# 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 CI/CD-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 für die Datenbank Docker-Container zu verwenden.

Neben dem fehlenden 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 deployt 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 CI/CD-Pipelines, bis zum Deployment mit Containern und Kubernetes zu bekommen.

### Definition Problemstellung

Microservice Architekturen sind heute sehr gefragt, 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. Diese sind dadurch später leichter individuell aktualisierbar und skalierbar. Dies ist vor allem für Webanwendungen und generell Anwendungen, die aus dem Internet zugreifbar sind sehr wichtig. Um Anwendungen letztendlich lauffähig zu machen, sind sehr viele Schritte vom Bauen der Anwendung, über das Testen, bis zum Deployment nötig. Wenn all diese Schritte manuell durchgeführt würden müssen, kommt es häufig zu sehr vielen Fehlern, die nicht reproduzierbar sind. Um diese Probleme zu vermeiden, soll eine Pipeline alle diese Aufgaben bis zum Deployment voll automatisiert durchführen. Durch das anschließende Deployment mit Kubernetes soll eine Skalierung möglich sein. Bei zugreifbaren Services ist diese Funktion fast unabdingbar, denn der Verkehr im Internet kann in kurzer Zeit sehr stark zu- oder abnehmen. Zusätzlich werden die entsprechenden Container von Kubernetes angesprochen, die aktuell nicht überlastet sind und so verteilt sich der Datenverkehr möglichst gut über alle Container.

### Lösungsansatz

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

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

Jeder Service soll eine vollständige CI/CD-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 auf Dockerhub gepusht.

Mit diesen Images kann dann das Deployment mit Kubernetes erfolgen. Falls dieser Ansatz nicht klappt, werden beide Microservices in der Cloud als Einzelanwendungen auf Heroku deployt.

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

### Implementierung inklusive Schwierigkeiten

In dem Projekt wurden 2 einfache Services implementiert. Ein User Service und ein Product 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 CI/CD-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, dass man einen Timeout einbaut. Dieser sorgt dafür, dass 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, dass es am sudo liegen könnte. Tatsächlich hat sich herausgestellt, dass bei Verwendung von sudo andere Umgebungsvariablen gesetzt sind. Dies ergibt 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 bereitgestellt 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 für jeden Pipeline Run inkrementiert.

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 es in einer Microservice Architektur ein entscheidender Punkt ist, dass jeder Service unabhängig von den anderen funktioniert, sollten die Repos getrennt sein.

Jedoch verwenden beide Services dieselbe 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 denselben Zweck, wie Funktionen beim Programmieren. Man exportiert 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, dass 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.

Die Skalierung der Microservice-Architektur erfolgte mit Kubernetes. Dazu wurde ein Replicaset angegeben, das insgesamt 4 verschiedene Pods für einen Service erstellen soll. Alle Konfigurationen für Kubernetes erfolgen durch sogenannte Yaml-Dateien. Der Container wird vom Docker-Hub gepullt und auf einem speziellen Port innerhalb des Pods gestartet. Zusätzlich wurde ein Service erstellt, der die Anfragen an die entsprechenden Pods weiterleitet. Diese Services brauchen für die beiden Microservices unterschiedliche Ports für den Knoten, auf dem sie laufen, um sich nicht in die Quere zu kommen. Sie laufen allerdings beide auf demselben Knoten, standardmäßig ist der Name „docker-desktop“. Es muss ein ansprechbarer Port für die Services angegeben werden und ein Port für die verschiedenen Ports, diese haben dann alle denselben und der Service entscheidet, an welchen Pod die Anfrage geleitet wird. Die Labels von Service und Replicaset müssen übereinstimmen, damit eine Verbindung durch Kubernetes hergestellt werden kann. Beide Microservices laufen im default namespace. Um vertrauenswürdige Informationen sicher zu speichern, muss bei Kubernetes eine spezielle Secrets-Datei angelegt werden. Diese muss dasselbe Label angeben, wie die Pods, für die sie die Secrets speichern soll. Es muss ein Secret-Typ angegeben werden, dieser spezifiziert, wie die Secrets von Kubernetes behandelt werden. Anschließend werden die Secrets im Feld „Data“ angegeben. Die Secrets sind Base64-encodiert, damit sie nicht gelesen werden können. Das Replica-Set greift jetzt auf diese Secrets zu und lädt sie als Umgebungsvariablen in die Container. Es muss der Name der Umgebungsvariablen, die Metadaten der Secrets-Datei auf die zugegriffen wird und der entsprechende Schlüssel in der Secrets-Datei angegeben werden. Ein paar grundlegende Kubernetes-Kommandos bezüglich starten und löschen von Deployments, Log-Informationen vom Pod ausgeben lassen und einiges mehr musste man sich aneignen.

