahl von Nodes laufen können.
- Die Features von Java 8 sollen konsequent genutzt werden, eine Verwendung mit älteren Java-Versionen ist nicht vorgesehen.

Java 8 ein Muss

Der letzte Punkt fällt jedem, der sich Lagom-Beispielcode ansieht, sofort ins Auge. Wer mit „Lambdas", „Optional" und „CompletionStage" noch auf Kriegsfuß steht, sollte einen Crash-Kurs belegen, denn in Lagom wimmelt es nur so davon *(siehe Listing 1)*. Hauptgrund dafür ist die asynchrone Natur des Frameworks. Lagom folgt nicht dem „Thread-per-Request"-Modell, wie es das Servlet-API ursprünglich vorgesehen hat.

In Web-Frameworks, die auf dem Servlet-API beruhen, ist dem eingehenden HTTP-Request ein Thread fest zugeordnet, der erst wieder freigegeben wird, wenn die HTTP-Response gesendet ist. Das setzt der Skalierbarkeit natürliche Grenzen – auch mit der schnellsten Hardware ist die Anzahl der nativen Threads, die das Betriebssystem parallel verwalten kann, begrenzt.

Lagom kommt mit wenigen Threads aus, da der ausführende Thread bei jeglicher Kommunikation – sei es mit der Außenwelt oder zwischen den Komponenten des Service – immer wieder freigegeben wird. Kaum etwas in Lagom ist ein einfacher Methodenaufruf – stattdessen ist der Standard-Rückgabewert eine „CompletionStage". Dieser können dann wiederum ein (oder

```
@Override
public ServiceCall<User, NotUsed> createUser() {
  return request -> {
    return friendEntityRef(request.userId)
         .ask(new CreateUser(request))
         .thenApply(ack -> NotUsed.getInstance());
  };
}
```

Listing 1: Lambdas, überall Lambdas

mehrere) Callback(s) übergeben werden, die aufgerufen werden, wenn die asynchrone Ausführung abgeschlossen ist.

Es wird schnell deutlich, dass Java ein bisschen „syntactic sugar" für diese asynchrone, Callback-basierte Programmierung fehlt, sodass der Code nicht immer ästhetisch ansprechend ist. Das darunter liegende Konzept überzeugt jedoch – um die parallele Rechenleistung moderner Multicore-Prozessoren auszunutzen, führt an der asynchronen Programmierung kein Weg vorbei.

Eine wichtige Begleiterscheinung dieser Art der Programmierung ist die Notwendigkeit, mit unveränderlichen (immutable) Objekten zu arbeiten. Das ist nicht weiter kompliziert, für viele allerdings sicher eher ungewohnt. Das typische Objekt in der Objektorientierung zeichnet sich ja durch die Eigenschaften aus, eine Identität und einen veränderlichen Zustand zu besitzen. Lagom erfordert ein Umdenken mehr in Richtung funktionaler Programmierung: Ich übergebe einer Funktion einen Wert (oder in Lagom auch häufig: sende eine Nachricht), der dann auch konstant bleibt. Es ist keine Zauberei, solche unveränderlichen Klassen in Java zu schreiben, aber es kommt doch etwas an Boilerplate-Code zusammen, denn es müssen beispielsweise „equals" und „hashCode" jedes Mal mit einer wertebasierten Variante überschrieben werden. Glücklicherweise ist die Arbeit mit solchen „value objects" oder „data objects" auch in der restlichen Java-Welt inzwischen recht gängig, und etwa die „@Value"-Annotation aus Projekt Lombok *(siehe „https://projectlombok.org")* hilft, Tipparbeit zu sparen. Was Java dann noch fehlt, sind passende Klassen für unveränderliche Collections. Hier kann zum Beispiel „pCollections" *(siehe „http://pcollections.org")* Abhilfe schaffen. Was asynchrone Programmierung und unveränderliche Daten angeht, vertritt Lagom einen klaren Standpunkt: So muss es programmiert sein. Es werden keine veränderlichen Daten geteilt, es gibt keine blockierenden Aufrufe. Das ist vielleicht für einige noch ungewohnt, aber dieser Trend wird sich sicher auch außerhalb von Lagom durchsetzen.

