Architektur Prototyp einer Spring Cloud Applikation

Studienarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Herbstsemester 2018

Autoren: Moritz Habegger & Micha Schena
Betreuer: Mirko Stocker

Abstract

Aufgabenstellung

Ziel der Arbeit ist es, einen Prototypen einer Microservice-Architektur zu bauen, der die wesentlichen Aspekte von DevOps berücksichtigt und auf Spring Cloud basiert. Dies beinhaltet die Strukturierung des Projektes in Microservices anhand einer Domain-Driven-Design-Analyse (DDD-Analyse), das Einrichten einer CI/CD-Umgebung mit Jenkins und das automatische Aktualisieren der Live-Instanzen ohne Unterbrechung.

Vorgehen / Technologien

Als Erstes definierte das Team als Applikation ein Blog-System, da es sich dabei um eine überschaubare und verständliche Domäne handelt. Als Nächstes wurde eine DDD-Analyse gemacht, um sinnvolle Grenzen zwischen den Microservices zu finden. Um das Risiko von Zeitverzögerungen zu minimieren, wurden einzelne Prototypen entwickelt, die das Zusammenspiel von Google Cloud Platform und Spring zeigen. Des Weiteren wurde die CI/CD-Umgebung, welche Jenkins und SonarQube beinhaltet, auf der Google Cloud Platform aufgesetzt und eingerichtet. Im letzten Abschnitt der Arbeit wurden die Komponenten zu der geplanten Blog-Applikation zusammengeführt.

Ergebnis

Es wurde gezeigt, wie die Microservices auf der Google Cloud Platform in Docker-Images mit Kubernetes bereitgestellt und an Services wie PubSub, Cloud Datastore, Memcache oder Cloud SQL angebunden werden können. Mit der DDD-Analyse wurde gezeigt, wie einzelne Kontexte, beispielsweise die Statistik-Funktion von der Haupt-Domäne (Blog-Funktion), getrennt und über einen Message-Bus (PubSub) asynchron aktualisiert werden können und somit Geschwindigkeit gewonnen werden kann. Mit der Umsetzung der CI/CD-Umgebung wurde ein Docker-in-Docker-Ansatz gezeigt, in dem Docker-Images in einem Docker-Container erstellt werden.

Management Summary

Ausgangslage:

Aktuell werden viele Webapplikationen als monolithische Applikationen implementiert. Das bedeutet, dass die Applikation ein untrennbares homogenes Gebilde aus funktionalen Elementen darstellt. Das führt dazu, dass es schwierig ist die Hochverfügbarkeit zu garantieren oder neue Änderungen einzuführen, denn wenn ein Teil ausfällt, ist die gesamte Applikation nicht verfügbar. Des Weiteren sind viele Testsysteme auf lokalen Servern aufgesetzt. Diese verursachen erhebliche Wartungskosten. Vom Wechsel lokaler Testsysteme zu solchen in der Cloud erhofft man sich Kosteneinsparungen bei der Wartung.

Vorgehen/Technologien:

Als Erstes definierte das Team als Applikation für den Prototypen ein Blog-System, da es sich dabei um eine überschaubare und verständliche Domäne handelt. Der Prototyp soll als Entscheidungsgrundlage für Software-Architekten dienen. Die evaluierten Technologien sind Docker und die Programmbibliothek Spring Cloud. Der Prototyp soll schlussendlich auf der Google Cloud Platform (GCP) laufen. Diese Arbeit soll zeigen, ob sich die neuen Technologien bewähren und als Basis für die Weiterentwicklung bestehender Spring Boot Systeme dienen können. Gleichzeitig wird getestet ob die Testumgebung in die Cloud ausgelagert werden kann, indem diese für den Prototypen auf GCP aufgesetzt wird.

Ergebnisse:

Der fertige Prototyp befindet sich im öffentlichen Repository[1].

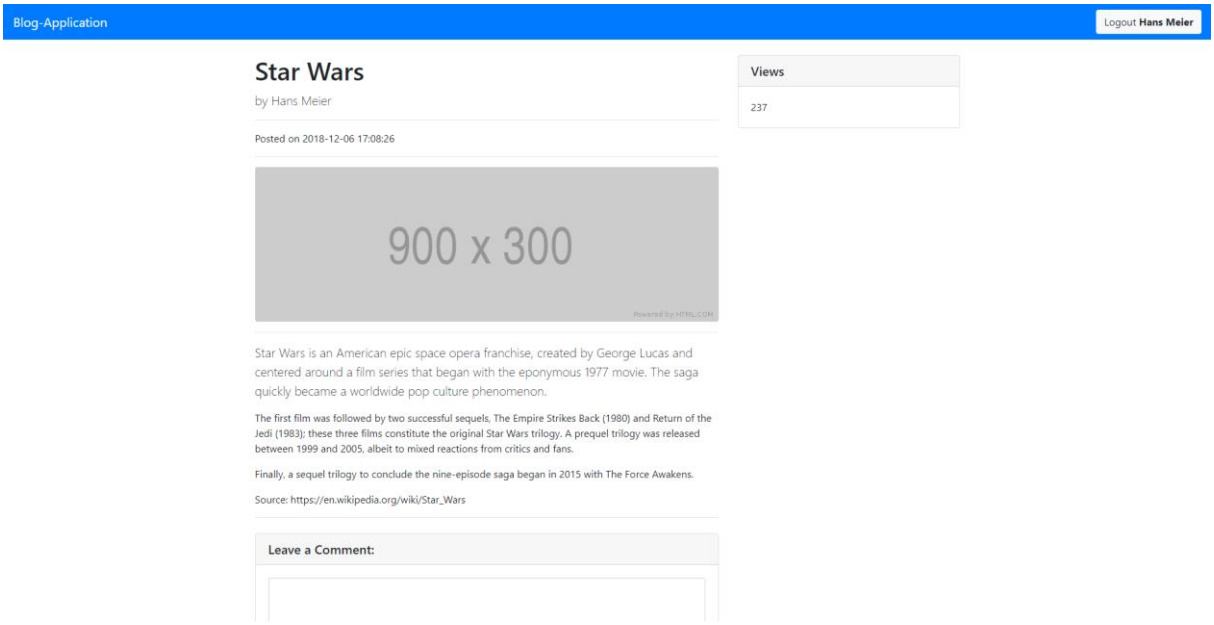

Abbildung 1 Blog Applikation

Der Prototyp konnte erfolgreich umgesetzt und in der Cloud bereitgestellt werden. Wie erhofft weist der Prototyp eine deutlich höhere Stabilität auf, da bei einem Ausfall einer Komponente der Rest der Applikation trotzdem verfügbar ist. Wenn zum Beispiel die Kommentar-Komponente ausfällt, wird der Artikel nach wie vor angezeigt, jedoch ohne die Kommentare. Bei der monolithischen Architektur würde der ganze Blog ausfallen, bis das Problem bei den Kommentaren behoben wurde.

[1] https://github.com/firegnome/spring-cloud-prototype-gcp

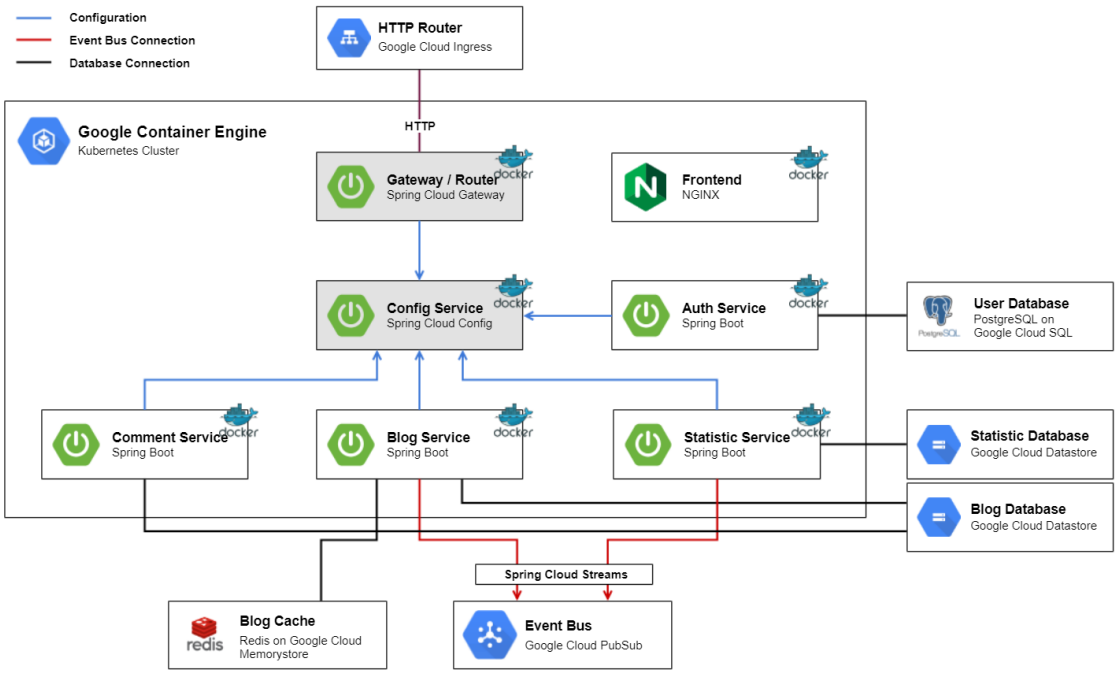

Abbildung 2 Blog Deployment

Der Zusatzaufwand, der eine solche Architektur verursacht, darf nicht vernachlässigt werden. Dennoch empfiehlt sich ein Umstieg auf eine verteilte Applikation, insbesondere, wenn das Entwicklungs-Team grösser wird und der Umfang der Applikation stark wächst. Der Aufwand für einen Umstieg auf diese Architektur wächst mit dem Umfang der Applikation, was einen frühen Umstieg nahelegt. Da Spring Cloud eine verwandte Technologie von Spring Boot ist, kann der Umbau in kleinen Schritten erfolgen, was das Risiko stark verringert.

Die Experimente mit dem Testsystem in der Cloud haben einige Hürden aufgezeigt, welche jedoch während der Umsetzung nachhaltig beseitigt werden konnten. Das bedeutet, dass mit der gesammelten Erfahrung einem Umstieg auf ein Testsystem in der Cloud nichts mehr im Weg steht.

Inhalt

Ausgangslage (Kontext)

Typ der Arbeit

Das Ziel dieser Arbeit ist es, ein Prototyp für ein Microservice-Deployment zu erstellen. Es soll festgestellt werden ob und wie gut das Zusammenspiel von Spring Cloud und Kubernetes unter Anbindung von Google Cloud Platform (GCP) spezifischer Services wie zum Beispiel PubSub, Cloud Datastore oder Cloud SQL funktioniert.

Fachliche Domäne

Diese Arbeit richtet sich an Software-Architekten, die sich für eine Microservice-Architektur auf der GCP mit Kubernetes interessieren.

Heutige Lösung

Heutzutage werden viele Web-Services als monolithische Applikationen programmiert, zum Beispiel mit Spring Boot. Aktuell gibt es aber einen Trend zu Microservice-Architekturen, da diese versprechen skalierbarer und einfacher wartbar zu sein. Diese Arbeit soll zeigen wie eine moderne Architektur mit Spring Cloud aussehen könnte.

