Technische Hochschule Augsburg

Fakultät für Informatik

Dokumentation

für die Semesteraufgabe im Fach NoSQL

NoSQL: Umsetzung der Semesteraufgabe mit Google Firestore

erstellt von

Peter Fischer -

Leonelle Tifani Kommegne Kammegne -

Michael Mertl - 2209076

Gregor Pfister -

Jana Sophie Schweizer - 2209427

Technische Hochschule Augsburg

An der Hochschule 1
D-86161 Augsburg
T +49 821 5586-0
F +49 821 5586-3222
www.tha.de
info@tha.de

Inhaltsverzeichnis

1 Ausgangssituation

In unserer neuen Position im Datenbank-Team haben wir von unserem Vorgänger eine relationale Kurs-Datenbank übernommen. Laut Vorgabe des Managements soll diese nun in eine moderne NoSQL-Datenbank überführt werden.

Da keine strukturierte Übergabe stattfand, stand uns lediglich ein SQL-Dump der alten Datenbank zur Verfügung. Um die Datenstruktur besser zu verstehen und einen Überblick über die enthaltenen Tabellen und deren Beziehungen zu gewinnen, haben wir diesen Dump zunächst in ein Entity-Relationship-Diagramm (ERD) überführt.

Abbildung 1 zeigt das daraus entstandene ER-Diagramm, das insgesamt neun relationale Tabellen umfasst:

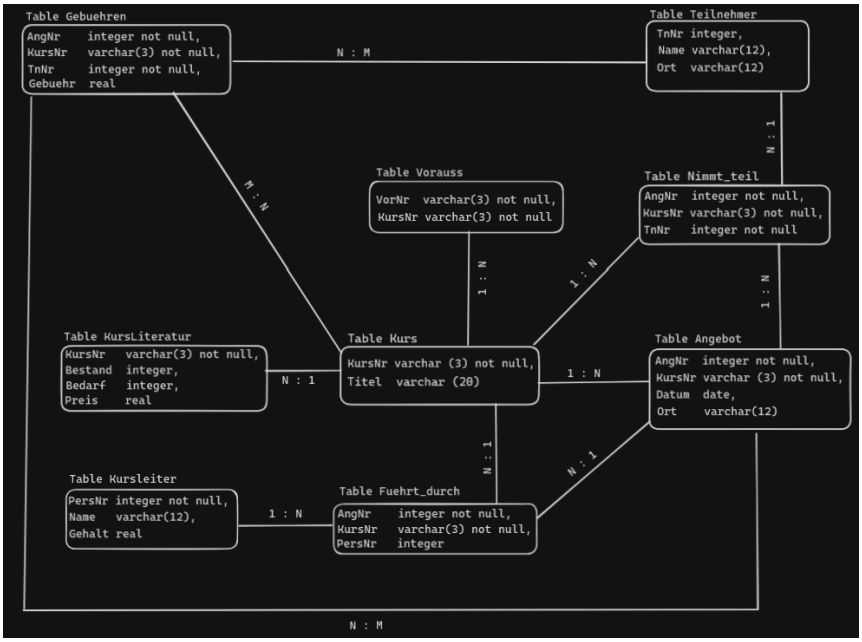

Abbildung 1: Entity-Relationship-Diagramm der ursprünglichen relationalen Struktur

Auf Basis dieser Analyse entschieden wir uns, die Struktur in eine dokumentenbasierte NoSQL-Datenbank zu übertragen. Dabei war es uns wichtig, die ursprüngliche logische Struktur möglichst beizubehalten, sie jedoch gleichzeitig so zu modellieren, dass sie den Prinzipien und Stärken eines dokumentenorientierten Datenmodells entspricht.

2 Entscheidung für eine NoSQL-Datenbank

Zur Auswahl einer geeigneten NoSQL-Datenbank haben wir zunächst das *DB-Engines Ranking* (https://db-engines.com/en/ranking) als Orientierungshilfe herangezogen. Dieses Ranking bietet eine fundierte Übersicht über die aktuell am weitesten verbreiteten Datenbanktechnologien.

Nach einer kurzen Recherche hinsichtlich Verbreitung, Dokumentation und Community-Support fiel unsere Wahl auf **Google Cloud Firestore**. Ausschlaggebend war neben den technischen Eigenschaften und der guten Integration in das Node.js-Ökosystem auch unser persönliches Interesse an der Arbeit mit Firebase-Technologien.

2.1 Google Cloud Firestore

Google Cloud Firestore ist eine dokumentenbasierte NoSQL-Datenbank, in der Daten in sogenannten *Collections* organisiert sind. Jede Collection kann beliebig viele *Dokumente* enthalten, die wiederum hierarchisch strukturierte *Subcollections* besitzen können.

