Umsetzung der Semesteraufgabe mit Google Cloud Firestore

von Peter Fischer, Leonelle Tifani Kommegne Kammegne, Michael Mertl, Gregor Pfister und Jana Sophie Schweizer

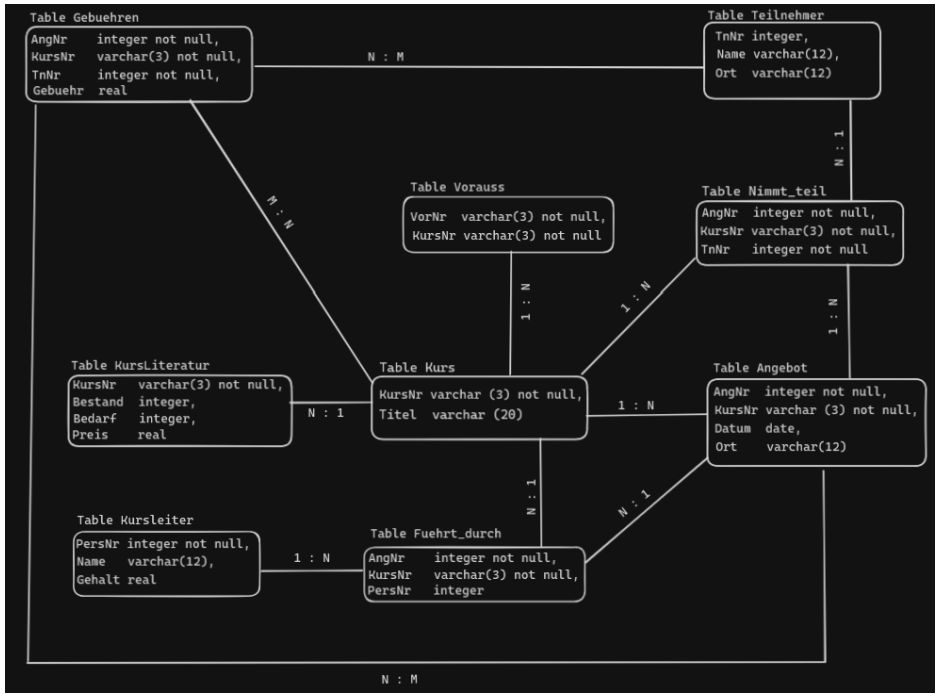
Ausgangssituation

Gruppe 05

Vorgänger

SQL-DUMP alte relationale Struktur

Management will jetzt eine NoSQL-Datenbank



Erstellung eines Entity-Relationship-Modell, um den SQL-DUMP besser zu verstehen

Auswahl einer NoSQL Datenbank

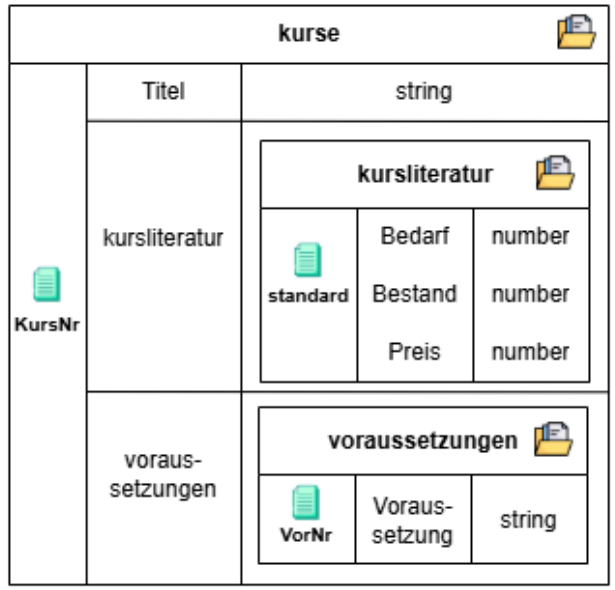
Entscheidung Firestore

DB-Engines Ranking: https://db-engines.com/en/ranking/document+store

- von Google
- gute Dokumentation
- gute Integration in Node.js
- persönliches Interesse

Google Cloud Firestore

Firestore ist eine dokumentenbasierte NoSQL-Datenbank:
- Daten sind in *Collections* organisiert
- Eine *Collection* hat beliebig viele *Dokumente*
- Ein *Dokument* kann selbst wieder *Collection* (dann *Sub-Collections* genannt) besitzen mit *Dokumenten*
- Dokumente bestehen aus Feldern in Form von Schlüssel-Wert-Paaren (ähnlich wie bei JSON-Objekten)



