UNIVERSITY OF APPLIED SCIENCES

Department of Computer Science

Seminar

Build und Deployment von Microservices

in Java

Eingereicht am:

20 Januar 2016

Eingereicht von:
Matthias Metzger (inf101522)
Im Stook 14
25421 Pinneberg
Tel.: (04101) 27 42 9
E-Mail: noobymatze@yahoo.de

Referent:
Prof. Dr. Andreas Häuslein
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Tel.: (041 03) 80 48-42
E-Mail: hs@fh-wedel.de

Betreut von:
Claudio Altamura
Not Available
Not Available
Not Available
Not Available
Not Available

Inhaltsverzeichnis

Abbildungsverzeichnis

List of Listings

1 Einleitung

Google, Amazon und Netflix sind nur einige wenige der bekanntesten Unternehmen, die in ihrer Unternehmensarchitektur Microservices verwenden. Dabei ermöglichen vor allem die Fortschritte in virtuellen Maschinen, Tests und Continuous Integration (CI) eine automatisierte und somit flexible, hochwertige Entwicklung von Softwareprojekten.

Die Umstellung einer monolithischen Anwendungsarchitektur auf eine Microservice Architektur oder deren Einführung vergrößert allerdings auch die Menge der nötigen Technologien und macht damit einen Wissensaufbau notwendig. Aus diesem Grund ist eine sorgfältige, vorausschauende Beleuchtung der häufigsten Probleme und deren Lösungsansätze ein wichtiger Bestandteil dieses Übergangs.

An diesem Punkt setzt diese Arbeit an. Im ersten Schritt wird der Begriff „Microservices“ definiert und anhand von verschiedenen Charakteristika erläutert. Dies ermöglicht den Lesern, Vor- und Nachteile einer solchen Architektur für ihr Unternehmen zu beurteilen. Darauf aufbauend wird anhand eines kleinen HelloWorld-Service mit dem Framework Dropwizard[1] der Aufbau einer Build-Pipeline untersucht. Mit der Hilfe des Continuous Intergration Servers Jenkins[2] wird eine geeignete Variante einer Build-Pipeline für den Build des HelloWorld-Service umgesetzt. Im dritten Schritt werden dann die Problemstellungen des Deployments erläutert und ebenfalls anhand des HelloWorld-Service dargestellt.

Die Sicherheitsaspekte bei der Verwendung von Containern, im Gegensatz zu virtuellen Maschinen sowie das Testing von Microservices ist allerdings nicht Thema dieser Arbeit. Trotzdem sollten diese Aspekte vor einer Einführung einer solchen Architektur eruiert werden.

1.1 Microservices

Mit Microservices wird zum einen ein Architekturstil zur Entwicklung von verteilten Systemen bezeichnet, zum anderen eine Menge von kleinen, autonomen Services, die zusammen arbeiten [New15, 2].

Ein einzelner Microservice stellt also einen unabhängigen Teil von Geschäftslogik für ein Gesamtsystem bereit und lässt sich so auch unabhängig vom Gesamtsystem einsetzen. Dieser Ansatz ähnelt der UNIX Philosophie auf Serviceebene. Diese propagiert die Idee, kleine, auf eine Aufgabe fokussierte Programme/Werkzeuge zu entwickeln, die durch verschiedene Mittel miteinander kombiniert werden können.

Den Gegensatz zu einer aus Microservices bestehenden Architektur stellt ein „Monolith“ dar. Dieser Begriff stammt ursprünglich aus dem Griechischen und bezeichnet einen aus einer einzigen Gesteinsart bestehenden Gesteinsblock. In der Softwareentwicklung ist damit eine Anwendung gemeint, die aufgrund ihrer Größe verschiedene Prinzipien der Softwareentwicklung verletzt. Zu diesen gehören

[1] http://www.dropwizard.io/0.9.1/docs/
[2] https://jenkins-ci.org/

neben Anderen das „Single Responsibility Prinzip“, das folgendes besagt: „Es sollte niemals mehr als einen Grund geben eine Klasse zu ändern“ ([Mar03, S. 10]), eine starke Kopplung verschiedener Module oder schwache Kohäsion. So ist die schnelle Integration von Änderungen zum einen durch die erhöhten Buildzeiten, zum anderen durch implizite Abhängigkeiten von Teilsystemen des Monotlihen untereinander schwierig. Weiterhin stellt vor allem die Skalierung der Mitarbeiter ein Problem dar, da diese ein grobes Verständnis für die gesamte Anwendung besitzen müssen.

1.1.1 Charakteristika

Aufgrund der mangelnden Definition von Microservices und der Microservice Architektur werden diese Begriffe über verschiedene Charakteristika definiert, die solche Systeme aufweisen [LF14]. Ein positiver Nebeneffekt einer solchen Beschreibung liegt in der Befähigung von Softwarearchitekten, Vor- und Nachteile sowie Einsatzgebiete dieser Technologie eigenständig und anwendungsspezifisch zu beurteilen.

Die Charakteristika, veröffentlicht von Martin Fowler und James Lewis in *Microservices*[3], werden im Folgenden genannt und kurz erläutert.

Services als Komponenten

Modularität ist eine fundamentale Technik in der Softwareentwicklung. Sie wird verwendet, um Probleme und Systeme auseinanderzubrechen, sie einzeln zu lösen und zu einem späteren Zeitpunkt zu einer Gesamtlösung zusammenzuführen. Auf Programmebene bedeutet dieses Konzept, gemeinsame Funktionalitäten in Funktionen oder Methoden, mit einer optional definierten Schnittstelle, zu extrahieren. Diese Funktionen und Methoden dienen dann als Grundbausteine für die Umsetzung der Geschäftslogik des Programms. Auf Serviceebene stellen Micro- und Nanoservices solche Grundbausteine für die Erstellung eines Produktes dar.

Der Vorteil dieses Ansatzes findet sich vor allem in der Möglichkeit, einzelne Microserivces unabhängig voneinander zu aktualisieren. Damit wird die flexible und agile Einbindung von Änderungen und Erweiterungen gewährleistet. Auch die Versionierung von Microservices findet so unabhängig voneinander statt.

Ein weiterer Vorteil liegt in der Möglichkeit unterschiedliche Technologien für verschiedene Microservices zu verwenden, womit eine bessere Anpassung an die Natur des jeweiligen Problems gegeben ist.

Ein wesentlicher Nachteil dieses Ansatzes auf Serviceebene liegt darin begründet, dass Microservices so zu verteilten Systemen mit all ihren Problemen und Vorteilen werden. Das bedeutet, dass neben den Fehlerfällen der Geschäftslogik gleichzeitig eine ganze Menge weiterer Fehler behandelt werden müssen.

Organisation um Geschäftsfähigkeiten

Traditionelle Teams werden nach ihren Fachbereichen aufgeteilt. So kann es passieren, dass Entwickler, Designer und Betriebswirte in verschiedenen Abteilungen arbeiten und für ein Projekt miteinander kommunizieren. Conway’s Gesetz besagt dabei, dass auf diesem Weg die Softwarearchitektur die Kommunikationsstruktur im Unternehmen spiegelt.

[3] http://martinfowler.com/articles/microservices.html

In der Microservices Architektur hingegen werden funktionsübergreifende Teams gefördert. Ein solches Team besteht aus Entwicklern, Designern und Betriebswirten, die gemeinsam für die Entwicklung und den Betrieb des Microservice verantwortlich sind.

Daraus ergeben sich vor allem klare und kurze Kommunikationswege zwischen den einzelnen Fachbereichen und eine klare Trennung der Verantwortlichkeiten für ein Produkt. Wird also eine Änderung in einem Microservice benötigt, der nicht in der Verantwortung des entsprechenden Teams liegt, wird ein Featurerequest an das verantwortliche Team gestellt.

In einer monolithischen Anwendung besteht zwar auch die Möglichkeit Teams auf diese Weise zu bilden, allerdings ist sie viel schwerer zu forcieren.

Produkte statt Projekte

Veraltetes Projektmanagement sieht vor, Projekte von einem Team entwickeln zu lassen und die entsprechende Anwendung dann einem Wartungsteam zu übergeben. Werner Vogels, Amazon CTO, fasst mit „You build it, you run it“ die moderne Art, Services zu entwickeln, zusammen. Da die Teams somit den gesamten Lebenszyklus des Service übernehmen, lässt sich dieser ansatzweise als Produkt betrachten. Die verantwortlichen Teams nehmen Funktionalitätsanfragen und Fehlerberichte von anderen Teams, ähnlich einem vertriebenen Produkt, entgegen und sorgen für die Umsetzung und Behebung.

