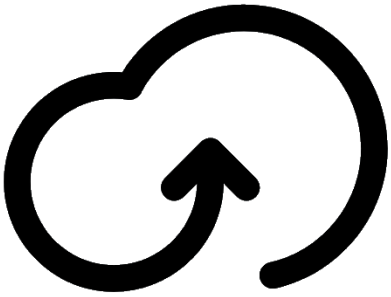


STUDENT SHARE

SHARE. NETWORK. HELP

Cloud Computing

Wintersemester 2022/2023

Hochschule **RheinMain**

Fachbereich Design Informatik Medien
Studiengang Informatik (Master of Science)

Projektabgabe: 17. Januar 2023

Gruppe: Jörn Bachmeier, Sebastian Braun, Sandra Kiefer
Dozent: Prof. Dr. Philipp Schaible

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

"Wissen ist die einzige Ressource, welche sich durch Gebrauch vermehrt." - Probst et al

Um dieses Zitat umzusetzen, wurde das Projekt "StudentShare" ins Leben gerufen. *StudentShare* ist eine Plattform für Studenten der Hochschule RheinMain, auf der sie ihre kompletten Unterrichtsmaterialien mit ihren Kommiliton:innen teilen können. So können Studenten auf mehr Ressourcen wie Mitschriften, Skripte oder Altklausuren von aktuellen oder ehemaligen Kommiliton:innen zugreifen und durch deren Gebrauch ihr Wissen erweitern.

An der Hochschule RheinMain gibt es viele verschiedene Plattformen, auf denen Dozenten ihre Materialien bereitstellen können. Jedoch gibt es keine Plattform, auf der auch Studenten ihre Materialien veröffentlichen können. Um den Informationsaustausch zwischen Kommiliton:innen zu fördern, möchten wir mit *StudentShare* eine zentrale Plattform bereitstellen, auf der nicht nur Materialien bereitgestellt werden, sondern durch einen Chat auch Nachrichten jahrgangsübergreifend ausgetauscht und Materialien angefragt werden können.

StudentShare soll als Web-Scale-Anwendung in der Google Cloud bereitgestellt werden und der Microservice-Architektur entsprechen. Teil der Anwendung müssen diverse Google Cloud Platform Produkte sein. Im Backend sollen eine App Engine, Cloud Functions und ein Containersystem wie Cloud Run oder eine Kubernetes Engine verwendet werden. Zur persistenten Datenspeicherung müssen eine SQL- und NoSQL-Datenbank sowie ein Cloud Storage für Multimedia-Daten zum Einsatz kommen. Zur sicheren Datenverarbeitung wird ein Identitätsmanagement mit spezieller Zugriffsverwaltung gefordert.

1.2 Anwendungsszenarien

Für Studenten, die neu an der Hochschule RheinMain sind, kann es teilweise sehr mühsam sein, sich zusätzliche Informationen zu einem belegten Modul zu beschaffen. Kontakt zu ehemaligen Studierenden oder Studenten aus höheren Semestern ist ohne weitere Hilfsmittel sehr schwierig. Auch fehlt die Information, ob und wo man sich Materialien wie Altklausuren oder Mitschriften beschaffen kann. Diesem Problem entgegenzuwirken, wurde die Plattform *StudentShare* ins Leben gerufen.

Sie soll eine zentrale, von Studenten gemanagte Anlaufstelle für Studenten der Hochschule RheinMain darstellen und das Leben aller Studenten vereinfachen.

Materialien wie Altklausuren können bei der Klausurvorbereitung zu einer besseren Note und zu einem sicheren Gefühl beitragen. Sie sind für Studierende ein wichtiges Element in der Vorbereitung, aber trotzdem oftmals nur schwer zugänglich. Möchte sich ein Student nun also Informationen zu einem Modul bei *StudentShare* beschaffen, muss er sich mit seiner Hochschul-E-Mail-Adresse auf der Plattform anmelden. Sollte noch keine Anmeldung durchgeführt worden sein, wird die Person durch den Anmeldeprozess geleitet. Im Idealfall sollte die E-Mail-Adresse mit dem Identity Access Management der Hochschule RheinMain gegengecheckt werden. Ist die E-Mail-Adresse im System hinterlegt, kann der User anschließend die Dateien seines Fachbereiches einsehen. Sollte sich der User zuvor bereits angemeldet haben, wird er nach erfolgreicher Authentisierung zur Startseite des Fachbereiches weitergeleitet. Mit den benötigten Berechtigungen kann er diese Dateien dann herunterladen und eigene Dateien seinen Kommiliton:innen zur Verfügung stellen.

Über den zentralen Chat sind alle gleichzeitig angemeldeten User miteinander verbunden. Hier können Informationen zu bisher gestellten Fragen nachgelesen und neue Fragen an alle User der Plattform gestellt werden. Auf diese Weise ist ein offener Austausch unter der Studierendenschaft gewährleistet und die User können sich gegenseitig unterstützen. Chatnachrichten werden in Echtzeit an alle angemeldeten User übermittelt und im Chat dargestellt, sodass eine schnelle Reaktion erfolgen kann.

Sobald ein User die Webseite schließt, wird er automatisch abgemeldet und seine Verbindung zum Chat wird getrennt. Alle Nachrichten und Dateien sind in den Datenbanken der Anwendung gespeichert und werden nach erneuter Anmeldung wieder zur Verfügung gestellt.

1.3 Anforderungen

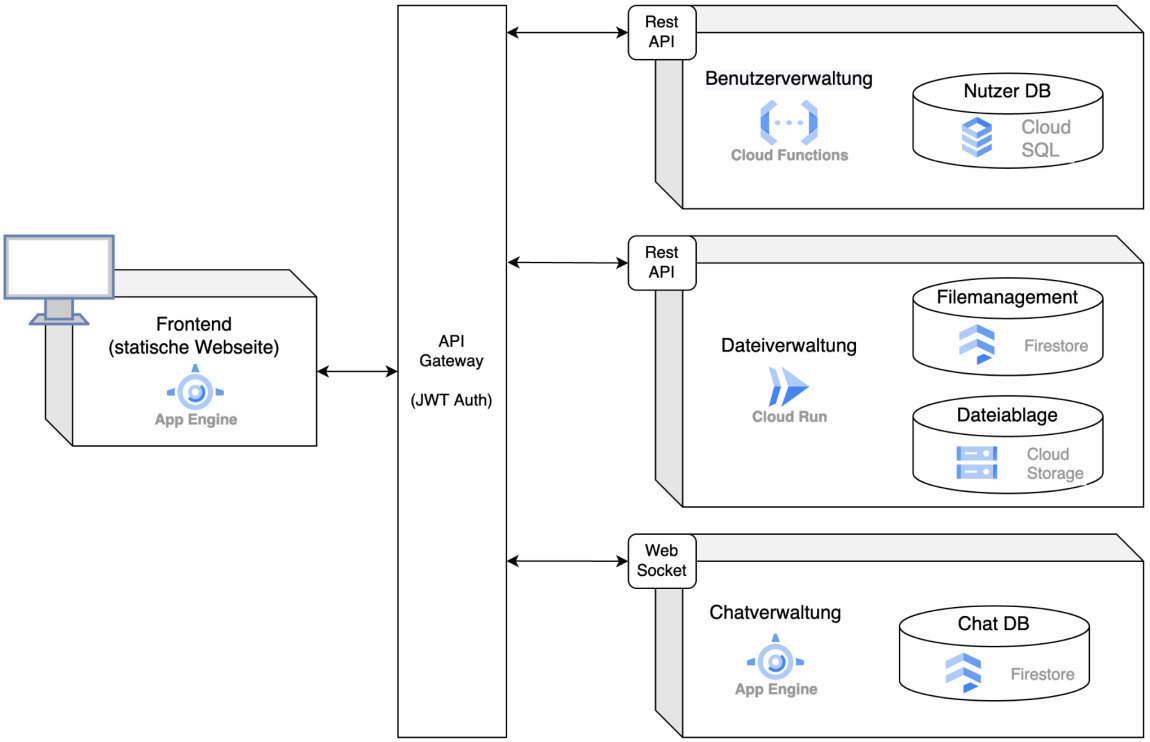
Vor Projektentwurf war bereits klar, welche technischen Anforderungen in diesem Projekt umgesetzt werden sollen. Die Herausforderung war es, die geforderten Technologien sinnvoll einzusetzen und in einer Anwendung zu vereinen. Basis für die Anwendung sollen diverse Cloud Computing Technologien der Google Cloud Platform sein.

