„WolkeSieben“ - die Cloud-Native Tiervermittlung

Klaus Fuhrmeister, Arne Müller, Niklas Sauer, Savina Diez

Februar 2021

Link zu den GitHub-Repositories: `https://github.com/CC-Wolke7`

Inhaltsverzeichnis

1 Motivation und Idee

Tiere aller Art verdienen ein fürsorgliches und sicheres Umfeld. Leider kann das nicht immer garantiert werden, sodass sich viele Besitzer dazu entscheiden, ihr Haustier abzugeben. Zur Weitervermittlung leisten Tiervermittlungen wie Ebay Kleinanzeigen oder Tiervermittlung.de einen wichtigen Beitrag für unsere Gesellschaft. Wir, ein Team von Tierliebhabern, unterstützen das Konzept von Tiervermittlungen, wollten es jedoch mit einer ansprechenden Benutzeroberfläche, intuitiven Funktionen und einer robusten Infrastruktur umsetzen, die theoretisch beliebig viele Vermittlungen übernehmen kann. Letzteres sollte mit Hilfe der Google Cloud Plattform erreicht werden.

Das Portal mit dem Namen „WolkeSieben“ erlaubt Besitzern, die ihr Haustier abgeben möchten, Angebote zu erstellen, diese mit Bildern und Videos zu versehen, hierzu Anfragen zu erhalten und Details mit Interessenten in Echtzeit zu besprechen. Durch das Hosting in der Cloud können daneben noch einige weitere Features effizient und skalierbar umgesetzt werden. Allgemein erlaubt diese Form des Hostings natürlich einen geräte-, zeit- und ortsunabhängigen Zugriff, sowie bessere Kostenstrukturen, da die Hardware, sowie die Wartung der Ressourcen entfallen.

Im Rahmen dieser Arbeit erläutern wir wie eine Vielzahl von Google Cloud Services zum Funktionsumfang und der Bereitstellung unserer Plattform beitragen werden. Folgende Services haben wir in unserer Applikation verwendet:

- **Cloud Run** zur Bereitstellung zweier Microservices
- **App Engine** zur Bereitstellung eines dritten Microservice
- **Cloud CDN** zur Bereitstellung des Frontends
- **Cloud Firestore** zur Speicherung von Chat-Daten
- **Cloud Memorystore** zur Skalierung des Echtzeit-Chats
- **Cloud Bigtable** zur Speicherung von „Gefällt mir“ Angaben
- **Cloud SQL** zur Speicherung von Benutzer Profilen, Angeboten, Merklisten, sowie Rassen-Abonnements
- **Cloud Storage** zur Speicherung von statischen Dateien wie dem Frontend, Profil- / Angebots-Medien und einer Admin-Oberfläche
- **Cloud Functions** zur Benachrichtigung über neue Angebote von abonnierten Rassen
- **Cloud Pub/Sub** zum Auslösen der o.g. Benachrichtigungen

Der Prototyp unserer Applikation, wie in Abbildung 1 demonstriert, ist ab sofort unter folgender Adresse aufrufbar: `https://cc-wolkesieben.de`.

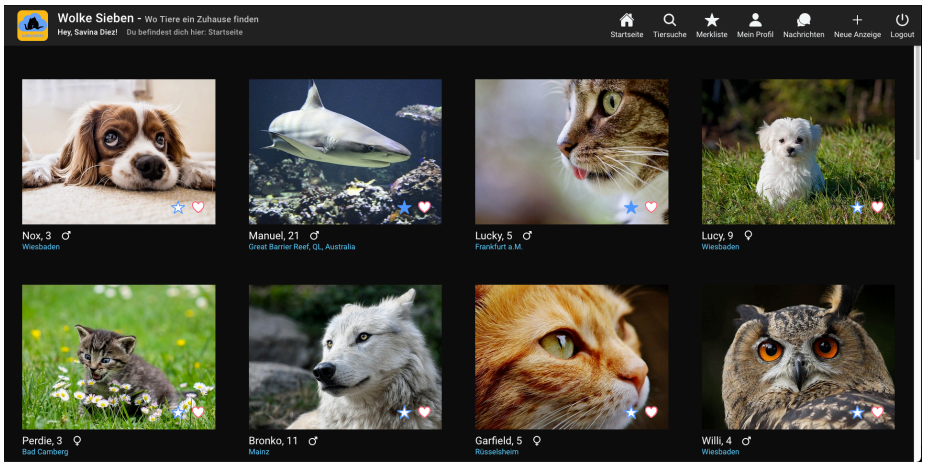

Abbildung 1: Startseite von cc-wolkesieben.de

2 Architektur

2.1 Microservices

Wie für Cloud-Anwendungen üblich setzen wir bei der Struktur des Backends auf einen Microservice-Ansatz, bei dem die ganzheitliche API in verschiedene, voneinander unabhängige Programme aufgeteilt wird. Diese Form der Bereitstellung bringt einige Vorteile mit sich, die hier herausgestellt werden.

Zum einen können die einzelnen Funktionen auf diese Weise in Umgebungen bereitgestellt werden, welche speziell auf diese zugeschnitten sind. In unserer Anwendung sieht man dies primär anhand der Aufteilung nach den verwendeten Datenbank-Typen (Polyglott Persistency). Im Umkehrschluss muss sich jeder Microservice somit auch nur mit denen für seine Domäne relevanten Technologien befassen. Dies erleichtert die Konfiguration und trägt dazu bei, dass sich Services einfacher unter den Entwicklern eines Team aufteilen lassen, ohne dass diese sich in die Quere kommen. Zu guter Letzt lassen sich die Möglichkeiten der Bereitstellung in der Cloud so optimal ausnutzen, da jeder Microservice auf der Infrastruktur aufgesetzt werden kann, die für ihn am geeignetsten ist, ohne den Rest der Anwendung zu einem Kompromiss zu zwingen. Dies wiederum erlaubt die unabhängige Skalierung der am meisten nachgefragten Microservices der ganzheitlichen API.