Intelligente Endpunkte, dumme Kommunikationswege

Microservices sind intelligente Endpunkte eines Systems. Die Kommunikation zwischen diesen Endpunkten ist ein elementarer Bestandteil einer Microservice Architektur. Damit Funktionalität nicht von diesen Endpunkten in die Kommunikationswege sickert, sollten die eingesetzten Technologien möglichst weitgehend von der Geschäftslogik freigehalten werden.

Dezentralisierte Führung

Durch die Komponentisierung einer Anwendung in Microservices ist die Wahl der Technologie nur eine Entscheidung, die von zentraler Stelle auf das entsprechende Team des Services übertragen wird. Die benötigten Schnittstellen lassen sich von den einzelnen Teams über Consumer-Driven-Contracts entwerfen.

Anstatt dass ein Service eine Schnittstelle bereitstellt, wird mit diesem Ansatz von den Konsumenten eine Schnittstelle gefordert, die dann durch den entsprechenden Microservice implementiert wird. Die Spezifikation lässt sich weiterhin direkt in die Build-Pipeline einbinden und ermöglicht so das Testen der Spezifikation vor dem produktiven Deployment.

Dezentralisierte Datenverwaltung

In einer monolithischen Anwendung werden in der Regel alle Daten in einer Datenbank gespeichert. Da Microservices abgeschlossene, autonome Teilsysteme sind, beinhaltet das auch die Speicherung der Daten. Daraus folgt, dass das Datenmodell auf verschiedene Services aufgeteilt wird.

Bounded-Contexts ist eine zentrale Idee im Domain-Driven-Design (DDD). Sie basiert auf der Beobachtung, dass Begriffe wie „Kunde“ oder „Produkt“ in verschiedenen Kontexten anders interpretiert werden. Das heißt, sie besitzen in unterschiedlichen Kontexten andere Eigenschaften. Ein Kunde

benötigt im Kontext der Authentifizierung beispielsweise im Wesentlichen ein Passwort und eine E-Mail-Adresse. Bei der Erstellung einer Rechnung werden vor allem auch Nach- und Vorname sowie Kontodaten benötigt.

Gibt es nun einen Microservice, der die Authentifizierung handhabt und einen, der für die Rechnungserstellung zuständig ist, dann werden diese unterschiedlichen Daten von den entsprechenden Microservices verwaltet und nicht in einer zentralen Datenbank persistiert.

Aufgrund der verteilten Natur einer Microservice Architektur stellt die Implementierung von Transaktionen eine Herausforderung dar. In einer verteilten Transaktion müssen die Ressourcen mehrerer Services koordiniert werden. Dies kann zu einer Verzögerung in den Antwortzeiten der Anwendung führen. Ist letzteres aus Geschäftsgründen nicht erwünscht, muss das Ziel der vollständigen Konsistenz der Daten aufgegeben werden.

Die Vorteile dieser Art der Datenverwaltung liegen im Wesentlichen in der Unabhängigkeit.

Automatisierung der Infrastruktur

Durch die hohe Anzahl von Services in einer Microservice Architektur wird ein hoher Grad der Automatisierung beim Build- und Deployment von Microservices benötigt. Die manuelle Verwaltung hunderter oder tausender solcher Services ist wirtschaftlich und technisch nicht tragbar. Aufgrunddessen ist es wichtig, den Build und das Deployment automatisiert ablaufen zu lassen.

Das Monitoring via Log-Analyse ist ebenfalls ein elementarer Bestandteil der Infrastruktur. Mit diesen Werkzeugen wird garantiert, dass ein Ausfall eines Microservice schnell bemerkt wird. Die Automatisierung ermöglicht dann weiterhin einen entsprechend schnellen Rollback des fehlerhaften Service oder einen Neustart.

Fehlertoleranz

Die verteilte Natur von Microservices sorgt dafür, dass neben den möglichen Fehlern der Anwendung selbst eine Menge von Fehlern aus dem Bereich der verteilten Systeme auftreten können. Beispielsweise können unerwartet Services ausfallen oder es kann eine Netzwerkpartition auftreten. Microservice Architekturen müssen daher von Beginn an auf entsprechende Fehler vorbereitet sein.

Ein Teil der Vorbereitung liegt im Aufbau der Monitoring Infrastruktur für die Services, um ausfallende Services und deren Fehlerursachen schnell entdecken und ihnen entgegenwirken zu können. Sind die Antwortzeiten beispielsweise aufgrund einer erhöhten Anzahl von Anfragen größer, besteht die Möglichkeit einer dynamischen, horizontalen Skalierung des betroffenen Service. Ein weiterer Container lässt sich dazuschalten und über einen Load-Balancer ansteuern.

Evolutionäres Design

Das letzte Charakteristikum stellt das evolutionäre Design der Anwendung dar. Durch die geringe Größe der Services lassen sich diese einfach ersetzen oder umstrukturieren. Wird eine neue Funktionalität benötigt, lässt diese sich als zusätzlichen Microservice implementieren, der in einer weiteren Pipeline deployed wird und so mit den anderen Services kommuniziert.

Auch monolithische Anwendungen lassen sich mit Microservices erweitern. So ist es möglich, langsam und stetig einzelne Services aus einer monolithischen Anwendung abzuspalten und diese in eine solche Architektur zu übertragen.

1.1.2 HelloWorld-Service mit Dropwizard

Zur verbesserten Illustration der möglichen Vorteile und Probleme beim Build- und Deployment eines Microservice wird ein kleiner, zustandsloser HelloWorld-Service mit dem Java-Framework Dropwizard entwickelt.

Dieser wird dann in die in Kapitel 2 vorgestellte Build-Pipeline gegeben.

Dropwizard

Dropwizard[4] wurde ursprünglich von dem Social Media Unternehmen Yammer[5] als Open-Source Bibliothek freigegeben. Die Bibliothek vereint die populärsten Open-Source Bibliotheken zur Entwicklung von RESTful Webservices im Java-Umfeld.

Die folgenden Bibliotheken werden verwendet:

- *Jetty*[6] als Webserver.
- *Jersey*[7] für eine REST Schnittstelle.
- *Jackson*[8] für die Verarbeitung von JSON.
- *Logback*[9] für Logging.
- *Hibernate Validator*[10] für die Validierung.
- *Metrics*[11] für die Bereitstellung von Metriken im Produktivbetrieb.
- *JDBI*[12] *und Hibernate* für die Datenbankinteraktion.
- *Liquibase*[13] für die Migration der Datenbank.

Anwendungsaufbau

Jeder mit Dropwizard entwickelte Microservice, besitzt eine Subklasse der von Dropwizard bereitgestellten `Application<T>` (`HelloWorldApplication`) und einen Subklasse der Klasse `Configuration` (`HelloWorldConfiguration`). Abbildung 1.1 illustriert diese Hierarchie.

Die `HelloWorldConfiguration` besitzt Eigenschaften, die mit einer beim Start der Anwendung als Kommandozeilenargument übergebenen Konfigurationsdatei im YAML Format korrespondieren. Im Kontext von Microservices wird damit die umgebungsorientierte Konfiguration des Service möglich. Einstellungen wie das Logging und die zu öffnenden Ports können dort getroffen werden.