Startpunkt der Anwendung muss eine statische Webseite sein, welche auf verschiedene REST APIs im Backend zugreifen soll. Das Backend soll nach dem Prinzip der Microservice-Architektur aufgebaut werden und die Services App Engine, Cloud Functions und eine Containerinstanz wie Cloud Run oder der Kubernetes Engine enthalten. Diese Services sollen ihre Daten in drei verschiedenen Datenbanktypen speichern, denn es sollen nicht nur organisatorische Daten der Anwendung gespeichert werden, sondern auch Multimedia-Daten wie Bild, Audio und Video. Zum Einsatz müssen also eine SQL- und eine NoSQL-Datenbank sowie ein Cloud Storage für die Multimedia-Daten kommen.

Die Anwendung muss durch ein Sicherheitskonzept mit Benutzerverwaltung, Authentifizierung und Autorisierung abgesichert werden. Zusätzlich muss sie resistent gegenüber den meisten Cyberangriffen sein und ihre Ressourcen selbstständig entsprechend dem aktuellen Traffic skalieren.

Wie diese Anforderungen in *StudentShare* vereint wurden, wird im nachfolgenden Kapitel über die Architektur unserer Anwendung genauer erläutert.

2 Architektur



StudentShare ist nach dem Prinzip der Microservice-Architektur aufgebaut. Die Anwendung besteht aus mehreren kleineren Einheiten. Zu diesen zählen die Microservices Benutzerverwaltung, Dateiverwaltung und Chatverwaltung, sowie ein statisches Frontend und ein API-Gateway. Jede Instanz kann unabhängig voneinander deployed und verwaltet werden und reduziert so den Wartungsaufwand der Software. Zudem können die Komponenten unabhängig voneinander ausgetauscht und erweitert werden. Die "Services können als Prozesse definiert werden, die häufig über ein Netzwerk unter Verwendung von technologieunabhängigen Protokollen kommunizieren"[1]. Die verwendeten Protokolle sind REST und die auf TCP basierende Technologie der Websockets. Jeder Service kann einzeln entwickelt und betrieben werden, was in der Entwicklungsphase auch so durchgeführt wurde.

Das Frontend bildet mit der statischen Webseite die Benutzeroberfläche der Anwendung. Von der Webseite aus werden alle Microservices aufgerufen, um die Funktionalität der Webseite anzusprechen und die Seite mit Daten zu füllen.

Beim ersten Aufruf der Seite gelangt man zum Login-Bildschirm, auf dem man sich mit der Google Third-Party-Authentifizierung autorisieren muss. Ist die E-Mailadresse im System hinterlegt, wird die Cloud Functions der Benutzerverwaltung aufgerufen. Dort wird der User mit seinen Daten in der

[1] https://www.leanix.net/de/wiki/vsm/microservices-architecture

SQL-Datenbank abgespeichert und erhält als Antwort einen JWT-Token, der in den Cookies des Browsers gespeichert wird. Dieser Token wird innerhalb der Anwendung für jede Anfrage an die Microservices für die Autorisierung mitgesendet.

In der Regel wird jede Anfrage vom API-Gateway bearbeitet. Dort wird zuerst das JWT-Token validiert und anschließend die Anfrage an den jeweiligen Service weitergeleitet und über den Service Account autorisiert. Ausnahme ist der Chat Microservice, da die Websockettechnologie (noch) nicht mit dem API-Gateway von Google kompatibel ist[2].

Dort findet die Validierung des JWT-Tokens im Microservice selbst statt. Ist das Token gültig, wird eine Verbindung zum Websocket-Server aufgebaut. Wird eine Nachricht abgeschickt, wird das Chatobjekt in der Firestore NoSQL Datenbank gespeichert. Anschließend wird die Nachricht an alle Clients gesendet, die aktuell mit dem Server verbunden sind.

Die Hauptfunktionalität der Anwendung liegt in der Dateiverwaltung. Ist der User angemeldet, wird über das API-Gateway der Microservice der Dateiverwaltung aufgerufen. Dieser läuft in einem Container in Cloud Run. Im Hintergrund arbeitet der Service mit zwei Datenbanken, um die Performance zu erhöhen und die Kosten der Aufrufe zu minimieren. Die Informationen der hochgeladenen Dateien werden als Key-Value-Paare in der Firestore NoSQL-Datenbank gespeichert. Die eigentlichen Dateien werden in einem Storage Bucket gespeichert und nur für den tatsächlichen down- oder upload der Datei aufgerufen. So werden die teureren Abrufe auf den Cloud Storage reduziert. User können ausschließlich Dateien anzeigen, die für ihre Fachbereiche freigegeben wurden, welche bei der erstmaligen Anmeldung ausgewählt werden mussten. Im Umkehrschluss bedeutet dies, dass Dateien, die der User hochlädt, auch nur für den gewählten Fachbereich sichtbar sind.

Die Infrastruktur der Anwendung kann durch Infrastructure-as-Code auf der Google Cloud Platform automatisiert hoch- oder runtergefahren werden. Dazu wird ein zentrales Shell-Skript gestartet, welches auf mehrere darunterliegende Skripte verweist, die jeweils eigenständig einen Service hoch- oder herunterfahren.

[2] https://issuetracker.google.com/issues/176472002?pli=1

3 Implementierung

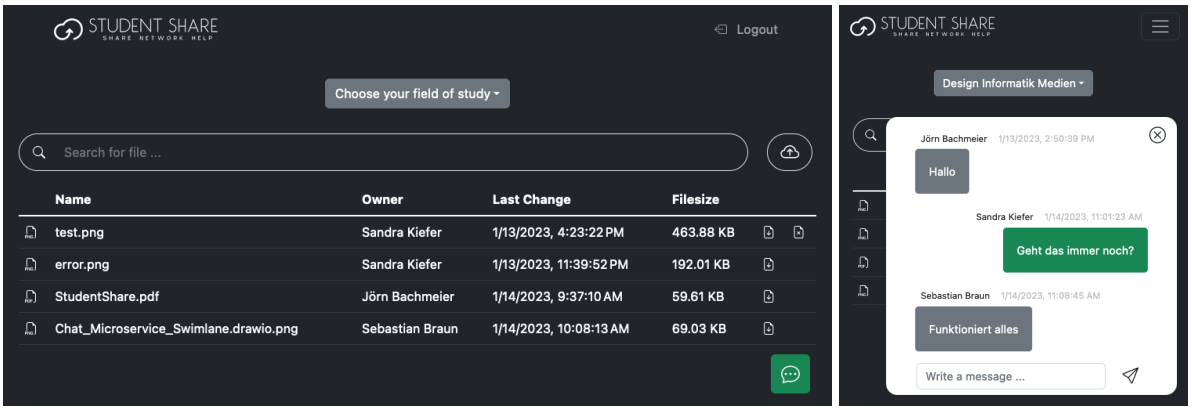
3.1 Statische Webseite

Das Frontend der Webanwendung ist als Single-Page-Application mit dem Framework Vue.js realisiert. Der statische Export der Webseite wird über ein Content Delivery Network (CDN) oder eine Google App Engine (GAE) den verschiedenen Clients zur Verfügung gestellt. Da für das Hosten der Webseite über das CDN eine Top-Level-Domain benötigt wird und diese nicht in der kostenlosen Testphase inbegriffen ist, nutzen wir die in der kostenlosen Testversion enthaltene GAE. Je nach Aufrufzahlen kann das Hosten der statischen Webseite in der GAE weniger kosten, da eine kostenlose Stufe der GAE bereitgestellt wird und dies bei herkömmlichen Hosting-Anbieter nicht der Fall ist.[3]

Die Webseite ist in zwei Pfade mit unterschiedlichen Ansichten unterteilt. Ruft der Nutzer die Webseite erstmals auf, gelangt er auf den Login-Screen mit einem Button für die Third-Party Authentifizierung und Autorisierung mit einem beliebigen Google Account des Nutzers. Nach erfolgreicher Authentifizierung und Autorisierung des Google-Accounts öffnet sich ein weiterer Dialog mit der Aufforderung, die angehörigen Fachbereiche (fundieren als Rollen des Nutzers) zu selektieren. Dies ist nur beim erstmaligen Anmelden auf der Webseite nötig, da diese zusätzlichen Informationen des Fachbereichs in späteren Schritten benötigt werden. Das vom Backend übergebene JSON Web Token (JWT) wird in den Cookies des Browsers gespeichert und bei jeder nachfolgenden Anfrage zur Autorisierung mitgeschickt.