Die einzelnen Dokumente bestehen aus Feldern in Form von Schlüssel-Wert-Paaren und ähneln in ihrer Struktur stark JSON-Objekten. Dieses flexible Datenmodell ermöglicht die effiziente Speicherung von semistrukturierten Informationen.

Zu den zentralen Merkmalen von Firestore gehören die Unterstützung verschachtelter Datenstrukturen, Arrays, Referenzen auf andere Dokumente sowie spezielle Datentypen wie Zeitstempel. Dadurch lassen sich auch komplexe, objektähnliche Datenmodelle direkt und modellnah abbilden **Kesavan.2023**, **Firebase.2025**, **FirebaseDatenmodell.2025**.

Die konkrete Umsetzung dieser Struktur in unserem Projekt wird in Kapitel 3 ausführlich dargestellt.

2.2 Abfragesprache von Firestore

Die Standardabfragesprache von Google Cloud Firestore ist keine deklarative Sprache wie SQL, sondern eine methodenbasierte API, die über verschiedene Programmiersprachen hinweg verfügbar

ist. Firestore stellt hierfür offizielle SDKs bereit, unter anderem für JavaScript, TypeScript, Python, Java und Kotlin **GoogleCloudQueries.2025**.

Die Interaktion mit der Datenbank erfolgt über eine Befehlskette aus Methodenaufrufen, mit denen sich Daten lesen, filtern, sortieren und paginieren lassen. Dabei orientieren sich die Methoden am zugrunde liegenden dokumentenbasierten Modell und bieten eine intuitive Möglichkeit, auf Daten zuzugreifen.

Abbildung 2 zeigt ein einfaches Beispiel für eine Abfrage in Firestore mit JavaScript. Hier wird nach allen Städten gesucht, bei denen das Feld `capital` auf `true` gesetzt ist, sortiert nach der Bevölkerungszahl:

```
const snapshot : QuerySnapshot<DocumentData, DocumentData> = await db.collection( collectionPath: 'cities')
    .where( fieldPath: 'capital', opStr: '==', value: true)
    .orderBy( fieldPath: 'population', directionStr: 'desc')
    .limit( limit: 5)
    .get();
```

Abbildung 2: Beispiel einer Abfrage in Firestore

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 bewusst für die Verwendung von **TypeScript** entschieden, um die Stärken der Firestore-API mit statischer Typprüfung zu kombinieren. Dies war insbesondere im Hinblick auf die Datenmigration aus der relationalen Struktur von Vorteil, da wir so die ursprünglichen Datentypen in Form von Interfaces abbilden und validieren konnten.

Wie genau wir mithilfe von TypeScript und sogenannten *Convertern* eine durchgehende Typsicherheit im Zugriff auf Firestore erreicht haben, wird ausführlich in Kapitel 4 erläutert.

2.3 Lokale Nutzung

Da Google Cloud Firestore eine cloudbasierte Datenbank ist, standen wir vor der Herausforderung, sie lokal zu betreiben – denn die Nutzung einer Cloud-Lösung war in unserem Fall nicht zulässig. Für diesen Zweck stellt Firebase die sogenannte *Local Emulator Suite* zur Verfügung. Dabei handelt es sich um eine Sammlung von Dienstemulatoren, die das Verhalten der echten Firebase-Dienste lokal nachbilden **EmulatorSuite.2025**.

Für unser Projekt reichte der Einsatz des Firestore-Emulators sowie der zugehörigen Benutzeroberfläche aus. Die Einrichtung erfolgte lokal durch die Installation von *Node.js*, dem *Java JDK*

sowie der *Firebase CLI*. Mit wenigen Konfigurationsbefehlen ließ sich anschließend die erforderliche Datei `firebase.json` generieren. Dieses Setup entspricht dem von Firebase empfohlenen Vorgehen für die lokale Entwicklung **EmulatorInsall.2025**.

Abbildung 3 zeigt einen Ausschnitt aus der Benutzeroberfläche des Emulators:

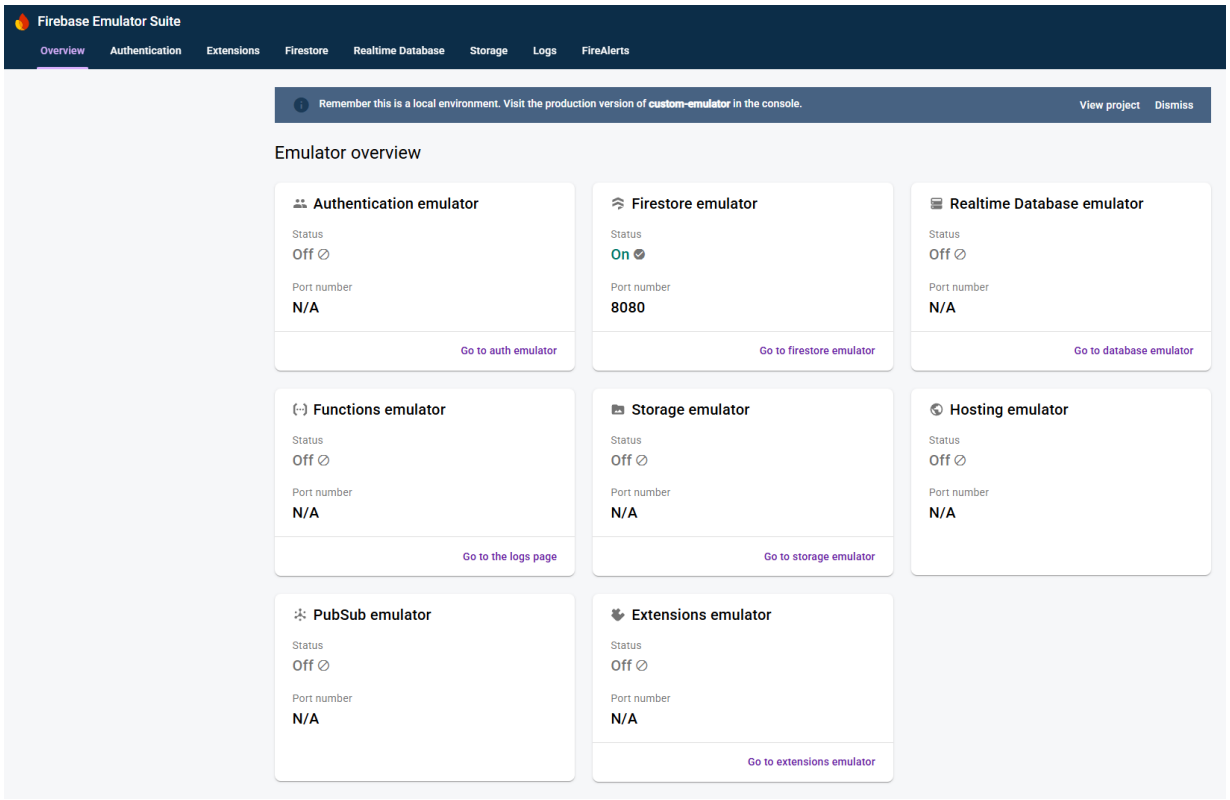

Abbildung 3: Auszug aus der UI der Local Emulator Suite

2.3.1 Versuchter alternativer Ansatz zur lokalen Nutzung

Neben der empfohlenen lokalen Installation der Emulator Suite haben wir auch versucht, die Firestore-Emulatorumgebung mithilfe von Docker bereitzustellen. Ziel war es, die notwendigen Tools nicht direkt auf unseren eigenen Systemen installieren zu müssen.

Dazu haben wir verschiedene Beiträge, Artikel und GitHub-Repositories recherchiert und unterschiedliche Ansätze ausprobiert, unter anderem:

1. https://github.com/PathMotion/firestore-emulator-docker
2. https://hub.docker.com/r/mtlynch/firestore-emulator/
3. https://medium.com/@jens.skott_65388/simplifying-firebase-emulation-with-docker-a-guide-to-local-development-and-testing-0c3c33fd92c7
4. https://github.com/ridedott/firestore-emulator-docker/blob/master/Dockerfile

Zwar ließ sich die Emulator Suite grundsätzlich innerhalb eines Docker-Containers starten und auch die Benutzeroberfläche aufrufen, jedoch traten in der praktischen Nutzung erhebliche Einschränkungen auf. Das Hauptproblem bestand darin, dass der Zugriff auf die Datenbank aus externen Skripten heraus nicht zuverlässig funktionierte – insbesondere die Weiterleitung der Ports (Expose) verursachte Schwierigkeiten.

Eine alternative Lösung wäre gewesen, sämtliche Skripte direkt innerhalb des Containers auszuführen. Dieses Vorgehen erschien uns jedoch in Bezug auf Wartbarkeit und Entwicklungsfluss zu umständlich. Aufgrund dieser Einschränkungen haben wir uns letztlich für den von Firebase vorgesehenen Weg mit lokaler Installation entschieden, der sich als unkompliziert und stabil erwiesen hat.