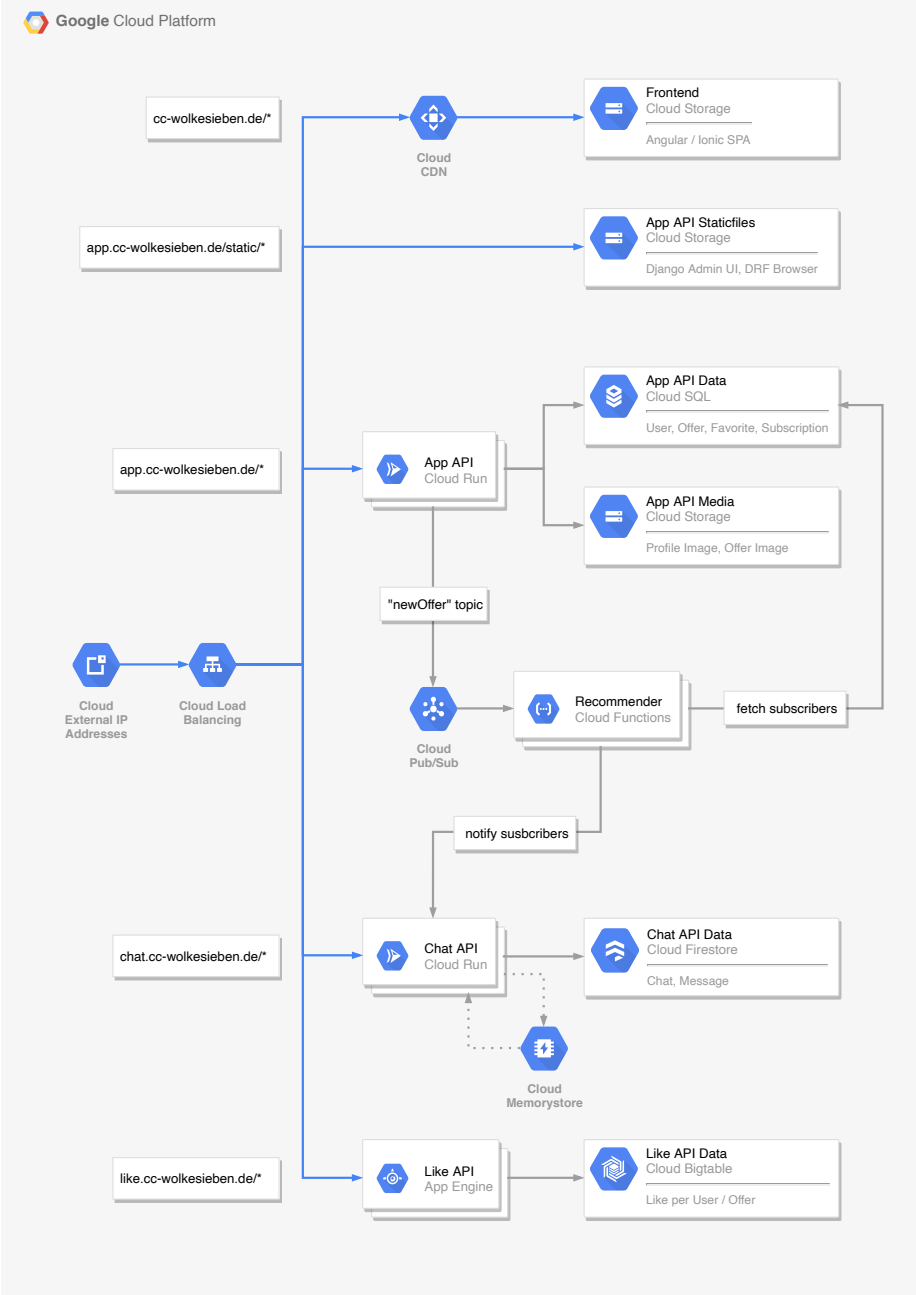

Abbildung 2: Architekturüberblick

2.2 Domain-Driven Design

Ursprünglich sollte sich die Architektur der einzelnen Microservices am Domain Driven Design-Pattern orientieren. Dieses sieht eine Unterteilung der Services wie folgt vor:

Domäne
: für die Kernfunktionen des Service

API
: für die Bereitstellung

Infrastruktur
: für die Anbindung an externe Systeme (z.B. Datenbanken)

Im Laufe der Entwicklung stellte sich heraus, dass sich das Pattern, welches eigentlich für DotNet-Anwendungen konzipiert wurde, nicht unmittelbar auf die von uns verwendeten Programmiersprachen und Frameworks übertragen lässt. Konkret sei dies am Beispiel von Django, einem Python Web-Framework, verdeutlicht: Einer der Hauptzwecke der Infrastruktur-Komponente ist es, die Anbindung verschiedener Datenbanken, sowie den Wechsel zwischen diesen, so einfach und standardisiert wie möglich zu gestalten. In Django besteht dieses Problem allerdings gar nicht, da die Datenbankanbindung komplett im Hintergrund stattfindet. Bei Verwendung des Django-REST-Frameworks entfällt außerdem ein Großteil der Arbeit, die zur Erstellung einer REST-API nötig ist, da auch hier vieles im Hintergrund passiert. Des Weiteren gibt Django eine umfassende Dateistruktur vor, die sich nur schwer mit dem Domain Driven Design vereinen lässt. Aus diesen Gründen haben wir uns bereits zu einem frühen Zeitpunkt der Entwicklung entschieden, uns nicht weiter an diesem Pattern zu orientieren.

3 Frontend

3.1 Technologie

Hinsichtlich dem Ziel, unsere Plattform einer möglichst großen Masse von Benutzern zugänglich zu machen, wurde das Frontend mit Ionic (Version 5) entwickelt. Obwohl augenscheinlich nur Browser Technologien zum Einsatz kommen (HTML, CSS, JavaScript), kann die mit Hilfe von Ionic entwickelte Single-Page-App anschließend auch in eine native iOS bzw. Android App konvertiert werden. Intern fußt Ionic auf dem von Google propagierten Angular Framework, welches das „Two-way Binding“ Paradigma umsetzt. Es unterstützt auch die von uns bevorzugte Programmiersprache TypeScript, ein Superset von JavaScript, sowie SASS als erweiterte Stylesheet-Sprache.