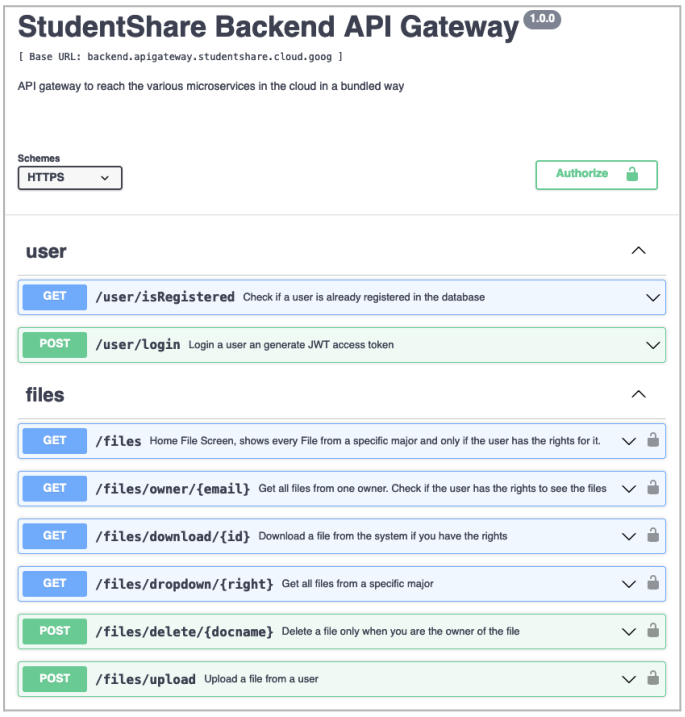
Anschließend wird der Nutzer auf den File-Screen weitergeleitet. Dort werden alle Dateien angezeigt, auf die der Nutzer Zugriff hat. Mithilfe des Dropdown kann der Nutzer sich nur die Dateien des jeweiligen Fachbereichs anzeigen lassen. Sind sehr viele Dateien hochgeladen, können über eine Suchleiste die Dateien gefiltert werden oder nach verschiedenen Kriterien die Dateien in der Liste sortiert werden. Zusätzlich ist das Hoch- und Herunterladen von Dateien möglich. Jedoch kann ein Nutzer nur die Dateien löschen, die ihm gehören und selbst hochgeladen worden sind. Zusätzlich kann auf der Webseite ein Chat aufgeklappt werden, über den die Nutzer sich austauschen und nach Hilfestellungen oder Dateien fragen können. Des Weiteren kann der Nutzer sich ausloggen, wird wieder zum Login-Screen zurückgeleitet und das JWT wird aus den Cookies im Browser gelöscht.

[3] https://cloud.google.com/appengine/docs/legacy/standard/python/getting-started/hosting-a-static-website



3.2 API-Gateway

Das API-Gateway sitzt zwischen dem Client und den verschiedenen Microservices im Backend (Benutzerverwaltung, Dateiverwaltung, Chatverwaltung). Das Gateway nimmt alle API-Aufrufe des Clients entgegen und gibt diese an die entsprechenden Services weiter.



Bei dem Erstellen des Gateways in der Google Cloud wird ein Service-Account hinterlegt, welcher zur Autorisierung der Anfrage an das Backend genutzt wird. Die entsprechenden Microservice sind nicht öffentlich verfügbar, erlauben jedoch nach entsprechender Autorisierung den Zugriff auf den Microservice mithilfe des Service-Accounts des Gateways. Dem Service-Account müssen dazu die benötigten Rollen und Zugriffsrechte erteilt werden. Somit sind die Microservices nur über das Gateway öffentlich im Internet zu erreichen.

Zusätzlich sind einige Pfade mit JWT gesichert, auf welche nur registrierte und angemeldete Nutzer zugreifen sollen (Datei- und Chatverwaltung). Das JWT wird vom Client an die Anfrage ans Gateway angehängt. Das Gateway validiert die Signatur des Tokens mit dem öffentlichen Schlüssel. Dieser Schlüssel wird zuvor von der Benutzerverwaltung beim Anmelden des Nutzers festgelegt und das Token mithilfe des privaten Schlüssels erstellt. Nur wenn ein gültiges JWT mitgeschickt wird, leitet das Gateway die Anfrage an den entsprechenden Microservice weiter.

Leider ist es momentan (aktueller Stand: Januar 2023) noch nicht möglich, einen Websocket über das Google API-Gateway bereitzustellen. Aus diesem Grund übernimmt der Websocket in diesem Projekt die Überprüfung der JWT auf deren Korrektheit beim Verbinden eines neuen Clients selbst. Sendet der Client ein ungültiges JWT, wird keine Verbindung zum Websocket aufgebaut.

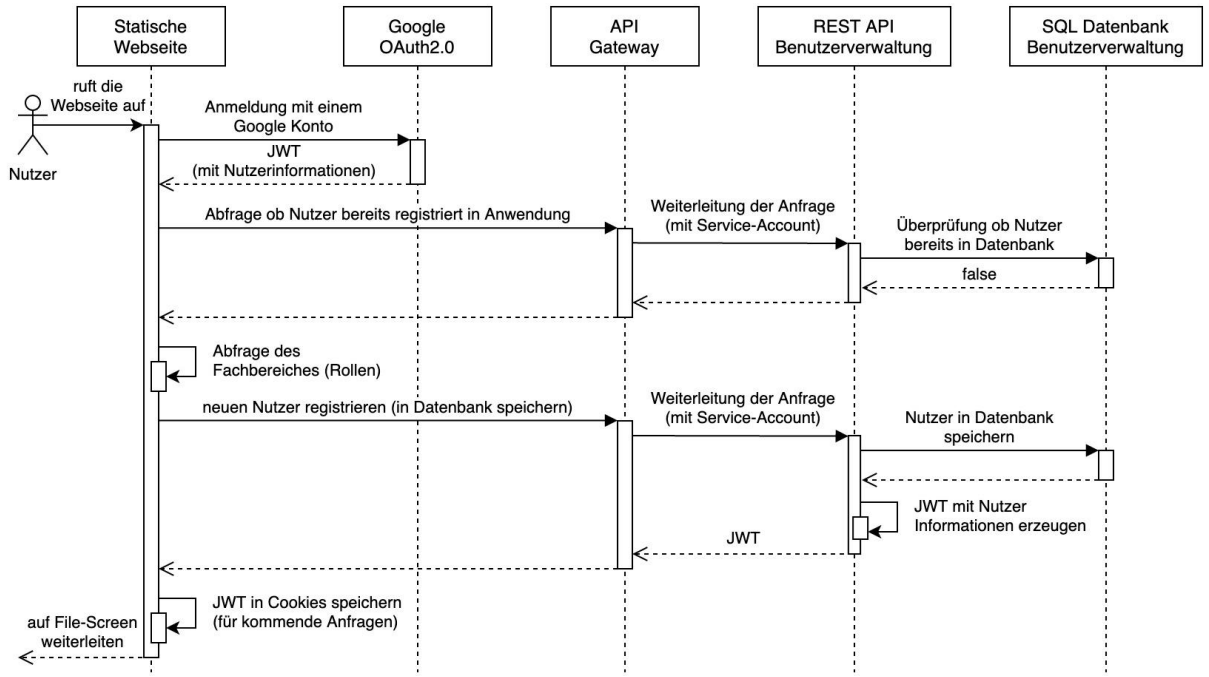
3.3 Benutzerverwaltung

Bei der Realisierung des Microservices Benutzerverwaltung haben wir uns für Google Cloud Functions entschieden. Dabei handelt es sich um ein Functions as a Service (FaaS) Produkt, welches ohne Server oder Container ausführbar ist und automatisch nach der jeweiligen Arbeitslast skaliert. Grundsätzlich zahlt man je Aufruf der Funktion und nur solange, wie die Funktion ausgeführt wird. Zusätzlich sind die ersten zwei Millionen Aufrufe pro Monat dabei kostenlos.[4] Google Cloud Run besitzt ebenfalls diese kostenlose Stufe, jedoch handelt es sich dabei um eine Container verwaltete Plattform.[5] Aufgrund der technischen Anforderungen des Projekts und dem Benutzen möglichst vieler Technologien der Google Cloud haben wir uns bei der Dateiverwaltung für Cloud Run und bei der Benutzerverwaltung für Cloud Functions entschieden.