3 Aufbau der Datenstruktur

Für die Überführung der in Kapitel 1 beschriebenen relationalen Struktur haben wir die Vorteile eines dokumentenbasierten Datenmodells – wie es Google Firestore bietet – gezielt genutzt. Die finale Datenstruktur ist dabei nicht von Anfang an in ihrer jetzigen Form entstanden, sondern wurde im Laufe der Projektarbeit mehrfach überarbeitet und optimiert. Ziel war es, eine Lösung zu entwickeln, die sowohl den ursprünglichen Datenbeziehungen gerecht wird als auch eine effiziente Umsetzung der geforderten Abfragen ermöglicht.

Die resultierende Struktur ist in Abbildung 4 visualisiert:

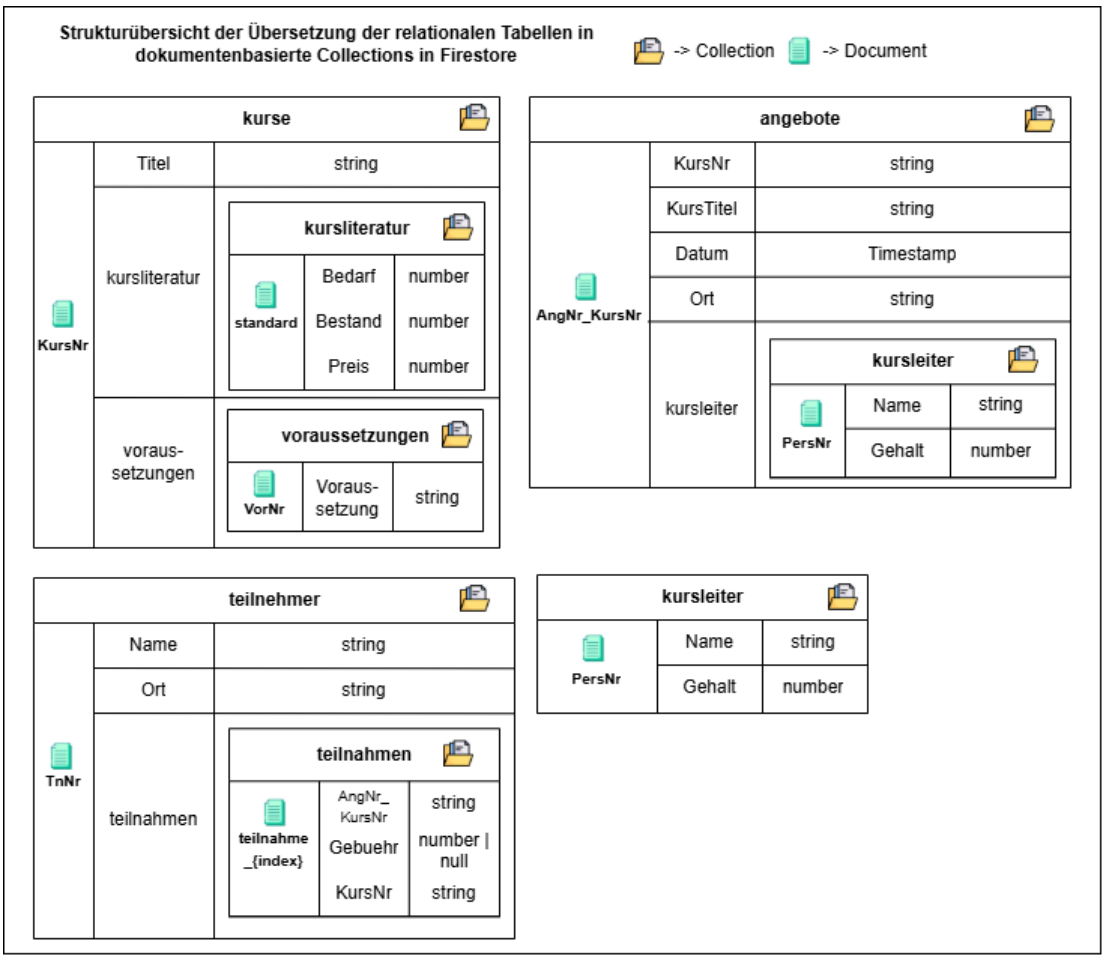

Abbildung 4: Struktur der Daten in Firestore

Aus den insgesamt neun relationalen Tabellen wurden vier Haupt-Collections abgeleitet: `kurse`, `angebote`, `teilnehmer` und `kursleiter`. Dabei wurden einige ursprünglich eigenständige Tabellen als Sub-Collections integriert:

- In `kurse` befinden sich die Sub-Collections `kursliteratur` (ursprünglich: `KursLiteratur`) und `voraussetzungen` (ursprünglich: `Vorauss`).
- In `teilnehmer` ist die Sub-Collection `teilnahmen` untergebracht (ursprünglich: `Nimmt_teil`), wobei die ehemals separate Tabelle `Gebuehren` nun als Feld innerhalb der Teilnahme gespeichert ist.
- Die Collection `kursleiter` entspricht nahezu direkt der alten Tabelle gleichen Namens.
- Die Collection `angebote` wurde am stärksten angepasst: Neben der Sub-Collection `kursleiter`, in der relevante Kursleiterdaten (redundant) gespeichert sind, wird dort zusätzlich der `KursTitel` direkt im Angebot abgelegt.