Problembeschreibung (Stand der Technik)

Motivation

Anlass für diese Arbeit gab das Fehlen eines kompletten und durchgehenden Beispiels einer Spring Cloud Applikation. Die meisten aktuellen Applikationen sind als monolithische Applikationen aufgebaut. Eine solche Applikation besteht zum Beispiel aus einer einzigen Spring Boot Applikation, die die komplette Applikations-Logik beinhaltet. Ein aktueller Trend ist, dass neue Änderungen so oft wie möglich veröffentlicht werden um ein schnelles Feedback zu diesen zu erhalten. Je mehr Leute jedoch an einer solchen Applikation arbeiten, desto öfter wird die Applikation aktualisiert, was zu verschiedenen Konflikten führen kann. Vom neuen Ansatz erhofft man sich, dass Änderungen einfacher und mit weniger Risiko eingeführt werden können.

DDD-Analyse

Anhand des Buches Domain Driven Design kompakt [Vernon17] wurde eine Domain-Analyse für ein Blog-System erstellt. Das Team orientierte sich am ersten Kapitel des Buches *Strategisches Design mit Bounded Contexts.* Als erster Entwurf ergab sich folgendes:

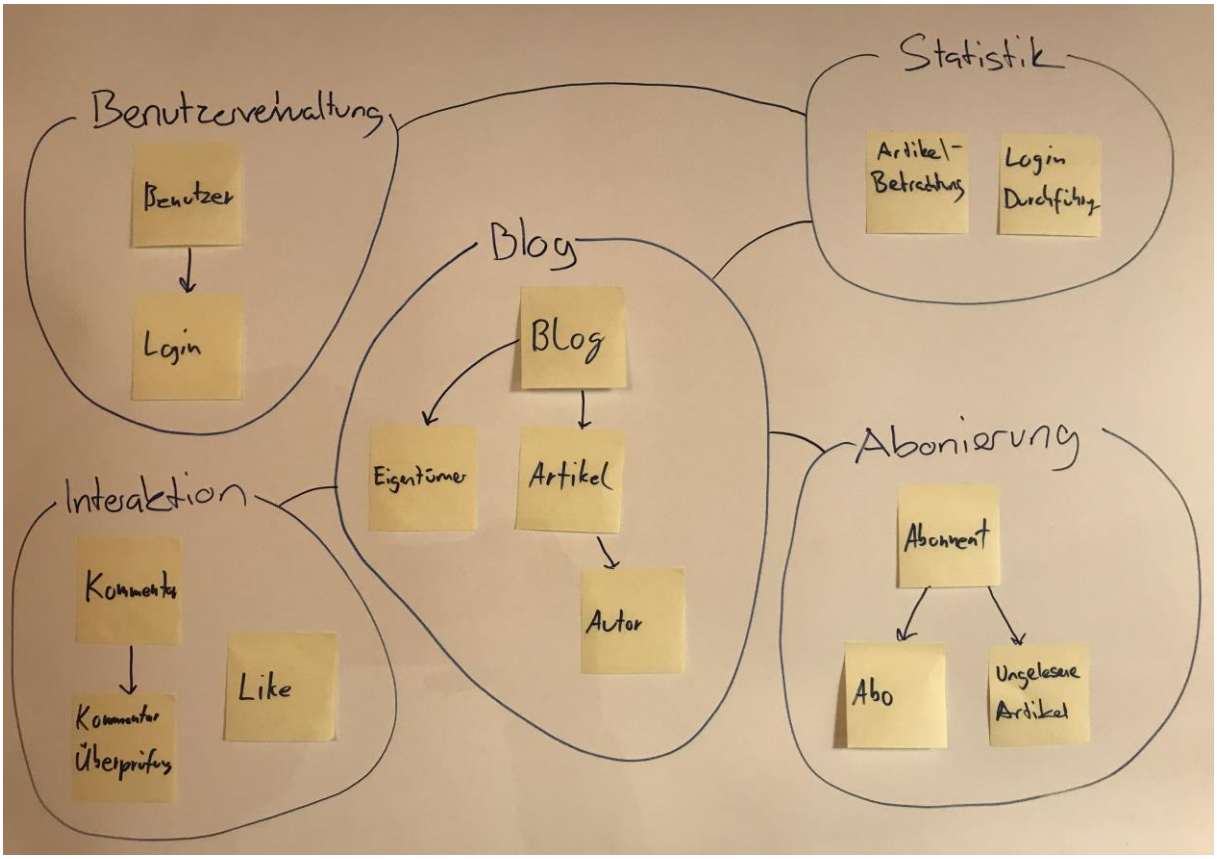

Abbildung 3 DDD-Analyse Entwurf

Domäne Blog

Die Core-Domäne Blog ist der Kern der Applikation. Sie ist in sich dafür verantwortlich, dass Blogs erstellt und bearbeitet werden können. In einem Blog können mehrere Artikel von Autoren verfasst werden. Der Eigentümer des Blogs kann Autoren das Erstellen von Artikel auf seinem Blog berechtigen.

Domäne Benutzerverwaltung

Die «Benutzerverwaltung» bildet als unterstützende Domäne das Login für das gesamte System ab. So kann zum Beispiel eine Änderung des Passwortes komplett innerhalb dieser Domäne ausgeführt werden, ohne eine andere zu beeinflussen.

Domäne Interaktion

Die unterstützende Domäne Interaktion ist für alle Benutzerinteraktionen auf dem Blog zuständig. Sie befasst sich ausschliesslich mit sozialen Aspekten wie Kommentar erstellen oder Like abgeben. Bei neuen Kommentaren auf einen Artikel muss dieser zuerst vom Autor überprüft und freigeschalten werden.

Da diese Domäne keine zeitkritischen Aktionen enthält, wurde sie aus der Core-Domäne entfernt und in eine eigene verschoben. Das hat den Vorteil, dass die Core-Domäne auch bei grosser Interaktions-Last weniger stark tangiert wird.

Domäne Abonnierung

Die unterstützende Domäne Abonnierung befasst sie sich mit dem Abonnieren von Blogs. Ein Leser kann sich Abonnenten für einen Blog erstellen. Beim Erstellen von neuen Artikeln auf dem Blog wird dieser als ungelesener Artikel beim Leser vermerkt.

Domäne Statistik

Die Statistik ist dafür zuständig alle Artikelbetrachtungen und Benutzerlogins nachzuvollziehen. Somit können Eigentümer und Autoren des Blogs zum Beispiel die Aufrufe, gemessen an Betrachtungen des jeweiligen Artikels, einsehen. Die Domäne reagiert auf die Events der Benutzerverwaltung bei einem Login eines Benutzers oder beim Aufruf eines Artikels.

Events

Für die Events ist eine Message-Bus angedacht. Auf diesen Message-Bus können sich die Domänen Registrieren und erhalten dann die gewünschten Messages, auf die sie reagieren können.

Funktionale Anforderungen

Um die funktionalen Anforderungen zu erfassen, haben wir ein Story-Mapping gemacht. Ziel des Story Mapping ist es, die ganze Funktionalität als Geschichte (Story) chronologisch durchzudenken. Das hat den Vorteil, dass man an sehr viel denkt, sich aber trotzdem nicht im Detail verliert. Anschliessend wurden die Features auf verschiedene Versionen und somit nach ihrer Priorität gruppiert.

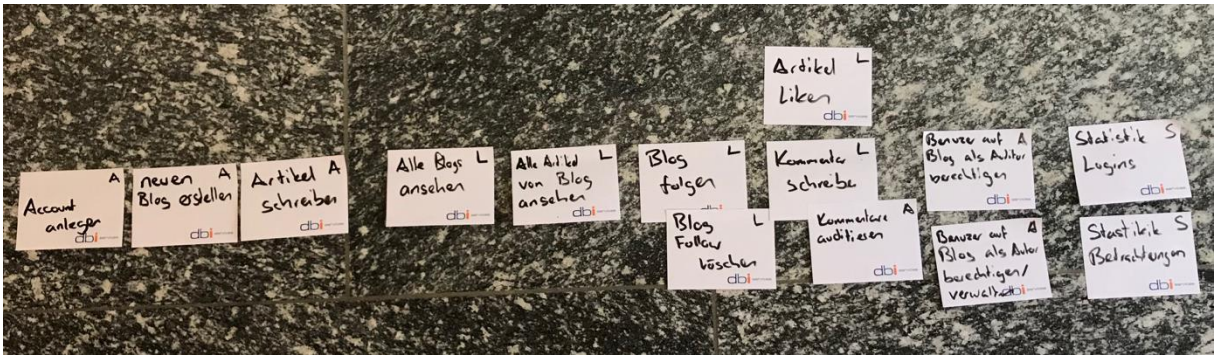

Abbildung 4 Stories nach Story-Mapping in chronologischer Reihenfolge

Es ist nicht das Ziel des Story-Mappings, alles möglichst genau zu dokumentieren, wie es funktionieren soll, sondern vielmehr, dass an alle Aspekte gedacht wird und ein gemeinsames Verständnis für die Stories entwickelt wird. Somit wurde versucht, dem Grundsatz aus dem Buch

User Story Mapping von Jeff Patton [Patton14] zu folgen: *Stories get their name from how they should be used, not what should be written.* Das Team entschied sich an dieser Stelle aktiv dagegen, die Stories im Detail zu beschreiben, sondern ergänzt sie in einem iterativen Prozess.

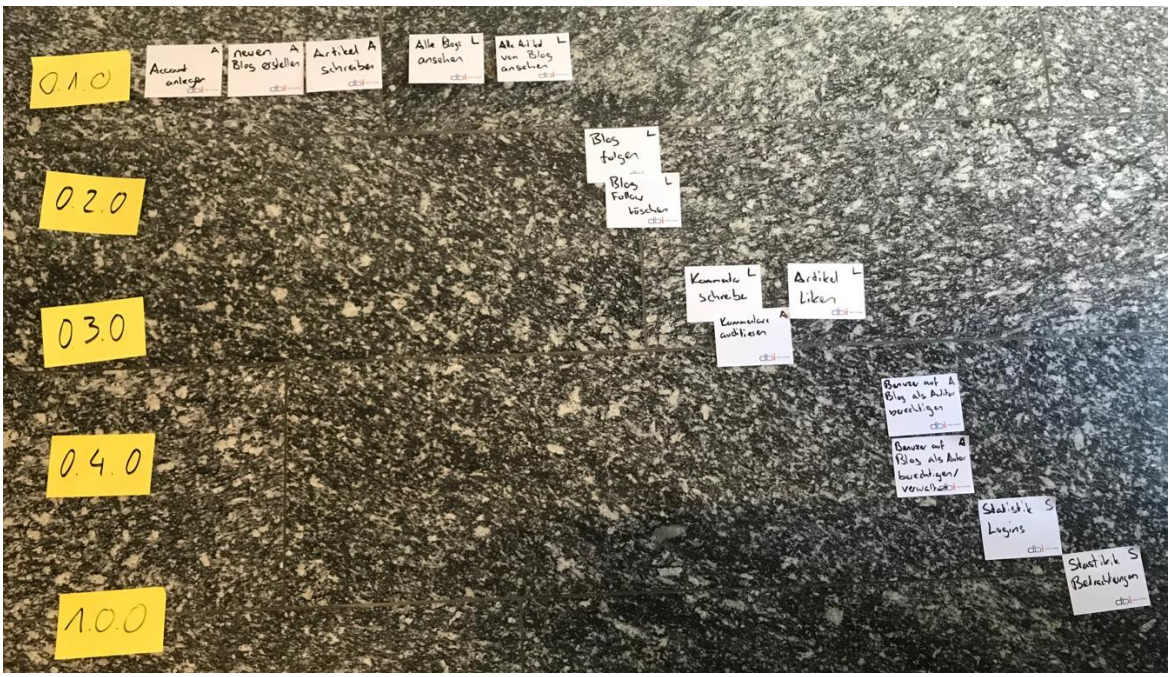

Abbildung 5 Stories gruppiert nach Versionen

Um die Stories in Zukunft weiter zu bearbeiten und verfeinern, wurden diese in GitLab erfasst.