3.2 Authentifizierung

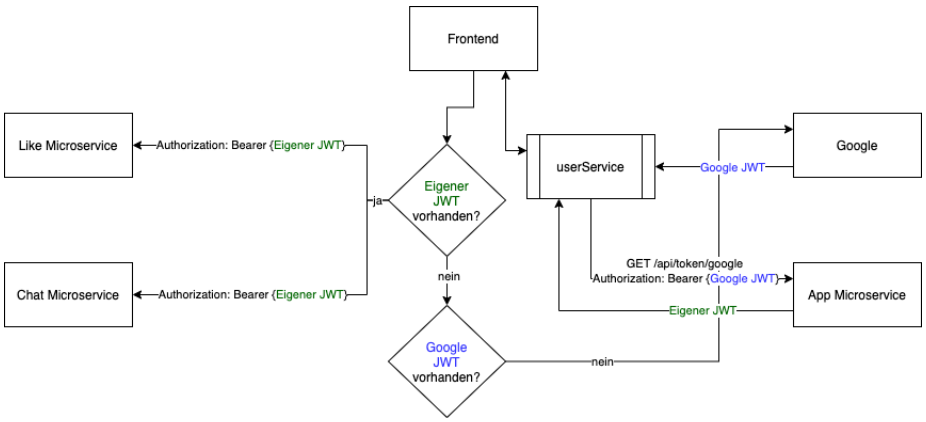

Abbildung 3: Authentifizierung mit Google

Die Authentifizierung eines Benutzers wird wie in Grafik 3 durchgeführt. Im ersten Schritt wird überprüft, ob bereits eine Session vorhanden ist. Diese ist durch das Vorhandensein eines gültigen JSON Web Token (JWT), ausgestellt vom App-Microservice (siehe Abschnitt 4), gekennzeichnet. Andernfalls ist der Benutzer nicht eingeloggt. In diesem Fall kann der Nutzer sich mit einem Social Provider (Facebook, Google, GitHub etc.) einloggen. Aktuell wird nur Google unterstützt. In jedem Fall wird jedoch eine OAuth 2.0 Anfrage an den jeweiligen Provider abgesetzt, die das Öffnen dessen Single-Sign-On Fenster zur Folge hat. Bei erfolgreichem Ausweisen der Identität (normalerweise durch Benutzername und Passwort, gegebenenfalls mit einem Einmalpasswort) liefert dieses ein sogenanntes ID Token zurück, das dem JWT Standard folgt. In dessen Payload findet sich neben den verifizierten Benutzer Details, wie Name und Email Adresse, auch die innerhalb der Domäne des jeweiligen Provider eindeutige ID des Nutzers. Um den Login-Vorgang abzuschließen, wird das ID Token an den App-Microservice gesendet, welcher die Gültigkeit anhand der Signatur prüfen kann und dieses gegen ein neues JWT austauscht. Die Notwendigkeit eines eigens ausgestellten JWT wird in Unterabschnitt 4.3 erläutert.

Genau genommen handelt es sich bei dem Rückgabewert des App-Microservice um ein JWT-Paar, nämlich ein Access- und Refresh-Token. Das Access-Token wird für künftige Requests an jeden der Microservices als `Bearer` Token im `Authorization` Header mitgesendet, worauf die Authentifizierung und Autorisierung in jedem Microservice basiert. Das Refresh-Token hingegen erlaubt es dem Frontend, ein neues Access-Token zu erhalten, wenn dieses seine zeitliche Begrenzung überschritten hat. Refresh-Tokens sind in der Regel um ein Vielfaches länger gültig.

Um die Login-Session auch bei Neuladen der Webseite beizubehalten, werden

beide Tokens im `localStorage` persistiert. Damit muss der oben beschriebene Login-Prozess erst wiederholt werden, wenn sich der Nutzer längere Zeit nicht im Portal bewegt hat und somit auch beide Tokens abgelaufen sind. Bei einem manuellen Logout werden beide Tokens zwangsläufig entfernt.

3.3 Google OAuth Integration

Zur einfachen Integration mit Angular, stellt Google das `gapi` NPM Package bereit. Es verwaltet den Zustand der Authentifizierung mittels eines `GoogleUser` Objektes, das auch eine Methode exponiert (`.getAuthResponse().id_token`), um das in Unterabschnitt 3.2 erwähnte ID Token abzurufen. Sollte der Benutzer nicht angemeldet sein, kann der OAuth 2.0 Prozess durch die statische Methode `gapi.auth2.init()` gestartet werden. Diese Methode sollte mit der in der Google Developer Konsole konfigurierten `CLIENT_ID` aufgerufen werden, um sicherzustellen, dass das ID Token aus dieser Login-Session und nicht der einer anderen Applikation stammt.

3.4 Deployment

3.4.1 Google Cloud Storage

Der Google Cloud Storage ist ein Cloud Speicher, um beliebige Binary Large Objects (BLOBs) zu speichern und in sogenannten Buckets (vgl. Ordner) zu verwalten. Es gibt kein Speicherlimit, stattdessen wird die Benutzung pro belegtem Gigabyte abgerechnet. Datensicherheit, Langlebigkeit sowie Verfügbarkeit werden durch die Google SLAs garantiert.

Mit dem Programm `gsutil` stellt Google eine Kommandozeilen-Schnittstelle zur Verfügung, über die auch der Frontend Bucket dieses Projektes regelmäßig aktualisiert wird.

3.4.2 Cloud CDN & Load Balancer

Das Cloud Content Delivery Network (Cloud CDN) ist ein Netzwerk von Google, welches überall auf der Welt Knotenpunkte verteilt hat und damit auf die schnelle Auslieferung von Inhalten optimiert ist. Um dieses zu nutzen, muss ein HTTP(S) Load Balancer eingerichtet werden, der Anfragen an eine Domain erhält und diese an den jeweiligen Google Cloud Storage Bucket weiterleitet. Das Cloud CDN ist dazwischengeschaltet und fungiert als unabhängiger, automatischer Cache in mehreren Regionen der Welt.

Für die Skalierung bedeutet dieses Konstrukt, dass der Bucket nicht skalieren muss, denn Anfragen entstehen lediglich während dem CDN Cache-Refill bzw. beim Upload in den Bucket selbst. Bei einer steigender Anzahl von Nutzern muss nur das Cloud CDN skalieren. Hierfür sorgt Google vorbildlich.

4 App-Microservice

Der App-Microservice stellt den Großteil der Backend-Funktionen unserer Anwendung bereit. Er ist in Python geschrieben, verwendet das Django Web-Framework und wird über Cloud Run bereitgestellt.