Abfragesprache von Firestore

Standardabfragesprache ist keine deklarative Sprache wie SQL, sondern eine methodenbasierte API, die über verschiedene Programmiersprachen hinweg verfügbar ist.
Firestore stellt hierfür offizielle SDKs zur Verfügung für z.B. JavaScript, Python, Java, Kotlin …

Web modular API | Web namespaced API | Swift | Objective-C | Kotlin Android | Java Android | Dart Flutter | Python | Python (Async) | C++ | Mehr

```
import { collection, doc, setDoc } from "firebase/firestore";

const citiesRef = collection(db, "cities");

await setDoc(doc(citiesRef, "SF"), {
    name: "San Francisco", state: "CA", country: "USA",
    capital: false, population: 860000,
    regions: ["west_coast", "norcal"] });
```

Einfaches Schreiben von Daten

Web modular API | Web namespaced API | Swift | Objective-C | Kotlin Android | Java Android | Dart Flutter | Java | Python | Python (Async) | Mehr

```
// Create a reference to the cities collection
import { collection, query, where } from "firebase/firestore";
const citiesRef = collection(db, "cities");

// Create a query against the collection.
const q = query(citiesRef, where("state", "==", "CA"));
```

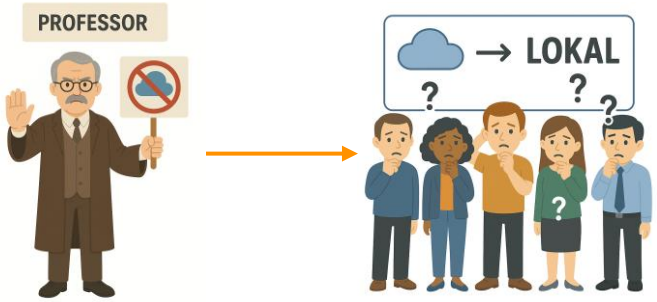
simple_queries.js

Einfache Abfrage von Daten

Im Gegensatz zu SQL müssen bei dieser Art von Abfragen Joins, Aggregationen und komplexere Operationen vom Client übernommen werden. Das bedeutet, dass manche Auswertungen – wie etwa das Zusammenführen mehrerer Datensätze – durch zusätzliche Logik im Anwendungscode umgesetzt werden müssen.

In unserem Projekt haben wir uns für TypeScript entschieden, um bei der Migration der relationalen Struktur, die ursprünglichen Datentypen zu erhalten und Typsicherheit zu gewährleisten.