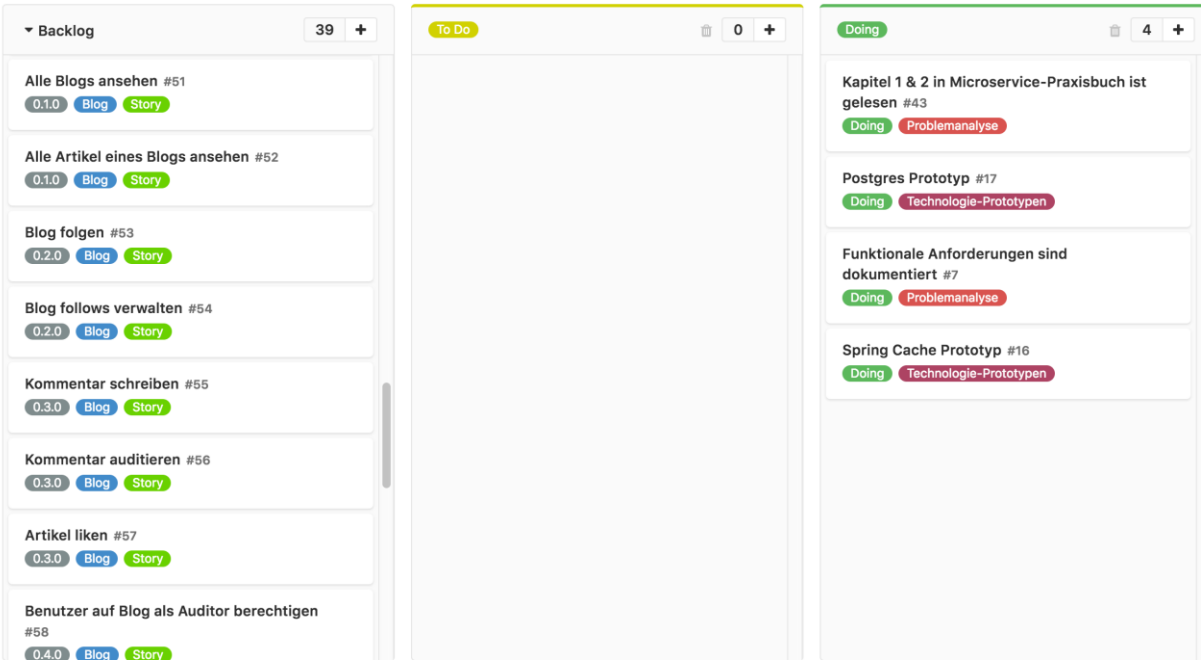

Abbildung 6 Stories auf GitLab-Board

Nicht Funktionale Anforderungen

Qualitätsattribut	Qualitätsmerkmal
Funktionalität	1. Die Applikation blockiert nach drei fehlerhaften Anmeldeversuchen die Anmeldung für 30 Sekunden.
Zuverlässigkeit	2. Das System kann in 99% von den Fällen 100‘000 Blog-Artikel pro Sekunde ausliefern. 3. Artikel werden erstellt, auch wenn der Abo Service nicht verfügbar ist. 4. Wenn der Abo Service nicht verfügbar war und Artikel erstellt wurden, führt er sie beim nächsten erfolgreichen Start nach. 5. Der Blog wird geladen, auch wenn der Kommentar-Service nicht verfügbar ist. 6. Der Blog wird geladen, auch wenn der Login-Service nicht verfügbar ist 7. Der Statistik-Service kommt automatisch wieder in einen konsistenten Zustand, auch wenn er für längere Zeit nicht verfügbar war.
Benutzbarkeit	8. Alle Input-Felder können über die Tabulatortaste erreicht werden. 9. Alle Input-Felder sind dem Inhalt entsprechend validiert. 10. Alle Input-Felder sind mit Labels versehen um von Screen-Readern gelesen werden zu können.
Effizienz	11. Neue Kommentare können in 99% der Fälle innerhalb 5 Sekunden ausgelesen werden. 12. Neue Likes können in 99% der Fälle innerhalb 5 Sekunden ausgelesen werden. 13. Neue Benutzerlogins können in 99% der Fälle innerhalb 5 Sekunden aus der Statistik ausgelesen werden. 14. Neue Blogs können in 99% der Fälle innerhalb 5 Sekunden aus der Statistik ausgelesen werden.
Änderbarkeit	15. Alle Unittests müssen erfolgreich durchlaufen, bevor der Code in den Master-Branch akzeptiert wird.
Übertragbarkeit	16. Die Applikation ist durch ein Installationsscript aufsetzbar. 17. Die gesamte Installation dauert nicht länger als 30 Minuten.

Tabelle 1 Qualitätsmerkmale

Das ausführende Referenzsystem ist die Google Cloud Platform. Für alle diese Qualitätsmerkmale wird zum Abschluss des Projektes ein Review vorliegen, die die Vollständigkeit der einzelnen Merkmale als erfüllt, beziehungsweise als nicht erfüllt, bewertet.

Lösungskonzept

Lösungsvarianten

Bei allen Lösungsvarianten wurde eine Microservice-Architektur mit Spring Cloud[2] als Basis genommen, da dies von der Aufgabenstellung gefordert wird. Spring Cloud ist eine Sammlung von Programmbibliotheken, welche es ermöglicht, Spring Boot Applikationen zu einer Microservice-Architektur zusammenzuführen. Die Aufteilung der Aufgaben in Microservices wurde anhand der DDD-Analyse gemacht. Beide Varianten zeigen eine asynchrone Microservice-Architektur mit einem Message-Bus, der die Services entkoppelt.

Microservices mit Anbindung an GCP-Services

Diese Variante zeigt ein Microservice-Deployment auf der Google Kubernetes Engine[3]. Speziell dabei ist, dass die jeweiligen Services direkt auf GCP-Services (Google Cloud Platform), wie beispielsweise Google Cloud Memorystore[4] (Redis), zugreifen. Die Microservices sind so gebaut, dass der Zugriff auf die GCP-Services über einen Abstraktions-Layer erfolgen, dessen Implementierung ausgetauscht werden kann. Das hat den Vorteil, dass nichts am Code selber geändert werden muss, um die Applikation auf einem anderen Cloud-Provider wie zum Beispiel AWS zu betreiben. Ein Beispiel für einen solchen Abstraktions-Layer ist Spring Cloud Streams[5], der den tatsächlichen Service, in diesem Falle Google PubSub[6], verbirgt. Das bedeutet, dass derselbe Code mit RabbitMQ[7] oder Apache Kafka[8] verwendet werden kann.

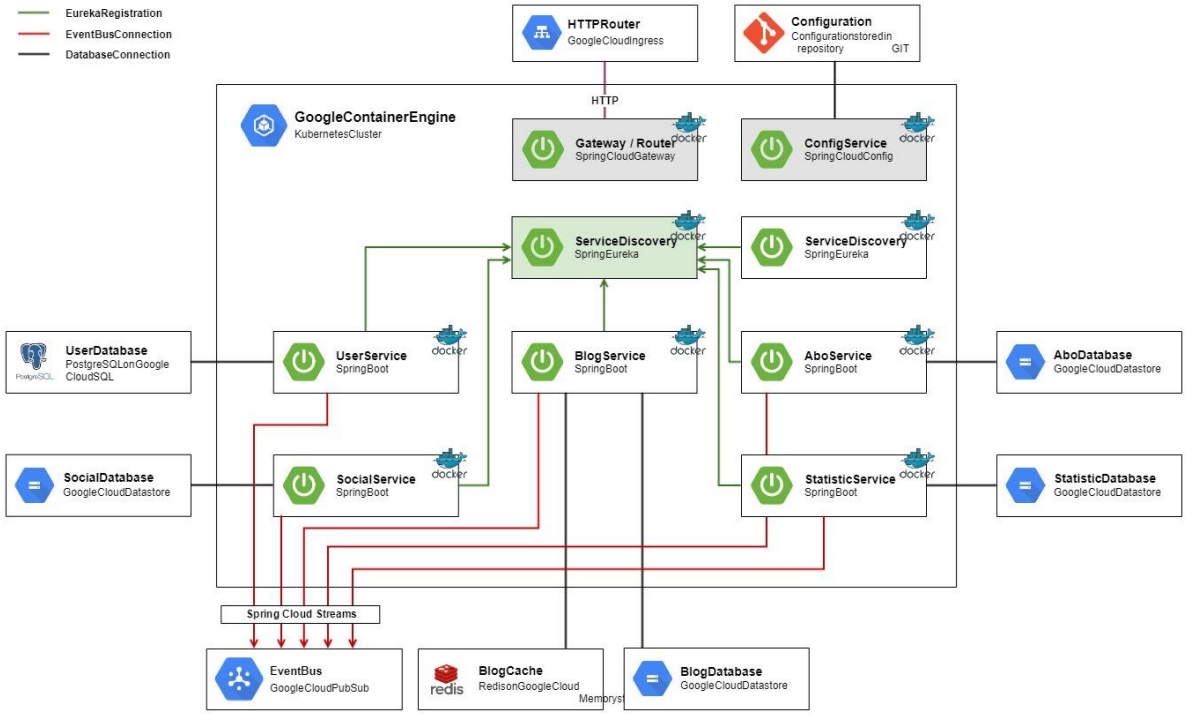

Abbildung 7 Lösungsvariante 1 - Anbindung an GCP-Services

[2] http://spring.io/projects/spring-cloud
[3] https://cloud.google.com/kubernetes-engine/
[4] https://cloud.google.com/memorystore/
[5] https://cloud.spring.io/spring-cloud-stream/
[6] https://github.com/spring-cloud/spring-cloud-gcp/tree/master/spring-cloud-gcp-pubsub-stream-binder
[7] https://github.com/spring-cloud/spring-cloud-stream-binder-rabbit
[8] https://github.com/spring-cloud/spring-cloud-stream-binder-kafka

Microservices mit allen Services auf Kubernetes

Bei dieser Variante werden alle benötigten Komponenten auf einen Kubernetes Cluster bereitgestellt. Das bedeutet, dass die Datenbank-Instanzen vom Betreiber des Clusters verwaltet werden und dieser für die Verfügbarkeit und Skalierung dieser verantwortlich ist. In dieser Variante wurde der Message Bus von der GCP (PubSub) verwendet. Dieser könnte ebenfalls noch in Form eines RabbitMQ oder Apache Kafka direkt auf dem Kuberentes Cluster realisiert werden.

