Umsetzung der Semesteraufgabe mit Google Cloud Firestore

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

Link zum Vortragsvideo: https://www.youtube.com/watch?v=TvPoCD6FcbE

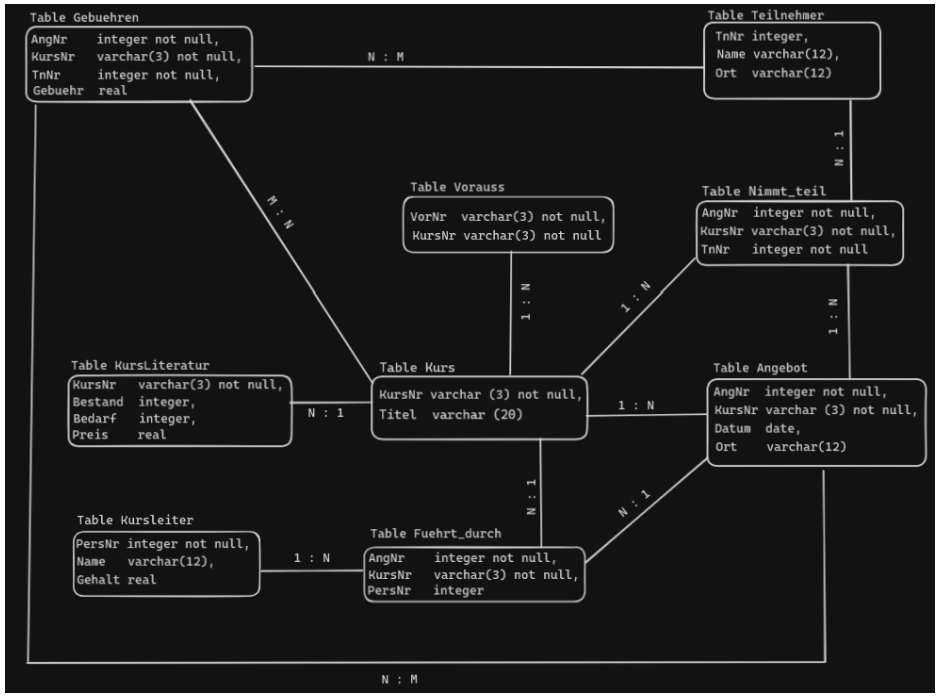
Ausgangssituation

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

[1], [2], [3]

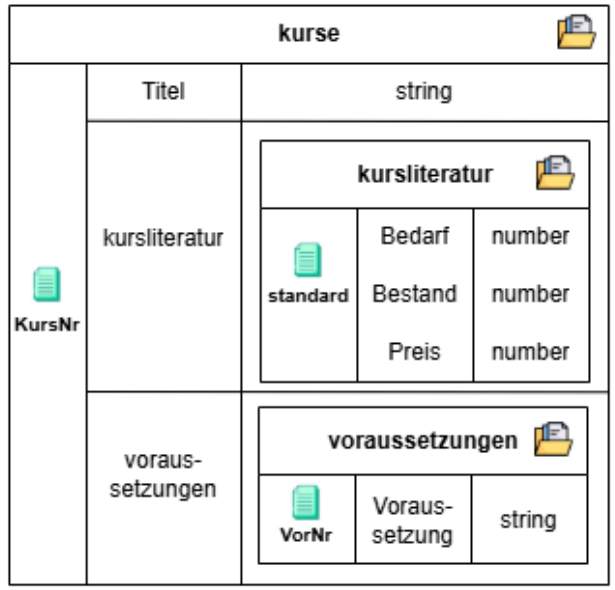
Entscheidung Firestore

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)



[4]

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

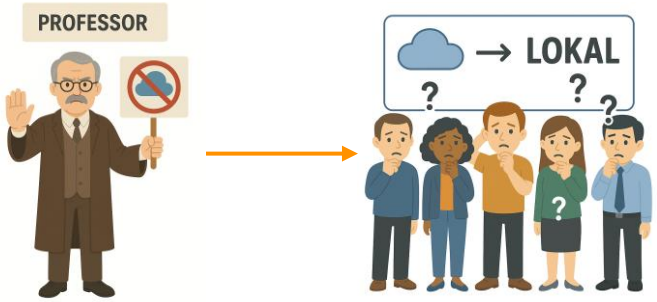
Einfache Abfrage von Daten

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.