Die redundante Speicherung des Kurstitels sowie der Kursleiterdaten wurde bewusst gewählt. In Firestore sind komplexe JOINs, wie sie in relationalen Datenbanken üblich sind, nicht möglich. Um performante Abfragen zu ermöglichen und unnötige Mehrfachabfragen zu vermeiden, ist dieses Vorgehen sogar empfohlen und gilt als *Best Practice* im Umgang mit NoSQL-Datenbanken **Estuary.2025**.

Insgesamt entspricht die von uns gewählte Struktur dem Paradigma einer dokumentenorientierten Datenbank. Sie erwies sich im Verlauf der Umsetzung als robust und praxistauglich, insbesondere im Hinblick auf die Bearbeitung der gestellten Aufgaben. Gleichzeitig bringt die Entscheidung für redundante Datenhaltung gewisse Herausforderungen mit sich – insbesondere bei `update`- und `delete`-Operationen. Diese Aspekte werden in den Kapiteln 6 und 7 näher erläutert.

4 Typsicherheit und Abfragen mit TypeScript

Wie bereits in Kapitel 2.2 erwähnt, haben wir uns bei der Implementierung für TypeScript als Abfragesprache entschieden. Der Hauptgrund dafür war, dass Firestore standardmäßig keine Typsicherheit bietet. Das Datenbanksystem selbst ist schemalos, sodass die Validierung von Datentypen vollständig in der Anwendungsschicht erfolgen muss.

Da wir möglichst nah an den Datentypen der ursprünglichen relationalen Struktur bleiben wollten, haben wir in TypeScript entsprechende Interfaces für alle Tabellen definiert. Für die Datenbankkommunikation kam das offizielle npm-Paket `@google-cloud/firestore` (https://www.npmjs.com/package/@google-cloud/firestore) zum Einsatz, das wir mit unseren Typdefinitionen kombiniert haben, um eine durchgängige Typsicherheit zu gewährleisten.

Abbildung 5 zeigt einen Auszug unserer zentralen Typdefinitionen:

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

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

Abbildung 5: Auszug aus unseren Typdefinitionen

Damit diese Typen nicht nur in unserem Anwendungscode verwendet werden, sondern auch beim Zugriff auf die Firestore-Datenbank wirksam sind, haben wir zusätzlich sogenannte *Converter* implementiert. Diese ermöglichen es, typisierte Daten aus der Datenbank zu lesen und wieder zurückzuschreiben – und zwar im Einklang mit den definierten TypeScript-Interfaces.

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

Abbildung 6: Definition unserer Converter

Wie in Abbildung 6 zu sehen, implementiert unser Converter-Objekt zwei zentrale Methoden: `fromFirestore` konvertiert das rohe Datenobjekt aus Firestore in einen konkreten TypeScript-Typ, während `toFirestore` dafür sorgt, dass nur strukturkonforme Objekte in die Datenbank geschrieben werden.

Im Zusammenspiel mit der Methode `.withConverter()` des Firestore-SDK konnten wir so eine durchgängige Typprüfung bei allen Lese- und Schreibvorgängen realisieren. Abbildung 7 zeigt ein Beispiel für den praktischen Einsatz des Converters im Code:

```
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
```

Abbildung 7: Beispiel für die Verwendung eines Converters

Praktische Vorteile.

Der Einsatz dieser Technik brachte mehrere Vorteile mit sich:

- **Reduktion von Fehlern:** Typfehler, vergessene Felder oder falsche Datentypen wurden bereits zur Entwicklungszeit durch den TypeScript-Compiler erkannt.
- **Bessere Wartbarkeit:** Änderungen an Datenstrukturen waren dank der zentralen Typdefinitionen leicht nachvollziehbar und systemweit konsistent anpassbar.
- **Konsistenz in beiden Richtungen:** Die gleiche Datenstruktur wurde sowohl beim Einlesen als auch beim Schreiben verwendet – ohne doppelte Validierung oder manuelle Typzuweisungen.

Abgrenzung zum Firestore-Modell.