4.1 Technologie

Die Wahl der Programmiersprache ergab sich dabei aus der Wahl des Web-Frameworks. Django bietet einige entscheidende Bausteine, welche die Entwickelung von Web-Anwendungen vereinfachen und von denen auch wir großen Gebrauch gemacht haben.

Der größte Vorteil ist die vollständige Abstraktion des Datenbankmodells durch das eingebaute Object-Relational Mapping (ORM). Dies ermöglicht das Anlegen und Verwalten einer SQL-Datenbank ohne echten SQL-Code schreiben zu müssen. Modelle und Relationen können mit Python-Code definiert und anschließend per Konsolen-Befehl in SQL-Code umgewandelt und auf die Datenbank übertragen werden. Die Entwickler können sich dadurch auf die reine Programmierung in Python konzentrieren.

Unter Zuhilfenahme des Django-REST-Frameworks werden auch die Standard REST-Funktionen (`GET`, `POST`, `DELETE`, etc.) für alle eigens angelegten Datenbank Modelle vor-implementiert, wodurch sich die Entwickler umso mehr auf die für die Anwendung benötigten Funktionen konzentrieren können. Dieses Plugin erleichtert auch das Testen der Anwendung, indem es eine Web-Oberfläche zur Interaktion mit der API bereitstellt.

Daneben haben sich die eingebauten Konzepte, die Anwendung produktionsreif zu machen, schon mehrfach in der Industrie bewährt.

4.2 Datenbank

Entsprechend dem Polyglott Persistency Prinzips, setzt der App-Microservice auf eine Cloud SQL Instanz für relationale Daten, sowie einen Cloud Storage Bucket für Medien sonstiger Art (z.B. Fotos und Videos). Hinsichtlich dem SQL-Datenbank Typen, genügt MySQL (Version 8) unseren Anforderungen.

An dieser Stelle sollte erwähnt werden, dass zum Zeitpunkt der Entwicklung kein offizieller Emulator für den Google Cloud Storage vorliegt. Daher wurde zum Testen der Datenbankanbindung auf eine Open-Source Lösung namens `fake-gcs-server` zurückgegriffen.

4.3 Authentifizierung

Wie bereits in Unterabschnitt 3.2 dargelegt, knüpft die Nutzer-Authentifizierung an den OAuth 2.0 Flow eines jeweiligen Social Provider. Um den Login-Vorgang abzuschließen, muss das Frontend einen weiteren Request an den App-Microservice stellen, um das ID-Token des Social Provider gegen ein neues JWT zu

tauschen, das von allen Services dieses Projekts akzeptiert wird. Dies hat zwei entscheidende Vorteile:

Universally Unique Identifier (UUID)
: Die im ID Token eines jeweiligen Social Provider gelistete Benutzer Identifikation („`sub`“ Payload-Claim) ist streng genommen nur in der Domäne des jeweiligen Social Provider eindeutig. Das heißt, ein Benutzer mit der Google ID „`xyz`“ hat nicht zwangsläufig die gleiche ID bei Facebook oder einer anderen Plattform. Soll jedoch eine künftige Integration mit weiteren Single-Sign-On Anbietern erfolgen, muss eine in der Domäne unserer Tiervermittlung eindeutige Benutzerkennung eingeführt werden.

  Zu diesem Zweck speichert der App-Microservice beim ersten Login mit dem ID Token eines jeweiligen Providers (z.B. `GET /api/token/google`) welche Kennung der Nutzer bei diesem Provider hat, legt zugleich aber auch für den Nutzer eine eindeutige UUID an, sodass ein Lookup in beide Richtungen künftig möglich ist.

Custom Claims
: Ein Nebeneffekt von eigens ausgestellten JWTs ist die Tatsache, dass die Payload („Claims“) der Tokens selbst festgelegt werden kann. Dies ist besonders für die Autorisierung anhand von „Scopes“ relevant.

Damit dieses Token auch von den anderen Microservices zur Authentifizierung verwendet werden kann, exponiert der App-Microservice einen öffentlichen Endpunkt (`POST /api/token/verify`) mit dem die Gültigkeit eines solchen JWTs überprüft werden kann. In Produktion sollte diese Methode allerdings nicht verwendet werden, da sie unnötig zur Anfragenlast beiträgt. Stattdessen sollte lediglich der Public Key des Schlüsselpaars, mit dem die Signatur des JWTs erstellt wurde, öffentlich gemacht werden, sodass die anderen Services diesen initial laden und alle weiteren Schritte zum Prüfen der Gültigkeit lokal ausführen.

4.4 Deployment

Schon bevor das Deployment Target (App Engine, Cloud Run, Kubernetes etc.) der einzelnen Microservices festgesetzt wurde, haben wir uns darauf festgelegt, möglichst alle Dienste auf Container-Basis zu entwickeln. Das bedeutet, Code wurde prinzipiell immer gegen ein Linux-basiertes Container Image gebaut und getestet, mit dem garantiert werden kann, dass alle Entwickler die gleiche Entwicklungsumgebung vorliegen haben und Builds sowie Fehler leicht reproduzierbar sind, unabhängig von der zugrundeliegenden Hardware.

Es bietet sich daher an, auch ein Subset der Microservices als Container bereitzustellen. Dies ist auch der Fall für den App-Microservice. Für das Containerbasierte Deployment gibt es auf der Google Cloud Platform neben Kubernetes auch eine simplere Variante, nämlich Cloud Run. Beide sind in der Lage, basierend auf einer Deployment-Spezifikation, die Container Anzahl entsprechend der Anfragelast zu skalieren.

Auch an dieser Stelle sei eine Limitierung der Google Cloud Plattform (vgl. Unterabschnitt 4.2) zu nennen. Eine Schwierigkeit, die uns beim Bereitstellen der Microservices, die Cloud Run benutzen, aufgefallen ist, ist die fehlende Möglichkeit, eine SSH-Sitzung in einem laufenden Container zu starten. Somit kann eine Fehleranalyse nur schwer stattfinden. Log-basiertes Debugging kann nicht alle Problemquellen identifizieren. Uns ist weiterhin bekannt, dass andere Cloud-Anbieter diese Möglichkeit sehr wohl zur Verfügung stellen. Genau genommen bietet Google das mit seinem Kubernetes-Dienst bereits indirekt an (`kubectl exec ...`). Zudem basiert Cloud Run auf Kubernetes. Dies kann sicherlich einen Einfluss auf die Wahl der Cloud-Plattform haben.