[5]

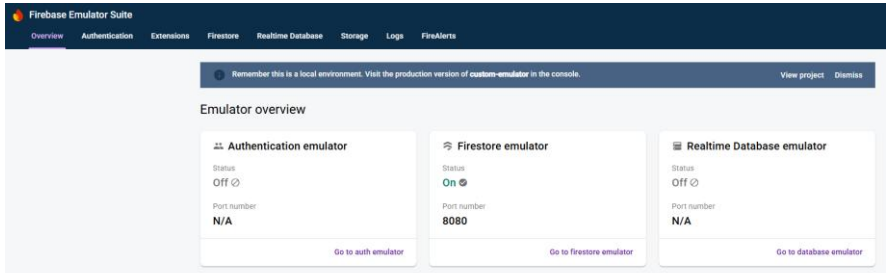
Lokale Nutzung von Firestore

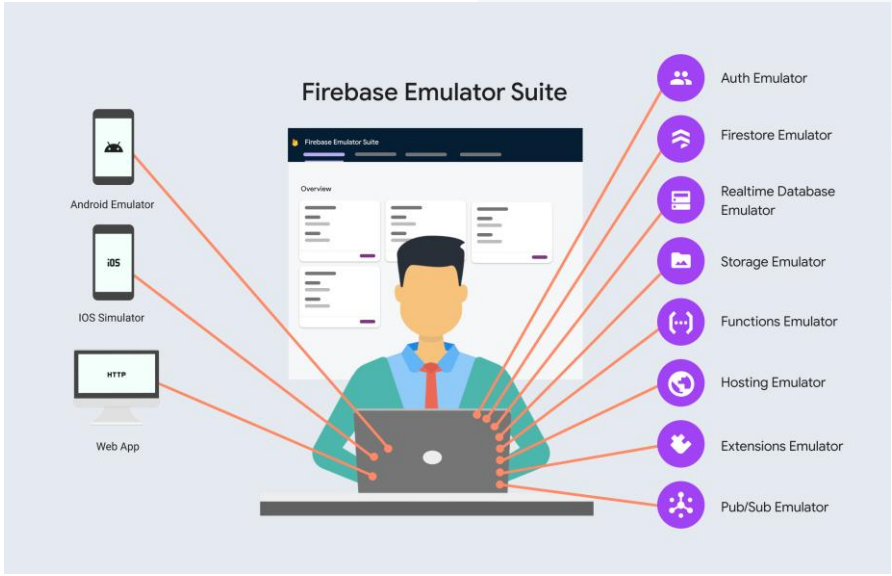


Cloud-Lösungen sind verboten

Gruppe 05

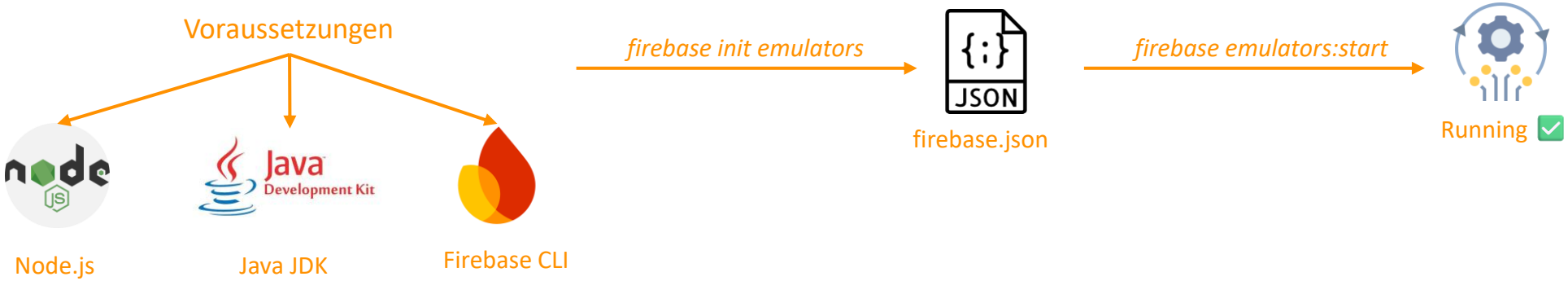
Firebase stellt eine *Local Emulator Suite* bereit, welche das Verhalten der echten Firebase-Dienste lokal nachbildet. Für unsere Zwecke benötigen wir nur den Firestore Dienst und die *Emulator-UI* für eine visuelle Darstellung.