Die `HelloWorldApplication` übernimmt das gesamte Bootstrapping des Service. Hier lassen sich die Ressourcen registrieren und Datenbankverbindungen öffnen. Die von Dropwizard vorgeschlagenen Pakete finden sich im Folgenden:

[4] http://www.dropwizard.io/0.9.1/docs/
[5] https://www.yammer.com
[6] http://www.eclipse.org/jetty/
[7] https://jersey.java.net/
[8] http://www.codehaus.org/
[9] http://logback.qos.ch/
[10] http://hibernate.org/validator/
[11] https://github.com/dropwizard/metrics
[12] http://www.jdbi.org/
[13] http://www.liquibase.org/

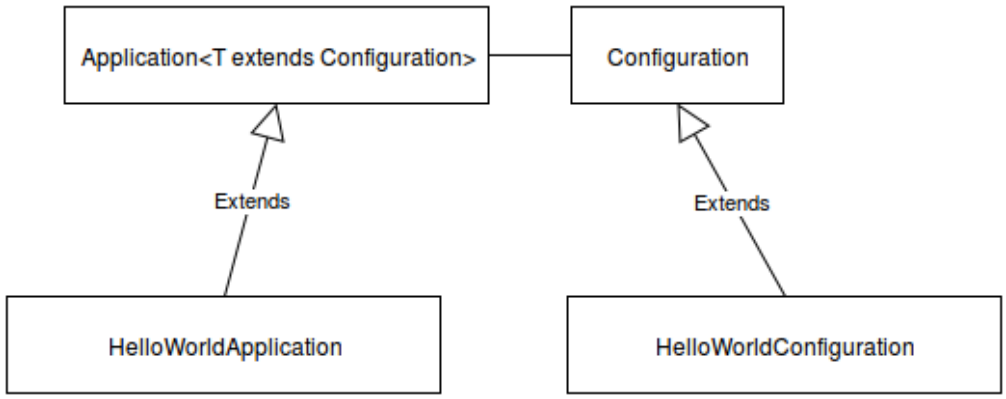

Abbildung 1.1: Dropwizard Klassen im Einsatz

- *api*: In diesem Paket finden sich die Objekt-Repräsentationen der REST Schnittstelle. Im Falle des HelloWorld-Service findet sich dort die Klasse „HelloWorld“ mit einem message-Attribut.
- *resources*: Diese beinhalten die REST Ressourcen, die die Repräsentationen freigeben. Für den HelloWorld-Service findet sich dort eine Klasse „HelloWorldResource“.
- *health*: Dieses Paket enthält Checks zum Beispiel für das Testen, ob die Datenbank oder die MessageQueue erreichbar ist.

Es gibt weitere Pakete, die allerdings für die Größe dieses Service nicht notwendig sind.

- *core*: Beinhaltet die Entitäten dieses Service.
- *db*: Beinhaltet Data-Access-Objects (DAOs).
- *views*: Diese beinhalten Templates für HTML Views.

HelloWorldResource

Die HelloWorldResource sieht dann folgendermaßen aus:

```
@Path("/")
public class HelloWorldResource {

  @GET
  @Produces(MediaType.TEXT_PLAIN)
  public String hello() {
    return "Hello World!";
  }
}
```

Im Folgenden wird für dieses Projekt eine Build-Pipeline erstellt.

2 Build

In dem vorangegangenen Kapitel wurde gezeigt, dass die Automatisierung des Build- und Deployment Vorgangs ein elementares Charakteristikum für die Kategorisierung einer Architektur als Microservice Architektur ist.

Zur Erreichung dieses Ziels wird Continuous Integration (CI) beziehungsweise Continuous Delivery (CD) verwendet. Continuous Integration ist eine Entwicklungstechnik, in der jeder Entwickler seine Änderungen häufig - in der Regel täglich - in den Hauptzweig einer Anwendung integriert. Jede dieser Integrationen wird von einem automatisierten Build inklusive Tests verifiziert [FF06]. Continuous Delivery ist die natürliche Erweiterung von Continuous Integration und bezeichnet eine Menge von Techniken und Prozessen, wie Build-Automatisierung und kontinuierliches Deployment.

Aus diesem Ansatz ergibt sich die Idee einer Build-Pipeline. Diese gibt für jedes Build-Artefakt eine Reihe von Phasen vor, die vor dem Deployment des Service durchlaufen werden müssen. Zur Umsetzung dieser Pipeline wird zunächst ein Build System für die entsprechende Programmiersprache benötigt. Dieses sollte reproduzierbare Builds inkl. Abhängigkeiten auf diversen System unterstützen.

Darauf aufbauend lässt sich dann ein CI Server zur Umsetzung der Pipeline verwenden. Bei fehlgeschlagenen Tests oder anderweitig auftretenden Fehlern können beispielsweise E-Mails an die Entwickler verschickt werden. Vor allem durch die jüngsten Entwicklungen im Bereich der Container und Virtualisierung lassen sich diese Schritte in Isolation ausführen. Im Folgenden werden die benötigten Schritte und Technologien genauer erläutert. Darauf aufbauend wird eine beispielhafte Pipeline für den HelloWorld-Service mit dem CI Server Jenkins aufgezeigt.

2.1 Build-Pipeline

Die Phasen der Build-Pipeline können beliebig gewählt werden. In Abbildung 2.1 lässt sich eine beispielhafte Pipeline für einen einzelnen Microservice erkennen.

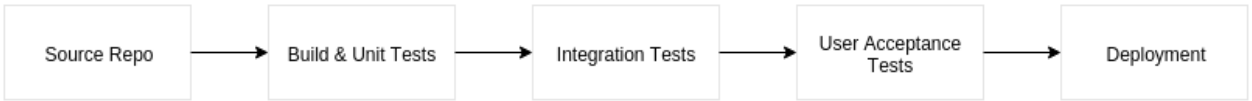

Abbildung 2.1: Beispielphasen einer Build-Pipeline

Der erfolgreiche Build und die erfolgreiche Ausführung der Unit Tests führen nun dazu, dass im Nachfolgenden die Intergrationstests und danach die Akzeptanztests ausgeführt werden. Diese Pipeline lässt sich beliebig erweitern oder verkürzen. Die statische Code Analyse mit Werkzeugen wie Checkstyle[1] oder SonarQube[2] lässt sich beispielsweise vor oder sogar parallel mit den Integrationstests ausführen.

[1] http://checkstyle.sourceforge.net/
[2] http://www.sonarqube.org/

Dies erhöht zum einen die Geschwindigkeit, mit der sich die Pipeline abarbeiten lässt, zum anderen auch die Qualität und das Vertrauen in den Quellcode.

2.1.1 Alternativen

Komplexere Fälle ergeben sich, wenn es darum geht, das Deployment mehrerer Microservices zu koordinieren. Für die Gestaltung der entsprechenden Build-Pipelines gibt es verschiedene Alternativen [New15, S 105 ff.].

Monolithischer Build, ein Repository

Den einfachsten Fall stellt eine Pipeline dar, in der es ein Quellcode Repository gibt, das durch einen monolithischen Build erstellt wird. In Abbildung 2.2, übernommen aus dem Buch *Building Microservices* von Sam Newman, lässt sich diese Art von Build-Pipeline erkennen.

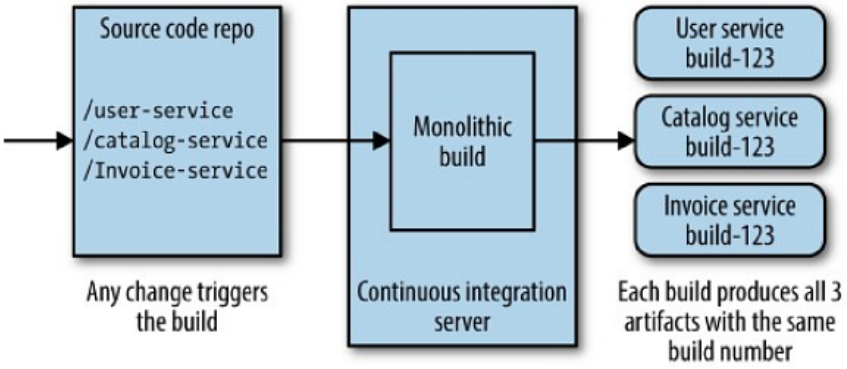

Abbildung 2.2: Monolithischer Build mit einem Quellcode Repo [New15, S. 105]

Die Gestaltung einer solchen Pipeline stellt sich in der Praxis allerdings als unpraktisch heraus. Sofern nicht an mehreren der Microservices Änderungen vorgenommen werden, führt die Änderung eines einzelnen Service zum unnötigen Build der jeweils anderen Services.

Weiterhin lassen sich innerhalb eines Commits mehrere Services auf einmal verändern, was eine nachträgliche Trennung der Verantwortlichkeiten und eine mögliche Fehlersuche erschwert. Auch die Verantwortlichkeit für die jeweiligen Services lässt sich auf diesem Weg nicht forcieren, womit jedes Team verantwortlich für alle Services ist.

