Technische Hochschule Augsburg

Fakultät für Informatik

Dokumentation

für die Semesteraufgabe im Fach NoSQL

NoSQL: Umsetzung der Semesteraufgabe mit Google Firestore

erstellt von

Peter Fischer - 2210654

Leonelle Tifani Kommegne Kammegne - 2209740

Michael Mertl - 2209076

Gregor Pfister - 2209779

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:

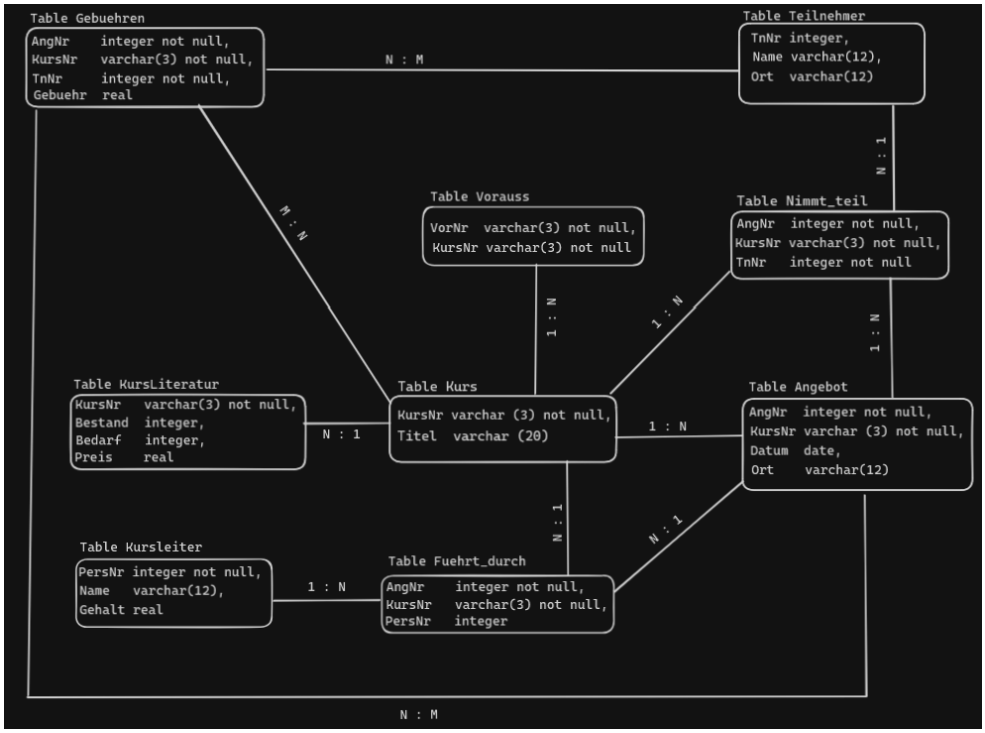

Abbildung 1: Entity-Relationship-Diagramm der ursprünglichen relationalen Struktur

Auf Basis dieser Analyse mussten wir uns für eine NoSQL-Datenbank entschieden. Die Entscheidung und die Gründe für die jeweilige NoSQL-Datenbank wird nachfolgend beschrieben in Kapitel 2 („Entscheidung für eine NoSQL-Datenbank“).

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, Google als Entwickler 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 *Sub-Collections* 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 [1], [2], [3]. Nachfolgend zeigt Abbildung 2 ein Datenstruktur-Beispiel in Firestore:

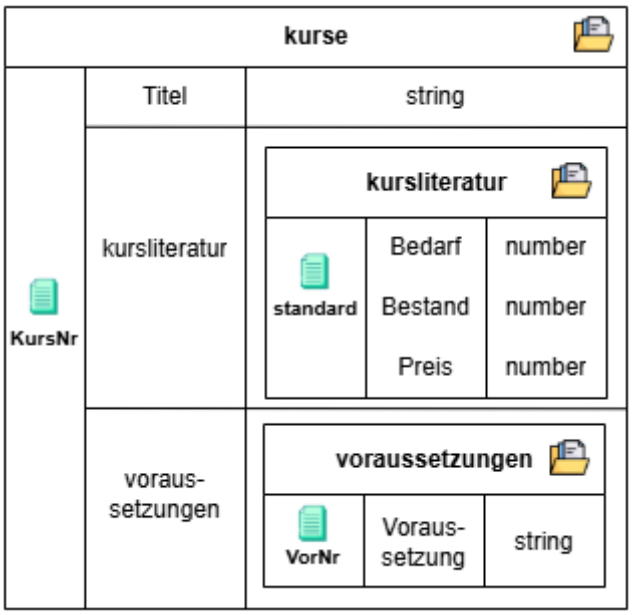

Abbildung 2: Beispiel einer dokumentenbasierten Firestore Datenstruktur

Abbildung 2 zeigt hierbei die *Collection* kurse, bei der die *Dokumente* jeweils einen Titel als string beinhalten und zwei *Sub-Collections*, die wiederum ihre eigenen *Dokumente* mit Schlüssel-Wert-Paare beinhalten. Die konkrete vollständige Umsetzung dieser Struktur in unserem Projekt wird in Kapitel 3 („Aufbau der Datenstruktur“) 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 [4].

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 3 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 3: 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 („Typsicherheit und Abfragen mit TypeScript“) 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 [5].

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 [6].

Abbildung 4 zeigt einen Ausschnitt aus der Benutzeroberfläche des Emulators:

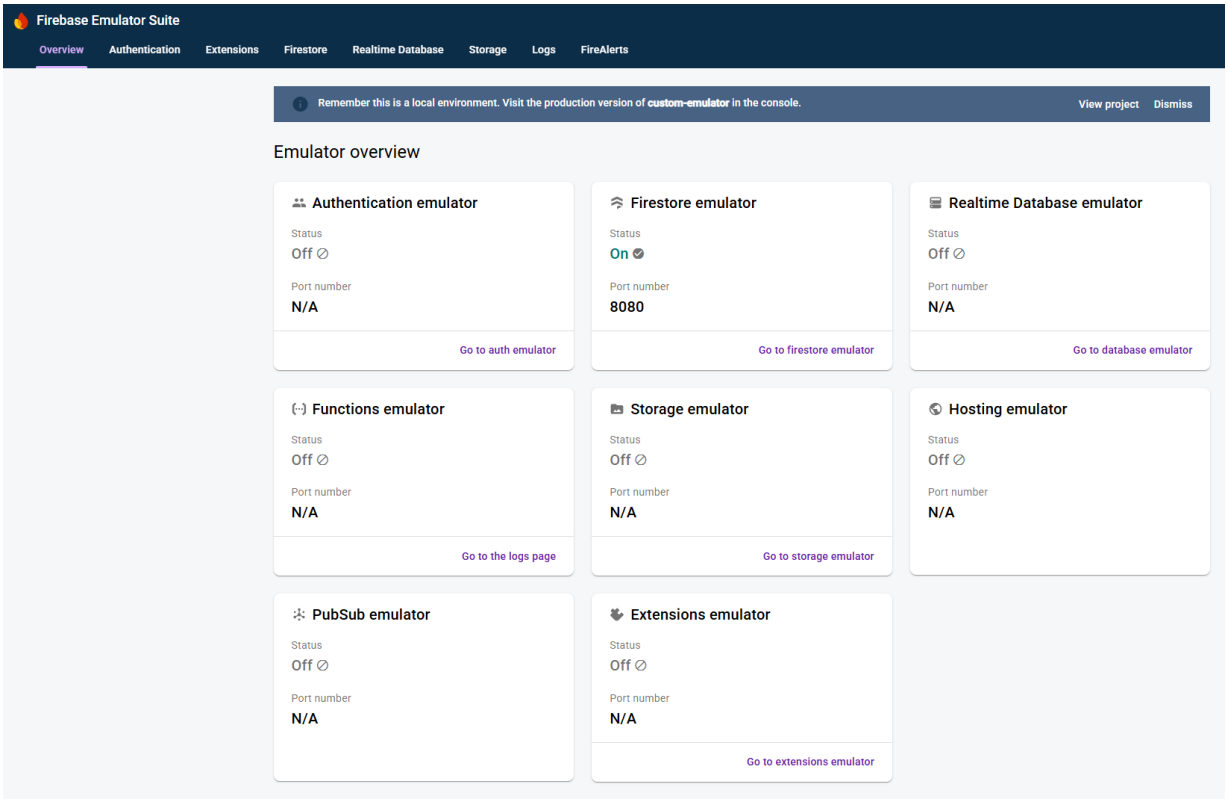

Abbildung 4: Auszug aus der UI der Local Emulator Suite

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

Diese verschiedenen Ansätze haben unterschiedlichen Projektmitglieder versucht und jeder ist auf andere Probleme gestoßen. Manche hatten bereits Probleme bei der Installation und haben zu keinem Zeitpunkt einen lauffähigen Docker erreicht und manche hatten Probleme bei der Anwendung des Containers. Dabei ließ sich die Emulator Suite zwar 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 verursachte Schwierigkeiten.

Eine alternative Lösung wäre dazu gewesen, sämtliche Skripte direkt innerhalb des Containers auszuführen. Dieses Vorgehen erschien uns jedoch in Bezug auf Wartbarkeit, Entwicklungsfluss und auf Grund der Tatsache, dass andere Projektmitglieder den Docker nicht ausführen konnten, 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 5 visualisiert:

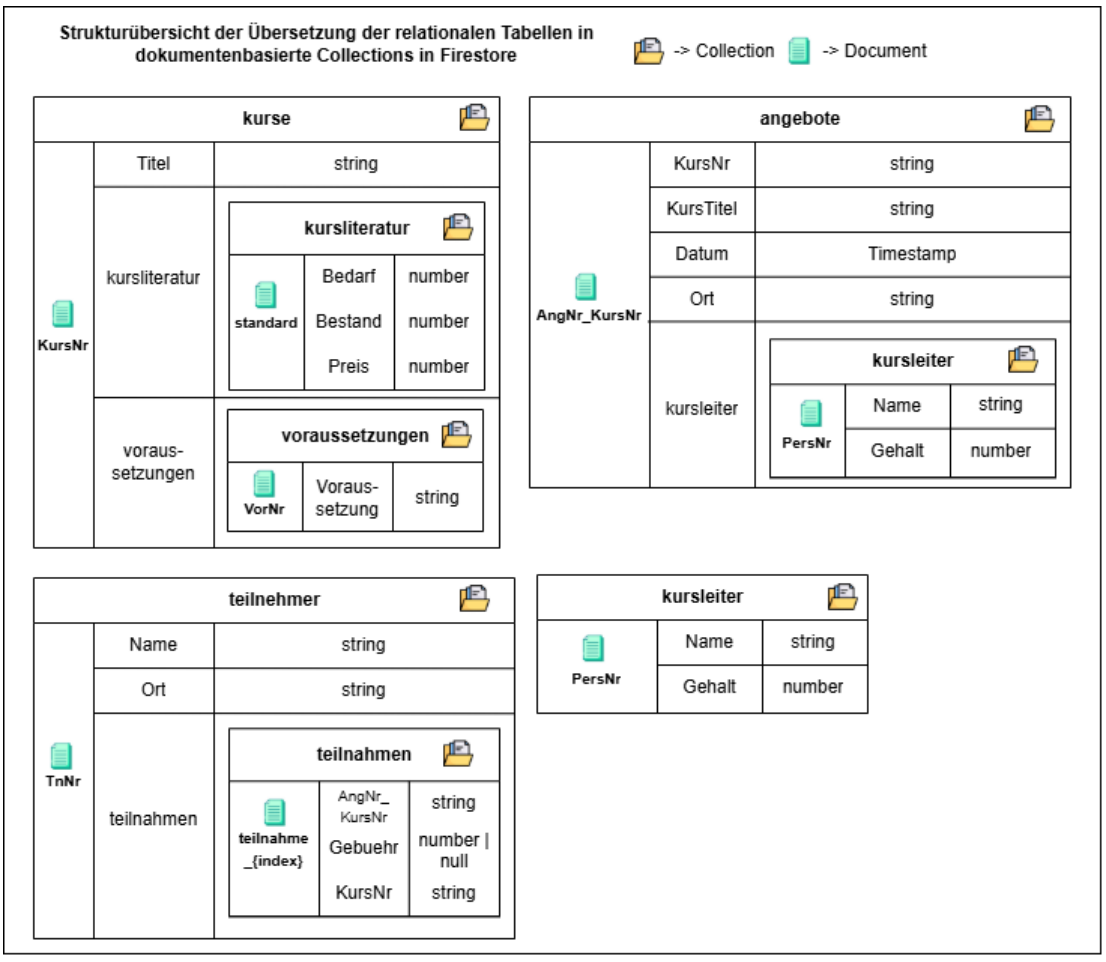

Abbildung 5: Struktur der Daten in Firestore

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 eines *teilnahme Dokuments* 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 diese strategische Denormalisierung sogar empfohlen und gilt als *Best Practice* im Umgang mit NoSQL-Datenbanken, um bestimmte, häufig vorkommende Abfragen um bis zu 50% zu beschleunigen [7], [8].

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 im Kapitel 6 („Herausforderungen bei den Update- & Delete-Abfragen“) 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 6 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 6: Auszug aus unseren Typdefinitionen

Damit diese Typen durchgängig in unserem Anwendungscode verwendet werden, ohne dass wir ständig die Daten beim Schreiben oder Lesen *casten* müssten, haben wir 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.

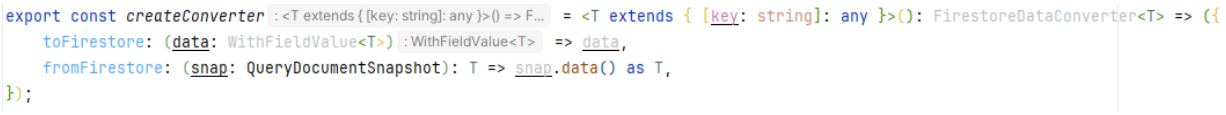

Abbildung 7: Definition unserer *Converter*

Wie in Abbildung 7 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 [9].

Im Zusammenspiel mit der Methode *.withConverter()* des Firestore-SDK konnten wir so eine durchgängige Typprüfung bei allen Lese- und Schreibvorgängen realisieren. Abbildung 8 zeigt ein Beispiel für den praktischen Einsatz des *Converters* im Code:

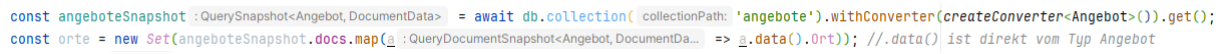

Abbildung 8: Beispiel für die Verwendung eines *Converters*

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. Abbildung 9 zeigt die Schritte zum initialen Laden der Kurs-Daten nach Firestore auf:

```typescript
// Converter erstellen für die Kurs-Daten
const kursConverter : FirestoreDataConverter<Kurs, DocumentData...> = createConverter<Kurs>();

// Aufrufen der Funktion zum Laden der Kurs-Daten
async function main() : Promise<void> {  no usages  new *
    await loadStructuredData<Kurs>(
        collectionName: 'kurse',
        filePath: 'data/kurse.json',
        kursConverter
    );
}

// Umsetzung des konkreten Schreibens der Kurs-Daten in Firestore
async function loadStructuredData<T>(  Show usages  new *
    collectionName: string,
    filePath: string,
    converter: FirestoreDataConverter<T>
) : Promise<void> {
    console.log( message: `  Lade ${collectionName} aus Datei: ${filePath}`);

    try {
        const data : Record<string, T> = loadJsonFile<T>(filePath);

        for (const [id, document] of Object.entries(data)) {
            const docRef : DocumentReference<T, DocumentData> = db.collection(collectionName).doc(id).withConverter(converter);

            if (collectionName === 'kurse') {
                const kurs = document as unknown as Kurs;
                const { kursliteratur, voraussetzungen, ...rest } = kurs;

                await docRef.set(rest as T);

                if (kursliteratur && Object.keys(kursliteratur).length > 0) {
                    await docRef.collection( collectionPath: 'kursliteratur').doc( documentPath: 'standard').set(kursliteratur);
                }

                if (voraussetzungen && voraussetzungen.length > 0) {
                    for (const v of voraussetzungen) {
                        await docRef.collection( collectionPath: 'voraussetzungen').doc(v).set({ Voraussetzung: v });
                    }
                }
            }
            // ... (weitere Collections hier behandeln)
```