Mit im Paket: Event Sourcing, CQRS …

Ist das erste „Hello World" geschrieben, kommt als zweiter Schritt meist: Daten in der Datenbank speichern, eine einfach CRUD-Anwendung schreiben; also eine Entity-Klasse definieren und mit einem ORM schnell auf eine Tabelle in der relationalen Datenbank abbilden. Wer das erwartet, unterschätzt die Ambitionen der Lagom-Entwickler deutlich.

Lagom sieht als den Standardfall für Persistenz das „Event Sourcing" vor. Für den langjährigen Java-EE-Entwickler erfordert das etwas Umdenken, hier zeigt das Framework nach Meinung des Autors jedoch seine größte Stärke (und seine Existenzberechtigung).

In einer CRUD-Anwendung speichern wir in einer relationalen Datenbank den aktuellen Zustand unserer Anwendung. Wenn wir nachverfolgen wollen, wie dieser zustande gekommen ist, greifen wir zu zusätzlichen Kniffen wie Audit-Tabellen, die diese Änderungen festhalten. Event Sourcing *(siehe „https://martinfowler.com/eaaDev/EventSourcing.html")* dreht dies einfach um – statt des Zustands sind alle Ereignisse gespeichert, die zu diesem Zustand geführt haben. Vereinfacht kann man es sich vielleicht so vorstellen: Das, was bisher in Datenbank-Tabellen war, ist nun nur noch im Speicher. In der Datenbank wird lediglich eine Art „commit log" geführt, in dem alle Einträge dauerhaft bestehen bleiben. Um weiterhin die Möglichkeit zu haben, Abfragen über den gesamten Datenbestand auszuführen, dient eine separate „Lese-Seite" (Command Query Responsibility Segregation (CQRS), siehe „https://martinfowler.com/bliki/CQRS.html").

Eine ernsthafte Einführung in Event Sourcing und CQRS würde den Rahmen dieses Artikels sprengen, daher ist auf die angeführten Links verwiesen. Es sei aber gesagt: Dieser Ansatz hat enorme Vorteile. Wer einmal eine relationale Datenbank mit Sharding über mehrere Server verteilen musste, wird Event Sourcing zu schätzen wissen. Da das relationale Modell auf der „Schreiben-Seite" wegfällt und nur noch ein einfaches „Event Journal" geführt wird, kann zum Speichern ohne Weiteres eine NoSQL-Datenbank wie Apache Cassandra eingesetzt werden, und der Skalierung sind quasi keine Grenzen gesetzt. Lesen und Schreiben lassen sich zudem getrennt voneinander skalieren, für die „Lese-Seite" können verschiedene Technologien eingesetzt werden, was unzählige Möglichkeiten eröffnet.

Was hat das nun mit Lagom zu tun? Lagom beinhaltet für die dauerhafte Speicherung das Modul „Lagom Persistence", das im Grunde ein Event-Sourcing-Framework ist. Lagom gibt dabei eine relativ starre Struktur vor und kommt mit allen notwendigen Voreinstellungen. Für den Entwickler heißt es: Ich muss nur die Klasse „PersistentEntity" beerben, die Events definieren, die meine Daten verändern dürfen, und die Event Handler implementieren, die diese Änderungen in meinem im Hauptspeicher gehaltenen Objekt vornehmen. Das war's. Lagom kümmert sich um alles andere: Dass die Events gespeichert werden (per Default in Cassandra, was beim Autor auf Anhieb funktionierte) und dass benötigte Objekte aus den bisherigen Events wiederhergestellt werden (und nach längerer Nichtnutzung auch wieder entfernt).