Zeitgleich stellt das einzelne Repository allerdings auch einen Vorteil dar, da sich der gesamte Quellcode an einer Stelle befindet und somit kein aufwändiges Umschalten zwischen verschiedenen Repositories benötigt wird.

Multiple Builds, ein Repository

Eine Variante des vorangegangenen Beispiels stellen mehrere Builds aus einem Repository dar. Dies wird in Abbildung 2.3 skizziert.

Bei diesem Ansatz kann ein (versehentlicher) Commit immer noch für die Veränderung mehrerer Services verantwortlich sein. Kann dieses Risiko organisatorisch in Kauf genommen werden, stellt dieser Ansatz kein Problem dar. Es ist trotzdem möglich einzelne Teams aufzuteilen und ein Commit eines Service führt nicht automatisch zum Build aller weiteren Services.

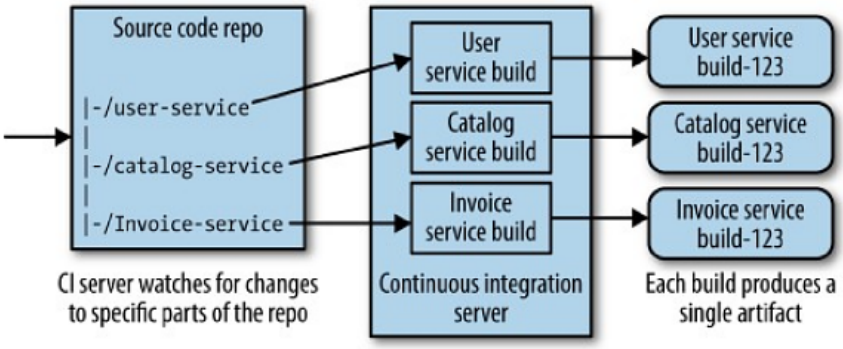

Abbildung 2.3: Multiple Builds mit einem Repository [New15, S. 106]

Multiple Builds, multiple Repositories

Um auch den letzten Nachteil einer unklaren Trennung auszuräumen, findet sich in Abbildung 2.4 der dritte und letzte der hier diskutierten Ansätze.

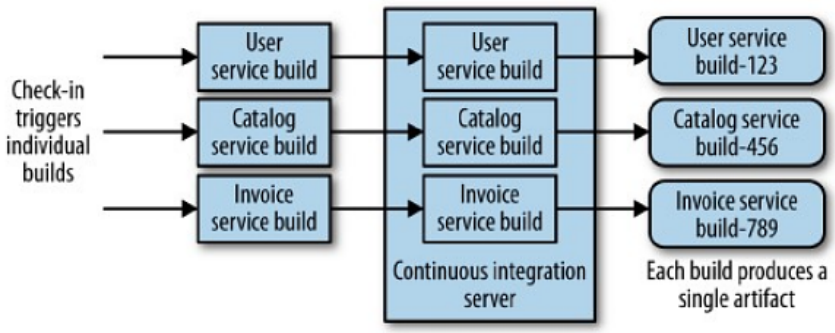

Abbildung 2.4: Multiple Builds mit mehreren Repositories [New15, S. 107]

Dabei fällt auf, dass jeder Service ein eigenes Repository bekommt. Damit ist es möglich die Zugriffsrechte zu steuern und die Kommunikation zwischen Teams bei erwünschten Änderungen zu forcieren.

Auch wird die Unabhängigkeit der Services gefördert, da es keinen gemeinsamen Build mehr gibt. Obgleich dieser Ansatz mit dem größten Aufwand verbunden ist, wird er aufgrund der genannten Vorteile bevorzugt.

In dem hier entwickelten HelloWorld-Service gibt es keinen solchen Build, da es nur einen einzigen Service gibt.

2.2 Build Systeme

Ein Build System ist ein Werkzeug zur automatisierten, reproduzierbaren Erstellung einer Anwendung oder in diesem Fall eines Microservice. Vor allem die Verwaltung von Abhängigkeiten ist ein elementarer Bestandteil eines Build Systems. Für die Programmiersprache Java, in der der HelloWorld-Service entwickelt wird, gibt es verschiedene Build Systeme.

Apache Ant

Apache Ant[3] steht für „Another Neat Tool“ und wurde im Jahr 2000 von James Duncan Davidson entwickelt, weil dieser ein Build Werkzeug für Java benötigte. Jedes mit Ant entwickelte Projekt beinhaltet eine Datei namens `build.xml`. Diese liegt im Wurzelverzeichnis eines Projektes und definiert verschiedene Ziele, die von den Benutzern per Kommandozeile aufgerufen werden können.

Im Gegensatz zu anderen Build Systemen schreibt Ant keine Verzeichnisstruktur vor und beinhaltet auch keinen Weg, um automatisiert Abhängigkeiten in eine JAR einzubinden. Diese Funktionalitäten müssen von den Projektentwicklern eigenständig definiert werden.

Apache Maven

Apache Maven[4] ist ebenfalls ein Build Werkzeug und wurde im Jahr 2004 erstmals veröffentlicht. Im Gegensatz zu Ant basiert es auf dem Prinzip „Convention over Configuration“ (CoC), zu Deutsch: Konvention über Konfiguration. Um dem Aufwand der Konfiguration zu entgehen, der beispielsweise bei Ant anfällt, wird dabei eine einheitliche Konvention festgelegt. Auf diesem Weg müssen nur Ausnahmen oder Abweichungen konfiguriert werden. Entsprechend ähnelt sich der Aufbau der Projekte, die mit Apache Maven als Build System erstellt werden, was den Einstieg bei bekannten Konventionen vereinfacht. Die Plugin basierte Architektur von Apache Maven ermöglicht weiterhin die Definition von spezifischen Schritten für den Build des Projektes.

Apache Maven hat vor allem eine einheitliche Verzeichnisstruktur für Java Projekte als Konvention eingeführt und bietet aufgrunddessen automatisch die Verwaltung von Abhängigkeiten der Anwendung. Dafür existiert ein zentrales Repository[5], in dem (Open-Source-) Bibliotheken zu finden sind, die sich in jedem Projekt einbinden lassen.

Auch bei einem Maven-Projekt gibt es eine zentrale Projekt-Datei: die `pom.xml`. Diese beinhaltet die gesamte Konfiguration für das Projekt. Letzters kann dabei auch aus mehreren Sub-Projekten bestehen. Im Kontext von Microservices ließen sich beispielsweise die Integrations- oder Performance-Tests in einem Sub-Projekt erstellen.

Gradle

Gradle[6] wurde im Jahr 2007 veröffentlich, ist ebenfalls ein flexibles, allgemeines Werkzeug für den Build von Projekten und setzt auf den Erkenntnissen von Apache Ant und Apache Maven auf.

Wie Apache Maven ermöglicht Gradle ebenfalls CoC Builds. Im Gegensatz zu Apache Maven erfolgt die Konfiguration allerdings durch eine auf Groovy basierende Domain-Specific-Language (DSL). Mit der Hilfe des Gradle Wrappers lässt sich eine einheitliche Version spezifisch für das Projekt verwenden. Damit benötigt auch der CI Server keine globale Installation von Gradle.

Gradle verwendet das Repository von Apache Maven, womit Zugriff auf fast alle Open-Source-Biblioteken von Java besteht. Im Gegensatz zu Maven erstellt Gradle inkrementelle Builds des Projektes, womit die Kompilierzeiten stark reduziert werden.

Aus diesen Gründen wird für diese Arbeit Gradle als Build Werkzeug verwendet. In Abbildung 2.5 lässt sich sowohl die Struktur der Anwendung, als auch die gesamte Gradle Konfiguration betrachten.