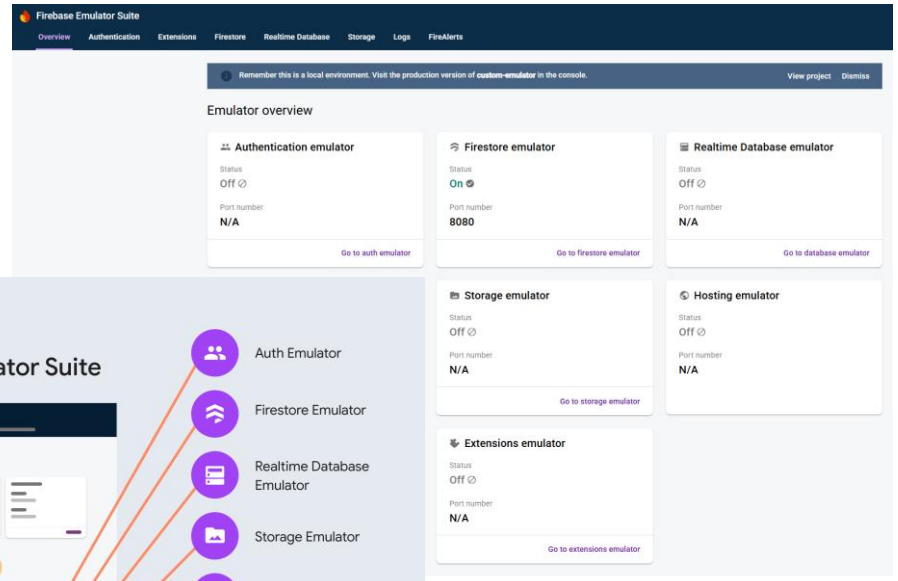
Lokale Nutzung von Firestore

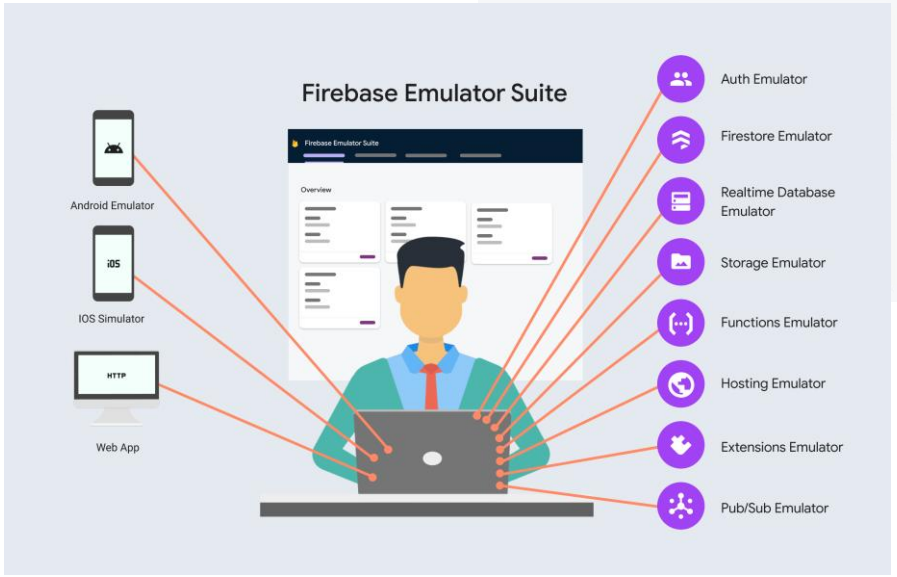


Cloud-Lösungen sind verboten

Gruppe 05

Firebase stellt eine *Local Emulator Suite* bereit, welche das Verhalten der echten Firebase-Dienste lokal nachbildet. Für unsere Zwecke benötigen wir nur den Firestore Dienst und die *Emulator-UI* für eine visuelle Darstellung.





Lokale Nutzung von Firestore

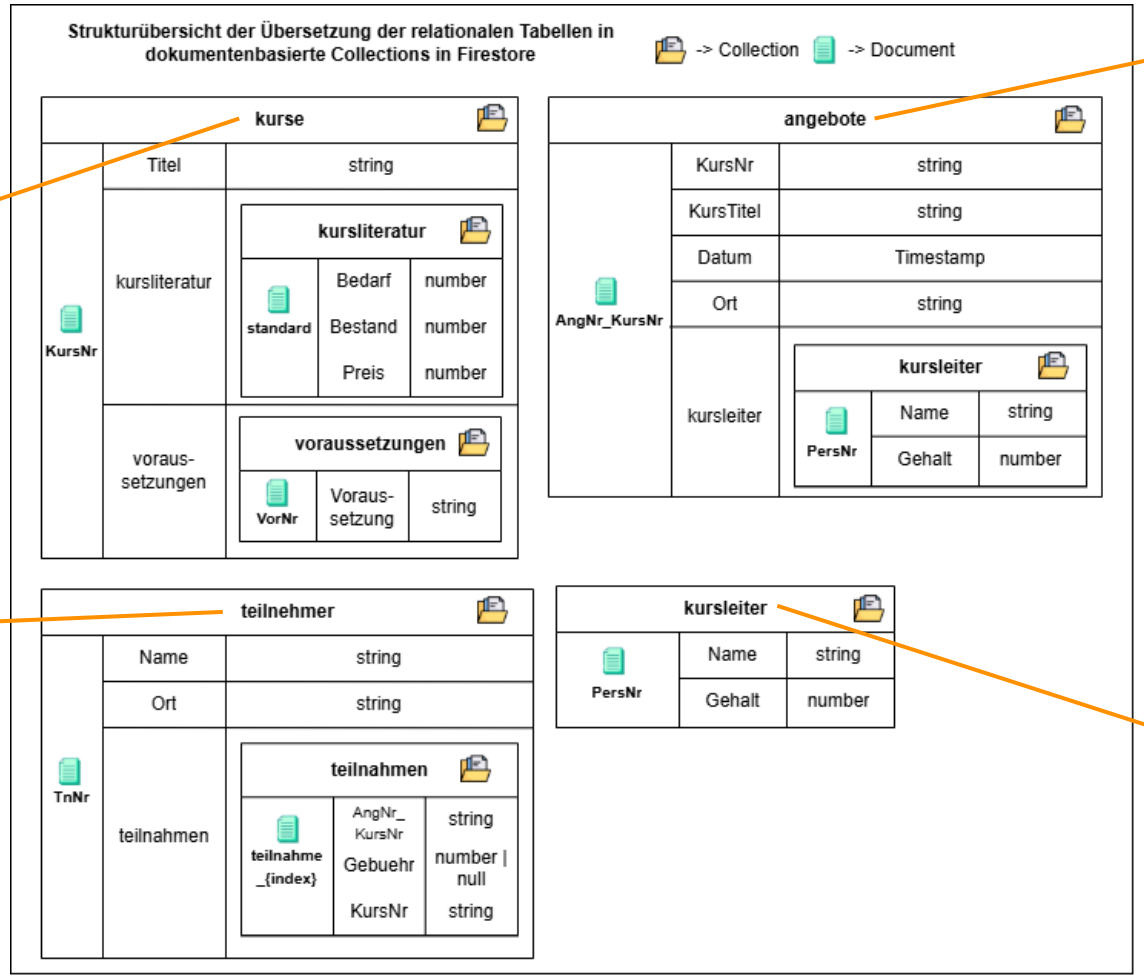
@Frage an alle: Hier noch eine Folie dazu, für das genaue Setup und die Probleme mit Docker?