5 Chat- und Like-Microservices

Die Komponenten, die das Chatten zwischen Nutzern sowie das Liken von Angeboten ermöglichen, werden in diesen Abschnitt gemeinsam gelistet, da sie im Prinzip der gleichen Quelle entspringen, nämlich einem Monorepo. Aus Gründen der Optimierbarkeit wurden diese beiden Services getrennt vom App-Microservice entwickelt.

5.1 Monorepo Strategie

Mono-Repositories beschreiben eine Strategie, bei der mehrere Projekte im gleichen versionierten Repository entwickelt werden. Dies bietet sich besonders bei Projekten an, die auf den gleichen Technologie-Stack setzen und hierdurch von einem vereinfachtem Dependency Management und leichterer Code-Wiederverwendung profitieren können.

Ein solcher Ansatz erzwingt dennoch nicht das gleiche Deployment Target oder generell den gleichen Build Prozess. In diesem Fall wurden die Chat- und Like-Microservices als zwei Teilbereiche einer einzigen API konstruiert, die jedoch durch die selbst entworfene Plugin-Architektur getrennt aktiviert und somit bereitgestellt werden können.

5.2 Technologie

Verglichen mit dem App-Microservice (siehe Unterabschnitt 4.1), ist die Wahl der Programmiersprache nicht vom Framework geprägt, sondern von der Sprache selbst. Gerade hinsichtlich der Echtzeit-Kommunikation mit Browsern per WebSockets, ein Core-Feature des Chat-Microservices, hat sich JavaScript als Backend-Pendant über die Jahre bewährt.

Auf der anderen Seite, gibt es, vergleichen mit Django, im NodeJS Umfeld viel seltener Frameworks, die eine solch rigide Struktur ansetzen. Das wohl bekannteste, Express, ist absolut leichtgewichtig und eher als Routing Mechanismus mit Middleware zu verstehen. Für größere Projekte ist dies in unseren Augen aber auf lange zeit kontraproduktiv. Deshalb, haben wir glücklicherweise

mit NestJS ein Framework gefunden, das den bewährten Konzepten von Angular folgt.

5.3 Authentifizierung

Die Authentifizierung setzt auf den in Unterabschnitt 4.3 beschriebenen Mechanismus, um die vom App-Microservice ausgestellten JWTs zu verifizieren und danach die Identität des Nutzers zu ermitteln. Ein Spezialfall stellt die Authentifizierung des Echtzeit-Chats dar, wie in Unterabschnitt 5.4 ausgewiesen wird.

5.4 Chat-Microservice

5.4.1 Datenbank

Die Besonderheit des Chats liegt zum einen an der Anforderung, diesen in Echtzeit anbieten zu können, zum anderen aber auch darin, dass sich das Format der Nachrichten populärer Messenger bekanntlich stark geändert hat. So ist es beispielsweise inzwischen möglich, per WhatsApp sowohl reine Textnachrichten, als auch Links, Bilder oder Dokumente zu versenden. Keiner der Nachrichten-Typen lässt sich sinnvoll durch einen relationalen Datenbank-Eintrag abbilden. Um auch unser Portal auf diesen Umstand vorzubereiten, setzen wir mit dem Chat-Microservice auf eine NoSQL Datenbank, dem Google Firestore. Es handelt sich hierbei um eine Dokumenten-basierte Datenbank, die in Kollektionen aufgeteilt ist und im Grunde genommen JSON-Dokumente von maximal 1 MB speichert [3].

5.4.2 Echtzeit-Chat

WebSockets Effektiv unterliegt jeder dem für einen Browser bestimmten Nachrichtenverkehr dem HTTP Protokoll [12, p. 1]. Somit kommen auch nur einige wenige Technologien in Frage, um Nachrichten in Echtzeit an Webseiten zu übertragen. Diese lauten:

- HTTP (Long-)Polling [15, p. 1]
- HTTP Streaming [17, p. 7]
- Server-Sent Events [11, pp. 43–44] [17, p. 13]
- WebSockets [12, p. 1]

Allerdings lassen sich nur mit WebSockets bidirektionaler Datenströme abbilden. Deshalb setzt dieser Teil der Anwendung auf diesen Transportkanal, der im NestJS Framework durch Gateways abstrahiert ist [4].

Skalierung Da WebSockets eine persistente Verbindung zwischen Client und Server einrichten, d.h. die Verbindung nicht wie traditionell nach jeder Anfrage schließen, sind besondere Vorkehrungen notwendig, um die Applikation horizontal zu skalieren. Denn es ist denkbar, dass ein Nutzer A, der mit Server-Instanz X verbunden ist, einem Nutzer B, der wiederum mit Server-Instanz Y in Verbindung steht, in Echtzeit eine Nachricht senden möchte. Dies bedarf der Kommunikation zwischen den horizontal skalierten Server-Instanzen X und Y. Hierfür setzt der Chat-Microservice auf Google Memorystore, einer In-Memory Datenbank, die zu 100% dem Redis Protokoll folgt [6].

Authentifizierung WebSockets sind einzig und allein mit HTTP verwandt, in dem Sinne dass dessen Handshake als eine Aufforderung zum WebSocket Upgrade verstanden werden kann [14, p. 57]. Das erlaubt es WebSockets mit traditioneller Web-Infrastruktur zu koexistieren [13, pp. 10–11]. Theoretisch könnte über die Header der Upgrade Anfrage auch eine Authentifizierung implementiert werden, so wie in Unterabschnitt 3.2 beschrieben. Aus unerklärlichen Gründen bietet jedoch die Browser WebSocket Schnittstelle keine Möglichkeit Header zu definieren [8]. Das funktioniert sehr wohl aus Sicht eines Backends [10]. Deshalb implementiert dieser Teil der Anwendung eine Authentifizierung auf Applikationsebene. So kann sich im ersten Schritt jeder Client mit dem Chat WebSocket verbinden. Danach ist dieser aufgefordert, eine `AUTH_REQUEST` Nachricht mit seinem JWT zu verwenden. Das Token wird wie gewohnt geprüft. Erst dann wird der bereits geöffnete WebSocket zur Liste der authentifizierten Nutzer hinzugefügt und kann künftig Nachrichten erhalten.