[3] http://ant.apache.org/
[4] https://maven.apache.org/
[5] http://search.maven.org/
[6] https://gradle.org/

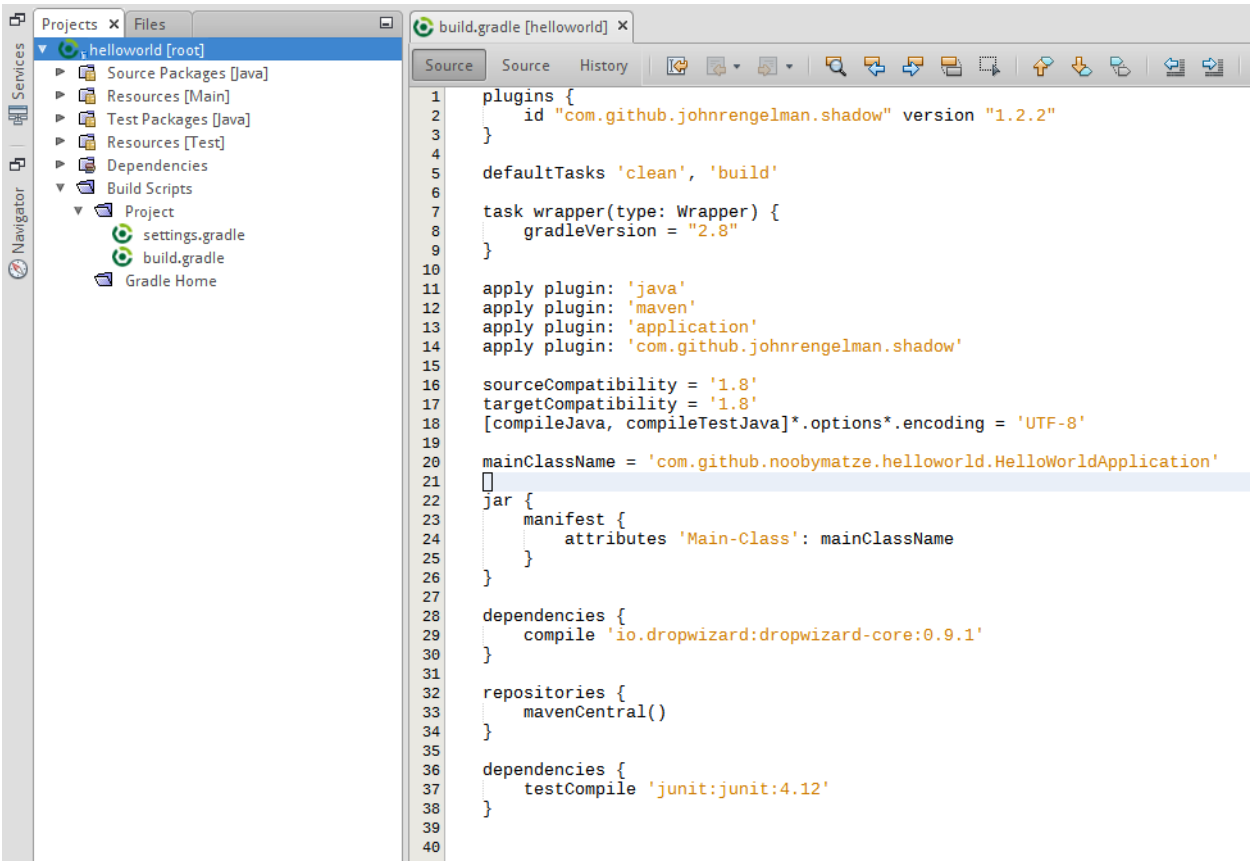

Abbildung 2.5: Gradle Projektstruktur und Konfiguration des HelloWorld-Service

Erwähnenswert ist dabei die Abhängigkeit in dem Abschnitt `dependencies` auf Dropwizard und das Shadow Plugin[7], das mit der Hilfe von `apply plugin: 'com.github.johnrengelman.shadow'` verwendet wird. Damit wird eine ausführbare JAR erstellt, die alle Abhängigkeiten beinhaltet und damit nur eine Java Runtime Environment (JRE) benötigt, um gestartet zu werden.

2.3 Testing

Das Testing von Microservices wird innerhalb dieser Arbeit nicht weiter vertieft. Dennoch ist es ein wichtiger Bestandteil des automatisierten Builds und soll in diesem Abschnitt kurz beleuchtet werden. Neben Unit Tests, Integration Tests, User Acceptance Tests, Functional Tests, End to end Tests und Performance Tests gibt es speziell in einer Microservice Architektur Tests, die aus Consumer-Driven-Contracts resultieren.

Unit Tests sollten von anderen Systemen unabhängige Teile der Geschäftslogik testen und somit eine geringe Zeit benötigen. Damit erlauben sie ein schnelles Feedback und können eigenständig ausgeführt werden.

Integration Tests testn die Integration mit anderen Systemen und stellen somit eine Möglichkeit dar, Datenbankmigrationen oder die Abhängigkeit zu anderen Microservices im Verbund zu testen. An dieser Stelle ist es möglich, die Tests von Consumer-Driven-Contracts einzubinden. Diese werden im vorhinein spezifiziert und stellen sicher, dass Consumer der von dem getesteten Microservice

[7] https://github.com/johnrengelman/shadow

bereitgestellten Schnittstelle mit dem Service kommunizieren können. Dies hat zur Folge, dass die entsprechenden Services in der Testphase nie gestartet werden müssen.

Sind diese Tests erfolgreich, lässt sich beispielsweise mit der Hilfe von Docker ein neues Abbild des Service erstellen. Ein User Acceptance Tester kann dieses in seiner lokalen Umgebung starten, die gewünschten Tests durchführen und die Build-Pipeline manuell in den nächsten Schritt starten lassen.

Performance Tests lassen sich parallel dazu oder über Nacht ausführen und stellen sicher, dass die Antwortzeiten eines Service' auch bei hoher Last stabil bleiben und dieser zuverlässig arbeitet.

2.4 CI Server

Der Continuous Integration Server ist für die konkrete Implementierung einer Build-Pipeline zuständig und verbindet damit alle separaten Phasen der Build-Pipeline. Jenkins[8], Hudson[9], Gitlab CI[10] und Travis[11] sind nur ein paar der möglichen Alternativen.

Für diese Arbeit wird Jenkins verwendet. Diese Vorgehensweise ergibt sich aus der Tatsache, dass Travis CI nur in der Enterprise Variante auf einem lokalen Server installiert werden kann. Weiterhin ist Jenkins der Open-Source-Nachfolger von Hudson. Während die Open Source Variante von Gitlab CI vollständig funktionsfähig ist, bietet die Enterprise Variante weitere Funktionalitäten an.

2.5 Build-Pipeline für HelloWorld-Service

Zur Erstellung einer Build-Pipeline für den HelloWorld-Service wird Jenkins zunächst mit einigen Plugins erweitert. Da der HelloWorld-Service mit der ursprünglich von Linus Torvalds entwickelten Versionsverwaltung Git[12] versioniert wird, wird das Git Plugin[13] verwendet. Weiterhin gibt es ein Build-Pipeline-Plugin[14], das es ermöglich, eine Build-Pipeline zu visualisieren. In Abbildung 2.6 lässt sich eine solche Ansicht erkennen.

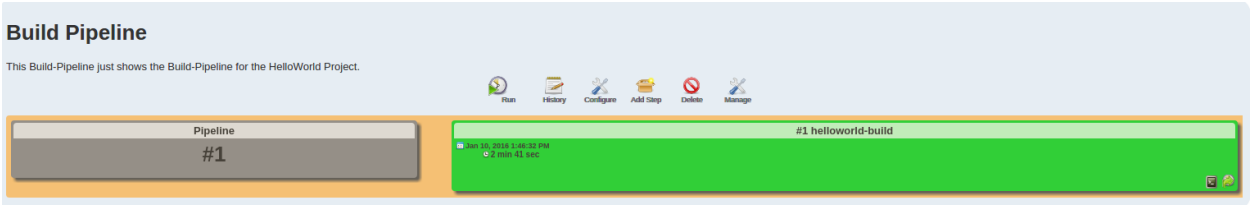

Abbildung 2.6: Build-Pipeline des HelloWorld-Service

Die einzelnen Schritte der Build-Pipeline werden dabei als Jobs in Jenkins definiert. Nur der erste Job entnimmt dabei den Stand des Quellcode Repositories. Die weiteren Jobs kopieren den Workspace ohne die Verzeichnisse der Versionsverwaltung. Zu diesem Zweck wird das Clone Workspace SCM Plugin[15] verwendet.

[8] https://jenkins-ci.org/
[9] http://hudson-ci.org/
[10] https://about.gitlab.com/gitlab-ci/
[11] https://travis-ci.org/
[12] https://git-scm.com/
[13] https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin
[14] https://wiki.jenkins-ci.org/display/JENKINS/Build+Pipeline+Plugin
[15] https://wiki.jenkins-ci.org/display/JENKINS/Clone+Workspace+SCM+Plugin

Die Konfiguration des Builds mit der Hilfe von Gradle lässt sich auf Abbildung 2.7 erkennen.