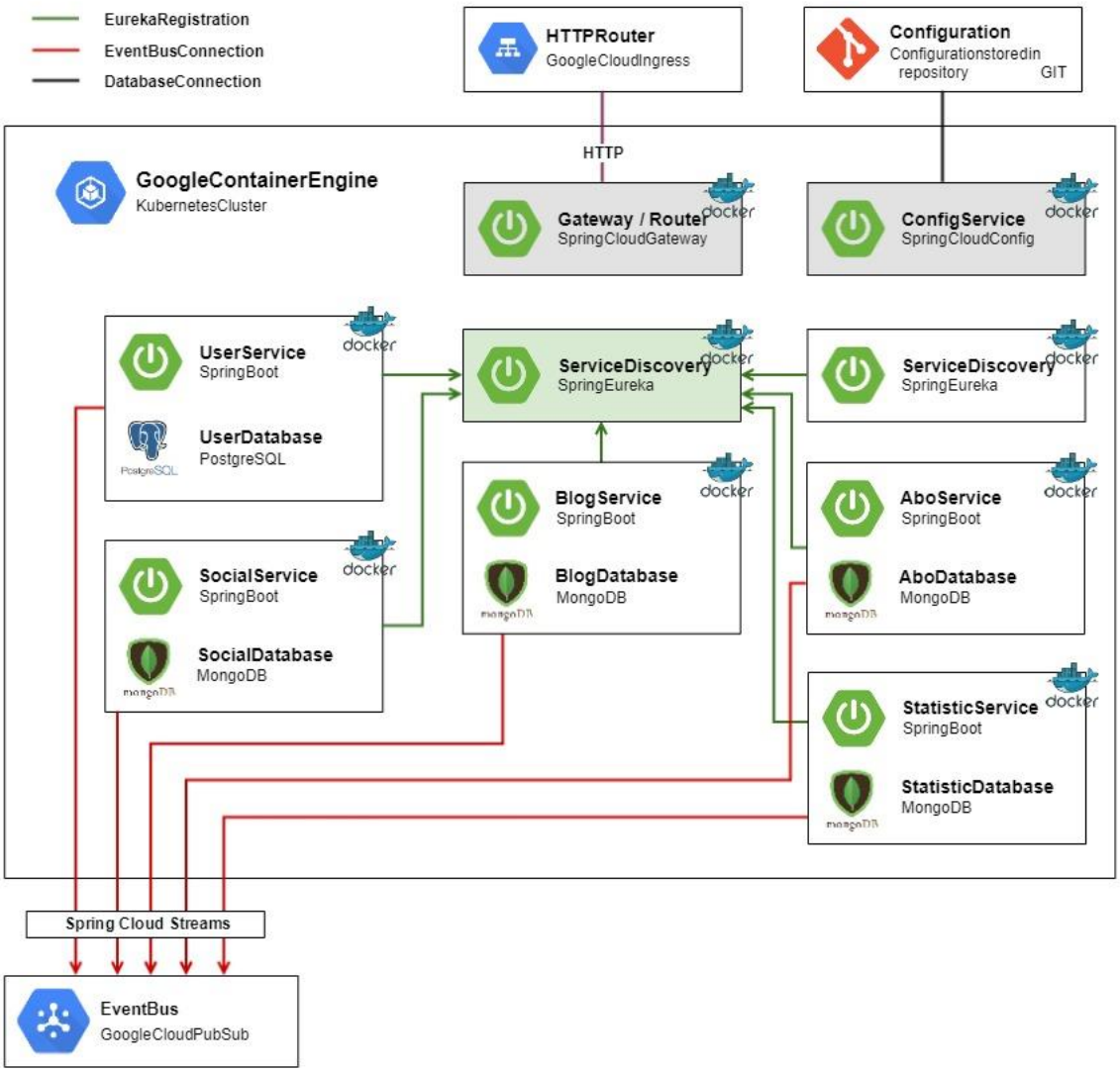

Abbildung 8 Lösungsvariante 2 - alles auf Kubernetes

Vergleich

	Vorteile	Nachteile
Lösungsvariante 1	• Durch GCP garantierte Verfügbarkeit der Datenbanken • Backups durch GCP erleichtert • Performante Konfiguration durch GCP • Einfache Erweiterung z.B. von Speicherplatz über GCP-Console. • Benötigt wenig Know-How im Bereich Betrieb von Datenbanken	• Aufwändige Konfiguration der Anbindung der Datenbanken / Services. • Cloud-Spezifische Konfiguration
Lösungsvariante 2	• Einfacher zu portieren, da alles nur von Kubernetes abhängig ist.	• Hochverfügbare Postgres Instanz ist sehr aufwändig zu warten. • Für jegliche Datenbanken müssen Docker-Volumes zur Verfügung gestellt werden, die korrekt gesichert werden müssen. • Sichere und performante Konfiguration der Datenbanken muss selber gemacht werden.

Tabelle 2 Lösungsvarianten

Entscheid

Die erste Variante wird, in Anbetracht der Aufgabenstellung, als die passendere Variante angesehen. Der grösste Nachteil dieser Variante, die aufwändige Konfiguration der Anbindungen, wird in dieser Studien-Arbeit grösstenteils eliminiert, da gute Konfigurations-Beispiele erarbeitet werden. Die Variante 1 birgt für den Erfolg der Studienarbeit aber auch einige Risiken, die vor allem in der Verfügbarkeit von Anbindungs-Libraries des Spring-Frameworks an die Infrastruktur von GCP liegen. Um diesem Risiko entgegenzuwirken wurde Zeit für Technologie-Prototypen eingeplant, die jeweils die Anbindung an einen wichtigen GCP-Service zeigen.

Adressierung von Qualitätsattributen

Die beschriebene Architektur erlaubt es die Verfügbarkeit der Applikation horizontal zu skalieren und adressiert somit die Effizienz und Zuverlässigkeit. Dieser Ansatz erlaubt es eine beliebige Anzahl von Microservices zu betreiben und kann somit in Zukunft auf verschiedene Szenarien reagieren. Die asynchrone Architektur garantiert, dass nicht die ganze Infrastruktur ausfallen kann. Microservices erlauben es auch, dass mehrere Teams gleichzeitig an der Applikation arbeiten und unabhängig voneinander Änderungen an verschiedene Microservices durchführen können.

Architektur

Die finale Architektur der Applikation ist in den README Dateien im öffentlichen Repository[9] dokumentiert.

[9] https://github.com/firegnome/spring-cloud-prototype-gcp

Design

User Interface Entwürfe

Das Frontend ist nicht die Hauptaufgabe dieser Arbeit, weswegen ihm keine hohe Priorität zugeordnet wurde. Deswegen wurde von KeenThemes das Metronic[10] Theme evaluiert. Dieses bietet bereits einige Beispiele, die in den Design Entwürfen verwenden werden. Schlussendlich sind dies nicht fertige Entwürfe, die so umgesetzt werden, sondern dienen eher zur Inspiration und Evaluation, wie es umgesetzt werden könnte.

Blogs Übersicht

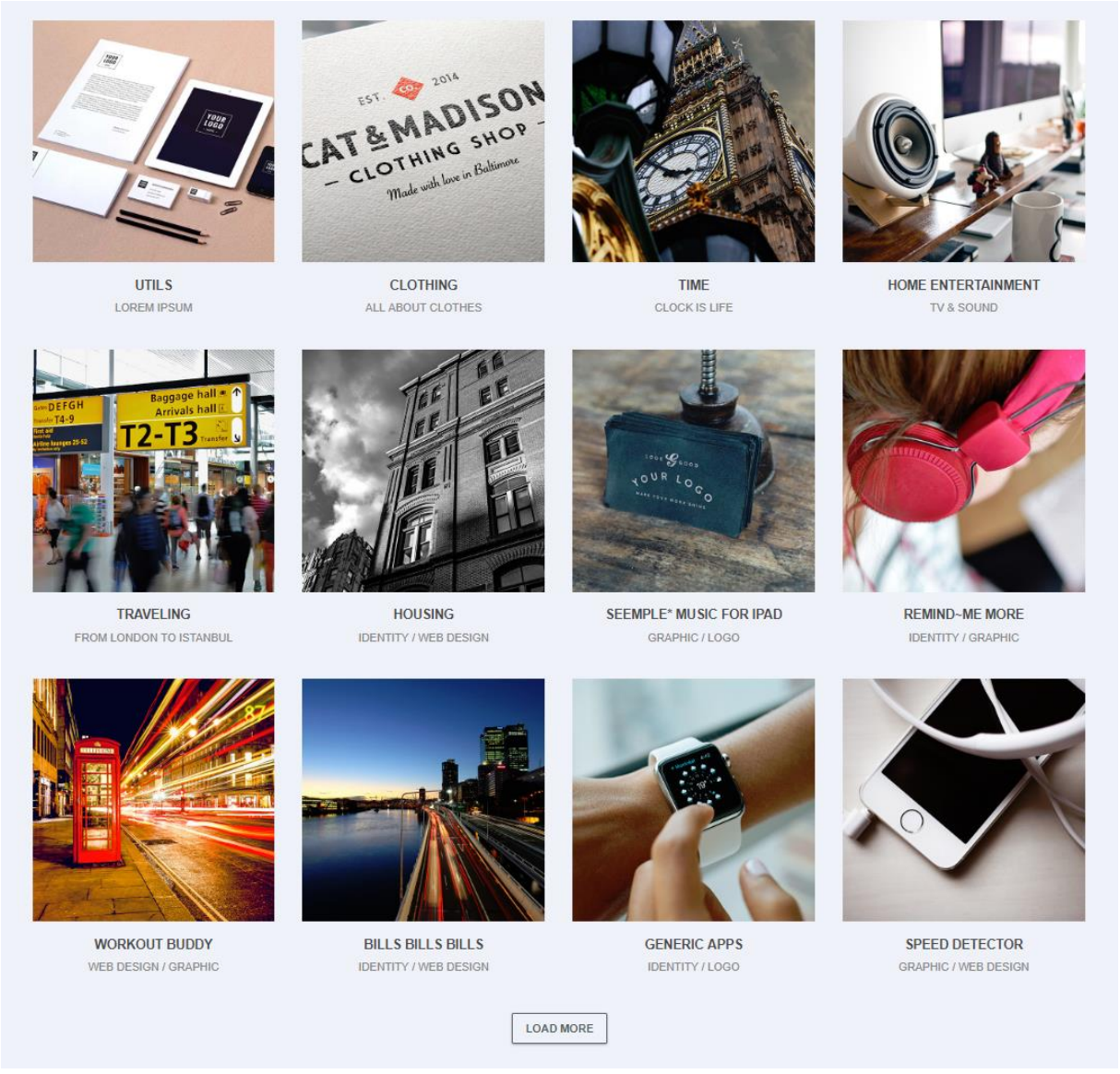

Abbildung 9 Blogs Übersicht[11]

Diese Variante bietet trotzdem eine gute Übersicht über die Blogs. Eine bereits sortierte Liste bei der API-Abfrage würde sich anbieten, um die beliebtesten Blogs anzuzeigen.