Die Benutzerverwaltung ist hauptsächlich für das Speichern und Verwalten der Nutzer in der Datenbank und das Erzeugen von JSON Web Token (JWT) verantwortlich. MySQL wird dabei als relationales Datenbank genutzt, da es sich bei den zu speichernden Nutzerinformationen um strukturierte und konsistente Daten handelt. Zudem kommt hinzu, dass nur beim Anmelden des Nutzers einmalig auf die Datenbank zugegriffen werden muss und danach die benötigten Informationen im JWT mitgespeichert und mitgeschickt werden. Würde es sich um größere Datensätze mit deutlich mehr Leseabfragen handeln, wäre es sinnvoll eine nicht-relationale Datenbank zu verwenden.[6] Die Tabelle der Nutzer in der Datenbank besitzt die Felder E-Mail (Primärschlüssel), Name und Fachbereiche (eine Liste in Form eines mit Komma getrennten Strings).

Um die Funktionsweise der Benutzerverwaltung besser erklären zu können, gehe ich im Folgenden die einzelnen Schritte durch, wenn sich ein Nutzer das erste Mal bei der Webanwendung anmeldet (Visualisierung der einzelnen Funktionsaufrufe im untenstehenden Sequenzdiagramm). Zu Beginn ruft der Nutzer die Webanwendung auf. Dort befindet sich ein Button für die Weiterleitung zur Anmeldung mit einem Google Account. Dazu wird der Nutzer auf einen Google OAuth 2.0 Endpunkt weitergeleitet und erteilt der Webanwendung die benötigten Berechtigungen. Ist die Autorisierungssequenz abgeschlossen, erhält der Client ein JWT von Googeln, welches die

[4] https://cloud.google.com/functions
[5] https://cloud.google.com/run
[6] https://datascientest.com/de/sql-vs-nosql-unterschiede-anwendungen-vor-und-nachteile



Accountinformationen des Nutzers beinhaltet.[7] Der Client schickt nun über das API-Gateway eine Anfrage an den Microservice Benutzerverwaltung, ob der Nutzer sich zum ersten Mal auf der Webanwendung anmeldet. Das heißt, es wird überprüft, ob seine Informationen in der Datenbank bereits gespeichert sind. Ist dies nicht der Fall, wird der Nutzer auf der Webseite aufgefordert, seine Fachbereiche anzugeben. Jetzt wird erneut über das API-Gateway eine Anmeldeaufforderung an das Backend geschickt, diese beinhaltet das von Google erzeugte JWT und die zuvor angegebenen Rollen (beziehungsweise Fachbereiche). In der Benutzerverwaltung wird zunächst die Signatur des JWT von Google auf ihre Gültigkeit überprüft und die Informationen des Nutzers herausgelesen. Anschließend wird der Nutzer mit seiner E-Mail, dem Namen und den Fachbereichen in der Datenbank gespeichert. Zuletzt wird ein neues JWT mit einem privaten Schlüssel mit dem RS256-Algorithmus asymmetrisch verschlüsselt. Dieses Token beinhaltet alle Informationen des Nutzers (siehe Beispiel eines decodierten JWT). Der Client speichert anschließend das Token mit den Informationen in den Cookies des Webbrowsers. Bei allen folgenden Anfragen wird im Header das Token mitgeschickt. Das JWT wird anschließend vom API-Gateway auf seine Gültigkeit überprüft und nur nach erfolgreicher Überprüfung an den entsprechenden Microservice im Backend weitergegeben.

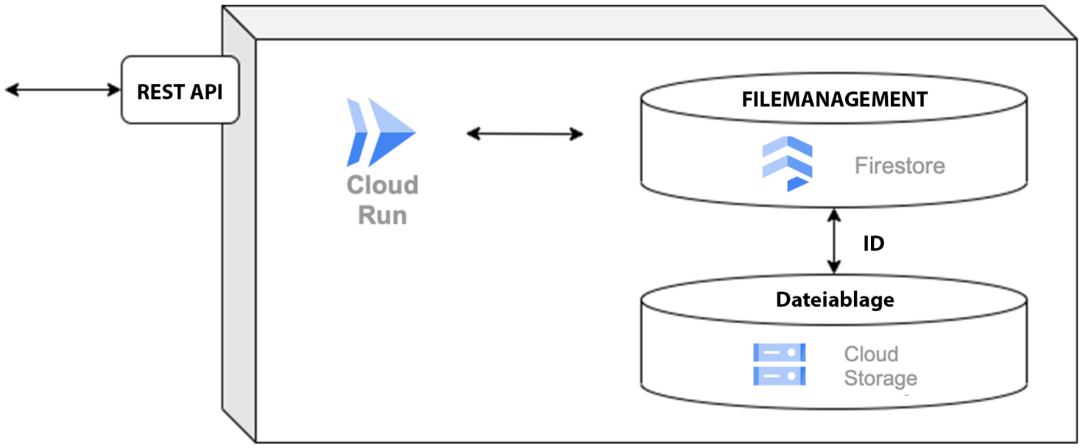
```
{
    alg: "RS256",
    typ: "JWT"
}.
{
    email: "sandra.kiefer.dev@gmail.com",
    sub: "sandra.kiefer.dev@gmail.com",
    name: "Sandra Kiefer",
    courses: "Design Informatik Medien",
    iss: "StudentShare",
    aud: "https://website-dot-studentshare.ey.r.appspot.com",
    iat: 1673778739,
    exp: 1673865139
}.
[signature]
```

[7] https://developers.google.com/identity/protocols/oauth2

3.4 Dateiverwaltung

Der File Microservice kümmert sich um alle Files der angemeldeten User und läuft über Cloud Run in einem Node JS Container in der Cloud. In diesem Container läuft ein Express.js Server, der über eine Rest Schnittstelle von dem API-Gateway angesteuert wird und alle Aufgaben der File Verwaltung übernimmt.

Die Files werden mit allen nötigen Informationen in zwei unterschiedlichen Datenbanken gespeichert:



Das File Objekt an sich wird in einem Google Cloud Storage Bucket gespeichert. Im Cloud Storage Umfeld sind Buckets eine Art Container, in dem Daten in Form von Objekten gespeichert werden. Im Gegensatz zu üblichem Storage erfolgt die Ablage nicht in Ordnerstrukturen und Dateien. Das Verschachteln ist nicht erlaubt. Der Bucket hat einen global eindeutigen Namen und ist über eine URL erreichbar. Die gespeicherten Objekte haben eindeutige IDs oder auch Dateinamen in dem Bucket. Um eine Datei herunterzuladen, benötigt es nur den Namen des Buckets und des Objektes. Deshalb muss dem Anwender nicht bekannt sein, wo der Standort des physischen Speichers ist. Außerdem ist die Anzahl an Objekten, die in einem Google-Bucket gespeichert werden können, unbegrenzt und damit ideal geeignet für seine Aufgabe in diesem Projekt.

Damit der User Informationen über die Dateien angezeigt bekommt und die Applikation weiß, welchem User er welche Daten anzeigen darf, braucht es weitere Informationen über die Datei und den Ersteller. Diese stehen in einer Google Cloud NoSQL Firestore Datenbank, die pro hochgeladenes File ein Dokument abspeichert, das alle nötigen Informationen enthält. Ein Dokument besteht aus mehreren Feldern, denen Werte zugeordnet sind. Die Dokumente werden in Sammlungen gespeichert und dienen in diesem Projekt als Akte für das zugeordnete File im Bucket. Das hat den Vorteil, dass alle Informationen, die in der Applikation immer wieder Verwendung finden, dank der

Echtzeit-Listeners schnell zu erreichen sind. Zusätzlich können den Abfragen mehrere verkettete Filter enthalten sein, um File-Listen schnell zu erzeugen oder zu sortieren. Dadurch müssen teure Anfragen auf die Daten im Bucket nur getätigt werden, wenn ein User das File herunterladen oder löschen möchte.