[6]

Lokale Nutzung von Firestore



Warum nicht die Emulator Suite im Docker laufen lassen, um Abhängigkeiten vom lokalen PC fernzuhalten?

- Installationsprobleme von vordefinierten Images
- Zugriffsprobleme auf die innerhalb des Dockers laufende Emulator Suite

Einzige Lösung -> Skripte ebenfalls im Container laufen lassen

=> Entschluss als Gruppe, den offiziellen Weg von Firebase mit ein paar lokalen Installationen zu nutzen ✅

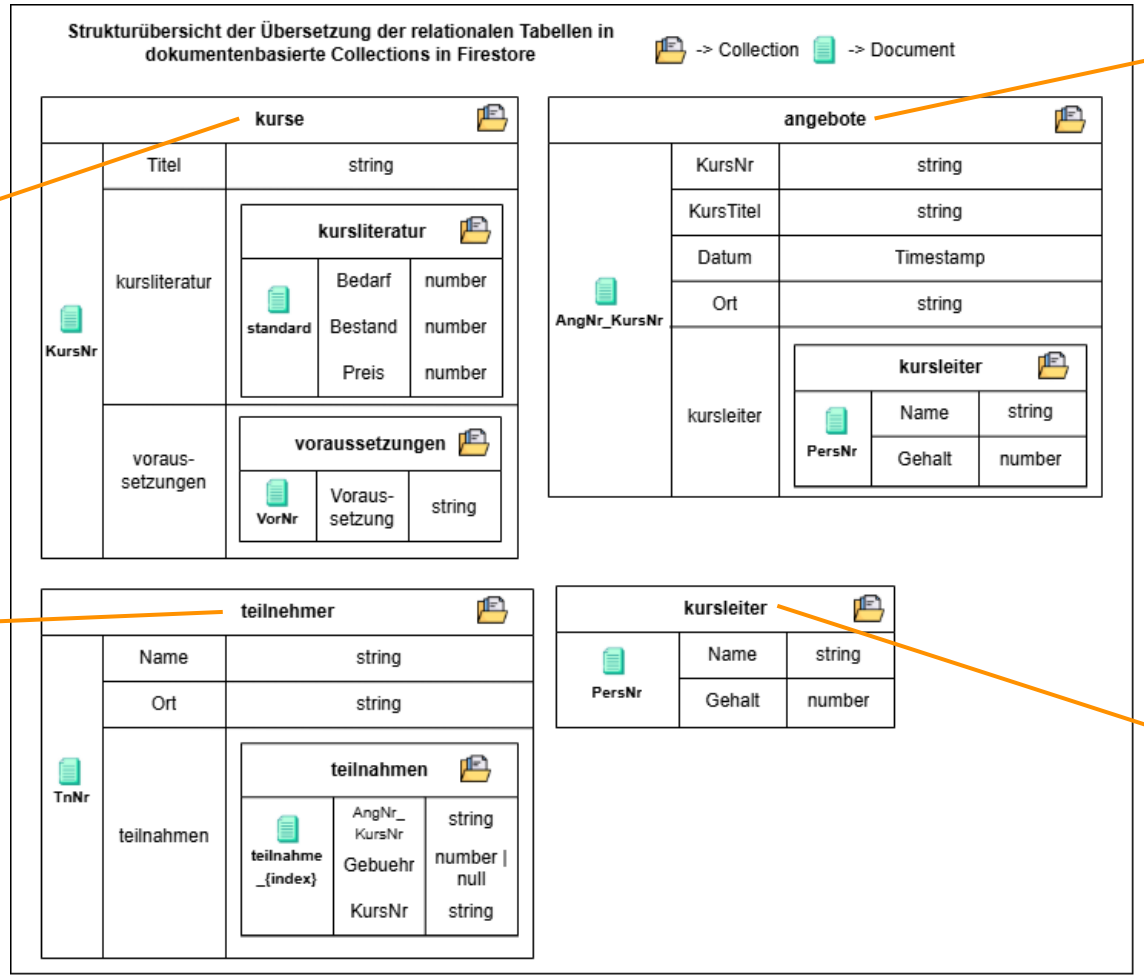
[7], [8]