Ohne weiteres Zutun können die Instanzen eines Lagom-Service, der auf verschiedenen Knoten läuft, ein Cluster bilden. Dann kümmert sich das Framework auch darum, die im Speicher gehaltenen Objekte über die Cluster-Knoten zu verteilen (sogenanntes

```
$ mvn archetype:generate \
    -DarchetypeGroupId=com.lightbend.lagom \
    -DarchetypeArtifactId=maven-archetype-lagom-java \
    -DarchetypeVersion=1.2.0
```

Listing 2

```
$ cd hello-world
$ mvn lagom:runAll
```

Listing 3

„Cluster Sharding"). Dabei können zur Laufzeit auch Knoten wegfallen oder hinzugefügt werden; Lagom kümmert sich um die benötigte Neuverteilung. Das Ganze wirkt sehr ausgereift und bis ins Detail durchdacht, sodass sich damit wirklich gut arbeiten lässt.

Frei nach dem Motto „Batterien sind enthalten, aber auswechselbar" kann man in der Standard-Einstellung mit Cassandra und Event Sourcing sofort loslegen – man kann allerdings auch andere Datenbanken verwenden und auch Services ganz ohne Event Sourcing implementieren. Es empfiehlt sich jedoch, Lagom Persistence auf jeden Fall anzusehen und sich vielleicht darauf einzulassen. Wer sich ohnehin schon einmal mit Event Sourcing und CQRS beschäftigen wollte, findet mit Lagom einen einfachen Einstieg.

… und ein Message-Bus

Wie erwähnt, kommt Lagom als Komplettpaket, und wenn man die Vorgabe Apache Cassandra nutzen möchte, kann man mit der Persistenz sofort loslegen. Das Gleiche gilt auch für asynchrones Messaging – Lagom bringt die Unterstützung für Apache Kafka direkt mit.

Asynchrones Messaging wird von anderen Frameworks etwas stiefmütterlich behandelt. Als Normalfall für die Kommunikation zwischen Services werden oft HTTP-Requests angenommen. Dies ist natürlich auch in Lagom möglich. Eine Besonderheit ist hier die Möglichkeit, sogenannte „Service Descriptor" zu definieren. Ein Service Descriptor kompiliert zu einer kleinen Client-Library, sodass ein typisiertes Interface für die Remote-Aufrufe zur Verfügung steht. Dieses kann per Guice dort injiziert werden, wo der Service verwendet werden soll – aus Entwicklersicht ist dieser dann eine einfache Java-Klasse, mit Typ-Check durch den Compiler (und Methoden-Vorschlägen beziehungsweise -Vervollständigung durch die IDE). Ein nettes Gimmick: Dies funktioniert auch mit WebSockets. Ein WebSocket präsentiert sich als „Akka Stream".

HTTP, REST und JSON in allen Ehren, birgt dieser Ansatz grundsätzlich die Gefahr, dass ein Request zu einer HTTP-Aufrufkette führt, die zusammenbricht, wenn ein Service nicht verfügbar ist. In anderen Worten: Ein nicht verfügbarer Service führt zu einer Fehler-Kaskade und setzt alle direkt oder indirekt von ihm abhängigen Services außer Gefecht. Lagom versieht solche Aufrufe schon standardmäßig mit einem „Circuit Breaker". Diese sorgen zwar für mehr Robustheit, lösen jedoch nicht das grundsätzliche (Architektur-)Problem.

Die Alternative zu diesem „Pull"-Ansatz (Daten bei Bedarf abrufen) ist der „Push" von Daten. Änderungen in einem Service führen dabei zu Domain Events, die über einen Message-Bus an andere Services publiziert werden. Die Services, die die entsprechenden Events abonnieren, sind dann in der Lage, eine eigene Datenbasis lokal vorzuhalten und zu aktualisieren. Es gibt natürlich noch weitere Anwendungsfälle für asynchrone Nachrichten zwischen Microservices. Die durch den Event ausgelöste Aktion kann beliebig sein und muss sich nicht auf die Aktualisierung lokaler Daten beschränken. Ähnlich wie bei der Persistenz kann man bei Lagom mit dem asynchronen Messaging sofort loslegen, ein vorkonfigurierter Apacha-Kafka-Message-Bus ist integriert.