In der Applikation werden viele Informationen angezeigt, die der User über die Daten wissen muss, wie zum Beispiel die Größe des Files, welchen Namen die Datei trägt und von wem und wann es hochgeladen wurde. Außerdem braucht die Applikationen Informationen über den Ersteller des Files, über die Sichtbarkeit und Rechte und die Verbindung zum eigentlichen File im Bucket. Das Dokument und das abgelegte File im Bucket tragen dabei den gleichen eindeutigen Namen als ID, damit es zu keinen Verwechslungen kommt.

Um die Eindeutigkeit des Users zu garantieren, wird nach dem Einloggen seine E-Mail Adresse im JWT Token gespeichert. Auch seine Rechte bzw. eingetragenen Kurse stehen in dem Token. Dieses Token wird über das API-Gateway an das Backend geschickt und ausgelesen, um den angemeldeten User von den anderen zu unterscheiden und ihm ausschließlich die Files anzuzeigen, für die er registriert ist. Es wird darauf geachtet, dass Files nur in den Fachbereich hochgeladen und angezeigt werden, für die sie gedacht sind. Auch darf ein User nur die Dateien, die er selbst hochgeladen hat, wieder löschen, aber jede Datei, die ihm angezeigt wird, herunterladen. Dafür werden zu jeder Zeit die eingetragenen Kurse des Users benötigt.

Der File Microservice besitzt verschiedene Endpunkte, um alle diese Funktionen und Abhängigkeiten umzusetzen (siehe Kapitel 3.2 API Gateway):

Nachdem sich der User über Google angemeldet und seine Kurse ausgewählt hat, gelangt er auf die Startseite der App und bekommt alle verfügbaren Files seiner ausgewählten Kurse angezeigt. Dafür werden seine User Informationen über das JWT Token an den Microservice geschickt. Das Backend liest das Token aus und erzeugt anhand der Kursinformationen eine Liste an verfügbaren Files im Bucket über die Dokumente im Firestore. Diese Liste wird an das Frontend geschickt und nach Upload-Datum sortiert.

Da ein User für mehrere Kurse registriert sein wird, kann er mit dem Dropdown-Menü nach ihnen filtern. Über diesen Endpunkt enthält das Frontend eine Liste an verfügbaren Files für nur einen ausgewählten Kurs. Falls der User die hochgeladenen Files einer Person ansehen möchte, kann er auf den Namen dieser Person klicken. Das Frontend bekommt über diesen Endpunkt alle Files einer bestimmten Person, aber nur für die Kurse, in denen beide registriert sind. Sollte die ausgewählte Person auch Files für andere Kurse hochgeladen haben, bekommt der User nur die Dateien angezeigt, die zu seinen Kursen passen. Dafür ist es wichtig, dass sowohl die E-Mail Adresse als auch die Liste

der Kurse im JWT Token stehen, damit die ausgewählte Person auch eindeutig von den anderen Usern unterschieden werden kann.

Wenn der User eine Datei gefunden hat, die er herunterladen möchte, dann kann er dies mit jedem für ihn angezeigten File machen. Der Endpunkt zum Herunterladen der Files benötigt nur die Id der Datei, der Rest kommt über das JWT Token. Im Backend wird noch einmal geprüft, ob der User die richtigen Rechte für die Aktion hat und ob die zu herunter ladende Datei existiert. Ist das der Fall, wird das File über die ID im Dokument ausgewählt und für den User heruntergeladen. Sollte es nicht funktionieren, bekommt der User eine entsprechende Nachricht.

Hat der User selbst Dateien, die er für andere Studenten in seinem Kurs zur Verfügung stellen möchte, dann kann er das jederzeit tun. Über diesen Endpunkt werden alle Informationen geschickt, die für die Erstellung eines Bucket-Dokument Paars benötigt werden. Aus dem Frontend wird das zu hochladende File als “FormData” mit “Axios” an das Backend geschickt. Dieses macht mit “Multer” daraus ein Objekt des Types “File”.

Als Erstes wird der Filename gesichert, um es im Frontend anzuzeigen. Mit dem Filename wird danach eine eindeutige ID erzeugt, damit sowohl das Dokument als auch das Objekt im Bucket leicht und eindeutig zu erreichen sind. Hat der Bucket noch kein File mit diesem Namen, wird sich der Name gesichert und ein “WriteStream” geöffnet, um das File in den Bucket zu laden. Ist dies erfolgreich gewesen, wird das Dokument mit allen wichtigen Informationen erzeugt.

```
export interface File {
  filename: string;
  file_id: string;
  owner: string;
  last_change: string;
  fileSize: string;
  rights: Array<string>;
  password: string;
  email: string;
  public: boolean;
}
```

Aus den Metadaten des Files erhält das Dokument “*date, filesize, filename, file_id*”, aus dem JWT Token “*owner, email, rights*” und in späteren Versionen könnte es über das Frontend noch “*password und visibility*” bekommen. Alle diese Informationen werden zusätzlich zu dem hochgeladenen File getrennt gespeichert, um jederzeit einen schnellen Zugriff auf sie zu haben.

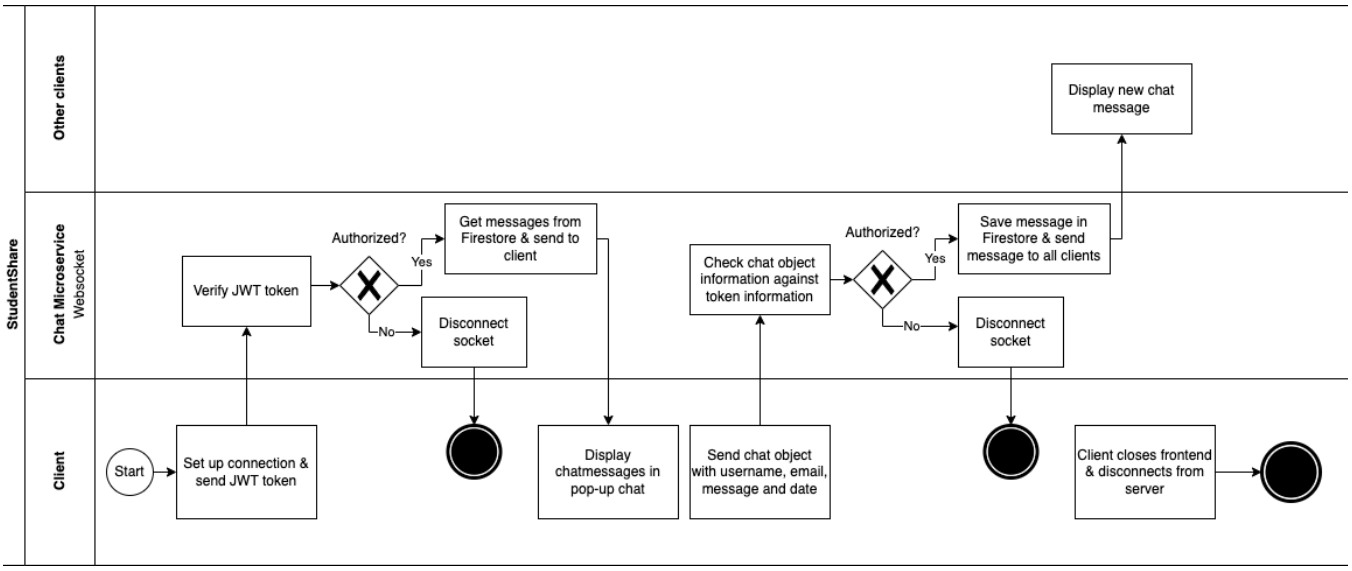
Möchte der User ein File wieder löschen, geht das nur, wenn die E-Mail Adresse des File Eigentümers und des Users übereinstimmen. Ist das der Fall, löscht das Backend das ausgewählte File über den Endpunkt mit der mitgegebenen ID aus dem Bucket und der Sammlung heraus und gibt dem Frontend einen Boolean über die Lage zurück.