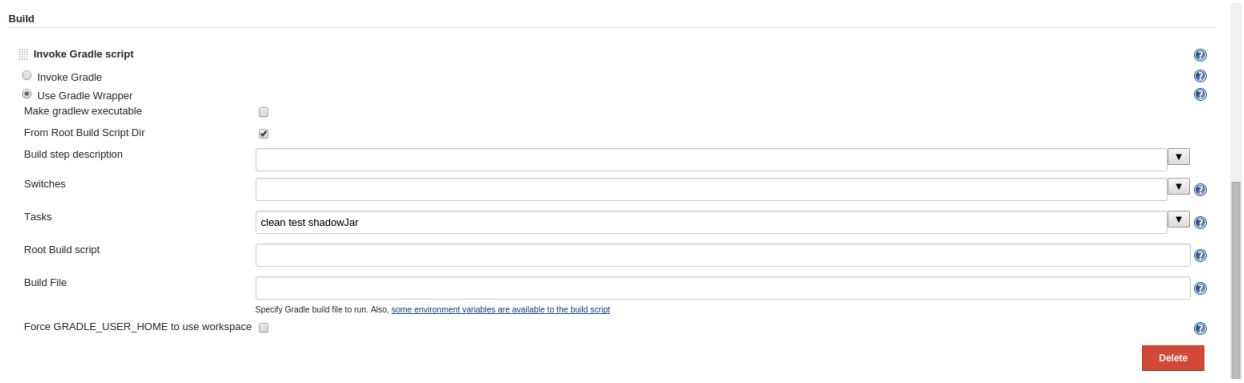

Abbildung 2.7: Gradle Konfiguration helloworld-build

Das Git Plugin sorgt im vorhinein dafür, dass der letzte Commit des Hauptzweigs (`master branch`) aus dem Github Repository[16] für den Build verwendet wird. Die von Gradle hier auszuführenden Tasks sind zunächst das Bereinigen („clean“) der bisherigen Build-Artefakte. Die Ausführung der Unit Tests („test“) und danach das Erstellen der ausführbaren JAR mit allen Abhängigkeiten via das „shadowJar“-Kommandos. Die Testergebnisse lassen sich weiterhin veröffentlichen, beziehungsweise mit dem Test Results Analyzer Plugin[17] analysieren.

Ein fehlschlagender Build sorgt durch die in Abschnitt 3.4 gezeigte Konfiguration des nachfolgenden Build-Schritts für ein sofortiges Stoppen der Build-Pipeline. Es lässt sich infolgedessen eine E-Mail an die Entwickler des Projekts verschicken, um die Fehler schnellstmöglich zu beheben. Parallel oder nachfolgend kann die Quellcode-Analyse mit SonarQube oder Checkstyle erfolgen. Weiterhin ist es möglich, einen Folgeschritt nur manuell auszulösen. Somit lassen sich User Acceptance Tests (UAT) abbilden, die den manuellen Eingriff von Testern benötigen. Ein zufriedenstellendes Ergebnis lässt sich dann manuell zum Deployment oder Pre-Deployment freigeben.

Mit diesen Überlegungen und Einstellungen lässt sich die in Abbildung 2.6 gezeigte Ansicht erstellen, um die resultierende Pipeline zu visualisieren.

[16] https://github.com/noobymatze
[17] https://wiki.jenkins-ci.org/display/JENKINS/Test+Results+Analyzer+Plugin

3 Deployment

Nachdem nun der Build in Form einer beliebig erweiterbaren Pipeline feststeht, eröffnet sich die Frage nach einem automatisierten Deployment der Microservices. Wie beim Build unternehmensspezifische Fragestellungen zunächst beantwortet werden müssen, ergeben sich diese ebenfalls bei Deployment der Microservices. Vor allem die Trade-offs einer Microservice Architektur werden hier diskutiert.

3.1 Service-Host-Mapping

Wie viele Services auf einem Host in Betrieb genommen werden sollten, ist eine der ersten Fragen, die sich beim Deployment von Microservices ergibt. Host wird in diesem Fall mit der von Sam Newman verwendeten Bedeutung einer isolierten Einheit, auf der sich ein Service installieren lässt, belegt [New15, vgl. S. 116].

3.1.1 Mehrere Services pro Host

Den einfachsten Ansatz stellt das Deployment mehrerer Services auf einem Host dar. Abbildung 3.1 skizziert diese Idee.

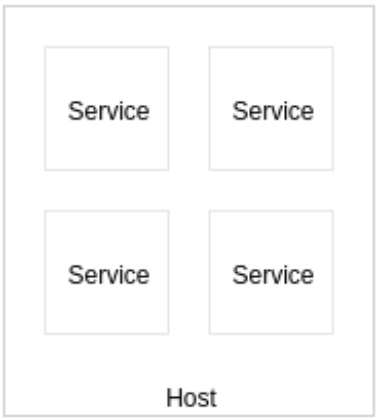

Abbildung 3.1: Mehrere Services pro Host

Dies vereinfacht vor allem die Verwaltung der Infrastruktur, da ein weiterer Service auf dem entsprechenden Host deployed werden kann. Ein zweiter Vorteil findet sich in der Tatsache, dass die Virtualisierung solcher Services Overhead für die Verwaltung der virtuellen Maschinen kostet und damit das Deployment auf einem Host einfacher ist.

Ein Charakteristikum von Microservices ist die Verwaltung des Service von den entsprechenden Teams, die für die Entwicklung verantwortlich sind. Mit mehreren Services pro Host stellt sich die Frage, welches Team für die Konfigruation des Hosts verantwortlich ist. Weiterhin können

übergreifend inkompatible Bibliotheken von den verschiedenen Services benötigt werden, womit die Konfiguration weiterhin erschwert wird.

Fällt einer der Services aus oder beansprucht mehr CPU-Zeit, stellt dies ein weiteres Problem dar. Welcher Service verantwortlich ist und aus welchem Grund, ist dabei nicht zwangsweise offensichtlich.

3.1.2 Einzelner Service pro Host

Eine Alternative zu dem vorherigen Deployment mehrerer Services auf einem Host findet sich in Abbildung 3.2.

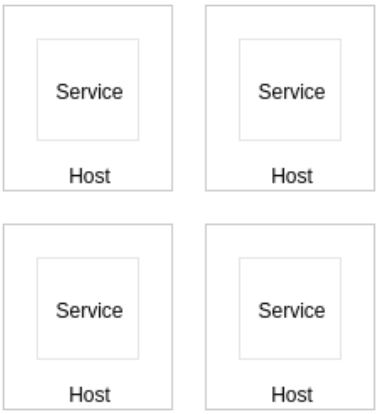

Abbildung 3.2: Mehrere Services pro Host

Dabei beherbergt jeder Host einen eigenen Service. Mit diesem Modell trägt jedes Team des entsprechenden Service die Verantwortung für dessen Deployment und kann damit beliebige Konfigurationen vornehmen. Auch ein Ausfall oder die plötzliche Verwendung einer größeren Menge von Ressourcen ist sofort einem einzelnen Service zuzuordnen, womit die Fehlersuche vereinfacht wird.

Die Nachteile liegen im Wesentlichen in der Verwaltung der größeren Anzahl von Hosts und dem damit verbundenen finanziellen und zeitlichen Kostenaufwand.

3.2 Kommunikation

Ein wichtiges Charakteristikum von Microservices aus Kapitel 1 stellt die Unabhängigkeit und Autonomität von Microservices dar. Trotzdem ist eine Kommunikation zwischen den Services oftmals unvermeidlich. Der einfachste Ansatz, um eine Kommunikation zwischen zwei oder mehr Services zu erlauben, ist die entsprechenden IP-Adressen der benötigten Endpunkte als Konfiguration beim Starten des Microservices festzulegen. Diese Idee birgt allerdings einige Nachteile.

Zum einen können sich IP-Adressen zwischen Deployments ändern, wenn diese dynamisch vergeben werden. Zum anderen lassen sich zusätzliche Instanzen nur mit einem Reboot der schon laufenden Services hinzufügen.