Abbildung 9: Beispiel für das Laden der Daten in die Firestore Datenbank

Abbildung 9 zeigt die Erstellung des *Kurs-Converters*, der anschließend in der *main-Methode* des TypeScript Skripts genutzt wird, um die *kurse.json* mithilfe der *loadStructuredData* Funktion nach Firestore zu schreiben. Hierbei werden in der Funktion die einzelnen Schritte für die *Sub-Collections kursliteratur und voraussetzungen* ebenfalls veranschaulicht, um die gesamten Kurs-Daten ordnungsgemäß initial zu laden.

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 drei zentrale Punkte exemplarisch erläutert, die in unserem Projekt den meisten Einfluss hatten:

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 über Teilnehmer zu erhalten, die einen Kurs am eigenen Wohnort gebucht haben (Aufgabe g) müssen mehrere Leseoperationen mit clientseitiger Logik durchgeführt werden, um den gewünschten Output zu erhalten. Nachfolgend veranschaulicht Abbildung 10 die konkrete Umsetzung der Aufgabe **g)** auf Grund von fehlender *JOIN*-Funktionalität in Firestore:

```
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}`);
        }
    }
}
```

Abbildung 10: Umsetzung von *JOINs* in Firestore anhand Aufgabe g)

   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. Wie bereits in Kapitel 3 („Aufbau der Datenstruktur“) erwähnt, haben wir deshalb zum Teil Redundanzen in unserer Datenstruktur für effizientere Abfragen.

2. **Eingeschränkte Aggregatfunktionen:** Firestore unterstützt inzwischen einfache Aggregationen wie *count()*, allerdings nur für die Gesamtanzahl von Dokumenten, die bestimmte Kriterien erfüllen [10]. Komplexe Aggregationen wie *GROUP BY* oder *HAVING COUNT(*)*, wie sie in SQL üblich sind, sind in Firestore weiterhin nicht möglich. In Aufgabe **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. Nachfolgend veranschaulicht Abbildung 11 die konkrete Umsetzung der Aufgabe **i)** auf Grund von fehlender eingeschränkter Aggregatfunktionen in Firestore:

```
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`);
    }
}
```

Abbildung 11: Umsetzung von *Count* in Firestore anhand Aufgabe i)

Das verlagert die Logik vollständig in die Anwendungsebene und erhöht damit den Entwicklungsaufwand.

3. **Kein direkter Zugriff auf *Sub-Collections* innerhalb *Collections*:** In Firestore ist der direkte Zugriff auf *Sub-Collections* 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 „flach“ sind und nur Dokumente aus einer bestimmten Sammlung oder Sammlungsgruppe zurückggeben werden können und keine Daten aus untergeordneten Sammlungen [11].
   Dadurch entstehen, insbesondere bei komplexeren Abfragen wie den Teilaufgaben **m)** und **n)**, in Schleifen verschachtelte mehrfache Datenbankzugriffe. Durch Anpassung unserer Datenstruktur und die Speicherung von Informationen redundant, konnten wir hier etwas entgegenwirken und die Anzahl der Iterationen verringern (siehe Kapitel 3: „Aufbau der Datenstruktur“).
   Will man allerdings für alle gleichnamigen *Sub-Collections* eine Operation ausführen, bei der die Informationen der *Haupt-Collection* keine Rolle spielen, z.B. in unserem Fall die gesamt Teilnahmen eines Angebots betrachten, ist das direkt möglich und ohne

mehrfache Nachladeprozesse. Hierzu stellt Firestore den Befehl *collectionGroup* bereit, der alle gleichnamigen *Collections* in der ganzen Datenbank direkt lädt [12].