In diesem Kapitel wurde erläutert, warum zwei unterschiedliche Datenbänke für diesen Microservice genutzt werden und welche Aufgabenbereiche abgedeckt werden. Im späteren Fazit wird besprochen, welchen Vorteil dabei das Aufteilen der Aufgaben in verschiedene Microservices hat.

3.5 Chatverwaltung

Für den Chat Microservice wird die Google App Engine verwendet, da für den Chat der Anwendung eine dauerhafte Websocketverbindung zu einem Server benötigt wird. In der App Engine läuft ein Express.js Server, der als Websocket-Server die Kommunikation zu den Clients übernimmt. Die Verbindung per Websocket ermöglicht es, neue Chatnachrichten ohne Verzögerung zu senden und zu empfangen. Für die Websockets wird die Libary Socket.io verwendet.

Der Chat Microservice ist mit der NoSQL-Datenbank Firestore verbunden, in der die Chatnachrichten mit Informationen über den User (Name + E-Mail) sowie die Uhrzeit der Chatnachricht gespeichert werden. Dies ermöglicht es, alle alten Nachrichten nach erfolgreicher Anmeldung im Frontend aus der Datenbank zu laden und diese im Chatverlauf anzeigen zu lassen. Eine NoSQL-Datenbank eignet sich für diesen Anwendungsfall sehr gut, da die Informationen jeweils als Key-Value-Paare gespeichert werden. Die Key-Attribute “Name”, “E-Mail”, “Date” und “Message” werden dann mit Informationen eines übermittelten Chatobjekts gefüllt und gespeichert. Mit Firestore lassen sich mehrere Tausend Chatobjekte pro Sekunde verarbeiten und speichern, sodass der Microservice fehlerfrei laufen kann.



Normalerweise müssen die Microservices nicht eigenständig die JWT-Token der User verifizieren. Der Chat Microservice ist jedoch ein Sonderfall. Im Regelfall übernimmt das API-Gateway die Authentifizierung und Autorisierung. Aufgrund der Inkompatibilität von Googles API-Gateway mit der Websocketverbindung muss der Chat Microservice dies eigenständig durchführen.

Zum Verbindungsaufbau wird vom Client das JWT-Token als Authentisierung im Header mitgeschickt. Im Backend wird dieses Token verifiziert. Sollte es ungültig sein, wird die Socketverbindung zum Client getrennt. Mit einem gültigen Token werden nun die Nachrichten im

Backend aus der NoSQL-Datenbank geladen und an den Client zum Frontend gesendet. Die Nachrichten werden dann nach Datum sortiert und im Pop-up-Chat dargestellt.

Sobald ein Client eine neue Nachricht sendet, wird ein Chatobjekt mit den Werten “Name”, “E-Mail”, “Date” und “Message” an den Server gesendet. Die Werte “Name” und “E-Mail” werden aus dem JWT-Token der aktuellen Session im Frontend an das Chatobjekt übergeben. Im Backend werden die Werte des Objektes dann gegen die Werte des beim Verbindungsaufbau mitgeschickten JWT-Tokens geprüft. Sollten diese Werte nicht identisch sein, wird die Socketverbindung zum Client getrennt. Bei identischen Werten wird das neue Chatobjekt in der NoSQL-Datenbank gespeichert und anschließend an alle anderen verbundenen Clients gesendet. Diese stellen die neue Chatnachricht dann in Echtzeit im Frontend dar.

Sollte ein Client die Verbindung zum Server verlieren oder die Webseite schließen, wird die Socketverbindung zum Server automatisch getrennt.

4 Analyse

4.1 Sicherheitskonzept

Die Webanwendung basiert auf dem Modell der Zero Trust Security. Darunter versteht man, dass sich alle Nutzer und Anwendungen gegenseitig grundsätzlich nicht vertrauen und einander immer authentifizieren müssen, beispielsweise mit JWT.

Jeder Microservice ist nach außen geschützt und kann nur mit einem zuvor festgelegten Service-Account erreicht werden. Dieser Service-Account ist im API-Gateway hinterlegt und wird zur Erzeugung eines Zugangstoken genutzt, welcher wiederum an die weitergeleitete Anfrage an den Microservice im Backend angehängt wird. Ist das Token ungültig oder wurde manipuliert, wird der Zugriff nicht gewährt.

Innerhalb der einzelnen Microservices ist der Zugriff auf die verschiedenen Datensicherungen (SQL, NoSQL, Cloud Storage) ebenfalls klar geregelt. Dem Microservice und der entsprechenden Datenbank wird ein Service-Account mit den benötigten Berechtigungen hinzugefügt. Ruft der Server im Microservice nun die Datenbank auf, findet erneut eine Authentifizierung über die Service-Accounts statt.

Damit ein Nutzer die Anwendung überhaupt verwenden kann, muss er sich zunächst authentifizieren und autorisieren. Dazu wird der OAuth 2.0 Client von Google verwendet. Die Google APIs verwenden zur Authentifizierung und Autorisierung das OAuth 2.0 Protokoll, welches einen hohen Standard darstellt. Das von Google erzeugte JWT wird an den Microservice Benutzerverwaltung übergeben und wird nur ausgeführt, wenn die Signatur und weitere Sicherheitsparameter im Token korrekt sind. Mit den gleichen Standards wird ein eigenes JWT Token erzeugt, damit weitere individuelle Informationen zum Nutzer darin gespeichert und übergeben werden können.

Außerdem sind alle Pfade, die die Dateiverwaltung und Chatverwaltung betreffen, mit dem zuvor erzeugten JWT gesichert. Das gültige Token muss bei jeder Anfrage im Header (Bearer Authentifizierung) mitgeschickt werden. Das API-Gateway prüft die Gültigkeit des Tokens und leitet nach erfolgreicher Überprüfung die Anfrage an den Microservice weiter. Jeder Microservice überprüft ebenfalls, ob die angeforderte Aktion mit den Informationen aus dem Token übereinstimmt. Dadurch kann beispielsweise niemand im Namen eines anderen eine Nachricht verfassen oder eine Datei löschen, welche der Nutzer gar nicht hochgeladen hat. Im Frontend kann ein Nutzer zusätzlich nur auf den Datei/Chat-Screen zugreifen, wenn er erfolgreich angemeldet ist und sich das JWT in den Cookies des Browsers befindet. Ist der Nutzer nicht angemeldet, kann dieser nicht auf die Unterseiten der Webseite zugreifen und wird wieder auf die Login-Seite zurückgeleitet.

Weiterhin ist sämtliche Kommunikation zwischen Browser und Server oder auch zwischen Gateway und Microservices mit dem Hypertext Transfer Protocol Secure (HTTPS) verschlüsselt und somit gesichert. Das SSL-Protokoll verschlüsselt die ausgetauschten Daten und sorgt damit, dass vertrauliche Informationen auf dem Übertragungsweg vor Ausspähen oder anderem Missbrauch geschützt sind. Durch diese Protokolle wird die Vertraulichkeit, Integrität und Authentizität gewährleistet.

Abschließend kann also festgehalten werden, dass die Anwendung gegen einen Großteil der bekanntesten und verbreitetsten Angriffsmethoden geschützt ist. Durch die oben beschriebenen Sicherheitsmethoden werden Angriffe wie zum Beispiel Cross-Cite-Scripting (XSS), Man-In-The-Middle Angriffe (MITM), Session Hijacking, Cross-Site-Request-Forgery (CSRF) und Denial of Service (DoS) erheblich erschwert oder zum Teil ganz unterbunden.

4.2 Skalierbarkeit

Einer der großen Vorteile der Google Cloud Platform Technologien ist, dass Ressourcen automatisiert und flexibel skalieren. So kann die Anwendung *StudentShare* ihre Kapazitäten automatisch an den Bedarf der Nutzer anpassen und bleibt auch bei höherem Traffic stabil.