3.2.1 Service Registry

Zu diesem Zweck gibt es die Idee einer Service Registry. Jeder Service registriert sich während des Boots bei der Service Registry und meldet sich beim Herunterfahren ab. Ein Client des entsprechenden Services erfragt vor dem Request bei der Service Registry mögliche Endpunkte und kann dann Anfragen an diese Services stellen.

3.2.2 API Gateway

Das API Gateway ist ein Muster in einer Microservice Architektur, um die Kommunikation mit den Microservices zu bündeln. Dabei kann das Gateway als Proxy dienen oder zusätzliche Schnittstellen bereitstellen, mit denen beispielsweise Daten über mehrere Microservices hinweg aggregiert werden.

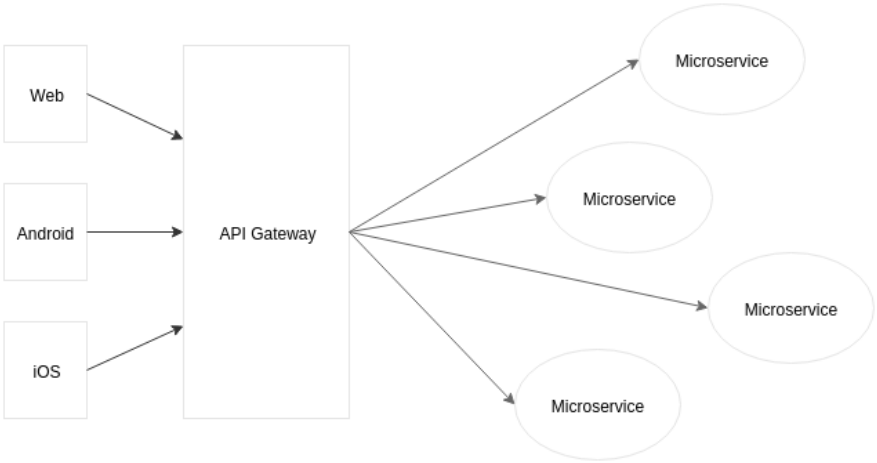

Abbildung 3.3: Überblick API Gatway

Abbildung 3.3 zeigt einen Überblick eines solchen Aufbaus. Erkennen lässt sich dabei, wie eine Anfrage eines UI zu mehreren Anfragen an verschiedene Microservices wird. Dies ist vor allem vorteilhaft für mobile Anwendungen, da mobile kabellose Netzwerke deutlich langsamer und kostspieliger für die Nutzer sind. Auch die Microservices selbst sind in der Lage, das API Gateway für den Datenaustausch zu nutzen und müssen auf diesem Weg keine festen Referenzen auf andere Services besitzen.

Ein Problem, das bei einer solchen Architektur allerdings auf Entwicklerseite entstehen kann, ist das Akkumulieren von Geschäftslogik innerhalb eines solchen Gateways. Dies führt erneut zu der Möglichkeit, Logik im gesamten System zu verteilen, anstatt diese innerhalb des entsprechenden Microservic zu kapseln.

Um diese Möglichkeit auszuschließen oder zumindest zu verhindern, lässt sich das API Gateway in mehrere kleinere Backends aufteilen, die jeweils mehrere Microservices aggregieren und sich so als Frontend für diese ansehen lassen [New15, S. 72].

Ein weiteres Problem von API Gateways liegt darin, dass diese den zentralen Anlaufpunkt für jegliche Anfragen an die Microservices darstellen. So ist es möglich, dass durch zu lange Timeouts und den Ausfall eines Microservice eine Kaskade ausgelöst wird, die diesen Fehler in viele Systeme propagiert und damit zum Ausfall des gesamten Systems führt.

Daher sollten entsprechende Muster wie Circuit Breaker implementiert werden. Diese agieren als Proxy vor Microservices und sorgen bei zuvielen Timeouts des eigentlich Service dafür, dass sie geöffnet werden und somit jede weitere Anfrage sofort in einem Timeout des Circuit Breakers

resultiert [Nyg07, S. 115 ff.]. Der Fehlerfall eines solchen Services wird dann via Monitoring an die verantwortlichen kommuniziert.

3.2.3 Message Queue

Message Queues, wie RabbitMQ[1], Apache Kafka[2] sind eine weitere Technologie, mit der sich die Kommunikation von Microservices koordinieren lässt [Wol15, vgl. S. 184 ff.].

Solche Systeme eigenen sich vor allem, wenn eine Event getriebene Architektur benötigt wird, da die Produzenten und Konsumenten einer Queue voneinander entkoppelt werden. Somit ist es möglich, bei einer größeren Anzahl von Nachrichten dynamisch die Anzahl der konsumierenden Services zu erhöhen. Die entsprechende Datenbank der Message Queue handhabt die persistente Speicherung der Nachrichten und ermöglicht somit sogar die Implementierung von Transaktionen.

Anderweitig müsste eine globale Transaktion alle Microservices umfassen, an denen eine Änderung vorgenommen werden muss. Dies birgt vor allem Risiken, wenn Timeouts oder Fehlerfälle auftreten, da die Transaktion das Schreiben anderer Daten möglicherweise für eine lange Zeit blockiert.

Mit der Hilfe von Message Queues kann das Empfangen und versenden von Nachrichten in eine Transaktion gewickelt werden. Bei Fehlern lassen sich Änderungen an den Daten zurückrollen und die Nachrichten werden nicht verschickt. Schon verschickte Nachrichten lassen sich bei diesem Vorgehen allerdings nicht wieder rückgängig machen.

3.3 Virtualisierung/Container

In Abschnitt 1.1.1 wurde auf die Automatisierung der Infrastruktur als ein Charakteristikum für Microservices eingegangen. Zur Umsetzung dieser Eigenschaft beim Deployment von Microservices werden die technologischen Fortschritte in der Virtualisierung verwendet.

Dabei existieren verschiedene Ansätze. Virtuelle Maschinen (VM), wie VirtualBox[3], VMWare[4] stellen die traditionelle Weise dar, in der sich Betriebssysteme isoliert auf einem Hostsystem starten lassen. Dabei gibt es einen Hypervisor, der im Wesentlichen zwei Aufgaben übernimmt. Zum einen sorgt er für die Zuteilung von CPU und Speicher zu der VM und zum anderen agiert er als Kontrollschicht, über die sich die VMs selbst manipulieren lassen [New15, vgl. S. 123].

Innerhalb der VM lässt sich ein isoliertes und vom Hostsystem unabhängiges Betriebssystem starten. Der Hypervisor selbst benötigt allerdings Ressourcen, die somit einen Effekt auf die Performanz des emulierten Betriebssystems haben kann.

3.3.1 Linux Container

Dieses Problem lässt sich mit Linux Containern umgehen. Statt einen Hypervisor zu verwenden, wird bei dieser Technologie ein eigener Prozessraum erstellt, in dem die Prozesse des Containers isoliert von den Host-Prozessen leben. Der Linux Kernel verwaltet Bäume von Prozessen. Ein Linux Container stellt einen Sub-Baum von Prozessen dar, die somit automatisch vom Linux Kernel verwaltet werden und damit den Hypervisor überflüssig machen.

[1] https://www.rabbitmq.com/
[2] http://kafka.apache.org/
[3] https://www.virtualbox.org/
[4] http://www.vmware.com/de

Diese Technik findet sich in allen modernen Linux Kerneln und kann somit nur durch eine VM mit einem Linux Kernel emuliert werden. Durch den unnötigen Hypervisor ist die Startzeit von Linux Containern kürzer und die Anzahl der möglichen simultan gestarteten Container ist höher.

Die Verwendung dieser Technologie statt virtuellen Maschinen sollte allerdings trotzdem abgewogen werden. Linux Container sind nicht vollständig von ihren Host-Systemen isoliert, womit die Möglichkeit besteht, dass nicht vertrausnwürdige ausführbare Dateien Schadcode auf dem Host-System einschleusen.

3.3.2 Docker

Auf Linux Containern aufbauend wurde eine Technologie namens Docker[5] entwickelt, mit deren Hilfe sich Abbilder und Container verwalten lassen.

Das Unternehmen, das Docker vertreibt (docker Inc), stellt dabei eine Registry bereit, in der Open-Source Basisabbilder von verschiedenen Betriebssystemen und Anwendungen veröffentlicht, heruntergeladen und erweitert werden können.