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- & Delete-Abfragen

Bei den gestellten Update- und Delete-Aufgaben handelte es sich um sehr einfache, die zum Großteil jeweils nur eine *Collection* betrafen und sehr einfach durchgeführt werden konnten.

Hätte es sich um Update- oder Delete-Abfragen gehandelt, die mehrere *Collections* betroffen hätten bzw. Daten, die von uns mehrfach gespeichert werden, dann wären wir auf neue Herausforderungen gestoßen, um Datenintegrität zu gewährleisten, da es bei Firestore kein *ON DELETE CASCADE* oder *ON UPDATE CASCADE* gibt. Nachfolgend wird ein genau solches Beispiel für den Delete-Fall, der aber auch identisch auf die Update-Operation übertragen werden kann bei der Umsetzung, aufgezeigt, um zu veranschaulichen, wie man solche Probleme mit Firestore umsetzen würde.

Beispiel: Löschen von einem Angebot

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 sowie der redundant gespeicherte Kursleiter im Angebot 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 zugehörigen 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* [13] 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. Nachfolgend zeigen Abbildungen 12, 13 und 14 die nötigen Schritte auf, um eine Kombination aus einer *Transaktion* und *Batch-Operationen* für die ordnungsgemäße Löschung eines Angebots:

```
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;
    }
}
```

Abbildung 12: Schritt 1: Angebote und Teilnehmer extrahieren

```
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).`);
        }
    `
```

Abbildung 13: Schritt 2: Transaktion starten und Angebot Dokument löschen

```
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();
}
```

Abbildung 14: Schritt 3: Alle Teilnahmen durchlaufen und zugehörige auf Lösch-Liste der Batch-Queue setzen mit commit für Durchführung am Ende

Alternativ könnte mit einem serverseitigen Mechanismus *Firebase Cloud Functions* [14] gearbeitet werden. Diese *Functions* reagieren automatisch auf Datenänderungen - wie beispielsweise in diesem Projekt einen Löschvorgang eines Angebots. Nachfolgend zeigt Abbildung 15 die Umsetzung einer solchen *Cloud Function*:

```
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;
```

Abbildung 15: Umsetzung einer *Cloud Function* mit Trigger beim Löschen 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.

7 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 und vor allem Performanz. Durch manuelle Skripte und gezielte angepasste sowie redundante Datenmodellierung konnten wir hier aber ebenfalls viel erreichen und den Einschränkungen entgegenwirken.

Abschließend lässt sich sagen, dass dokumentenbasierte Datenbanken wie Firestore ideal für Projekte mit semistrukturierten Daten und klar definierten Zugriffsmustern geeignet sind. Das Projekt hat nicht nur unsere Kenntnisse in Firestore, TypeScript und NoSQL-Datenmodellierung vertieft, sondern auch ein praktisches Verständnis dafür geschaffen, wie sich ein klassisches relationales Datenbankmodell in eine moderne, flexible NoSQL Datenbank überführen lässt.

Literaturverzeichnis

[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 (siehe S. 3).

[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 (siehe S. 3).

[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 (siehe S. 3).

[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 (siehe S. 4).

[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 (siehe S. 5).

[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 (siehe S. 5).

[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/ (siehe S. 9).

[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 (siehe S. 9).

[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 (siehe S. 11).

[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 (siehe S. 14).

[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 (siehe S. 15).

[12] Firebase, *Einfache und kumulierende Abfragen in Cloud Firestore ausführen*, Zuletzt aktualisiert am: 26.05.2025. Zuletzt abgerufen am: 26.05.2025, 26.05.2025. Adresse: https://firebase.google.com/docs/firestore/query-data/queries?hl=de#collection-group-query (siehe S. 16).

[13] 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 (siehe S. 17).

[14] 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 (siehe S. 19).