# Projektbericht Cloud Computing

### Motivation

Die IT Branche ist im ständigen Wandel und regelmäßig setzen sich neue Lösungsansätze durch. In den letzten Jahren sind Microservice Architekturen, Automatisierungspipelines, Containerisierung und Orchestrierung Begriffe die überall auftauchen. Oftmals dauert es jedoch bis die neuen Konzepte in den Unternehmen so umgesetzt werden wie sie gedacht sind. Dies mussten wir auch im Betreuten Praktischen Semester (BPS) schon erfahren.

Monolithische Programme sollen immer öfter durch Microservice Architekturen abgelöst werden. Jedoch gibt es häufig noch Abhängigkeiten zwischen den Services die so nicht bestehen sollten. Auch bei CiCd Pipelines gehen oftmals nicht über die Tests hinaus.

Aber insbesondere bei der Containerisierung und der damit verbundenen Orchestrierung fehlt meistens noch viel Expertise. So wurden zum Beispiel in Projekten während des BPS in Memory Datenbanken verwendet anstatt Docker Container zu verwenden für die Datenbank.

Neben dem nicht Einsatz der Technologien ist aber auch der falsche Einsatz häufig ein Problem. Wie zum Beispiel der Einsatz von Kubernetes weil es im Trend ist obwohl die Skalierung für die Anwendung überhaupt kein Faktor ist, weil die Nachfrage viel zu klein ist. In solchen Fällen werden dann manchmal einzelne Pods in Kubernetes Deployed und der Nutzen geht komplett verloren.

Es ist somit auch nicht sonderlich überraschend, dass DevOps Spezialisten aktuell extrem gefragt sind und die Position zu den Bestbezahltesten in der Industrie gehört.

Deshalb ist unsere Motivation und unser Ziel mit diesem Projekt einen breiten Einblick in alle Schritte von Microservices über CiCd Pipeline bis zum Deployment mit Containern und Kubernetes zu bekommen.

### Definition Problemstellung

Microservice Architekturen sind sehr gefragt heute auch wenn diese nicht immer Sinnvoll sind oder richtig umgesetzt werden. Innerhalb des Projektes soll deshalb eine Microservice Architektur erstellt werden in denen jeder Service komplett unabhängig funktioniert. Die Pipeline soll vom Testing bis zum Deployment alle Aufgaben voll automatisiert durchführen. Durch das anschließende Deployment mit Kubernetes soll eine Skalierung möglich sein.

### Lösungsansatz

Für das Projekt soll eine einfache Microservice Architektur umgesetzt werden. Da die Funktionalität der Services nicht im Vordergrund steht in diesem Projekt werden diese sehr einfach gehalten. Die Services werden in Go implementiert. Man kann gut Argumentieren das Go die beste Sprache für Microservices ist.

Für die Datenbanken wird MongoDB verwendet. Diese werden mit MongoDB Atlas deployed.

Jeder Service soll eine vollständige CiCd Pipeline besitzen. Hierfür sollen im ersten Schritt alle Tests erfolgreich durchlaufen und die Code Coverage mit einem Threshhold von 75% überschritten werden.

Im zweiten Schritt wird von jedem Service ein Dockerimage erzeugt und dieses wird auf Dockerhub gepusht.

Mit diesen Images kann dann das Deployment mit Kubernetes erfolgen.

######################

### Implementierung inklusive Schwierigkeiten

In dem Projekt wurden 2 einfache Services implementiert. Ein User Service und ein Produkt Service. Diese sollen simple Funktionalitäten bieten die jeder Online Shop braucht.

Der User Service kann User in der Datenbank speichern, abfragen und löschen.

Der Product Service kann Produkte in der Datenbank speichern, abfragen und löschen.