Firestore selbst stellt – anders als relationale Datenbanken – kein festes Schema zur Verfügung. Durch die Kombination aus TypeScript und Convertern konnten wir dieses Defizit vollständig auf Anwendungsebene kompensieren. Besonders im Rahmen der Migration einer bestehenden SQL-Datenbank erwies sich dieser Ansatz als äußerst hilfreich.

4.1 Initiales Laden der Daten nach Firestore

Nachdem die Datenstruktur definiert und die Nutzung von TypeScript-Typen sowie Convertern zur Gewährleistung der Typsicherheit erläutert wurde, bestand der nächste Schritt darin, die

Ausgangsdaten in die Firestore-Datenbank zu überführen. Dies war notwendig, um anschließend alle *read*-, *update*- und *delete*-Operationen ausführen zu können.

Die bereitgestellten Daten wurden zunächst in mehreren `.json`-Dateien organisiert. Mithilfe eines eigens entwickelten *Load-Skripts* konnten diese anschließend automatisiert in Firestore importiert werden. Dabei war es entscheidend, dass für jede Datei der passende `Converter` in Kombination mit den zuvor definierten TypeScript-Typen verwendet wurde.

Besonderes Augenmerk lag auf der korrekten Behandlung von Sub-Collections, da diese – anders als in relationalen Datenbanken – nicht automatisch mit dem Hauptdokument gespeichert werden, sondern explizit separat geschrieben werden müssen.

Obwohl dieser Initialimport technisch gesehen relativ einfach umzusetzen war, musste er sehr sorgfältig durchgeführt werden, um sicherzustellen, dass sämtliche Daten im richtigen Format und an der korrekten Stelle gespeichert werden. Nur so konnten wir garantieren, dass alle nachfolgenden Abfragen erwartungsgemäß funktionieren.

5 Herausforderungen bei den Read-Abfragen

Bei der Umsetzung der Leseabfragen in Firestore traten mehrere Einschränkungen zutage, die im Vergleich zu klassischen relationalen Datenbanken zusätzliche Komplexität mit sich brachten. Im Folgenden sind zwei zentrale Punkte exemplarisch erläutert:

1. **Keine Unterstützung für JOINs:** In SQL können Daten aus mehreren Tabellen mithilfe von `JOIN`-Operationen direkt miteinander verknüpft werden. In Firestore existiert eine solche Funktionalität nicht. Um beispielsweise die Informationen zu einem Angebot gemeinsam mit dem zugehörigen Kurstitel zu laden (wie in Abfrage **d**), mussten zunächst alle Dokumente aus der Collection `angebote` gelesen und anschließend für jedes einzelne Dokument das passende Kursdokument aus der Collection `kurse` manuell nachgeladen werden. Dies führt bei größeren Datenmengen zu einem erheblichen Mehraufwand und einer höheren Anzahl an Leseoperationen – was sich negativ auf die Performance und potenziell auf die Kosten auswirken kann.
2. **Eingeschränkte Aggregatfunktionen:** Firestore unterstützt inzwischen einfache Aggregationen wie `count()`, allerdings nur für die Gesamtanzahl von Dokumenten, die bestimmte Kriterien erfüllen. Komplexe Aggregationen wie `GROUP BY` oder `HAVING COUNT(*)`, wie sie in SQL üblich sind, sind in Firestore weiterhin nicht möglich. In Abfrage **i** (Kurse mit mindestens zwei Teilnehmern) musste daher in Firestore ein eigenes Zählerobjekt im Anwendungscode erstellt werden, das sämtliche Teilnahme-Dokumente durchläuft und die Anzahl pro Kursangebot manuell berechnet. Das verlagert die Logik vollständig in die Anwendungsebene und erhöht damit den Entwicklungsaufwand.

Darüber hinaus ist der direkte Zugriff auf Sub-Collections ebenfalls nicht ohne Weiteres möglich – etwa, wenn man aus einer übergeordneten Collection alle enthaltenen Sub-Dokumente aggregiert betrachten möchte. Auch hierfür sind zusätzliche Leseoperationen und individuelle Nachladeprozesse erforderlich. Das liegt daran, dass Abfragen in Firestore "flachßind und nur Dokumente aus einer bestimmten Sammlung oder Sammlungsgruppe zurückggeben und keine Daten aus untergeordneten Sammlungen **FirebaseComparison.2025**.