[10] Metronic KeenTheme, http://keenthemes.com/preview/metronic/
[11] Metronic von KeenTheme

Blog Posts Übersicht

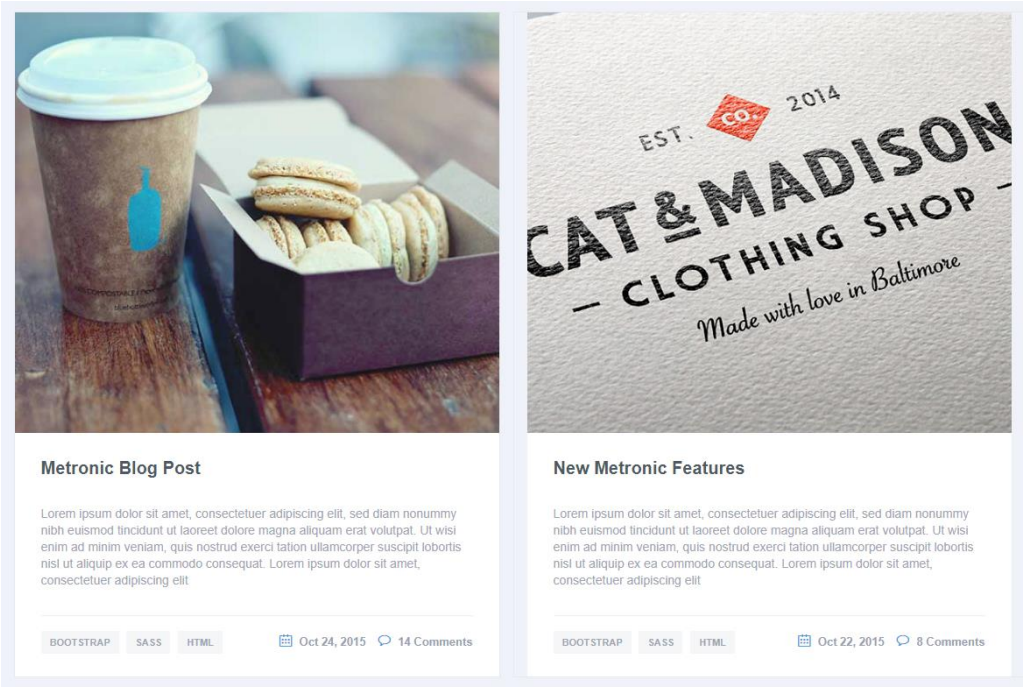

Abbildung 10 Blog Posts Übersicht[12]

Blog Comments

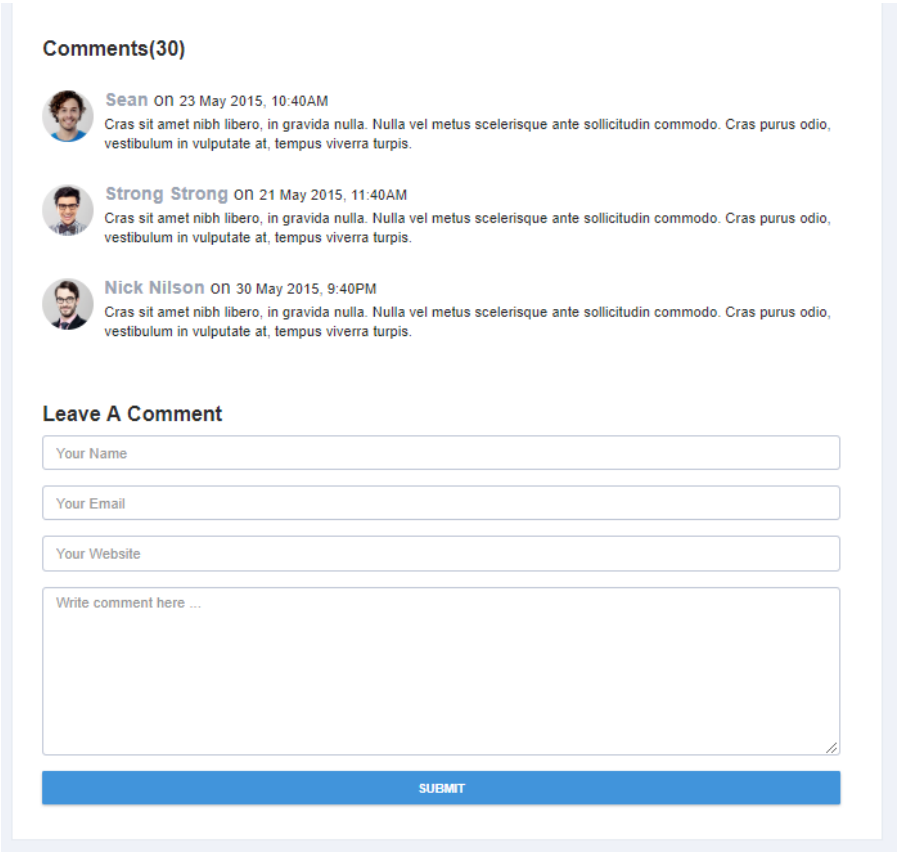

Abbildung 11 Blog Comments[13]

[12] Metronic von KeenThemes
[13] Metronic von KeenThemes

Blog Post

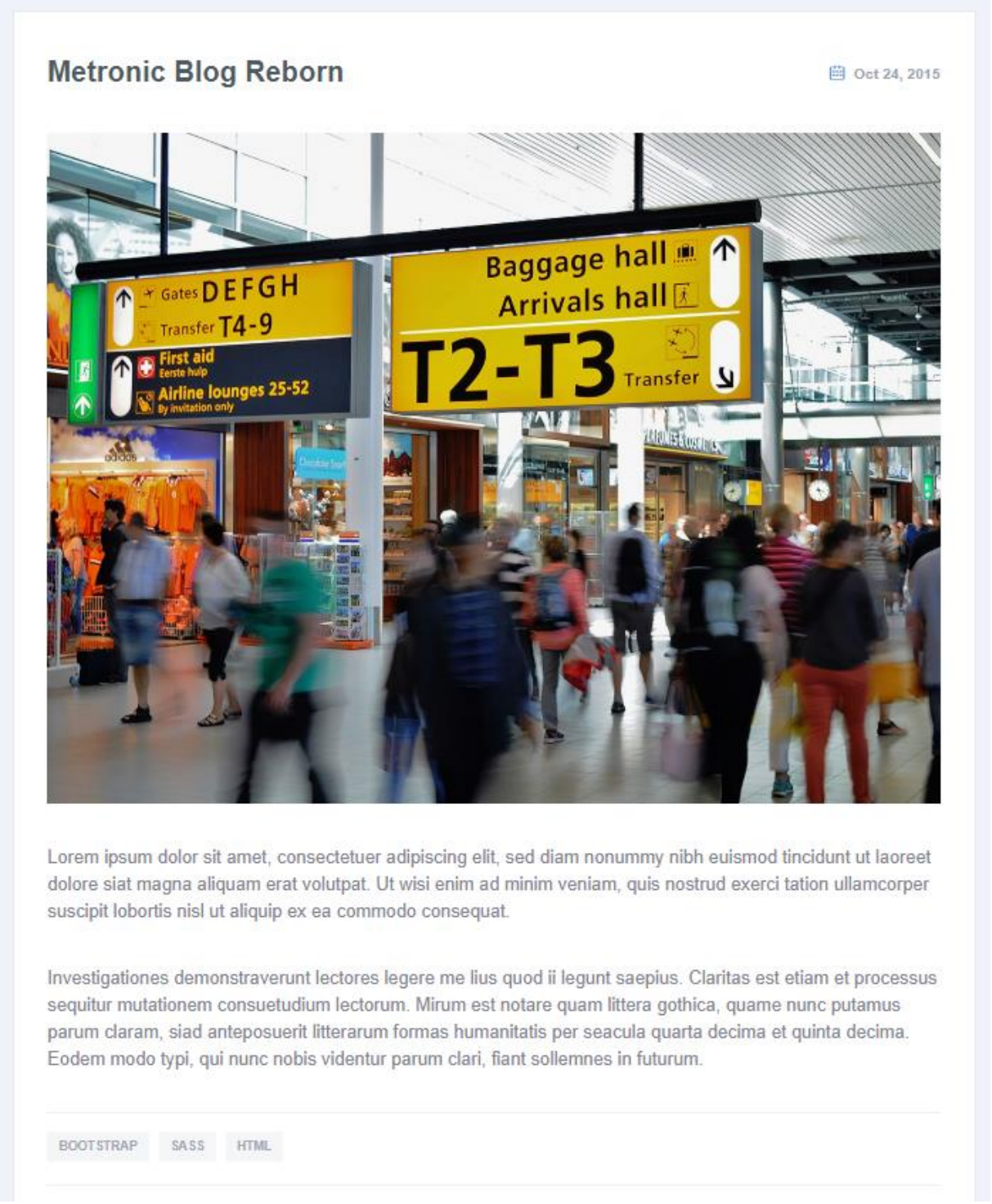

Abbildung 12 Blog Post[14]

[14] Metronic von KeenThemes

Umsetzung

Experimente

Die Architektur verlangte den Einsatz einer Vielzahl, dem Team meist unbekannten, Technologien, die angewendet oder konfiguriert werden mussten. Das stellte ein grosses Risiko für den Erfolg des Projektes dar, weshalb entschieden wurde, dass in der ersten Phase Experimente durchgeführt werden um Erfahrungen mit diesen Technologien zu sammeln.

Übersicht der Experimente

Experiment	**Ziel**	**Eingesetzte Technologien**
JPA PostgreSQL	Herauszufinden, wie eine Spring Applikation mit einer PostgreSQL-Datenbank verbunden werden kann.	• Spring Boot • JPA • PostgreSQL
Docker Getting Started	Herauszufinden, wie man ein Docker Image mit einer Spring Applikation erstellt.	• Docker • Spring Boot
Docker Compose Spring Boot JPA PostgreSQL	Herauszufinden, wie eine Spring Applikation mit einer PostgreSQL-Datenbank verbunden werden kann, wenn alles auf Docker läuft.	• Docker • Docker-Compose • Spring Boot • JPA • PostgreSQL
Cloud SQL for PostgreSQL with Spring Cloud	Herausfinden, wie man auf GCP eine PostgreSQL-Datenbank erstellen kann.	• Docker • Spring Boot • JPA • Cloud SQL for PostgreSQL
Kubernetes Spring Boot Webapplication	Herausfinden, wie man eine Spring Boot Applikation auf Kubernetes deployt und über einen LoadBalancer im Web verfügbar macht.	• Docker • Spring Boot • Kubernetes
Kubernetes and Cloud SQL with Spring Boot and PostgreSQL	Herausfinden, wie eine Datenbank von Cloud SQL von einem Kubernetes Cluster aus angebunden werden kann.	• Docker • Spring Boot • Kubernetes • Cloud SQL Proxy • JPA • Cloud SQL for PostgreSQL
Kubernetes and Datastore with Spring Boot	Herausfinden, wie der Cloud Datastore in Spring Boot verwendet und angebunden werden kann.	• Docker • Spring Boot • Kubernetes • Cloud Datastore
Kubernetes and PubSub with Spring Boot	Herausfinden, wie PubSub über Spring Cloud Streams angeschlossen werden kann und für das asynchrone Messaging eingesetzt werden kann.	• Docker • Spring Boot • Spring Cloud Streams • Kubernetes • PubSub
Kubernetes and Redis Cache with Spring Boot	Herausfinden, wie Redis (Memorystore) über Spring Cache angeschlossen werden kann.	• Docker • Spring Boot • Spring Cloud Streams • Kubernetes • Redis

Tabelle 3 Experimente

Prototyp

Nachdem diese Experimente durchgeführt worden waren, sind diese zu einem Prototyp zusammengeführt worden. Der Prototyp hat schlussendlich folgende Architektur:

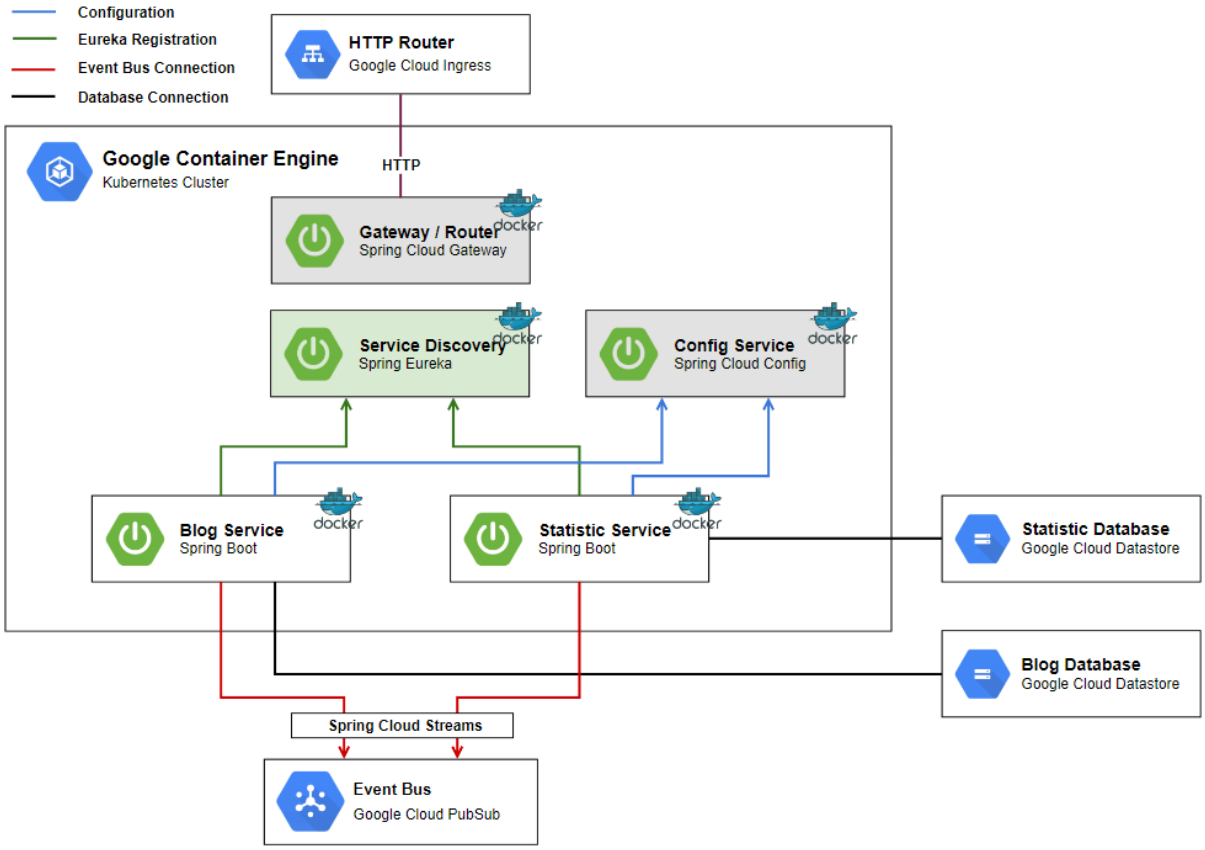

Abbildung 13 Blog Prototyp Deployment

Implementierungsdetails

Kubernetes Deployments und Services

Um Deployments und Services in Kubernetes zu erstellen, werden Konfigurationsdateien benötigt, welche in der Regel im Format YAML[15] sind, können aber auch JSON[16] sein.

Da **kubectl** die Standardanmeldedaten zur Authentifizierung gegenüber dem Cluster verwendet, muss der **application-default** von **gcloud** gesetzt sein (der normale auth-Befehl reicht nicht für alle Komponenten aus)[17]:

```
gcloud auth application-default login
```

Damit die Microservices im Cluster miteinander kommunizieren können, brauchen sie die Netzwerkadresse des jeweiligen Gegenübers. Dies könnte erreicht werden, indem die Deployments erstellt werden und nach dem Erstellen die IP-Adresse ausgelesen wird und statisch in den jeweiligen Microservice einprogrammiert wird. Jedoch ist dies kein dynamischer Ansatz und entspricht keiner langfristigen Lösung. Der bessere Ansatz ist die Verwendung des DNS-Servers des Clusters. Dieser löst Namensanfragen in die jeweilige IP-Adresse auf und ist für jeden Deployment-Namen immer aktuell.

Um dies einzurichten, muss ein Service-Endpoint erstellt werden:

```yaml
kind: Service
apiVersion: v1
metadata:
  name: statistic-microservice
spec:
  selector:
    app: statistic-microservice
    tier: backend
  ports:
  - protocol: TCP
    port: 80
    targetPort: http-server
```

Wichtig ist hierbei, dass der **selector** mit **app** und **tier** (hier **backend**) übereinstimmt. Ansonsten wird der Service den Pod nicht finden. Mithilfe dieses Services kann nun anstatt die IP-Adresse des Statistic-Microservices direkt der **name** verwendet werden:

```
http://statistic-microservice/api/statistic/..
```

Jenkins mit kubectl

Damit **kubectl** Befehle von einer Jenkins-Pipeline aus verwendet werden können, braucht der Jenkins das **Kubernetes Cli** Plugin[18]. Verwendet wird dieses dann mit **withKubeConfig** im Jenkinsfile. Ausserdem muss das Kommandozeilen-Programm **kubectl** auf dem Jenkins-Server installiert sein, denn das **Kubernetes Cli** Plugin liefert nur die vereinfachte Konfigurierbarkeit der Zugangsdaten. Da Jenkins jedoch in einem Docker-Container auf Kubernetes läuft und auf einem vorgefertigten Image von Google basiert, können diese Binaries nicht einfach installiert werden.

[15] https://en.wikipedia.org/wiki/YAML
[16] https://en.wikipedia.org/wiki/JSON
[17] https://cloud.google.com/kubernetes-engine/docs/clusters/operations#configuring_kubectl
[18] https://wiki.jenkins.io/display/JENKINS/Kubernetes+CLI+Plugin

Um trotzdem diese Funktionalität zu erhalten, wurde ein Docker-Image verwendet, welches die Binaries für kubectl beinhaltet:

```
docker.image('gcr.io/cloud-builders/kubectl').inside('--entrypoint ""') {
    withKubeConfig([credentialsId: 'added_credentials',
                    serverUrl: 'https://YOUR_CLUSTER_IP']) {
        sh "kubectl get pods"
    }
}
```

Speziell bei diesem Docker-Image ist der angepasste Entrypoint.

.inside('--entrypoint "") wird verwendet um den Standard-Entrypoint des Docker-Images zu überschreiben.

Die genaue Anleitung für das Aufsetzen von **kubectl** in Jenkins findet sich im öffentlichen Repository[19] unter dem Absatz **Kubernetes Cli**.

Asynchrone Microservices mit Event-Bus

Ein spezieller Aspekt der Architektur sind die asynchron verbundenen Microservices. Das bedeutet, dass ein Microservice einen Event veröffentlichen kann und sich danach nicht mehr darum kümmern muss. Dieses Konzept wurde beim Statistik-Microservice eingesetzt. Sobald beim Blog-Microservice ein Blog-Post angefragt wird, veröffentlicht dieser einen Event der besagt, dass der jeweilige Post gelesen wurde. Der Statistik-Microservice kann sich diese Informationen in seine eigene Datenbank speichern ohne dabei die Laufzeit des Blog-Microservice zu beeinträchtigen. Das ist ein sehr wichtiges Konzept, wenn die Applikation wächst und sehr viele verschiedene Aktionen bei einem einzelnen Aufruf des Benutzers stattfinden müssen.

Wichtig zu erwähnen ist hierbei, dass der Name des Bindings in der Konfiguration exakt gleich geschrieben werden muss, wie der Methodenname des Service:

Konfiguration:

```
spring:
  cloud:
    stream:
      bindings:
        blogEvents:  # spring internal name
          destination: blog_events # google cloud pubsub name
          content-type: application/json
      gcp:
        pubsub:
          bindings:
            blogEvents: # spring internal name
              consumer:
                auto-create-resources: true
```

Service:

```
@Output
MessageChannel blogEvents();
```

[19] https://github.com/firegnome/spring-cloud-prototype-gcp/tree/master/_devtools/jenkins

Cloud Datastore Ancestors

Der Cloud Datastore ist eine dokumentorientierte Datenbank[20] die den Vorteil hat, dass man eventuell konsistente Abfragen machen kann. Im Cloud Datastore können Beziehungen als Ancestors abgebildet werden. Das kann zum Beispiel die Beziehung einer Aufgabe zu einer Aufgabenliste sein. In diesem Beispiel ist die Aufgabenliste der Ancestor der Aufgabe. Durch diese Beziehung gehören diese zwei Typen zu einer Entität-Gruppe. Abfragen, die diese Beziehung berücksichtigen, werden standardmässig stark konsistent ausgeführt[21], was dazu führen kann, dass Abfragen auf diese länger dauern. Man kann dieses Verhalten bewusst deaktivieren und die eventuelle Konsistenz bevorzugen.

Dies wurde bei der Abhängigkeit zwischen Blog und Blog-Post gemacht:

```
Query<Entity> query = Query.newEntityQueryBuilder()
        .setKind(BLOG_POST_KIND)
        .setFilter(StructuredQuery.PropertyFilter.hasAncestor(
                datastore.newKeyFactory().setKind(BLOG_KIND).newKey(blogId)))
        .build();
```

In diesem Beispiel wird bewusst auf die starke Konsistenz verzichtet und der Kompromiss eingegangen, dass möglicherweise nicht alle Blog-Posts geladen werden, wenn gerade ein neuer Blog-Post erstellt wurde.

Zugriffsberechtigung der Benutzer

Um die Benutzer für Zugriffe auf bestimmte Ressourcen zu autorisieren kommen verschiedene Technologien in Frage:

- Basic Authentication in Kombination mit SESSIONID[22]
- OAuth2 Tokens (geeignet für Delegated Access in Zusammenhang mit einer externen API)
- JWT Tokens (geeignet für Microservices)

Die Basic Authentication in Kombination mit einer SESSIONID ist die einfachste Form der Authentifizierung. Sie ist jedoch für eine Microservice-Architektur ungeeignet, da jeder Microservice fähig sein müsste, festzustellen, ob die mitgelieferte SESSIONID gültig ist. Das würde eine zentrale Einheit wie eine Datenbank oder ein Cache verlangen, die bei jedem Request aufgerufen werden müsste.

OAuth2 Tokens würden das Problem mit den verteilten Services lösen, sind jedoch viel aufwändiger zu implementieren. Des Weiteren sind sie für die Kommunikation mit einer externen API geeignet.

Wir haben uns für die JWT Tokens entschieden, da sie perfekt für den Einsatz in einem Microservice-System sind. Wenn der Auth-Service den Benutzer mit seinem Benutzernamen und Passwort authentisiert, stellt er diesem ein JWT Token aus, signiert dessen Inhalt mit einem privaten Schlüssel und gibt dem Benutzer dieses mitsamt Signatur zurück. Wenn der Benutzer jetzt einen Zugriff auf eine geschützte Ressource machen will, sendet er dieses Token mit. Der betroffene Microservice kann den Benutzer anhand des mitgelieferten Tokens autorisieren und auf die Ressource zugreifen, da er ebenfalls im Besitz des privaten Schlüssels ist, mit dem das Token signiert wurde.

[20] https://en.wikipedia.org/wiki/Google_Cloud_Datastore
[21] https://cloud.google.com/datastore/docs/concepts/structuring_for_strong_consistency#datastore-datastore-basic-entity-java#consistency_guarantees
[22] https://www.baeldung.com/spring-security-basic-authentication

Überprüfung des Tokens in einem Microservice:

```
DecodedJWT jwtToken = JWT.require(Algorithm.HMAC512(SECRET.getBytes()))
        .build()
        .verify(token.replace(TOKEN_PREFIX, ""));
```

Ausstellung des Tokens mit der Ergänzung von Benutzername und Benutzer ID:

```
String token = JWT.create()
        .withSubject(((User) auth.getPrincipal()).getUsername())
        .withExpiresAt(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
        // Add userId so that other micro services can identify this user
        .withClaim("userId", user.getId().toString())
        .sign(HMAC512(SECRET.getBytes()));
```

Hier wird ein weiterer Vorteil der Signatur des Tokens deutlich. Im oben gezeigten Beispiel wird das Token durch die **userId** ergänzt. Das Token kann mit der benötigten Information angereichert werden, die die Microservices brauchen. Durch die Signatur wird garantiert, dass diese Daten nicht verändert wurden und somit kann diesen vertraut werden.