5.4.3 Deployment

Das Deployment Target ist Cloud Run. Hinweise hierzu finden sich in Unterabschnitt 4.4.

5.5 Like-Microservice

„Gefällt mir“ Angaben haben über die Jahre wortwörtlich die Herzen der Nutzer erobert. Wir gehen deshalb davon aus, dass ein großer Teil der API Anfragen auf diese Domäne zurück zu führen sein wird. Im Umkehrschluss muss diese also individuell skalieren können.

Als Deployment Target ist aus rein experimentellen Gründen die Google App Engine vorgesehen. In Produktion würde auch dieser Microservice per Cloud Run bereitgestellt werden, zumal er bereits als Container Image vorliegt (vgl. Unterabschnitt 4.4).

Nennenswert ist zuletzt der gewählte Datenbank-Typ. Es handelt sich hierbei um Google Cloud Bigtable, ein weiteres NoSQL-Produkt mit extrem niedriger Latenz und Skalierbarkeit bis auf Millionen Anfragen pro Sekunde [2]. Im Kontext von „Gefällt mir“ Angaben kann man sich die Bigtable wie eine Excel Kalkulationstabelle vorstellen: für verschiedenen Angebote (Zeile) wird der „Gefällt mir“ Zustand (Zelle) eines jeweiligen Nutzers (Spalte) niedergeschrieben. Dabei

kann der Zustand der „Gefällt mir“ Angabe durch einen Integer repräsentiert werden (`1 = Like` / `0 = Neutral` / `-1 = Dislike`). Die Angaben pro Angebot können durch die von Google bereitgestellten MapReduce Operationen effizient aggregiert werden [2].

6 Recommender Cloud Function

Angetrieben von der Idee, Benutzer über neue Angebote ihrer liebsten Rassen zu benachrichtigen, stellt die Cloud Recommender Function den einzigen Service dar, der nicht im Internet exponiert ist, d.h. keine frei zugängliche API hat. Stattdessen wird sie durch den Google Cloud PubSub Service ausgelöst. PubSub repräsentiert den asynchronen, ereignisgesteuerten Programmierstil, bei dem Komponenten eines Systems Zustandsänderungen in Form von Events publizieren [9]. Dabei ist im Allgemeinen nicht bekannt, welche anderen Komponenten diese Nachrichten empfangen werden oder wann sie diese verarbeiten [16, pp. 1–2] [18, p. 3]. Letzteres stellt kein Problem dar, denn dieser Benachrichtigungs-Service wurde als nicht-zeitkritisches Feature definiert.

Auch gibt es unterschiedliche Ansätze dazu, wie aussagekräftig ein solches Ereignis ist. In diesem Anwendungsfall wäre es also denkbar, das komplette neu eingetroffene Angebot oder nur die ID dessen via PubSub zu publizieren [19, p. 3]. Weiterhin benötigt der Service Kenntnis darüber, welche Nutzer die Rasse des Angebots abonniert haben.

Aktuell geht die Recommender Cloud Function davon aus, dass das komplette Angebot publiziert wird. Verantwortlich dafür ist der App-Microservice (siehe Abschnitt 4). Intern realisiert dieser das zuverlässige Publishing via PubSub durch einen `post-save` Hook auf die „Angebote“ Tabelle der Datenbank. Basierend auf der im Angebot enthaltenen Rasse, kann die Recommender Cloud Function dann eine Liste der abonnierten Benutzer vom App-Microservice abrufen. In Produktion sollte diese doppelte Kopplung zum App-Microservice vermieden werden. Diese kann durch eine replizierte read-only Datenbank Instanz oder einer speziell angepassten Datenbank-View durchbrochen werden. In jedem Fall kann die Cloud Function anschließend die abonnierten Nutzer per Chat-Microservice (siehe Unterabschnitt 5.4) benachrichtigen. Durch das Publizieren des gesamten neuen Angebots kann eine umfangreiche Nachricht (Name, Bilder, URL, etc.) an die Nutzer konstruiert werden.

7 Security

Bei der Implementierung wurde darauf geachtet, dass sowohl Betreiber als auch Nutzer gegen die gängigsten Angriffe geschützt sind. Das Abhören, Manipulieren oder Verletzen der Integrität von Daten darf nicht möglich sein. Im Folgenden werden entsprechende Lösungen präsentiert.

7.1 Verschlüsselung der Kommunikation

Eine unverschlüsselte Kommunikation zwischen Nutzer und Webserver bietet Angriffen wie Man-in-the-Middle Attack, Cross-Site-Scripting (XSS) und Session Hijacking ein breites Spielfeld. Wir nutzen ausschließlich HTTPS, wodurch sämtliche Kommunikation verschlüsselt vonstatten geht und ein Angreifer keine Informationen extrahieren kann, ohne die dem Zertifikat zugrunde liegende Public-Key Kryptographie zu brechen.

7.2 Authentifizierung und Autorisierung

Die Nutzer-Authentifizierung wurde ausführlich in Unterabschnitt 3.2 und Unterabschnitt 4.3 erklärt. Sämtliche Kommunikation mit den im Internet exponierten Services erfordert somit das Ausweisen der Identität, indem die von uns erstellen JWTs der jeweiligen Anfrage beigelegt werden müssen. Die Dauer der Gültigkeit der Access-Token ist frei konfigurierbar, standardmäßig jedoch auf 5 Minuten gesetzt. Auch die Authentifizierung unter den Services ist geregelt und erfordert spezielle Service Tokens. Jeder Microservice verwaltet hierzu eigenständig eine Whitelist und kann den Tokens darin eine Identität zuschreiben, wie beispielsweise der des Recommender Bot.

Auch die Autorisierung, also die Festlegung, ob ein gewisser Nutzer eine bestimmte Aktion ausführen darf, liegt im Verantwortungsbereich der Microservices. So darf beispielsweise kein Nutzer im Namen eines anderen Nachrichten versenden oder Angebote hochladen.