Nachdem die Pods, der Service und die Secrets von Kubernetes deployt wurden, kam es zu einem Error innerhalb der Pods. Die Pods konnten aus diesem Grund nicht starten und der Service hatte keine Endpoints, auf die er zugreifen konnte. Nachdem die Logs ausgegeben wurden, wurde klar, dass es sich um einen Fehler beim Datenbankzugriff handelte. Kubernetes konnte den Host nicht auflösen. Zahlreiche Versuche des Debuggings, wie interaktive Container starten und DNS-Einträge angeben, schlugen fehl. Die Datenbank (MongoDB-Atlas) ist von Go direkt und innerhalb eines Docker-Containers mit den entsprechenden Umgebungsvariablen erreichbar. Die Vermutung liegt nahe, dass wir nicht aus dem Kubernetes-Cluster rauskommen, also keine externe IP-Adresse anpingen können. Denn innerhalb eines interaktiven Containers war es auch nicht möglich eine externe IP-Adresse anzupingen. Dies könnte allerdings auch daran liegen, dass dieser interaktive Container keinen Service verwendet, der eine interne Adresse nach außen mapped. Unsere Pods im Replicaset verwenden dagegen einen Service, der die Kommunikation nach außen bereitstellt.

Schließlich haben wir die Microservices auf Heroku deployt. Anfangs gab es da noch kleine Bugs, in einem Microservice (Product-Service) konnten die Einträge, beziehungsweise Verzeichnisse in der anliegenden Datenbank nicht gefunden werden. Im anderen Service (User-Service) wurde eine Authentifizierung verlangt, obwohl wir keine angelegt hatten. Das Problem war die Dockerfile im obersten Verzeichnis in beiden Repositories. Dort wurde das falsche Image aus dem Docker-Hub gepullt. Nach der Anpassung auf die richtigen Images funktionierte die Ansprache der Services einwandfrei. Die Umgebungsvariablen können bei Heroku ähnlich wie die Secrets in GitHub-Actions angegeben werden.

Ein Ausblick in die weitere Vorgehensweise wäre die Angabe der Umgebungsvariablen bei GitHub statt diese direkt in einer Kubernetes-Datei anzugeben. Zusätzlich, wenn die Services in Kubernetes laufen würden, könnte man diese auf Okteto deployen.

Die Umgebungsvariablen werden in den Secrets bei GitHub gespeichert und können dann in der Pipeline mithilfe von einer vorgefertigten Action in Kubernetes geladen werden. Sie sind dann automatisch Base64-encoded, es muss dazu der Namespace, der Typ des Secrets, der Name des Secret-Kontexts, also die Metadaten, angegeben werden. Danach können die Daten von den Secrets in GitHub eingelesen werden.

Das Okteto-Deployment kann auch über eine Pipeline in GitHub laufen. Es muss dazu eine vorgefertigte Action in GitHub für Okteto aufgerufen werden. Jetzt kann der Namespace in Okteto angegeben werden und anschließend die Pipeline von Okteto getriggert werden. Es gibt auch weitere Möglichkeiten, wie ein Build auf Okteto, deployen von Stacks oder das Zerstören der Deployment-Pipeline, diese wären in diesem Fall aber nicht notwendig.