Aufbau Datenstruktur

Aus neun relationalen Tabellen wurden vier *Haupt-Collections* gebaut: *kurse, angebot, teilnehmer, kursleiter*

kurse beinhaltet in seinen Dokumenten jeweils *Sub-Collections*, dir vorher eigene Tabellen waren -> *kursliteratur* und *voraussetzungen*

teilnehmer beinhaltet in seinen Dokumenten jeweils eine *Sub-Collection*, dir vorher eine eigene Tabelle war -> *teilnahmen* -> mit dem neuen Feld *Gebuehr*, was vorher ebenfalls eine eigene Tabelle war



angebote beinhaltet in seinen Dokumenten jeweils eine *Sub-Collection*, die ebenfalls eine Haupt-Collection ist -> *kursleiter.*

Außerdem ist ein weiters Feld *KursTitel* hier hinzugekommen.

Die redundante Speicherung von *KursTitel* und der *Collection kursleiter* als *Sub-Collection* erleichtert uns Abfragen, die sonst mit *JOINs* gelöst werden oder mit vielfachen Lesen von mehreren *Collections* -> ***Best Practice*** in dokumentenbasierten Datenbanken

kursleiter beinhaltet keine *Sub-Collections* oder Änderungen gegenüber der alten relationalen Tabelle

Typsicherheit und Abfragen mit TypeScript

Firestore bietet standardmäßig keine Typsicherheit, daher haben wir uns für TypeScript entschieden, um diese zu gewährleisten

→ Für die Definition dieser Typsicherheit und den Abfragen an die Datenbank nutzen wir das *npm-Paket @google-cloud/firestore* (*https://www.npmjs.com/package/@google-cloud/firestore*)

Definition der Typen →

```ts
export interface Kursliteratur {  S
    Bestand: number;
    Bedarf: number;
    Preis: number;
}

export interface Kurs {  Show usages
    Titel: string;
    kursliteratur?: Kursliteratur;
    voraussetzungen?: string[];
}
```

→ Bau eines *Custom Converters* →

```ts
export const createConverter : <T extends { [key: string]: any }>() => F...  =  Show usages
    <T extends { [key: string]: any }>(): FirestoreDataConverter<T> => ({
    toFirestore: (data: WithFieldValue<T>) : WithFieldValue<T>  => data,
    fromFirestore: (snap: QueryDocumentSnapshot): T => snap.data() as T,
});
```

Nutzung für eine durchgängige Typprüfung beim Schreiben / Lesen / Updaten und Löschen der Daten ←

```ts
const kursConverter : FirestoreDataConverter<Kurs, DocumentData...  = createConverter<Kurs>();
const angebotConverter : FirestoreDataConverter<Angebot, DocumentD...  = createConverter<Angebot>();
const kursleiterConverter : FirestoreDataConverter<Kursleiter, Docume...  = createConverter<Kursleiter>();
const teilnehmerConverter : FirestoreDataConverter<Teilnehmer, Docume...  = createConverter<Teilnehmer>();

const docRef : DocumentReference<T, DocumentData>  = db.collection(collectionName).doc(id).withConverter(converter);

const angeboteSnapshot : QuerySnapshot<Angebot, DocumentData>  = await db.collection( collectionPath: 'angebote').withConverter(createConverter<Angebot>()).get();
const orte = new Set(angeboteSnapshot.docs.map(a : QueryDocumentSnapshot<Angebot, DocumentDa...  => a.data().Ort)); //.data() ist direkt vom Typ Angebot
```