Die Probleme bei der Umsetzung kamen in erster Linie aufgrund der Verwendung der Programmiersprache Go. Da wir noch nicht mit der Programmiersprache vertraut waren mussten wir uns erst in die Sprache und die Tools einarbeiten. Go ist eine Kompilierte, statisch typisierte Sprache die im Auftrag von Google entwickelt wurde. Die Programmiersprache ist recht einfach im Grundsatz und besitzt nur 25 Keywords. In Go gibt es keine Klassen oder Vererbung, lediglich Structs denen Methoden zugeordnet werden können. Durch Interfaces lässt sich mit Composition arbeiten. Sie wurde 2009 veröffentlicht und versucht viele Probleme zu lösen die bei älteren Sprachen die typisch für Services eingesetzt werden, wie zum Beispiel Java, vorkommen. Während man bei der Service Entwicklung in Java auf Frameworks wie SpringBoot oder Quarkus zurückgreift und dort einen hohen Overhead hat, sind alle notwendigen Funktionalitäten bereits nativ in Go vorhanden. Dennoch haben wir uns dafür entschieden mit der Gin Library zu arbeiten. Dadurch spart man sich etwas Komplexität auf Kosten der Laufzeit.

Ein Problem das viel Zeit in Anspruch genommen hat, war die Methode zur Serialisierung und Deserialisierung von JSON und BSON. Go besitzt Pointer jedoch keine Pointer Arithmetik. Bei der Deklaration von Methoden kann man diese für Pointer oder Values des Objektes definieren. Definiert man eine Methode für einen Objekt Pointer so ist diese aber auch lediglich bei Pointern bekannt und kann von Object Values nicht aufgerufen werden. Wir haben versucht diese Methode über ein Object Value zu nutzen und diesen Fehler lange nicht gefunden.

Die Implementierung der CiCd Pipeline wurde mit GitHub Actions umgesetzt. Hier sind einige Schwierigkeiten aufgetreten. In der ersten Version wurde innerhalb der Pipeline händisch ein Docker Container für die Datenbank aufgesetzt. Die Datenbank wird für die Tests der Services natürlich benötigt. Während dieses Vorgehen funktioniert hat, so war es kein sehr schöner Lösungsansatz.

GitHub Actions bieten die Möglichkeit sogenannte Services zu verwenden. Diese können zum Beispiel für Datenbanken genutzt werden. Man gibt einfach das Image der gewünschten Datenbank und die benötigten Variablen wie Username und Passwort an und GitHub startet einen Datenbank Container. Die Variablen können beliebig belegt werden, da der Container nur während der Tests existiert. Selbstverständlich werden die Variablen bei den Tests nochmal benötigt und sollten deshalb global gespeichert werden. Beim nutzen des Services muss man darauf achten das man einen Timeout einbaut. Dieser sorgt dafür das genug Zeit zum Aufsetzen der Datenbank da ist bevor die Tests beginnen.

Bei der Durchführung der Tests kam es lange zu erheblichen Problemen. Die Tests werden in unserm Fall in Ubuntu durchgeführt. Zur Durchführung wird das Go native Kommando „go test“ verwendet. Im ersten Ansatz haben wir das Kommando „sudo go test“ verwendet. Mit diesem konnten wir jedoch die Tests nicht durchführen, da die Umgebungsvariablen nicht bekannt waren. Dieser Fehler hat extrem viel Zeit gekostet. Prof. Speiser hatte die Idee das es am sudo liegen könnte. Tatsächlich hat sich herausgestellt das bei Verwendung von sudo andere Umgebungsvariablen gesetzt sind. Dies macht rückblickend auch Sinn war uns jedoch nicht bewusst.

Für die Codecoverage verwenden wir kein third party tool sondern ebenfalls die Toolchain von Go. Mit Hilfe eines Bash Skriptes wird aus dem Codecoverage Report der von Go durch setzen der entsprechenden Flag erzeugt wird, die Codecoverage heraus geparsed. Anschließend wird diese mit einem Wert X abgeglichen der wie gewünscht gesetzt werden kann. Wir haben unseren Threshold auf 75% gesetzt.

