

PERSONAL MULTI-TECHNOLOGY PROJECTS

Personal Catalogue: React & Firebase

A real, working catalogue for books, CDs, DVDs, and Blu-rays — a managed Firestore document store held up chapter by chapter against its own self-managed MongoDB sibling, a real client-SDK-only architecture with no backend server of its own, Security Rules built to close every gap Firestore's own lack of a schema layer leaves open, and a real deployment mounted onto an existing Astro site, closing with an honest, itemized self-managed-vs-managed scorecard across the whole course.

10 Chapters

Philip Osztromok

Generated with Claude

CHAPTER 1: PROJECT OVERVIEW & FIREBASE SETUP

Personal Catalogue: React & Firebase

Chapter 1 · Project Overview & Firebase Setup

This is the fourth and last of four courses building the exact same personal cataloguing tool in four genuinely different architectures — Personal Catalogue (PHP & MySQL), Personal Catalogue (Django & PostgreSQL), and Personal Catalogue (React, Express & MongoDB) are all already complete. That third sibling already made the case for a document database over a rigid relational schema; this course takes the same underlying idea — genuinely differently-shaped item records — and answers it with a **managed** document database instead of a self-run one, so the two can be read side by side as self-managed vs. managed NoSQL.

What the App Actually Does

The shared spec every course in the quartet builds toward, unchanged from the other three:

- **Catalogue four kinds of item** — books, CDs, DVDs, and Blu-rays — so it's possible to quickly check "do I already own this CD?" or "what Python books do I have?"
- **Tag books** with free-form labels (a FastAPI book might get Python, Web Development, Programming, Web Frameworks) so related books can be found by topic later.
- **Add items manually** for the first release — title, author/artist, format-specific details, tags where relevant.
- **Search and browse** the collection to answer the "do I have this?" question quickly.

Barcode scanning is explicitly out of scope for this first release — a real, obvious way to add items faster, but deliberately deferred rather than allowed to hold up a release that needs to ship quickly. This is a genuinely personal, single-user tool — nothing in this spec, or in any of its three completed siblings, calls for real user accounts, so this course doesn't build any either.

Why Firestore for This One

The MongoDB sibling's own real argument still holds here: a book's fields (author, ISBN, publisher) and a DVD's fields (director, runtime, disc format) genuinely don't overlap much beyond title and year, so forcing all four item types into one rigid, shared shape either wastes a lot of null columns or splits into several tables that need joining back together for any "show

"me everything" view. Firestore, Firebase's own document database, gives the identical genuine flexibility — a single `items` collection can hold a book document and a DVD document side by side, each with only the fields that item type actually needs.

Where the two variants genuinely diverge is what happens *after* that flexibility is granted. The MongoDB sibling's own Chapter 2 built real, per-type schema validation on top of its flexible collection using Mongoose discriminators — a document that doesn't match its declared item type's schema is rejected before it ever reaches the database. Firestore has no built-in equivalent to that at all: any document, in any shape, can be written to any collection, and Firestore itself will never object. Whatever validation this course ends up wanting has to come from somewhere else — Chapter 2 goes deep on exactly what "somewhere else" means once real code is being written against it.

FLEXIBLE IS NOT THE SAME AS VALIDATED

"Firestore is schemaless" is true, but it's easy to read that as a pure advantage over the MongoDB sibling's own Mongoose-enforced schemas. It's really a genuine tradeoff: nothing stops a bug from accidentally writing a CD document with a DVD's own field names, and Firestore's own database layer will accept it without complaint. Chapter 2 covers the real options for closing that gap back up.

COURSE	STACK	DATA MODEL
Personal Catalogue: PHP & MySQL	Classic LAMP	Normalized relational tables, real foreign keys
Personal Catalogue: Django & PostgreSQL	Python/Django ORM	Normalized relational tables via the ORM
Personal Catalogue: React, Express & MongoDB	MERN-style	One heterogeneous document collection, self-managed, with per-type schema validation via Mongoose discriminators
Personal Catalogue: React & Firebase	React + managed BaaS	One heterogeneous Firestore collection, managed by Firebase itself, with no built-in schema validation at all

Where Does the Backend Actually Live?