Aufbau Datenstruktur

Aus neun relationalen Tabellen wurden vier *Haupt-Collections* gebaut: *kurse, angebote, 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

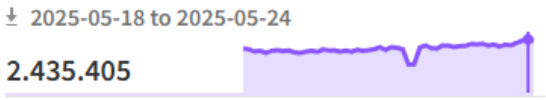
kursleiter beinhaltet keine *Sub-Collections* oder Änderungen gegenüber der alten relationalen Tabelle

[9]

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

```
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*

```
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

```
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. Beispiel aus Aufgabe g):

```
console.log('\n👥 Teilnehmer am eigenen Wohnort:');
const teilnehmerSnapshot = await db.collection('teilnehmer').withConverter(createConverter<Teilnehmer>()).get();
for (const tnDoc of teilnehmerSnapshot.docs) {
    const teilnehmer = tnDoc.data();
    const teilnahmenSnap = await tnDoc.ref.collection('teilnahmen').withConverter(createConverter<Teilnahme>()).get();
    for (const teilnahme of teilnahmenSnap.docs) {
        const { AngNr } = teilnahme.data();
        const angebot = await db.collection('angebote').doc(AngNr).withConverter(createConverter<Angebot>()).get();
        if (angebot.exists && angebot.data()?.Ort === teilnehmer.Ort) {
            console.log(`- ${teilnehmer.Name}: ${angebot.data()?.Ort}`);
        }
    }
}
```

=> Hauptgrund für unsere extra gebauten Redundanzen in der Datenstruktur

Herausforderungen bei der Umsetzung der Read-Abfragen

2. **Eingeschränkte Aggregatfunktionen**: Firestore unterstützt *COUNT* nur für die Gesamtanzahl an Dokumenten, die bestimme Kriterien erfüllen und komplexe Aggregationen with *GROUP BY* oder *HAVING COUNT* gar nicht. Clientseitiger Code nötig, um diese Operationen umzusetzen. Beispiel aus Aufgabe i):

```
console.log( message: '\n Kurse mit mindestens 2 Teilnehmern (alle Angebote zusammengefasst):');

// 1. Alle Angebote laden: Map<AngNr, KursNr>
const angeboteSnapshot4 : QuerySnapshot<Angebot, DocumentData> = await db.collection( collectionPath: 'angebote').withConverter(createConverter<Angebot>()).get();
const angebotZuKurs = new Map<string, string>();
for (const doc of angeboteSnapshot4.docs) {
    const { KursNr } = doc.data();
    angebotZuKurs.set(doc.id, KursNr);
}

// 2. Alle Teilnahmen über Collection Group Query laden
const teilnahmenSnapshot1 : QuerySnapshot<Teilnahme, DocumentData> = await db.collectionGroup( collectionId: 'teilnahmen').withConverter(createConverter<Teilnahme>()).get();
const kursTeilnehmerCounter: Record<string, number> = {};

for (const teilnahme of teilnahmenSnapshot1.docs) {
    const { AngNr } = teilnahme.data();
    const kursNr : string | undefined = angebotZuKurs.get(AngNr);
    if (kursNr) kursTeilnehmerCounter[kursNr] = (kursTeilnehmerCounter[kursNr] || 0) + 1;
}

// 3. Alle Kurse laden: Map<KursNr, Titel>
const kurseSnapshot1 : QuerySnapshot<Kurs, DocumentData> = await db.collection( collectionPath: 'kurse').withConverter(createConverter<Kurs>()).get();
const kursTitelMap = new Map<string, string>();
for (const doc of kurseSnapshot1.docs) {
    kursTitelMap.set(doc.id, doc.data().Titel);
}