Herausforderungen bei der Umsetzung der Read-Abfragen

1. Keine Unterstützung für *JOINs*: In SQL werden Daten aus mehreren Tabellen mithilfe von *JOIN-Operationen* miteinander verbunden. In Firestore existiert eine solche Funktionalität nicht und deswegen müssen hier alle *Collection* extra gelesen werden und einzeln dann nachgeladen werden, um *Collection* miteinander zu verbinden.
   => Diese Einschränkung hat ebenfalls dazu geführt, dass wir Redundanzen in unsere Datenstruktur eingebaut haben, damit wir Anfragen erleichtern, die sonst mit *JOINs* umgesetzt werden.

2. Keine Aggregatfunktionen wie *COUNT*, *AVG* oder *GROUP BY*: **@Leonelle** bist du dir sicher -> https://firebase.google.com/docs/firestore/query-data/aggregation-queries?hl=de . Du hast schon recht, dass es kein HAVING COUNT (*) gibt, aber COUNT, AVG oder GROUP BY gibt es schon.

3. Kein direkter Zugriff auf *Sub-Collections*: Wenn man eine Sub-Collection einer Collection laden möchte, dann ist das nicht direkt möglich und man muss immer die übergeordnete Collection ebenfalls laden. Hier sind zusätzliche Leseoperationen und individuelle Nachladeprozesse erforderlich.

⇒ Komplexe Auswertungen benötigen oft zusätzliche clientseitige Logik.

⇒ **@Gregor** bitte fertig machen etc.

Herausforderungen bei der Umsetzung der Update- & Delete Abfragen

1. Einfache Update & Delete Anfragen sind leicht umzusetzen

2. Sobald aber Daten, wie in unserem Fall z.B. Kursleiter, an mehreren Stellen gespeichert sind, also redundant, wie es bei dokumentenbasierten Datenbanken üblich ist, steht man vor Herausforderungen. Um Datenkonsistenz und Integrität zu gewährleisten, muss man selbst dafür extra Logik einbauen.

3. Kein ON DELETE CASCAE oder ähnliches wie bei SQL

4. Nutzung von *Batch Writes* oder *Transaktionen* -> *https://firebase.google.com/docs/firestore/manage-data/transactions?hl=de*

*=> **@Jana** und **@Peter** bitte das hier noch fertig machen*

Fazit

Migration einer relationalen Kursdatenbank in eine dokumentenbasierte NoSQL-Datenbank wie Firestore stellte eine herausfordernde wie ich lehrreiche Aufgabe dar mit vielen Unterschieden und Erkenntnissen:

1. Die Abfrageflexibilität durch z.B. JOINs in SQL wird in Firestore anders erreicht und erfordert und grundsätzliches Umdenken -> Daten müssen redundant gespeichert werden, Abfragen logisch vereinfacht und viele Operationen in die Anwendungsschicht verlagert, wo extra Code nötig ist.

2. Durch TypeScript und Convertern konnten wir dem schemalosen Ansatz von Firestore eine starke Typsicherheit entgegensetzen.

3. Firestore ist ideal geeignet für semistrukturierte Daten und klar definierte Zugriffsmuster. Ebenfalls muss man sich Einschränkungen bewusst sein – im Hinblick auf nicht mögliche relationale Operationen und automatische Konsistenz

=> Insgesamt hat das Projekt nicht nur unsere Kenntnisse in Firestore, TypeScript und NoSQL-Datenmodellierung vertieft, sondern uns auch ein praktisches Verständnis dafür geschaffen, wie herausfordernd eine Umsetzung einer relationalen Datenbank in eine dokumentenbasierte Datenbank ist.

Quellen

- Firebase Logo und Farben -> https://firebase.google.com/brand-guidelines
- Icons -> https://www.flaticon.com/
- Menschen in Comic-Art von ChatGPT
- Beispiel Abfragen -> https://firebase.google.com/docs/firestore/query-data/queries?hl=de#web
- Emulator UI -> https://firebase.google.com/docs/emulator-suite?hl=de
- Converter -> https://firebase.google.com/docs/reference/node/firebase.firestore.FirestoreDataConverter
- Quellen mit Infos noch aus Doku kopieren?