The MongoDB sibling needed a real Express server regardless — MongoDB isn't reachable directly from a browser, so some backend process was always going to exist to hold the

connection and expose CRUD routes. Firestore removes that requirement outright: its official client SDK can read and write documents directly from React, with no server of this course's own in between at all. That's a genuine architectural fork worth naming honestly now rather than assuming an answer — direct client-to-Firestore access is faster to build but puts more weight on Firestore's own Security Rules to do the job an Express route's own validation code would otherwise do; a small backend API layer keeps that validation in familiar, testable code at the cost of reintroducing the very server this variant doesn't strictly need. Chapter 3 resolves this for real rather than leaving it open for the rest of the course.

A QUESTION THIS COURSE DOESN'T ANSWER YET

Every real secret this project might need — an API key for a barcode-lookup service, say — belongs behind a server, never in browser-shipped code. But barcode scanning is explicitly deferred past this course's own first release, and nothing else in the shared spec needs an external, secret-holding API call. Whether that means this variant can skip Cloud Functions (Firebase's own serverless functions, the natural home for a real secret) entirely for v1 — genuinely more BaaS-pure than the site's own Food Tracker (React + Firebase) course, which needed one from its own Chapter 3 to proxy two external lookups — is worth watching for as later chapters actually get built, not asserted here.

A Real Deadline, Not Just a Framing Device

This project genuinely needs a working first release quickly, exactly as its three completed siblings each describe. That constraint shapes real decisions here too: a plain add-item form before any real polish, and — a genuine, welcome side effect of picking Firebase specifically — no server of this course's own to write, configure, and keep running, which is itself real time saved against every one of its three siblings. Chapter 8 covers where the remaining corners were deliberately cut given that time pressure.

Setting Up a Firebase Project

Create a project at the Firebase console, then enable Firestore in **production mode** — not test mode, which allows unrestricted reads and writes and would leave real validation entirely unaddressed from day one. With the project created, bring the Firebase SDK into the React app:

```

npm install firebase

// firebase.js
import { initializeApp } from "firebase/app";
import { getFirestore } from "firebase/firestore";

const firebaseConfig = {
  apiKey: "AIzaSy...",
  authDomain: "catalogue-xxxx.firebaseio.com",
  projectId: "catalogue-xxxx",
};

const app = initializeApp(firebaseConfig);
export const db = getFirestore(app);

```

THAT CONFIG OBJECT IS NOT A SECRET

Pasting `apiKey` directly into frontend source code looks alarming coming from a "never expose your API key" mindset. It isn't one: Firebase's client-side config values identify *which* Firebase project a request is talking to — they don't grant access to anything by themselves. Actual access control is enforced entirely by Security Rules, evaluated on Firebase's own servers for every read and write regardless of what config values the client presents. A genuine secret — a real, third-party API key that must never reach the browser — is exactly what Cloud Functions exist for, which is exactly the open question raised above.

DEVELOP AGAINST EMULATORS, NOT THE LIVE PROJECT

`npm install -g firebase-tools`, then `firebase login` and `firebase init emulators` sets up a local Firestore emulator. Building against that instead of the real cloud project from day one avoids racking up usage and avoids polluting real catalogue data with test writes while this course's own schema is still being worked out in Chapter 2.