// 4. Ausgabe: Nur Kurse mit mindestens 2 Teilnehmern
for (const [kursNr, anzahl] of Object.entries(kursTeilnehmerCounter)) {
    if (anzahl >= 2) {
        const titel : string = kursTitelMap.get(kursNr) ?? kursNr;
        console.log( message: `- ${titel}: ${anzahl} Teilnehmer`);
    }
}
```

[11]

Herausforderungen bei der Umsetzung der Read-Abfragen

3. **Kein direkter Zugriff auf *Sub-Collections* innerhalb *Collections***: Wenn man eine *Sub-Collection* einer bestimmten *Collection* laden möchte, oder dessen Werten lesen will, dann ist das nicht direkt möglich und man muss immer die übergeordnete *Collection* ebenfalls laden.
   => zusätzliche Leseoperationen und individuelle Nachladeprozesse erforderlich
   => Bei Abfragen, die zum Beispiel Informationen über die Kursleiter für Angebote beschaffen sollen, müssen nach dem Laden der Angebote noch die jeweiligen Kursleiter pro Angebot nachgeladen werden, um auf deren Namen oder Gehalt zugreifen zu können, da diese in einer *Sub-Collection* sind. Beispiel aus Aufgabe m):

```
const angeboteSnapshot6 = await db.collection('angebote').withConverter(createConverter<Angebot>()).get();

for (const angebot of angeboteSnapshot6.docs) {
    const { KursNr, KursTitel } = angebot.data();

    // Sub-Collection 'kursleiter' laden
    const kursleiterSnap = await angebot.ref.collection('kursleiter').withConverter(createConverter<Kursleiter>()).get();

    if (!kursGehaelterMap.has(KursNr)) {
        kursGehaelterMap.set(KursNr, { titel: KursTitel ?? KursNr, gehaelter: [] });
    }

    for (const leiter of kursleiterSnap.docs) {
        kursGehaelterMap.get(KursNr)?.gehaelter.push(leiter.data().Gehalt);
    }
}
```

Herausforderungen bei der Umsetzung der Read-Abfragen

3. **Kein direkter Zugriff auf *Sub-Collections* innerhalb *Collections***: Wenn man eine *Sub-Collection* einer bestimmten *Collection* laden möchte, oder dessen Werten lesen will, dann ist das nicht direkt möglich und man muss immer die übergeordnete *Collection* ebenfalls laden.
   => zusätzliche Leseoperationen und individuelle Nachladeprozesse erforderlich
   => Bei Abfragen, die zum Beispiel Informationen über die Kursleiter für Angebote beschaffen sollen, müssen nach dem Laden der Angebote noch die jeweiligen Kursleiter pro Angebot nachgeladen werden, um auf deren Namen oder Gehalt zugreifen zu können, da diese in einer *Sub-Collection* sind. Beispiel aus Aufgabe m):

Zwischenfazit nach den Read-Abfragen:
=> Komplexe Auswertungen benötigen oft zusätzliche clientseitige Logik.
⇒ Je nach Häufigkeit bestimmter Abfragen kann es sinnvoller sein auf *Sub-Collections* zu verzichten und Daten eher redundant direkt in den Dokumenten abzuspeichern, um das Nachladen zu verhindern
   ⇒ Ein Beispiel hierfür ist der Kurstitel, der häufig in Verbindung mit Angeboten benötigt wird. Um zu vermeiden, dass für jedes geladene Angebot ein weiterer Zugriff auf die „Kurse“ *Collection* notwendig ist, haben wir diesen redundant direkt im Angebot gespeichert.
   ⇒ Dadurch konnten wir in mehreren Abfragen die Anzahl der Datenbankzugriffe reduzieren
⇒ Diese Erkenntnisse haben unsere finale Datenstruktur maßgeblich geprägt.

[12], [13]

Herausforderungen bei der Umsetzung der Update- & Delete Abfragen

1. **Einfache Update & Delete Anfragen sind leicht umzusetzen:** Die initialen gestellten Aufgaben betrafen jeweils hauptsächlich nur eine *Collection* und waren sehr einfach und schnell umzusetzen.

2. In der Praxis werden aber oft Anfragen durchgeführt, die Daten betreffen, die an mehreren Stellen gespeichert sind, also redundant, wie es bei dokumentenbasierten Datenbanken üblich ist
   => **Problem**: Kein *ON DELETE CASCADE* oder *ON UPDATE CASCADE, u*m Datenkonsistenz und Integrität zu gewährleisten
   => **Lösung in Firestore**: Nutzung von *Transaktionen und Batch Operation oder Cloud Functions*
   => Zählungen oder andere Logik müssten trotzdem manuell am Client stattfinden
   => Dokumente müssen einzeln geladen und verglichen werden
   => Entwickler müssen selbst für Konsistenz sorgen

Beispiel: Löschen von einem Angebot
- Angebot selbst löschen
- Alle zugehörigen Teilnahmen und Gebühren, die bei den Teilnehmern gespeichert sind
- Wie sieht sowas genau aus?

Beispiel Transkationen und Batch Operation