Dadurch entstehen, insbesondere bei komplexeren Abfragen wie den Teilaufgaben m) und n), in Schleifen verschachtelte mehrfache Datenbankzugriffe. Das ist notwendig, um in unserem Beispiel Kurse mit Angeboten und Kursleiten zu kombinieren, um so Beispielsweise zu jedem Angebot den Kurs (für den Kursnamen) und den Kursleiter (für den Kursleiternamen bzw das Gehalt des Kursleiters) nachzuladen. Um diese Mehrfachabfragen zu reduzieren haben wir uns entschieden

den Kurstitel zusätzlich redundant in der Collection `angebote` abzuspeichern. Dadurch konnten wir mehrere Datenbankzugriffe einsparen und beispielsweise in Aufgabe m die Anzahl der nötigen Datenbankzugriffe fast halbiert werden. Diese strategische Denormalisierung von Daten wird in dokumentenbasierten Datenbanken häufig verwendet um bestimmte häufig vorkommende Abfragen um bis zu 50% zu beschleunigen **MoldStud.2025**.

Diese Einschränkungen zeigen exemplarisch, dass NoSQL-Systeme wie Firestore zwar flexibel in der Datenmodellierung sind, jedoch bei komplexeren Auswertungen oft zusätzliche clientseitige Logik erforderlich machen.

6 Herausforderungen bei den Update-Abfragen

Ein wesentliches Merkmal dokumentenbasierter NoSQL-Datenbanken wie Firestore ist die bewusste Entscheidung für Redundanz, um Abfragen zu beschleunigen und komplexe JOINs zu vermeiden. Diese Modellierungsfreiheit bringt jedoch Herausforderungen mit sich – insbesondere bei `update`-Operationen.

Ein konkretes Beispiel in unserem Projekt ist der redundante `KursTitel`, der sowohl in der Collection `kurse` als auch mehrfach in der Collection `angebote` gespeichert wird. Wird der Titel eines Kurses geändert, muss sichergestellt werden, dass alle zugehörigen Angebote ebenfalls aktualisiert werden, um Inkonsistenzen zu vermeiden.

In relationalen Datenbanken würde dies durch eine zentrale Normalisierung und Fremdschlüsselbeziehungen automatisch sichergestellt werden. In Firestore hingegen ist die Konsistenz zwischen redundanten Feldern Aufgabe der Anwendung.

Zur Lösung dieses Problems bietet Firestore sogenannte *Batch Writes* und *Transaktionen* an. Bei einem Batch Write können mehrere Dokumente in einem einzigen atomaren Schritt aktualisiert werden. Transaktionen ermöglichen zusätzlich, innerhalb eines Konsistenzkontextes zu lesen und zu schreiben – was insbesondere dann hilfreich ist, wenn Bedingungen geprüft werden müssen, bevor geschrieben wird.

In unserem Projekt war der Einsatz von `Batch Writes` die bevorzugte Methode, da wir genau wussten, welche Felder aktualisiert werden müssen und keine Abhängigkeiten zwischen den Lese- und Schreibvorgängen bestanden. Änderungen am Kurstitel konnten so gezielt auf alle betroffenen `angebote`-Dokumente angewendet werden.

Trotzdem bleibt zu beachten, dass Firestore keine automatische Änderungsweitergabe (wie `ON UPDATE CASCADE` in SQL) unterstützt. Entwickler:innen müssen sich dieser Limitierung bewusst sein und gegebenenfalls eigene Routinen oder Skripte entwickeln, um die Konsistenz sicherzustellen.

7 Herausforderungen bei den Delete-Abfragen

Neben der Aktualisierung redundanter Daten stellt auch das Löschen von Dokumenten in Firestore eine besondere Herausforderung dar – insbesondere dann, wenn Daten mehrfach gespeichert oder in Sub-Collections organisiert sind.

Ein typisches Beispiel im Rahmen dieses Projekts ist das Löschen von einem Angebot. Während in einer relationalen Datenbank mit referenzieller Integrität und `ON DELETE CASCADE` automatisch alle abhängigen Einträge (z. B. zugehörige Teilnahmen oder Gebühren) entfernt werden können, existiert eine solche automatische Löschkaskade in Firestore nicht.

Wird ein Dokument in einer Haupt-Collection gelöscht (z. B. das Angebot für einen bestimmten Kurs), so bleiben zugehörige Dokumente in Sub-Collections oder redundanten Kopien (etwa in `Teilnehmer/Teilnahmen`) bestehen – es sei denn, sie werden explizit mitgelöscht. Das kann zu Inkonsistenzen und verwaisten Daten führen, wenn keine zusätzliche Logik zur Datenpflege implementiert wird.

Zur Umsetzung konsistenter Löschvorgänge muss also anhand einer eigenen Lösch-Logik-Implementierung festgelegt werden, dass alle zugehörigen Dokumente, Sub-Dokumente und Kopien ebenfalls gelöscht werden. Um dies zu verdeutlichen, beziehen wir uns direkt auf das Löschen eines Angebots in diesem Projekt. In diesem Fall muss nicht nur das Angebot selbst, sondern zusätzlich alle zugehörigen Teilnahmen und Gebühren entfernt werden. Ein Vorteil unserer Datenspeicherung liegt in der Speicherung der Teilnahme zusammen mit den Gebühren direkt im `Teilnehmer`-Dokument. Dadurch müssen wir lediglich die zughörigen Teilnahmen, vom betroffenen Angebot löschen, um auch die Gebühren zu entfernen.