Where This Course Is Headed

Data modeling in Firestore, held up directly against the MongoDB sibling's own schema (Chapter 2); resolving the client-SDK-vs-backend-API question raised above for real (Chapter 3); one React form handling several genuinely different item shapes (Chapter 4); tags — attaching,

managing, and filtering by tag on books (Chapter 5); search, and Firestore's own real, honest query limitations compared with MongoDB's richer query language (Chapter 6); the list and detail views in React (Chapter 7); fast manual entry given the project's own real deadline (Chapter 8); deployment via Firebase Hosting (Chapter 9); and a capstone integrating this catalogue into the existing Astro-based site, closing with a direct, final comparison of self-managed MongoDB against managed Firestore across the whole project (Chapter 10).

Hands-On Exercises

EXERCISE 1

Explain the real, load-bearing difference between the MongoDB sibling's own Mongoose discriminators and a plain Firestore collection — specifically, what guarantee discriminators give at write time that Firestore does not, and what has to replace that guarantee in a Firestore-based app.

 [View solution](#)

EXERCISE 2

Create a Firebase project with Firestore enabled in production mode, wire up the client SDK in a React app, and confirm a working `db` reference. Then explain why Firebase's client-side config object is safe to include directly in frontend source code, when a real third-party API key is not.

 [View solution](#)

EXERCISE 3

Given that barcode scanning is deferred and nothing else in the shared spec needs a real, secret-holding API call, explain why this variant might not need any Cloud Functions at all for its first release — and name the one real thing that would change that answer once barcode scanning is eventually picked up.

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- **The shared app** — a personal catalogue for books/CDs/DVDs/Blu-rays, tags on books, manual entry first, barcode scanning deferred, single-user throughout
- **Why this variant** — Firestore's own document flexibility matches the MongoDB sibling's, but managed by Firebase rather than self-run
- **The real tradeoff** — Firestore has no built-in schema validation the way Mongoose discriminators provide; something else has to fill that gap (Chapter 2)
- **The open architectural question** — direct client-to-Firestore writes vs. a small backend API layer, resolved in Chapter 3
- **No Cloud Functions confirmed needed yet** — barcode scanning, the one feature that would need a real secret, is deferred past this course's first release
- **Deployment target** — Firebase Hosting (Chapter 9)
- **Next chapter:** Data Modeling in Firestore, Compared Directly Against the MongoDB Sibling's Own Schema

CHAPTER 2: DATA MODELING

Personal Catalogue: React & Firebase

Chapter 2 · Data Modeling

Chapter 1 promised two things for this chapter: the real Firestore document shapes, built to sit side by side with the MongoDB sibling's own schema for direct comparison, and a real answer to the validation gap that comparison exposed. This chapter delivers both — though not every piece of the second one lands here; where the resulting validation actually gets *called from* is still an open architectural question, resolved for real in Chapter 3.

What the MongoDB Sibling Built

Personal Catalogue (React, Express & MongoDB)'s own Chapter 2 built a base `Item` schema (`title`, `releaseYear`, `notes`) and four Mongoose **discriminators** — `Book`, `Cd`, `Dvd`, `Bluray` — each adding real, type-specific fields on top of that shared base, all writing into one physical `items` collection distinguished by a stored `itemType` field. Every discriminator schema still enforces its own required fields at write time: a `Cd` document with no `artist` is rejected by Mongoose before MongoDB ever sees it.

This chapter builds the same real shapes — the same field names, the same required fields, even the same `itemType` field name — in Firestore. What Firestore can't reuse is the enforcement mechanism itself: there is no Firestore equivalent of `Item.discriminator()`. A Firestore collection has no declared shape at all, for any document, ever.