Jenkins auf Kubernetes

Ein weiteres erwähnenswertes Detail der Arbeit ist der Docker-in-Docker-Ansatz in Kombination mit dem Jenkins. Das Ziel war es, die Pipelines auf Docker-Images laufen zu lassen. Diese Images ermöglichen es, auf einfache Art und Weise komplizierte Umgebungen bereitzustellen. So kann zum Beispiel ein Headless Chrome für Angular E2E-Tests alleine mit der Referenzierung des richtigen Images bereitgestellt werden. Die grosse Herausforderung lag jedoch darin, dass der Jenkins selber auch in einem Docker-Container auf Kubernetes läuft. Da am Ende der Pipeline für die neue Version der Applikation ein Docker-Image erstellen werden muss, muss dieses im Docker Image des Jenkins gemacht werden. Das würde voraussetzen, dass Docker in diesem Image installiert ist. Es gibt grundsätzlich drei Ansätze dieses Problem zu lösen, Docker-in-Docker (DinD), Docker-out-of-Docker (DooD) oder Kaniko[23]. Um das neue Applikations-Image zu builden, könnte also der DooD-Ansatz gewählt werden und der Docker-Daemon des darunterliegenden Kubernetes-Clusters verwendet werden. Die zweite Variante ist Docker auf dem Jenkins-Image zu installieren und zu starten.

DooD

Der DooD-Ansatzt greift auf den Docker-Daemon des darunterliegenden Systems, in unserem Fall Kuberenetes von Google Cloud, zu. Wenn ein Docker-Container gestartet wird, wird dieser also neben dem aktuellen Container gestartet. Dieser Ansatz wird oft wegen Sicherheitsbedenken kritisiert, da dem Container, in unserem Fall dem Jenkins, sehr viel Berechtigung auf die darunterliegende Maschine gegeben wird. Es kann auch zu Inkonsistenzen führen, da das darunterliegende System eigentlich von Kubernetes verwaltet wird und mit einer solchen Massnahme würde man Container neben Kubernetes starten und hat keine gute Kontrolle mehr über das System. Der DooD-Ansatz könnte auch mit einer externen virtuellen Maschine gemacht werden, die nicht Teil des Kubernetes-Clusters ist und auf der Docker installiert ist. Diese Maschine könnte dann den Docker-Daemon für die Jenkins-Container freigeben und diese könnten dann die Container auf dieser Maschine starten. Der Nachteil dieser Lösung ist, dass sie nicht automatisch skaliert. Es müssen manuell weitere virtuelle Maschinen hinzugefügt werden um die Kapazität zu erhöhen. Wenn alles auf Kubernetes läuft, profitiert man von der automatischen Skalierung.

[23] https://medium.com/hootsuite-engineering/building-docker-images-inside-kubernetes-42c6af855f25

DinD

Beim DinD-Ansatz wird Docker auf dem Docker-Image installiert und gestartet. Der grosse Nachteil dieser Variante ist, dass sehr viel Ressourcen gebunden werden, da kein gemeinsamer Cache für Docker-Images aus dem Repository vorhanden ist und weil es eine Virtualisierung in der Virtualisierung ist.

Die Vorteile sind jedoch sauber getrennte Berechtigungen und die automatische Skalierung dieser Container.

DinD mit SideCar

Das Team hat sich für ein Konzept entschieden, das auf dem DinD-Ansatz basiert jedoch noch etwas weiter geht. Bei diesem Deployment werden zwei Container gestartet, einer mit dem Jenkins und einer auf dem Docker installiert ist. Auf beiden Containern sind die Docker-Binaries installiert, jedoch läuft der Docker-Daemon nur auf dem Container mit Docker. Dieser gibt dem Jenkins-Container den Docker-Daemon über einen Port frei und somit sieht es für den Jenkins aus, als würde Docker auf seinem System laufen. Es hat aber den Vorteil, dass der Daemon keinerlei Berechtigungen auf das Jenkins-System hat, da er auf einer anderen virtuellen Maschine läuft. Hier wurde bewusst der Kompromiss des zusätzlichen Ressourcenverbrauchs eingegangen, um eine saubere Trennung zu erreichen.

Google Kaniko

Ein weiteres Projekt, das zur Diskussion stand war Kaniko von Google. Dieses Projekt ermöglicht das Erstellen von Docker-Images in einem Docker-Container ohne einen laufenden Docker-Daemon. Dieser Ansatz ist sehr sauber und effizient, erfüllt jedoch nicht die Anforderungen. Das Ziel ist es nicht nur Docker-Images erstellen zu können, auch die Pipeline soll auf verschiedenen Docker-Images laufen gelassen werden können. Aus diesem Grund kam dieses Projekt nicht in Frage.

Ergebnis

Das Resultat der Arbeit, der Architektur Prototyp einer Spring Cloud Applikation, findet sich im öffentlichen Repository[24].

Für das Ergebnis wurde nicht das geplante Theme von KeenThemes verwendet, da dies eine Kostenpflichtige Lösung ist aber diese Arbeit unter einer Open Source Lizenz zur weiteren Verwendung veröffentlicht wird. Deshalb wurde anstatt dieser Theme Bootstrap verwendet.

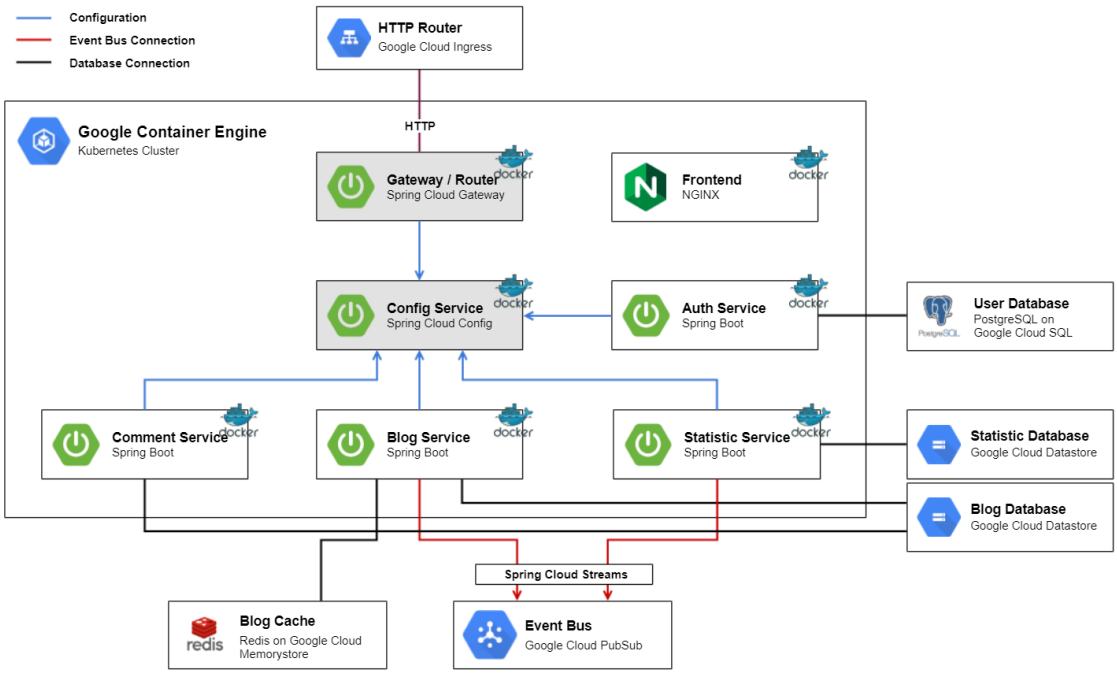

Abbildung 14 Deployment

Alle Microservices sind im selben Kubernetes Cluster auf GCP erstellt und der Gateway leitet die Anfragen an den jeweiligen Microservice weiter.

Pfad	Microservice
/api/blog/**	Blog Service
/api/comment/**	Comment Service
/api/statistic/**	Statistic Service
/auth/**	Auth Service
/**	Frontend

Tabelle 4 Routing

[24] https://github.com/firegnome/spring-cloud-prototype-gcp

Das Ergebnis lässt sich am besten an einem Beispiel diskutieren.

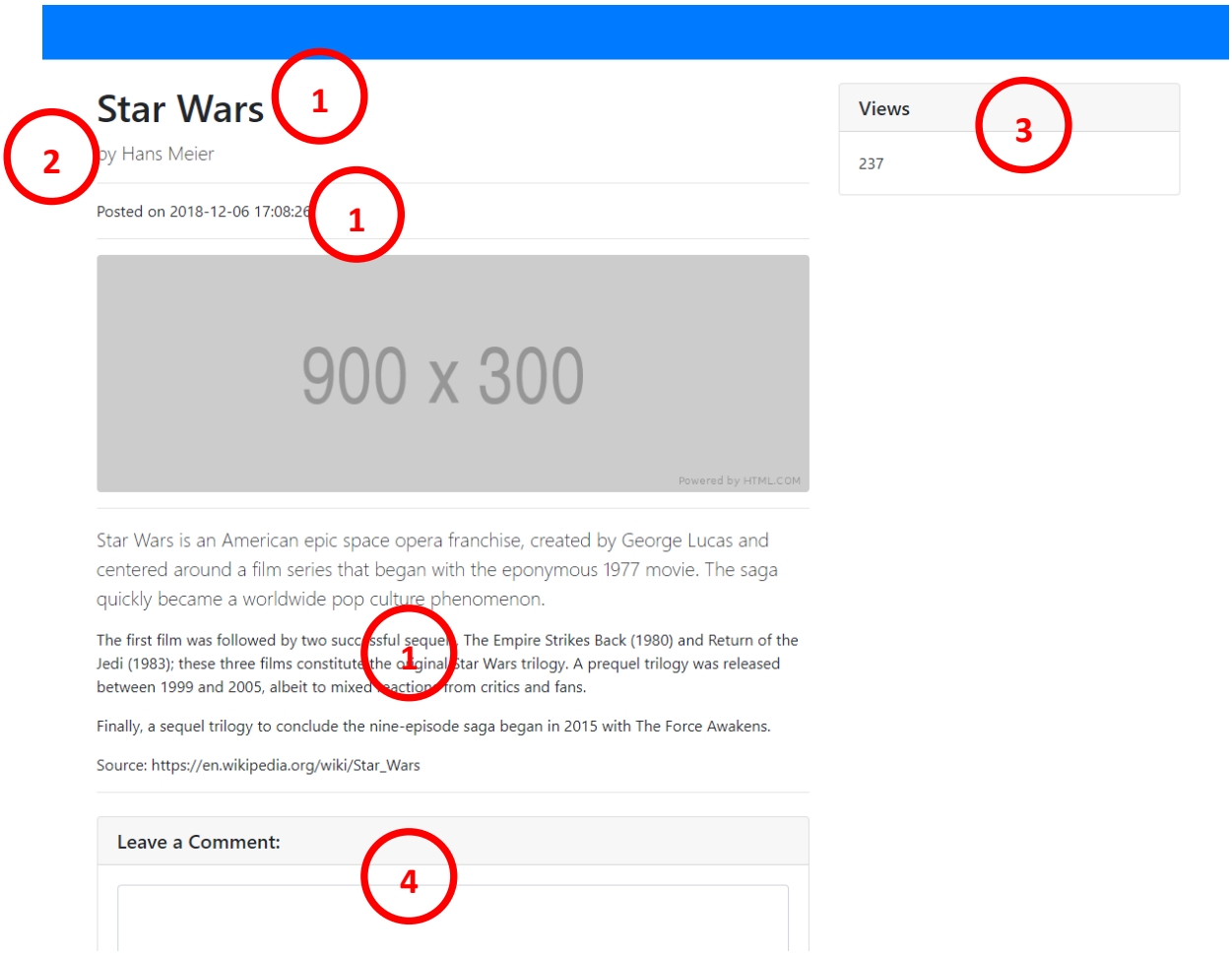

Abbildung 15 Blog Post Vorschau

Im oben gezeigten Abbild sieht man einen Blog-Post. Die eingekreisten Nummern im Abbild weisen auf den Ursprung der jeweiligen Daten hin. Die Nummern stehen für folgende Microservices:

Nummer 1: Blog-Microservice
Nummer 2: Auth-Microservice
Nummer 3: Statistic-Microservice
Nummer 4: Comment-Microservice

Wenn eine Ressource beim Laden dieses Blog-Posts nicht verfügbar ist, werden diese Daten nicht angezeigt. Das kann man sich zum Beispiel bei den Views vom Statistic-Microservice vorstellen. Die Zahl, die indiziert wievielmal der Blog-Post schon gelesen wurde, würde hier nicht angezeigt werden und hier im konkreten Fall würde die Box mit den Views überhaupt nicht angezeigt werden. Wenn jetzt das Entwickler-Team, das für den Statistic-Microservice verantwortlich ist, eine fehlerhafte Änderung in die produktive Umgebung einbaut und dieser Service nicht mehr verfügbar ist, wird der Blog-Post trotzdem angezeigt. Neben der Geschwindigkeit ist diese Art von Ausfallsicherheit einer der grossen Vorteile einer verteilten Applikation.