Da hier grundsätzlich oft ein hoher Aufwand von Schleifen-Operationen (z. B. for-Schleifen) und darin enthaltenen Löschvorgängen zu Grunde liegt, bietet sich das Konzept der `Transaktionen` und `batch`-Operationen **Firebase.TransaktionenBatch** in Kombination für die Durchführung der Lösch-Operation an. Firestore ermöglicht es, mehrere Löschvorgänge in einer Transaktion zu bündeln, sodass sie entweder alle erfolgreich ausgeführt oder im Fehlerfall zurückgerollt werden. Dies stellt sicher, dass die Datenbank in einem konsistenten Zustand bleibt und keine Teilergebnisse hinterlässt. Dabei gilt es zu beachten, dass Firestore vorgibt, dass Lese-Operationen immer vor Schreib-Operationen in den Transaktionen stehen müssen und eine maximale Anfragegröße von 10 Mebibyte (MiB) pro Transaktion gilt. Alternativ könnte mit einem serverseitigen Mechanismus *Firebase Cloud Functions* **Firebase.CloudFunctions** gearbeitet werden. Diese Functions reagieren

automatisch auf Datenänderungen - wie beispielsweise in diesem Projekt einen Löschvorgang eines Angebots. Diese Lösung war in unserem Projekt jedoch nicht erlaubt, da ausschließlich lokal mit der Emulator Suite gearbeitet wird und dementsprechend keine Cloud-Integrationen erlaubt sind.

Somit liegt die Verantwortung für konsistentes Löschen bei der Anwendungsschicht. Diese Tatsache ist ein bedeutender Unterschied zu relationalen Systemen, der insbesondere bei komplexeren Datenmodellen berücksichtigt werden muss.

8 Fazit

Die Migration einer relationalen Kursdatenbank in eine dokumentenbasierte NoSQL-Datenbank wie Google Cloud Firestore stellte sich als ebenso herausfordernd wie lehrreich heraus. Während relationale Datenbanken durch ihr starres Schema und eingebaute Beziehungen wie `JOINs`, `GROUP BY` oder referenzielle Integrität eine hohe Datenkonsistenz und Abfrageflexibilität bieten, verlangt das Arbeiten mit Firestore ein grundsätzlich anderes Denken: Daten müssen häufiger redundant gespeichert, Abfragen logisch vereinfacht und viele Operationen in die Anwendungsschicht verlagert werden.

Durch die Verwendung von TypeScript und Convertern konnten wir diesem schemalosen Ansatz eine starke Typsicherheit entgegensetzen und so sowohl bei der Datenmodellierung als auch bei der Datenverarbeitung präzise und konsistente Strukturen gewährleisten. Besonders hilfreich war dieser Ansatz im Kontext der ursprünglichen relationalen Struktur, da wir so viele der alten Konzepte typisiert beibehalten konnten.

Die Herausforderungen im Bereich der Lese-, Update- und Delete-Operationen haben deutlich gemacht, dass NoSQL-Systeme zwar hohe Flexibilität bieten, dafür aber auf Kosten automatischer Konsistenzmechanismen. Durch manuelle Scripte, gezielte Datenmodellierung und die Nutzung von Batch-Operationen konnten wir dennoch eine robuste und nachvollziehbare Lösung implementieren.

Die Entscheidung für Firestore hat sich trotz der technischen Einschränkungen insgesamt als sinnvoll erwiesen – vor allem aufgrund der leichten Einstiegsmöglichkeiten, der guten lokalen Entwicklungsumgebung (Emulator Suite) sowie der vollständigen Kontrolle über Datenstruktur und Zugriff.

Abschließend lässt sich sagen, dass dokumentenbasierte Datenbanken wie Firestore ideal für Projekte mit semistrukturierten Daten und klar definierten Zugriffsmustern geeignet sind. Gleichzeitig ist es entscheidend, sich der Einschränkungen bewusst zu sein – insbesondere im Hinblick auf relationale Operationen, Aggregationen und automatische Konsistenz.

Das Projekt hat nicht nur unsere Kenntnisse in Firestore, TypeScript und NoSQL-Datenmodellierung vertieft, sondern auch ein praktisches Verständnis dafür geschaffen, wie sich klassische relationale Datenbankmodelle in moderne, flexible Cloudsysteme überführen lassen.