Zu Beginn wird die App Engine betrachtet, die zum Hosten der Webseite und für den Chat Microservice verwendet wird. Googles App Engine kann automatisch auf eine höhere Zahl von Anfragen der User reagieren, indem von einer Anwendung mehrere Instanzen zur Verfügung gestellt werden. Tritt unerwartet höherer Traffic auf, skaliert Google selbstständig die Anzahl der Instanzen hoch und stellt diese parallel zur Verfügung. Dies gilt auch im gegenteiligen Fall. Sollte die Anwendung mal keinen Traffic haben, z.B. nachts, kann die Anzahl der Instanzen auch auf null runterskaliert werden. In Kombination mit der Firestore NoSQL-Datenbank bleibt der Chat Microservice flexibel, denn “Firestore skaliert automatisch und ohne Ausfallzeiten[8]”. So können tausende von Chatnachrichten pro Sekunde und Millionen gleichzeitiger Verbindungen verarbeitet werden. Alles natürlich nur abhängig von der tatsächlichen Kommunikation über den Microservice.

Die Cloud Functions ergeben mit der SQL Datenbank eine performante Verbindung für die Benutzerverwaltung, da beide jeweils einzeln, aber trotzdem parallel hoch- und herunterskalieren. Die Cloud Functions können, ebenso wie Cloud SQL, je nach Anzahl der parallelen Anfragen mehrere Instanzen zur Verfügung stellen. Der einzig limitierende Faktor hierbei ist das Limit von 100.000 maximalen Verbindungen[9] zur SQL-Datenbank, was aber mit den Anforderungen des Projekts von 100.000 Req/s übereinstimmt. Daher sollte auch die maximale Anzahl der Cloud Function Instanzen

[8] https://cloud.google.com/architecture/building-scalable-apps-with-cloud-firestore
[9] https://cloud.google.com/sql/docs/quotas#limits

auf 100.000 begrenzt werden. So können erhöhte Wartezeiten oder Fehler beim Aufrufen verhindert werden, denn innerhalb der Funktion kann bei mehr als 100.000 Instanzen keine Verbindung zur Datenbank mehr aufgebaut werden. Sollten mehr als 100.000 Verbindungen pro Sekunde benötigt werden, empfiehlt Google die Anwendung auf mehrere Google Cloud Projekte aufzuteilen.

Die Dateiverwaltung, bestehend aus Cloud Run, Firestore NoSQL-Datenbank und Cloud Storage lässt sich als letzten Microservice genauso einfach skalieren, wie die anderen Microservices. Cloud Run stellt automatisch mehr Containerinstanzen zur Verfügung, um alle eingehenden Anfragen und Ereignisse erfolgreich zu verarbeiten[10]. Sollte der Service wiederum nachts keinen Traffic empfangen, werden die Containerinstanzen standardmäßig auf null herunterskaliert. Dateipfade und Informationen über die Berechtigungen für die Datei werden in Firestore gespeichert. Je mehr Dateien hochgeladen werden, desto mehr Speicherplatz wird von den Informationen belegt. Der Cloud Storage hat mit seinen Buckets extrem schnelle Lese- und Schreibgeschwindigkeiten und so gut wie kein Speicherlimit.[11] Je nach Anzahl der hochgeladenen Dateien wird automatisch Speicherplatz innerhalb der Buckets verwendet.

Zuletzt bietet das API-Gateway von Google eine einfachere Möglichkeit, die Anfragen auf die Anwendung zu koordinieren. Das Gateway erstellt neue Instanzen zur Verarbeitung der Last, wenn der Traffic zunimmt. Zusätzlich ist es ein Dienst, der auch auf null skaliert. Gateway-Instanzen werden gelöscht, wenn kein Traffic vorhanden ist[12].

Somit erfüllt *StudentShare* die Anforderungen einer Web-Scale Anwendung und ist automatisch ohne manuelle Änderungen auf keine, einige, 100, 10.000 und 100.000 Req/s skalierbar. Dabei ist es egal, ob sich Tausende User gleichzeitig anmelden, Dateien hoch- oder herunterladen oder Chatnachrichten verschicken. Jeder Microservice ist unabhängig von den Anderen und skaliert automatisch eigenständig.

4.3 Kosten

Mit dem Gedanken ein Projekt wie dieses vollständig in der Cloud betreiben zu wollen, sollten die Kosten ständig im Auge behalten werden. Dafür werden in Tabelle 1 die Grundpreise der einzelnen Komponenten angegeben, plus zwei Szenarien, in welche Richtung sich die Anwendung entwickeln könnte.

Im ersten Szenario wäre die App Hochschule RheinMain intern und nur für Studenten auf dieser Hochschule verfügbar. Das macht laut Google circa 14.000 Studenten auf 70 unterschiedliche

[10] https://cloud.google.com/run/docs/about-instance-autoscaling?hl=de
[11] https://medium.com/swlh/how-well-does-your-database-scale-a3827951757e
[12] https://cloud.google.com/api-gateway/docs/deployment-model

Studiengänge. Im zweiten Szenario würde die App jedem Studenten in ganz Deutschland zur Verfügung stehen. Das macht ungefähr 3 Millionen Studenten in 20.951 unterschiedlichen Studiengängen. Des Weiteren wurde geschätzt, dass jeder Student sich pro Tag einmal anmeldet, maximal 11 Funktionen der App benutzt, 5 Nachrichten schreibt und über den Monat verteilt 5 Dokumente hochlädt. Die Gesamtkosten belaufen sich dabei monatlich.

Das ganze wurde versucht, mit den Pricing Angaben der Google Cloud Dokumentation für jedes Szenario so detailliert wie möglich hochzurechnen. Außerdem werden die Free Tiers mit eingerechnet, wodurch Szenario eins aufgrund der übersichtlichen Größe oft keine Kosten in der Cloud erzeugt.

Das Frontend, auf dem ein möglicher User landet, wird in einer Standard App Engine vom Typ F2 gehalten, die jede Stunde im Monat ansprechbar sein muss. Zusammen mit den Deployment- und Erhaltungskosten der Funktionen und Container (Build, etc.), die oftmals in Buckets gespeichert werden, sind das die Standardkosten, die für jedes Szenario benötigt werden.

Die Anmeldefunktion und das Erzeugen des JWT Tokens für das API Gateway geschieht über Cloud Functions. Die Nutzerinformationen werden dabei in einer Cloud SQL Datenbank gespeichert. Das Event für die Cloud Functions ist ein Standard Event und wird pro Anmeldung pro Nutzer 2 mal ausgelöst. Die in der SQL Datenbank gespeicherten Informationen sind kleine Dateien mit wenigen KB pro Datei. Dennoch ist die SQL Datenbank eine teure Komponente und erzeugt für ihren Nutzen einen hohen monatlichen Betrag. Die anschließende Kommunikation mit der Dateiverwaltung, geschieht mit dem API Gateway, wodurch hohe Aufrufzahlen generiert werden. Der Chat ist davon ausgenommen.

Chatnachrichten, die älter als ein Monat sind werden gelöscht. Außerdem erzeugt die Masse der Nutzer viele Nachrichten, die in Firestore erzeugt und gelagert werden müssen. Das ist ein beträchtlicher Kostenpunkt, an dem zukünftige Optimierungen stattfinden könnten. Des weiteren läuft der Chat über einen Websocket in einer App Engine. Dieser Kostenpunkt kann je nach Nutzerzahl stark ansteigen, da mehrere Instanzen benötigt werden.

Zu guter Letzt hat noch jeder User 5 Dokumente an seinen Account pro Monat gekoppelt, wodurch hohe Kosten in der Storage Unit erzeugt werden. Die Informationen der Dateien werden jedoch ausgelagert in eine weitere Firestore Sammlung, auf die sehr oft zugegriffen wird. Read Operationen sind für den Bucket teuer (Class A), aber für die Firestore Sammlung nicht. Deswegen sind die Kosten der Operationen auf den Bucket fast zu vernachlässigen und vergleichsweise billig über Firestore.

Aus all diesen Komponenten ergibt sich ein monatlicher Betrag, der bei einer kleinen Anwendung dank der Free Tiers gering ausfällt, aber je nach höhe der Nutzerzahl stark ansteigen kann. Es gibt potential, die Kosten etwas zu senken, obwohl die Komponenten kostengünstig versucht wurden einzusetzen. Insgesamt würde die Anwendung in Szenario zwei, 360$ am Tag, bzw. 10815$ im Monat kosten oder auch 0,003$ pro User.