```
const angebotTeilnahmeZaehler: Record<string, number> = {};
for (const teilnehmerDoc of teilnehmerSnapshot.docs) {
    const teilnahmenSnap :QuerySnapshot<DocumentData, DocumentData>  = await teilnehmerDoc.ref.collection( collectionPath: 'teilnahmen').get();
    for (const t of teilnahmenSnap.docs) {
        const { AngNr } = t.data() as Teilnahme;
        angebotTeilnahmeZaehler[AngNr] = (angebotTeilnahmeZaehler[AngNr] || 0) + 1;
    }
}
const zuLoeschendeAngebote: string[] = [];

await db.runTransaction(async (transaction :Transaction ) => {
for (const angebotDoc of angeboteSnapshot.docs) {
        const angebotId :string  = angebotDoc.id;
        const teilnehmerAnzahl :number  = angebotTeilnahmeZaehler[angebotId] || 0;

        if (teilnehmerAnzahl < 2) {
            const kursleiterSnap :QuerySnapshot<DocumentData, DocumentData>  = await angebotDoc.ref.collection( collectionPath: 'kursleiter').get();
            kursleiterSnap.docs.forEach(kursleiterDoc :QueryDocumentSnapshot<DocumentData, Docum...  => {
                transaction.delete(kursleiterDoc.ref);
            });

            transaction.delete(angebotDoc.ref);
            zuLoeschendeAngebote.push(angebotId);

            console.log( message: `🗑 Angebot ${angebotId} gelöscht in Transaktion (nur ${teilnehmerAnzahl} Teilnehmer).`);
        }
```

Zuvor: Snapshots von Angeboten und Teilnehmer
Hier: Teilnehmeranzahl pro Angebot zählen

Transaktion starten und Angebot mit Kursleiter Sub-Collections in Transaktion löschen

Notwendig für Batch Löschung

Beispiel Transkationen und Batch Operation

```
let batch : WriteBatch  = db.batch();
let opCount : number  = 0;
const MAX_BATCH_OPS = 490;

for (const teilnehmerDoc of teilnehmerSnapshot.docs) {
    const teilnahmenSnap : QuerySnapshot<DocumentData, DocumentData>  = await teilnehmerDoc.ref.collection( collectionPath: 'teilnahmen').get();
    for (const teilnahmeDoc of teilnahmenSnap.docs) {
        const { AngNr } = teilnahmeDoc.data() as Teilnahme;
        if (zuLoeschendeAngebote.includes(AngNr)) {
            batch.delete(teilnahmeDoc.ref);
            console.log( message: `🗑️ Teilnahme ${teilnahmeDoc.id} gelöscht (bezog sich auf Angebot ${AngNr}).`);

            opCount++;
            if (opCount >= MAX_BATCH_OPS) {
                await batch.commit();
                batch = db.batch();
                opCount = 0;
            }
        }
    }
}
if (opCount > 0) {
    await batch.commit();
}
```

Vorbereiten Batch mit empfohlenem Limit <= 500

Laufe durch alle Teilnehmer/Teilnahmen und setze zugehörige Teilnahmen auf „Lösch-Liste“ bzw. Batch-Queue zur Löschung

Batch-Queue Limit erreicht -> Batch-Vorgang durchführen und starte neuen Batch

Batch-Queue ausführen sofern noch etwas in Queue ist

Alternative: Cloud Function Trigger

Voraussetzung: Cloud