The Same Shapes, No Enforcement

Firestore's own flat document model actually makes reproducing the shapes easy — a document can hold any fields, so the base fields and a given item type's own fields just sit together directly on one document, with no separate base-versus-subtype schema objects needed at all:

```
// addItem.js
import { collection, addDoc } from "firebase/firestore";
import { db } from "../firebase";
```

```

const itemsRef = collection(db, "items");

// A Book document
await addDoc(itemsRef, {
  itemType: "Book",
  title: "FastAPI: Modern Python Web Development",
  releaseYear: 2023,
  notes: "",
  author: "Bill Lubanovic",
  isbn: "978-1098135508",
  publisher: "O'Reilly Media",
  tags: ["Python", "Web Development", "Web Frameworks"],
});

// A Cd document, same collection
await addDoc(itemsRef, {
  itemType: "Cd",
  title: "The Suburbs",
  releaseYear: 2010,
  artist: "Arcade Fire",
  tracklist: ["The Suburbs", "Ready to Start", "Modern Man"],
});

```

Both documents land in the same `items` collection, side by side, exactly as the MongoDB sibling's own discriminators arrange things — but nothing about that `addDoc` call checked that a `Cd` document actually needs an `artist`, or that `itemType` is even one of the four real values this app understands. A typo — `itemType: "CD"`, capitalized differently from every other document — would be accepted by Firestore exactly as readily as a correctly-typed one.

```

items collection (one Firestore collection, four document shapes)
├─ { itemType: "Book"   } title, releaseYear, notes, author, isbn, publisher, tags[]
├─ { itemType: "Cd"     } title, releaseYear, notes, artist, tracklist[]
├─ { itemType: "Dvd"    } title, releaseYear, notes, director, runtimeMinutes, regionCode
└─ { itemType: "Bluray" } title, releaseYear, notes, director, runtimeMinutes, resolution

```

Field for field, this is the identical shape the MongoDB sibling settled on. The diagram above could be relabeled "MongoDB" and nothing in it would need to change — the real divergence

between the two courses isn't in what gets stored, it's in what happens the instant before storage.

Closing the Validation Gap: a Real Shape Validator

Chapter 1 left this open rather than asserting an answer. Here's the answer, at least at the "what should be checked" layer: a small, standalone function that knows this app's own real rules — which fields each item type requires, and which fields have a restricted set of valid values, the direct equivalent of Mongoose's own `required: true` and `enum` options:

```
// validateItem.js
const REQUIRED_BY_TYPE = {
  Book: ['author'],
  Cd: ['artist'],
  Dvd: [],
  Bluray: [],
};

const VALID_ITEM_TYPES = Object.keys(REQUIRED_BY_TYPE);

export function validateItem(data) {
  if (!data.title) {
    throw new Error('Every item needs a title.');
```

```
  }
  if (!VALID_ITEM_TYPES.includes(data.itemType)) {
    throw new Error(`itemType must be one of ${VALID_ITEM_TYPES.join(', ')}, got "${data.itemType}"`);
  }
  for (const field of REQUIRED_BY_TYPE[data.itemType]) {
    if (!data[field]) {
      throw new Error(`A ${data.itemType} needs a ${field}.`);
    }
  }
  if (data.itemType === 'Bluray' && data.resolution && !['1080p', '4K'].includes(data.resolution)) {
    throw new Error(`resolution must be "1080p" or "4K", got "${data.resolution}"`);
  }
}
```

Run against the two documents above, plus two deliberately broken ones:

```

validateItem({ itemType: 'Book', title: 'x', author: 'y' }); // passes, no error
validateItem({ itemType: 'Cd', title: 'x' }); // throws: "A Cd needs a ar
validateItem({ itemType: 'CD', title: 'x', artist: 'y' }); // throws: "itemType must be
validateItem({ itemType: 'Bluray', title: 'x', resolution: '720p' }); // throws: resolu

```

That's the exact same class of rejection the MongoDB sibling's own discriminators give for free — a `Cd` with no `artist` never reaches storage — reproduced by hand in plain JavaScript, because Firestore itself will never reject it on its own.

THIS FUNCTION DOESN'T CARE WHERE IT'S CALLED FROM

`validateItem()` is deliberately written with no dependency on React, Express, or Firestore itself — it just takes a plain object and throws or doesn't. That's on purpose: whether it ends up running in the browser immediately before an `addDoc` call, inside a small backend route, or both, is exactly the client-SDK-vs-backend-API question Chapter 1 raised and Chapter 3 resolves. Writing the check itself as a standalone function now means that decision doesn't force this code to be rewritten later.