Komponente	Informationen	Grundpreis	~14.000 Nutzer/Monat	~3.000.000 Nutzer/Monat
Cloud Functions	1 Anmeldung/Tag/Nutzer * 2 Aufrufe 256MB Speicher 400MHz CPU 300ms each time	$0,40 (ab 2M Aufrufe pro 1M) $0,12 (ab 5GB/Monat pro GB) $0.0000025 (ab 400k pro GB-sec) $0.0000100 (ab 200k pro GHz-sec)	FREE - (840k Aufrufe)	$71,20 - (~180M) FREE $45,85 - (~13,5M GB-sec) $299,60 - (~21,6M GHz-sec)
	deployment costs Outbound Data to Google APIs	FREE FREE	FREE	FREE
Cloud Run	10 Aufrufe/Tag/Nutzer Preisstufe 2 (europe-west3)	$0,40 (ab 2M Aufrufe pro 1M) $0,00000350 (ab 360k GB-sec/Monat) $0,00003360 (ab 180k vCPU-sec/Monat)	$0,88 - (4,2M Aufrufe) FREE - (6,3k GB-sec) FREE - (12,6k vCPU-sec) (max. 100 gleichzeitige Nutzer)	$360 - (900M Aufrufe) FREE - (135k GB-sec) $3,02 - (270k vCPU-sec) (max. 1k gleichzeitige Nutzer)
	Deployment	$0.026 (deployment costs/Monat)	$0,026 (deployment)	$0,026 (deployment)
App Engine	F1 - auto. - 256mb - 600MHz 11 Aufrufe/Tag/Nutzer 5000 Aufrufe/Instanz	$0.06 Instanz/Stunde (ab 28 Stunden/Tag) $0.12 outgoing traffic/GB (ab 1GB traffic/Tag)	FREE - (1 Instanzen/Monat) FREE - (210mb outgoing traffic)	$1029.6 - (25 Instanzen/Stunde) $5,28 - (45GB outgoing traffic/Tag)
Cloud SQL	1 vCPU - 4GB Memory - 10GB SSD 100kb pro Datei	$0.0496 pro vCPU/Stunde $0.0084 pro GB/Stunde (Memory) $0.204 pro GB/month (SSD) $0,19 pro GB (internet egress)	$36.21 - (1 vCPU/Monat) $24,53 - (Memory/Monat) $0,29 - (1,4GB/Monat) $0,27 - (1,4GB/traffic) 1 Instanz	$1231,14 - (1 vCPU/Monat) $834,02 - (Memory/Monat) $87 - (300GB/Monat) $57 - (300GB/traffic) 34 Instanzen
Firestore (File)	10 Read Op/Tag/Nutzer 0,5 Write Op/Tag/Nutzer 0,5 Delete Op/Tag/Nutzer 5 Dokumente/Nutzer 1 KB max. Größe/Dokument	$0,06 Reads (ab 50k/Tag pro 100k Dok.) $0,18 Writes (ab 20k/Tag pro 100k Dok.) $0.02 Deletes (ab 20k/Tag pro 100k Dok.) $0,18 Storage (ab 1 GB pro Monat)	$1,23 - (~4,2M Reads/Monat) FREE - (~210k Writes/Monat) FREE - (~210k Deletes/Monat) FREE - (~70mb Storage/Monat)	$539,97 - (~900M Reads/Monat) $80,96 - (~45M Writes/Monat) $8,99 - (~45M Deletes/Monat) $81 - (~15GB Storage/Monat)
Firestore (Chat)	1 Read Op/Tag/Nutzer 5 Write Op/Tag/Nutzer 5 Delete Op/Tag/Nutzer 5 Dokumente/Tag/Nutzer 100 B max. Größe/Dokument	siehe Zeile drüber	FREE - (~420k Reads/Monat) $2,25 - (~2.1M Writes/Monat) $0,30 - (~2.1M Deletes/Monat) FREE - (~210mb Storage/Monat)	$53,1 - (~90M Reads/Monat) $808,9 - (~450M Writes/Monat) $89,90 - (~450M Deletes/Monat) $8,1 - (~45GB Storage/Monat) 4701,06
Cloud Storage	5 Dateien pro Nutzer max. Größe pro Datei sind 10mb europe-west3	$0,023 pro GB	$16,10 - (max. 700 GB)	$3450 - (max. 150 TB)
	Network egress auf dem selben Kontinent	FREE	-	-
	1 Operation pro Tag Class B = *get* für den download Free = *delete*	$0,004 pro 10k Operationen	$0,084 - (~210k Op./Monat)	$18 - (~45M Op./Monat)
API Gateway	12 Aufrufe/Tag/Nutzer gesamter Traffic von Firestore, Storage, SQL, Functions	$3.00 (ab 2M Aufrufe pro 1M <1Mil) $1.50 (ab 2M Aufrufe pro 1M >1Mil) $0.105 (EU to EU pro GB <10TB) $0.060 (EU to EU pro GB >150TB)	$9,12 - 5,04M Aufrufe 701,68GB	$1617 - 1,08Mil Aufrufe 150,36TB
Others	Cloud Build Cloud Container Registry Artifact Registry Frontend App Engine	First 120 build-minutes/day are free Kosten der Storage Unit (Bucket) FREE 259mb Frontend Instanthaltung	- $36.50	- $36.50
	-	-	$164,47	$10815,50

Tabelle 1: Kostenaufteilung aller genutzten Cloud-Dienste, anhand von zwei Szenarien

5 Fazit und Ausblick

Mit dem Projekt *StudentShare* wurden verschiedene Ansätze und Komponenten aus der Cloud miteinander vereint und erprobt. Angefangen mit einer statischen Webseite, die mit Vue und TypeScript umgesetzt wurde, folgen viele Endpunkte zu verschiedensten Funktionalitäten, aufgeteilt in unterschiedliche Microservices (Benutzer-, Datei- und Chatverwaltung), die mit dem Frontend über unabhängige REST APIs kommunizieren. Der User wird aufgefordert, sich mit einer Third-Party-Authentifizierung anzumelden, woraufhin der Benutzer-Microservice ein JWT erzeugt, mit Informationen über den User, mit dem sich bis zum Ende der Session über ein API-Gateway authentifiziert werden muss. Damit können sich die Microservices über eine No-Trust-Policy gegenseitig authentifizieren, ohne Anfragen an die User SQL Datenbank zu stellen, wodurch ein mögliches Bottleneck verhindert wird und die Sicherheit in dem System von außen gewährleistet ist. Außerdem wurden Multimedia- und organisatorische Daten voneinander in unterschiedlichen Datenbanken getrennt. Des Weiteren sind die Microservices durch die Verwendung von Infrastructure as a Service (IaaS) getrennt voneinander skalierbar. Die Kosten beschränken sich dabei auf ein Minimum.

Allerdings gab es bei der Durchführung des Projektes auch immer wieder Schwierigkeiten mit einzelnen Cloud-Komponenten, aufgrund von Beta-Zuständen und noch nicht implementierten Features. Daher wurde viel Zeit aufgewendet, “Workarounds” zu finden, in der nicht immer einfach gestalteten Dokumentation.

Doch nachdem nun die Infrastruktur in der Cloud und Kommunikation der einzelnen Komponenten funktioniert und die Aufgabenbereiche in unabhängige Microservices aufgeteilt wurden, können zukünftige Features einfacher in das Projekt integriert werden. Das ist ein großer Vorteil der Cloud- und Microservice-Architektur. Sind weitere Features in den unterschiedlichen Microservices geplant, wie unterschiedliche Chaträume pro Fachbereich, Dateivorschauen, Datei privat halten oder Passwort sichern, so können sie leicht integriert werden, ohne auf viele Abhängigkeiten achten zu müssen. Ein weiterer Vorteil ist die automatische und unabhängige Skalierbarkeit der Google Cloud Komponenten durch Google selbst. Damit werden nur so viele Ressourcen verbraucht, wie sie benötigt werden, wodurch die Kosten gesenkt werden können.