Es ist auch möglich direkt bei Okteto das GitHub-Repository zu hinterlegen. Die Namespaces und Kontexte müssen dann entsprechend auf der Seite über die grafische Oberfläche erstellt werden. Es wird dann ebenfalls deployt. Ein letzte Möglichkeit ist das Nutzen des Okteto Command Line Interface. Die CLI schaut nach entsprechenden Manifest-Dateien, die Namen wie „okteto.yml“ haben. Es gibt zusätzliche Befehle für Okteto, das Interface wurde bereits installiert, allerdings wurde noch kein Deployment ausprobiert, da Kubernetes an sich mit den Services nicht läuft. Namespaces und Kontexte müssen hier natürlich auch entsprechend vergeben werden. Okteto ist speziell für Kubernetes Deployments gedacht und bis zu einer gewissen Größe von 10 Pods in einem Cluster und 5GB Datenspeicher kostenlos nutzbar. Es kann auch nur 1 Cluster gleichzeitig kostenlos deployt werden.

### Bewertung

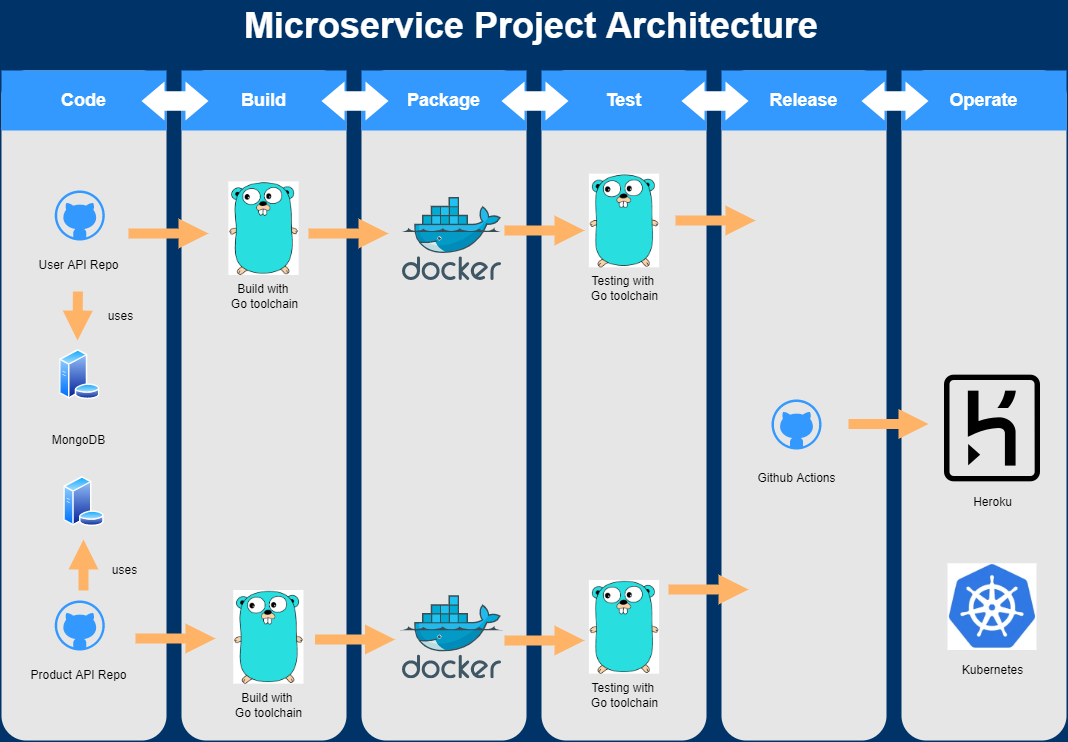
##################

### 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 die 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.