Review Nicht-Funktionale Anforderungen

Rückblickend hat es nicht viel Sinn gemacht, nicht Funktionale Anforderungen zu definieren. Die Applikation war in der Arbeit nur ein Mittel zum Zweck. Das Ziel war es, die architektonischen Merkmale einer asynchronen Microservice-Architektur zu zeigen und darum wurden die Entscheidungen, was entwickelt werden soll, daran gemessen, was sie zu diesem Ziel beitragen können. Das führte dazu, dass geplante nicht Funktionalen Anforderungen nicht erfüllt wurden.

Qualitätsmerkmal	Erfüllt	Bemerkung
1. Die Applikation blockiert nach drei fehlerhaften Anmeldeversuchen die Anmeldung für 30 Sekunden.	**nicht erfüllt**	Feature nicht implementiert
2. Das System kann in 99% von den Fällen 100‘000 Blog-Artikel pro Sekunde ausliefern.	**nicht erfüllt**	Nicht getestet
3. Artikel werden erstellt, auch wenn der Abo Service nicht verfügbar ist.	**nicht erfüllt**	Feature nicht implementiert
4. Wenn der Abo Service nicht verfügbar war und Artikel erstellt wurden, führt er sie beim nächsten erfolgreichen Start nach.	**nicht erfüllt**	Feature nicht implementiert
5. Der Blog wird geladen, auch wenn der Kommentar-Service nicht verfügbar ist.	**erfüllt**	
6. Der Blog wird geladen, auch wenn der Login-Service nicht verfügbar ist	**erfüllt**	
7. Der Statistik-Service kommt automatisch wieder in einen konsistenten Zustand, auch wenn er für längere Zeit nicht verfügbar war.	**erfüllt**	
8. Alle Input-Felder können über die Tabulatortaste erreicht werden.	**erfüllt**	
9. Alle Input-Felder sind dem Inhalt entsprechend validiert.	**nicht erfüllt**	Email bei login ist nicht erfüllt.
10. Alle Input-Felder sind mit Labels versehen um von Screen-Readern gelesen werden zu können.	**nicht erfüllt**	Nicht getestet
11. Neue Kommentare können in 99% der Fälle innerhalb 5 Sekunden ausgelesen werden.	**erfüllt**	
12. Neue Likes können in 99% der Fälle innerhalb 5 Sekunden ausgelesen werden.	**nicht erfüllt**	Feature nicht implementiert
13. Neue Benutzerlogins können in 99% der Fälle innerhalb 5 Sekunden aus der Statistik ausgelesen werden.	**nicht erfüllt**	Feature nicht implementiert
14. Neue Blogs können in 99% der Fälle innerhalb 5 Sekunden aus der Statistik ausgelesen werden.	**nicht erfüllt**	Feature nicht implementiert
15. Alle Unittests müssen erfolgreich durchlaufen, bevor der Code in den Master-Branch akzeptiert wird.	**erfüllt**	
16. Die Applikation ist durch ein Installationsscript aufsetzbar.	**erfüllt**	
17. Die gesamte Installation dauert nicht länger als 30 Minuten.	**erfüllt**	

Tabelle 5 Review Nicht Funktionale Anforderungen

Schlussfolgerung

Die geplante Architektur konnte erfolgreich in einer Beispiel-Applikation umgesetzt werden. Die gewünschten Technologien wurden an verschiedenen Beispielen gezeigt. Das wichtigste Thema der Arbeit war die asynchrone Microservice-Architektur. Ein weiterer wichtiger Teil ist, dass die Beispiel-Applikation mit Spring Cloud implementiert und auf der Google Cloud Platform bereitgestellt wurde. Das Spring Framework ist eine sehr umfangreiche Bibliothek, wenn es um die Unterstützung im Bereich von Microservices geht. Mit diesem Framework hätte auch eine andere Applikation wie synchrone Microservices oder eine monolithische Applikation programmiert werden können.

In der Beispiel-Applikation konnten nur sehr bedingt die Vorteile der asynchronen Kommunikation gezeigt werden und die Diskussion, ob diese Architektur Sinn macht, müsste weitergeführt werden. Eine grosse Schwäche dieser Architektur ist der Mehraufwand, der die Kommunikation verursacht. Die Beispiel-Applikation zeigt vielmehr den Einsatz der Technologien und wie sie eingesetzt werden könnten. Über dieses Thema gibt es im Internet jedoch bereits eine umfangreiche Diskussion[25] [26] und das Ergebnis dieser Arbeit eignet sich nicht als Grundlage um über den Einsatz einer asynchronen Architektur zu entscheiden. Vielmehr soll sie die technischen Konsequenzen an einem praktischen, vollständig bereitgestellten Beispiel zeigen.

Die DDD-Analyse war ein sehr interessantes Vorgehen. Der Ansatz Domänen-Grenzen, die in der realen Welt existieren, in die Software-Architektur zu übertragen, um eine starke Kohäsion zu erreichen, scheint sinnvoll zu sein. Es gibt bestimmt auch andere sehr gute Vorgehensweisen um Grenzen zwischen den Microservices zu finden. So kann es sicher auch sinnvoll sein, Teile der Applikation aus Geschwindigkeitsgründen in separate Microservices auszulagern. Auch bei diesem Thema kann diese Arbeit nur die Ansätze der DDD-Analyse zeigen, hoffentlich jedoch dazu anregen, diese Gedanken in grössere Projekte einfliessen zu lassen und als Fallbeispiel dienen.

Die Arbeit zeigt einen sehr umfangreichen Einsatz vieler Technologien auf der Google Cloud Platform und soll als Anhaltspunkt und Entscheidungsgrundlage für Software-Architekten dienen. Auch wurde versucht die Installation möglichst einfach zu gestalten, um die Hürden so klein wie möglich zu halten.

Eine Applikation, die auf den gezeigten Technologien aufbaut, hat das Potential für Millionen von Menschen verfügbar zu sein. Das zeigt auch Netflix, das ein starker Treiber des Spring Cloud Projektes ist und im Jahr 2017 die 100 Millionen[27] Benutzer mit ihren Services bedient haben. Das Ergebnis der Arbeit soll es Software-Architekten einfacher machen mit dieser Technologie zu experimentieren.

[25] https://jaxenter.de/microservices-warum-57795
[26] http://blog.covis.de/die-vorteile-und-herausforderungen-einer-microservices-architektur
[27] https://www.tagesspiegel.de/wirtschaft/nutzerzahlen-us-streamingdienst-netflix-hat-in-wenigen-tagen-100-millionen-abonnenten/19679750.html

Glossar

Begriff	Beschreibung
Microservice	Ein Microservice ist ein unabhängiger Teil einer komplexen Anwendungssoftware und erledigt eine kleine Aufgabe. Dies ermöglich modularen Aufbau einer Anwendungssoftware.
DevOps	DevOps (Development und IT Operations) beschreibt die Tätigkeit von Softwareentwicklung, Konfigurieren und Unterhalten von Systemen.
CI/CD-Umgebung	Continuous Integration (CI) und Continuous Delivery beschreibt die Automation von Testen der Software über Codequalität-Überprüfung bis hin zum Erstellen und Bereitstellen der Applikation.
Google Cloud Platform (GCP)	Google Cloud Platform ist ein Produkt von Google, das Cloud Services anbietet.
Docker	Mit Docker können virtuelle Umgebungen erstellt, geteilt und betrieben werden.
Kubernetes	Kubernetes basiert auf Docker, hat jedoch noch weitere Eigenschaften, die sich vor Allem in grösseren Deployments auszahlen.
Jenkins	Jenkins ist ein Test und Automatisierungsserver für CI/CD.
Netflix	Netflix ist ein US-Amerikanisches Unternehmen, welches sich auf das Streaming von Filmen spezialisiert hat.
Bootstrap	Bootstrap ist ein Open-Source Frontend-CSS-Framework für die Erstellung von Webseiten.
Monolithische Applikation	Eine Applikation welche die komplette Applikationslogik in einer einzigen Instanz beinhaltet.
Testsysteme	Testsysteme werden von Entwicklern eingesetzt, um ein Abbild des produktiven Systems zu haben, um dort die neuen Features zu testen.
Programmbibliothek	Eine Programmbibliothek beinhaltet Programm-Code, der von anderen Applikationen verwendet werden kann.
Microservice-Deployment	Eine Bereitstellung von mehreren Services, die zusammen eine Applikation darstellen.
Horizontales Skalieren	Die Eigenschaft einer Applikation, die auf mehrere virtuelle oder physikalische Computer verteilt werden kann.
Core-Domäne	Diese Domäne beinhaltet die zentralen Elemente einer Applikation.
Unterstützende Domäne	Eine unterstützende Domäne beinhaltet Elemente, die die Core-Domäne unterstützen.
Message-Bus	Eine Nachrichten-Warteschlagen, die sequenziell abgearbeitet wird.
Story-Mapping	Eine Vorgehensweise um Anforderungen zu erfassen.
User Story	Eine User Story beschreibt Anforderungen in der Alltagssprache.
Kubernetes Cluster	Ein Zusammenschluss mehrerer virtuellen oder physikalischen Computer auf denen Docker-Container bereitgestellt werden können.
Docker-Volumes	Eine virtuelle Festplatte die von Docker verwaltet wird und von Docker-Containern verwendet werden kann.
Docker-Container	Eine Instanz eines Docker-Images.
Docker-Image	Eine vorkonfigurierte Instanz eines Betriebssystems, welches von Docker als Docker-Container bereitgestellt werden kann.

Frontend	Der Teil der Applikation, der vom Benutzer gesehen wird. In einer Web-Applikation ist dieser häufig mit HTML und JavaScript umgesetzt.
Service-Endpoint	Der Port auf dem ein Service verfügbar ist.
Gateway	Das ist die Komponente, welche die Microservices mit einem anderen Netzwerk, zum Beispiel dem Internet, verbindet.

Tabelle 6 Glossar

Quellen/Referenzverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[Vernon17] Vaughn Vernon, «Domain-Driven Design kompakt», 2017, Übersetzt von Carola Lilienthal, Henning Schwentner

[Patton14] Jeff Patton, «User Story Mapping», September, 2014