Erweitern lassen sich diese mit der Hilfe einer Datei namens `Dockerfile`. Innerhalb dieser Datei können verschiedene Befehle verwendet werden, um das entsprechende Abbild zu konfigurieren. In Listing 3.1 findet sich die Konfiguration für das Docker-Abbild des HelloWorld-Service.

```
FROM java:7-jre
RUN mkdir /opt/helloworld
COPY helloworld.yml /opt/helloworld
COPY build/libs/helloworld-all.jar /opt/hellworld
EXPOSE 8000
WORKDIR /opt/helloworld
CMD ["java", "-jar", "helloworld-all.jar", "server", "helloworld.yml"]
```

Listing 3.1: Dockerfile HelloWorld-Service

Mittels des Kommandos `docker build` lässt sich aus diesem `Dockerfile` ein eigenes Abbild erstellen. Dabei wird der Ordner, in dem das Dockerfile liegt als Kontext verwendet, sodass innerhalb dieses Ordners die Konfigurationsdateien liegen müssen, damit der `COPY` Befehl funktioniert.

Abbildung 3.4: Docker Build Konfiguration

[5] https://www.docker.com/

3.4 HelloWorld-Service mit Docker

Das Deployment des HelloWorld-Service erfolgt mit Docker. In Listing 3.1 findet sich das `Dockerfile`, mit dem sich ein Abbild erstellen lässt. Der Befehl `docker build -t helloworld .` sorgt in Jenkins für die Erstellung des Images. Zu diesem Zweck muss auf dem Host Docker installiert werden. Weiterhin muss der Docker-Daemon ohne Root-Rechte anzusprechen sein.

Die Konfiguration für den Deployment-Schritt des HelloWorld-Service lässt sich in Abbildung 3.4 erkennen. Auffällig ist dabei im Wesentlichen der Befehl, um den Docker Container zu erstellen und zu deployen.

Der Grund für die Länge des Befehls liegt darin, dass es keine Möglichkeit gibt, das laufende Abbild mit einem neu erstellten Docker Container auszutauschen. Dazu muss der Docker Container zunächst gestoppt werden. Erst dann lässt sich das neue Abbild mit `docker run -d -p 8000:8000` starten. Das `-d`-Flag steht bei diesem Befehl dafür, den Container im Hintergrund zu starten.

In diesem konkreten Fall ist es allerdings auch möglich, dass noch kein Container gestartet wurde, weswegen diese Möglichkeit mit in den Befehl einbezogen werden muss. Der Autor dieser Arbeit hat diese Möglichkeit zunächst außer acht gelassen, was sich anhand des roten, fehlgeschlagenen Builds in Abbildung 3.5 erkennen lässt, die die vollständige Build-Pipeline für das Praxisbeispiel darstellt.

Ein Starten dieser Pipeline führt zunächst zu einem Build und der Ausführung der Unit Tests mit Gradle. Danach wird mit dem oben diskutierten Befehl ein Docker Abbild erstellt und dieser gestartet. Existiert schon ein gestarteter Container, wird dieser gestoppt und das neue Abbild gestartet.

Abbildung 3.5: Vollständige Build-Pipeline

4 Fazit

Im ersten Kapitel wurde gezeigt, welche Bedeutung der Begriff „Microservices“ und die damit einhergehende Architektur besitzt. Dabei hat sich herausgestellt, dass keine einheitliche Definition existiert, sondern es laut Fowler und Lewis viel mehr eine Menge von Charakteristika gibt, die diesen Begriff illustrieren. Dazu gehören Services als Komponenten, Organisation um Geschäftsfähigkeiten, Produkte statt Projekte, Intelligente Endpunkte - dumme Kommunikationswege, dezentralisierte Führung, dezentralisierte Datenverwaltung, Automatisierung der Infrastruktur, Fehlertoleranz und evolutionäres Design.

Auf dieser Basis aufbauend wurde die Idee von Continuous Integration/Continuous Delivery zur Automatisierung der Infrastruktur eingeführt. Mit diesen Techniken geht der Entwurf einer Build-Pipeline einher, für die es mehrere Alternativen gibt. Es wurde motiviert in einer Microservice Architektur für jeden Service ein eigenes Repository und eine eigene Build-Pipeline zu implementieren. Bei diesen Überlegungen hat sich schon ein wesentlicher Trade-off bei der Einführung der Microservices im Gegensatz zu einer monolithischen Anwendung gezeigt: Der Aufwand für den hinreichenden Wissensaufbau und die Automatisierung der Infrastruktur ist um ein vielfaches höher als bei der Entwicklung eines Monolithen.

Die sehr einfach gehaltene Beispiel-Pipeline für den HelloWorld-Service zeigt dabei wie sich Continuous Integration mit dem CI Server „Jenkins“ praktisch umsetzen lässt.

In Kapitel 3 hat sich gezeigt, welche Überlegungen für das Deployment der Microservices wichtig sind. Dabei wurde beleuchtet, welche Arten von Service-to-host-Mapping existieren, wie die Kommunikation zwischen den Microservices potenziell erfolgen kann und welche Schwierigkeiten dabei entstehen können.

Auch das Monitoring und die automatisierte Benachrichtigung bei Ausfällen oder Fehlern sind wichtige Bedenken bei der Einführung von Microservices. In Fehlerfällen hilft hier vor allem die klare Verantwortung der entwickelnden Teams für den jeweiligen Microservice, um zu einer schnellen Fehlerbehebung beizutragen.

Sind alle Bedenken und Probleme allerdings hinreichend eruiert, besteht die Möglichkeit für Unternehmen die „Time to Market“, also die Zeit, bis eine neue Funktionalität oder eine Fehlerbehebung deployed wird zu verringern. Bekanntermaßen wird von Amazon alle 11,6 Sekunden eine neue Änderung in das Produktivsystem übernommen.

Diese Entwicklungsgeschwindigkeit zeigt die Vorteile einer Microservice-Architektur. Der hohe Automatisierungsgrad ermöglicht die schnelle Reaktion auf Ausfälle oder Fehler und erhöht somit zum einen die Robustheit des Systems, zum anderen die Qualität. Diese wird durch die, für einen solchen Aufbau elementaren, Tests weiter verbessert.

Ein weiterer positiver Aspekt besteht in der Skalierung von Mitarbeitern. Während unvorhergesehene Abhängigkeiten die Starrheit einer monolithischen Anwendung erhöht und somit die Einarbeitung von Änderungen selbst mit mehreren Mitarbeitern verlängert, fördert die klare Trennung der Teams bei Microservices die Fähigkeit mehrere Änderungen parallel einzuarbeiten.

4.1 Ausblick

Das Ziel dieser Arbeit war, einen generellen Überblick über die Techniken und Probleme bei dem Build- und Deployment von Microservices zu verschaffen. Die Einführung einer solchen Architektur in einem konkreten Unternehmen erfordert hingegen noch eine genauere Untersuchung hinsichtlich der Anforderungen an die Infrastruktur, Sicherheitsaspekten und der möglichen Migration von einer monolithischen Architektur zu einer Microservice Architektur. Dies lässt sich allerdings nur von Architekten tätigen, die mit den vorhandenen Technologien und vor allem mit dem eigentlichen Geschäft des entsprechenden Unternehmens vertraut sind.

Insbesondere in kleineren Unternehmen stellt sich die Frage, ob der Overhead der Einführung auf Dauer tatsächlich einen Produktivitäts- und Qualitätsgewinn ergibt. Die gezeigten Techniken bezüglich des Aufbaus einer Build-Pipeline lassen sich schließlich auch auf eine monolithische Anwendung übertragen. Wird keine ständige Aktualisierung der Anwendung benötigt, steht die Microservice Architektur in Frage.

In diesen Fällen eigenen sich möglicherweise andere Architekturstile, wie beispielsweise die Schichtenarchitektur, Client-Server Architektur oder die Enterprise Service Bus Architektur. Die Art des zu vertreibenden Produktes muss hier über den Einsatz dieser verschiedenen Architekturen entscheiden.

Literaturverzeichnis

[FF06] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 2006.

[LF14] James Lewis and Martin Fowler. Microservices, 2014.

[Mar03] Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[New15] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 2015.

[Nyg07] Michael Nygard. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.

[Wol15] Eberhard Wolff. *Microservices*. dpunkt.verlag GmbH, 2015.