```
import * as functions from 'firebase-functions';
import * as admin from 'firebase-admin';

admin.initializeApp();
const db = admin.firestore();

export const cleanupOnAngebotDelete = functions.firestore
  .document('angebote/{angebotId}')
  .onDelete(async (snap, context) => {
    const angebotId = context.params.angebotId;
```

Kreieren Funktion – die automatisch getriggert wird wenn ein Angebot gelöscht wird
-> ID aus URL holen

Im Anschluss wieder Löschverfahren durch Beispielsweise Batch

Fazit

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

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

=> 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.

Firebase

Vielen Dank für Ihre Aufmerksamkeit ☺

Literatur

[1] R. Kesavan, D. Gay, D. Thevessen, J. Shah und C. Mohan, „Firestore: The NoSQL Serverless Database for the Application Developer“, in 2023 IEEE 39th International Conference on Data Engineering (ICDE), 2023, S. 3376–3388. doi: 10.1109/ICDE55515.2023.00259.

[2] Firebase, Cloud Firestore, Zuletzt aktualisiert am: 13.05.2025. Zuletzt abgerufen am: 21.05.2025. Adresse: https://firebase.google.com/docs/firestore?hl=de.

[3] Firebase, Cloud Firestore-Datenmodell, Zuletzt aktualisiert am: 13.05.2025. Zuletzt abgerufen am: 21.05.2025, 13.05.2025. Adresse: https://firebase.google.com/docs/firestore/data-model?hl=de.

[4] G. Cloud, Query and filter data, Zuletzt aktualisiert am: 15.05.2025. Zuletzt abgerufen am: 21.05.2025, 15.05.2025. Adresse: https://cloud.google.com/firestore/native/docs/query-data/queries.

[5] Firebase, Einführung in die Firebase Local Emulator Suite, Zuletzt aktualisiert am: 08.05.2025. Zuletzt abgerufen am: 21.05.2025, 8.05.2025. Adresse: https://firebase.google.com/docs/emulator-suite?hl=de.

[6] Firebase, Local Emulator Suite installieren, konfigurieren und integrieren, Zuletzt aktualisiert am: 08.05.2025. Zuletzt abgerufen am: 21.05.2025, 8.05.2025. Adresse: https://firebase.google.com/docs/emulator-suite/install_and_configure?hl=de.

[7] J. Richman, 7+ Google Firestore Query Performance Best Practices for 2024, Zuletzt aktualisiert am: 21.08.2024. Zuletzt abgerufen am: 21.05.2025, 21.08.2024. Adresse: https://estuary.dev/blog/firestore-query-best-practices/.

[8] G. Andersen und M. R. Team, How Data Models Affect Normalization & Denormalization in NoSQL Databases, Zuletzt aktualisiert am: 11.05.2025. Zuletzt abgerufen am: 23.05.2025, 11.05.2025. Adresse: https://moldstud.com/articles/p-how-data-models-affect-normalization-denormalization-in-nosql-databases .

[9] Firebase, FirestoreDataConverter, Zuletzt aktualisiert am: 22.07.2022. Zuletzt abgerufen am: 26.05.2025, 27.07.2022. Adresse: https://firebase.google.com/docs/reference/node/firebase.firestore.FirestoreDataConverter.

[10] Firebase, Daten mit Aggregationsabfragen zusammenfassen, Zuletzt aktualisiert am: 26.05.2025. Zuletzt abgerufen am: 26.05.2025, 26.05.2025. Adresse: https://firebase.google.com/docs/firestore/query-data/aggregation-queries?hl=de#use_the_count_aggregation.

[11] Firebase, Datenbank auswählen: Cloud Firestore oder Realtime Database, Zuletzt aktualisiert am: 13.05.2025. Zuletzt abgerufen am: 23.05.2025, 13.05.2025. Adresse: https://firebase.google.com/docs/firestore/rtdb-vs-firestore?hl=de.

[12] Firebase, Transactions and batched writes, Zuletzt aktualisiert am: 23.05.2025. Zuletzt abgerufen am: 24.05.2025, 2025. Adresse: https://firebase.google.com/docs/firestore/manage-data/transactions?hl=de.

[13] Firebase, Cloud Functions for Firebase, Zuletzt aktualisiert am: 18.05.2025. Zuletzt abgerufen am: 24.05.2025, 2025. Adresse: https://firebase.google.com/docs/functions.

Bild- und Icon-Quellen

- Firebase Logo und Farben -> https://firebase.google.com/brand-guidelines
- Icons -> https://www.flaticon.com/
- Menschen in Comic-Art von ChatGPT generiert-> https://chatgpt.com/