Mit Lagom entwickeln

Der „full stack"-Ansatz von Lagom nährt vielleicht die Befürchtung, dass für die Benutzung erst einmal eine eintägige Installations-Sitzung notwendig ist, um die Voraussetzungen (Cassandra, Kafka etc.) zu schaffen. Die Entwickler haben jedoch besonderes Augenmerk darauf gelegt, dies zu vermeiden. Lagom kommt mit einem eigenen Maven-Plug-in (alternativ wird auch „sbt" als Build-Tool unterstützt). Neben allen Lagom-Microservices, die man als Unterprojekte definiert hat, startet dieses auch die komplette benötigte Infrastruktur – einen integrierten Service-Lookup, einen Gateway-Service sowie die Embedded-Varianten von Cassandra und Kafka.

In der Praxis ist das wirklich recht beeindruckend: Ein Lagom-Demoprojekt auschecken, „mvn lagom:runAll" ausführen und das Terminal füllt sich mit „… started"-Meldungen. Nach einer Weile läuft das System, ohne dass eine einzige Konfiguration oder Installation vorgenommen werden musste. Der schnellste Weg zu einer laufenden Lagom-Anwendung führt über einen Maven-Archetype *(siehe Listing 2)*.

In Folge werden ein paar Namenswünsche abgefragt. Das Verzeichnis, das erzeugt wird, trägt die gewählte „artefactId" als Namen, etwa „hello-world" *(siehe Listing 3)* – und schon hat man eine laufende Lagom-Anwendung. Eine besondere IDE-Unterstützung bringt Lagom nicht mit, die gängige Maven-Integration von IDEA oder Eclipse reichen aber völlig aus: Lagom-Projekte lassen sich ohne Weiteres in diese Entwicklungsumgebungen laden. Sehr angenehm beim Programmieren ist das „hot reloading" – eine Änderung im Source-Code eines Lagom-Service führt dazu, dass beim nächsten Request ohne Neustart des Service neu kompiliert wird und die Änderungen sofort sichtbar sind.

Fazit

Obwohl es noch recht jung ist, wirkt Lagom in den meisten Bereichen gut durchdacht. Für zukünftige Versionen erwartet der Autor vor allem, dass der Austausch von Infrastruktur-Komponenten noch einfacher wird. Es ist jetzt schon möglich, etwa Cassandra durch eine relationale Datenbank zu ersetzen. Man könnte sich vorstellen, dass sich in zukünftigen Versionen beispielsweise auch Kafka leicht durch etwas anderes auswechseln lässt. Sicher wird es auch Verbesserungen geben bezüglich Monitoring und der Integration mit Orchestrierungs-Plattformen wie Kubernetes.

Lagom beeindruckt mit seiner Radikalität und Leistungsfähigkeit. Natürlich löst es nicht alle Probleme und auch mit Lagom schreiben sich die Microservices nicht von allein. Was dem Autor vor allem gefällt, ist, dass es inspiriert. Er hat selten ein Framework kennengelernt, das so sehr zum Nachdenken über Software-Architektur anregt wie Lagom.

Wer sich selbst inspirieren lassen möchte: Beispiel-Applikationen und die komplette Dokumentation stehen unter *„http://www.lagomframework.com"*.

Lutz Hühnken
lutz.huehnken@reactivesystems.eu

Lutz Hühnken ist freiberuflicher Solutions Architect und Trainer. Aktuell beschäftigt er sich mit der Entwicklung von Microservices mit Scala, Akka und Lagom. Er tweetet als „@lutzhuehnken" und bloggt unter „https://huehnken.de".