Wenn alle Tests und die Codecoverage erfolgreich waren, wird ein Docker Image erzeugt und auf Dockerhub gepusht. Hierfür verwenden wir die offiziellen GitHub Action Templates die von Docker bereit gestellt werden. Es kam dabei zu keinen Problemen.

In einem späteren Schritt haben wir diesen Prozess weiter ausgebaut. Nun wird in Abhängigkeit des Branches und in Abhängigkeit der Action ein das Image erzeugt und gepusht. Bei jedem erfolgreichen Push wird ein Image erzeugt und mit einer Image number hochgeladen. Die Nummer zählt automatisch hoch. Hierfür nutzen wir die von GitHub bereitgestellte Variable „github.run\_number“. Diese wird inkrementiert für jeden Pipeline Run.

Falls ein push auf den Main Branch stattfindet so wird zusätzlich noch das Latest Image erstellt und gepusht. Dadurch haben wir immer die History der alten Images aber auch das neue Main Image als Latest. Dieselbe Logik könnte man direkt erweitern für Test oder Staging branches. Hier bietet einem GitHub Actions sehr gute Optionen an.

Unsere zwei Services befinden sich in getrennten Repos. Da in einer Microservice Architektur ein entscheidender Punkt ist das jeder Service unabhängig von den anderen Funktioniert sollten die Repos getrennt sein.

Jedoch verwenden beide Services die selbe Pipeline. Dadurch hatten wir in der ersten Implementierung viel duplizierte Configuration innerhalb der yaml Files. Da die Config Files jedoch wie Code behandelt werden sollten greift auch hier das Prinzip „Don’t Repeat Yourself (DRY)“. Insbesondere wenn man von einem wirklichen Produktivsystem ausgeht mit vielen Services wäre es sehr ineffizient mehrere identische Config Files in verschiedenen Repos zu haben.

GitHub Actions bietet hier die Möglichkeit Templates zu verwenden. Templates erfüllen den selben Zweck wie Funktionen beim Programmieren. Ich exportiere die Config an einen Ort und kann diese immer wieder verwenden. Diese Templates haben wir auch bereits benutzt als wir die Docker Actions verwendet haben.

Für unser Template haben wir ein neues Repo erzeugt. Das Template unterscheidet sich eigentlich nur in einem Punkt von der Config die wir davor hatten. Am Anfang des Templates muss man explizit die benötigten Variablen deklarieren. Es können Inputs und Secrets deklariert werden. Der Unterschied besteht nur darin das Secrets geheim sind, da hier die Secrets benutzt werden sollten die im Repo definiert sind welches das Template nutzt.

Nach der Deklaration haben wir lediglich den Code der zuvor in beiden Service Repos war in das Template eingefügt. Nun kann in den Workflows der Service Repos mithilfe der URL und durch Übergabe der Inputs und Secrets die Action ausgeführt werden.

### Bewertung

##################

### Fazit

###############

## Anhang

### Architekturschaubild

###############

### Source Code

[https://GitHub.com/JulianTeschner/cc\_product\_service](https://github.com/JulianTeschner/cc_product_service)

[https://GitHub.com/JulianTeschner/cc\_user\_service](https://github.com/JulianTeschner/cc_user_service)

<https://GitHub.com/JulianTeschner/go_test_build_action>

## Individuell Beiträge

Für das Umsetzung haben wir viel im Pair-programming gearbeitet. Dennoch hatte jeder ein Gebiet für das er die Hauptverantwortung übernommen hat.

### Beitrag Simon

###############

### Beitrag Julian

Ich habe mich primär auf die Umsetzung der Services in Go und die Pipeline konzentriert. Ich hatte schon länger das Interesse Go zu lernen und in diesem Umfeld war es sowohl passend von der Technologie als auch ein angemessener Rahmen für ein erstes Projekt in einer neuen Sprache.

Auch die Pipeline stand in meiner Verantwortung. Da wir auf der Arbeit eine sehr unzufriedenstellende Pipeline haben, hatte ich großes Interesse hier besser zu werden.