7.3 Weitere Lösungen

Weitere Mechanismen zur Gewährleistung der Sicherheit sind:

- Datenbankzugriffe erfolgen im App-Microservice ausschließlich über Django Queries. Diese sind vor SQL-Injektion geschützt, da ihre Abfragen mit Hilfe von Query-Parametrisierung konstruiert werden. Der SQL-Code einer Abfrage wird getrennt von den Parametern der Abfrage definiert. Da Parameter vom Benutzer bereitgestellt werden können und deshalb als unsicher gelten, werden sie vom zugrunde liegenden Datenbanktreiber escaped. [7]
- Im App-Microservice wurde eine Middleware eingerichtet, die Schutz gegen Cross-Site Request Forgery (CSRF) bietet.
- Da grundsätzlich kein User-Input in das Dateisystem einfügt wird, können Directory Traversal-Angriffe nicht stattfinden.

8 Continuous Integration und Delivery

Beim Build und Deployment der verschiedenen Komponenten unserer Anwendung haben wir den Ansatz der Continuous Integration sowie der Continuous

Delivery (CI/CD) verfolgt. Das bedeutet, neu eingecheckter Code wird kontinuierlich gegen existierenden geprüft, indem hierfür, sofern vorhanden, jedes Mal eine extensive Test Suite durchlaufen wird. Fehlerhafte Builds werden somit markiert und Pull Requests frühzeitig blockiert. Daneben lassen sich in diesem Integrations-Verfahren auch andere Schritte wie das Prüfen auf einen Styleguide oder Linter kombinieren. Basierend auf diesen ersten Qualitäts-Indikatoren, lässt sich eine Pipeline zur automatischen, kontinuierlichen Bereitstellung des neu eingecheckten Codes einrichten. Abbildung 4 zeigt diesen CI/CD Prozess für den App-Microservice.

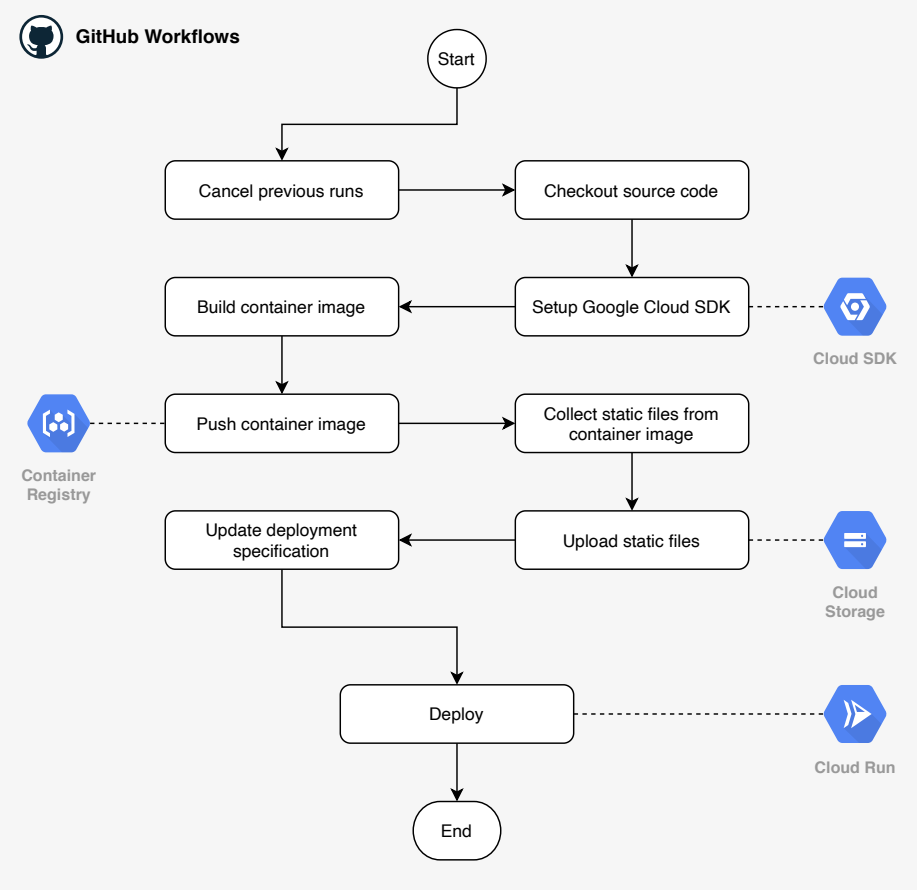

Abbildung 4: GitHub Workflow

Mit GitHub Actions können beide dieser Konzepte umgesetzt werden. Denn neben der Git-basierten Versionierung von Code oder kollaborativen Tools wie Issues, Projects, Teams etc., bietet GitHub neuerdings einen kostenlosen Dienst an, um CI/CD an die Breite Masse seiner Open-Source Entwickler zu bringen [1].

Die dafür notwendige Konfiguration wird samt Quellcode verwaltet. Allgemein tun wir das auch mit sonstigen Build Artefakten wie z.B. der Dockerfile oder Deployment Spezifikation eines jeden Microservices. Hiermit erstellen wir eine vollumfängliche Übersicht zu jeder Version der Software, inklusive der Umgebung und Beschreibung wie diese Software gebaut und bereitgestellt wurde.

9 Kostenanalyse

9.1 Annahmen zum Nutzerverhalten

Wir treffen die Annahme, dass es im Wesentlichen zwei Nutzergruppen gibt. Eine Gruppe nutzt die Anwendung vor allem zum Stöbern und Kaufen („Interessenten“), die andere nutzt sie zum Anbieten von Tieren („Verkäufer“). Ein kleiner Anteil der Benutzer wird in beiden Rollen agieren. So kommen wir zu der Annahme, dass ein Benutzer im Durchschnitt zwei Inserate zu jedem Zeitpunkt online hat. Weiterhin gehen wir davon aus, dass jedes Angebot im Durchschnitt zwei Bilder und ein Video beinhaltet, was bei einer Größe von 2 MB pro Bild und 16 MB pro Video zu einem Gesamtspeicherplatz von etwa 20 MB pro Angebot führt.

Um den Datenverkehr zu berechnen, gehen wir davon aus, dass ein registrierter Benutzer pro Tag ein Angebot aufruft sowie fünf mal die Startseite besucht. Diese Zahl haben wie etwas höher angesetzt, da auch nicht registrierte Benutzer die Startseite aufrufen können und die Anzahl dieser Aufrufe schwierig vorherzusagen sind.

Tabelle 1 fasst diese Annahmen zusammen.

Anzeigen pro Benutzer	2
Speicherplatz pro Anzeige	20 MB
Datenverkehr pro Nutzer pro Monat	600 MB
Anzahl Likes pro Anzeige und Benutzer	2
Anzahl Chats pro Benutzer pro Monat	2
Anzahl Nachrichten pro Chat	6
Größe einer Nachricht	20 KB

Tabelle 1: Annahmen zum Nutzerverhalten

9.2 Auswertung

Basierend auf den Werten in Tabelle 1 kann man mit Hilfe des Google Cloud Platform Preisrechners eine Kostenschätzung für die in unserem Projekt verwendeten Cloud Services erhalten [5].

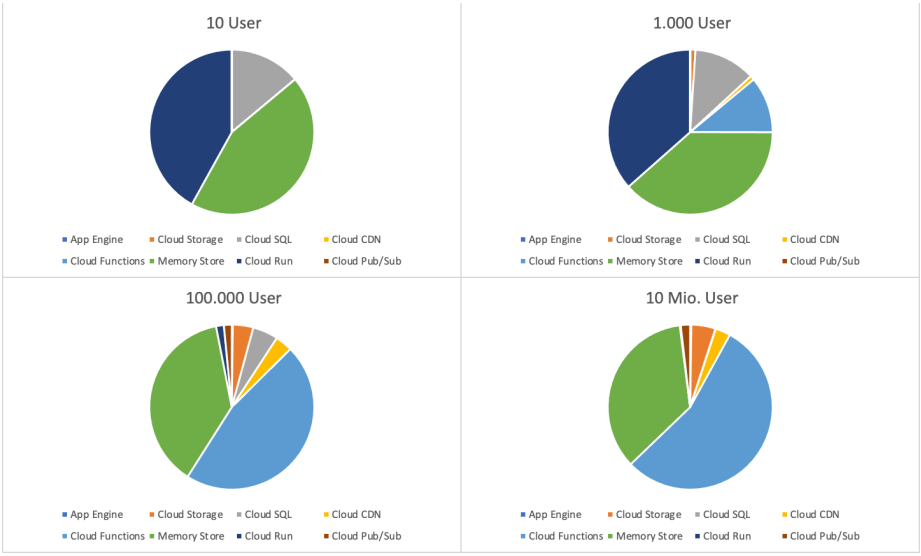

Abbildung 5: Kostenanteil der einzelnen Services

Abbildung 5 zeigt den Kostenanteil der Cloud Services im Verhältnis zu den Gesamtkosten. Hier ist besonders auffällig, dass der Cloud Memory Store bei jeder Nutzerzahl einen ähnlichen Anteil hat, während die Cloud Function mit steigender Anzahl der Benutzer einen größeren Anteil an den Gesamtkosten hat. Diese Verteilung liegt an der unterschiedlich starken Skalierung der Dienste.

	10 User	1K User	100K User	10M User
Gesamtkosten	72,62 €	83,37 €	2.088,48 €	177.059,05 €
Gesamtkosten pro User	7,26 €	0,08 €	0,02 €	0,02 €

Tabelle 2: Gesamtkosten

Um aussagekräftige Werte zu den Gesamtkosten zu erhalten, haben wir in Tabelle 2 neben den Gesamtkosten zusätzlich die Kosten pro Benutzer in der jeweiligen Gruppe berechnet. Hier ist deutlich erkennbar, dass die Kosten pro Benutzer mit steigender Anzahl abnehmen. Das liegt vor allem daran, dass gewisse Cloud-Dienste unabhängig von der Anzahl der Benutzer, also als Fixpreis, berechnet werden. Mit steigender Zahl der Benutzer sinken somit die Gesamtkosten pro Benutzer.

Literatur

[1] Automate your workflow from idea to production. https://github.com/features/actions. Accessed: 2021-02-21.

[2] Cloud bigtable. https://cloud.google.com/bigtable/?hl=de. Accessed: 2021-02-21.

[3] Cloud firestore data model. https://firebase.google.com/docs/firestore/data-model. Accessed: 2021-02-21.

[4] Gateways. https://docs.nestjs.com/websockets/gateways. Accessed: 2021-02-21.

[5] Google cloud pricing calculator. https://cloud.google.com/products/calculator. Accessed: 2021-02-21.

[6] Memorystore. https://cloud.google.com/memorystore?hl=de. Accessed: 2021-02-21.

[7] Security in django. https://docs.djangoproject.com/en/3.1/topics/security/. Accessed: 2021-02-14.

[8] Websocket. https://developer.mozilla.org/de/docs/Web/API/WebSocket. Accessed: 2021-02-21.

[9] What is pub/sub? https://cloud.google.com/pubsub/docs/overview. Accessed: 2021-02-21.

[10] ws: a node.js websocket library. https://github.com/websockets/ws. Accessed: 2021-02-21.

[11] Eliot Estep. Mobile html5: Efficiency and performance of websockets and server-sent events. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden, June 2013.

[12] John R Fallows, Frank J Salim, David B Gaunce, and Siddalingaiah Eraiah. Enterprise client-server system and methods of providing web application support through distributed emulation of websocket communications, October 2016. US Patent 9,459,936.

[13] Ian Fette and Alexey Melnikov. The websocket protocol. Request for Comments 6455, Internet Engineering Task Force, December 2011.

[14] Roy Fielding and Julian Reschke. Hypertext transfer protocol (http/1.1): Message syntax and routing. Request for Comments 7230, Internet Engineering Task Force, 2014.

[15] Manish Gupta, Natwar Modani, and Parul A Mittal. Event-triggered notification over a network, July 2004. US Patent 6,763,384.

[16] Annika Hinze, Kai Sachs, and Alejandro Buchmann. Event-based applications and enabling technologies. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pages 1–15, Nashville, TN, USA, July 2009.

[17] Salvatore Loreto, P Saint-Andre, Sd Salsano, and G Wilkins. Known issues and best practices for the use of long polling and streaming in bidirectional http. Request for Comments 6202, Internet Engineering Task Force, April 2011.

[18] Jean-Louis Maréchaux. Combining service-oriented architecture and event-driven architecture using an enterprise service bus. *IBM developer works*, pages 1269–1275, April 2006.

[19] Brenda M Michelson. Event-driven architecture overview. Technical report, Patricia Seybold Group Research Service, February 2006.