A FUNCTION CALL IS NOT THE SAME GUARANTEE AS A REJECTED WRITE

Calling `validateItem()` before every `addDoc()` in this app's own code is a real safeguard — but it's a safeguard that lives in application code, not in Firestore itself. Anyone with direct write access to this Firestore project (via the console, a script, or a different app entirely) can still write a malformed document straight past this function without ever calling it. Firestore Security Rules, covered as part of Chapter 3's own architecture decision, are the one mechanism that can make a check like this hold regardless of what code is doing the writing.

One Real Advantage Firestore Gets for Free

The MongoDB sibling's own Chapter 2 flagged a genuine limitation worth revisiting here: a document fetched through the base `Item` model doesn't expose `author`, `artist`, or any other discriminator-specific field, even though the field is really there in the underlying MongoDB document — Mongoose only attaches those extra fields when a document is loaded through its own matching discriminator model.

Firestore has no equivalent problem, for a simple reason: it has no separate base-model and subtype-model objects at all. Every document fetched from the `items` collection comes back

as a plain object carrying every field it was actually written with:

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

const snapshot = await getDocs(collection(db, "items"));
const everything = snapshot.docs.map((d) => ({ id: d.id, ...d.data() }));

// everything[0] is a real Book object with author/isbn/publisher/tags already present
// no separate "load through the Book model" step needed to see them.
```

ONE FREE WIN, ONE REAL COST

Firestore's own lack of a model layer means there's no "base model can't see subtype fields" trap to fall into — every fetched document is simply complete. The real cost sits on the other side of the same coin: the MongoDB sibling's own `Book.find()` automatically filters to just Book documents because it's a genuinely separate model tied to that discriminator. A Firestore query against `items` has no such automatic filtering at all — finding just the Book documents needs an explicit `where("itemType", "==", "Book")` clause every single time, covered properly once Chapter 6 builds real search.

QUESTION	MONGODB SIBLING (MONGOOSE DISCRIMINATORS)	THIS COURSE (FIRESTORE)
Base fields	<code>title</code> , <code>releaseYear</code> , <code>notes</code>	Same three fields, same names
Type marker	<code>itemType</code> (the <code>discriminatorKey</code>)	<code>itemType</code> (a plain field, same name, deliberately)
Required-field enforcement	Built in, via <code>required: true</code> on each discriminator schema	Not built in — a real, hand-written <code>validateItem()</code> function
Restricted values (e.g. Bluray resolution)	Built in, via <code>enum: [...]</code>	Not built in — checked inside the same validator
Fetching a mixed list of every item type	One query, via the base <code>Item</code> model	One query, against the plain <code>items</code> collection
Do subtype fields show up on a generic fetch?	No — only via the matching discriminator model	Yes — every document is already complete

QUESTION	MONGODB SIBLING (MONGOOSE DISCRIMINATORS)	THIS COURSE (FIRESTORE)
Fetching only one item type	Automatic — query through that discriminator model	Manual — an explicit <code>where("itemType", "==", ...)</code>

Hands-On Exercises

EXERCISE 1

Write real Firestore documents for all four item types (Book, Cd, Dvd, Bluray) into the same `items` collection using `addDoc`, then write a single `getDocs` query that returns all four back out, confirming every subtype field is present on each returned object with no extra step.

 [View solution](#)

EXERCISE 2

Extend `validateItem()` to also require a `director` field on Dvd and Bluray documents, then run it against one valid and one invalid document of each new type to confirm the new rule actually fires.

 [View solution](#)

EXERCISE 3

Explain, in your own words, why calling `validateItem()` from application code is a real safeguard but not the same guarantee Mongoose discriminators give — and name the one Firestore-native mechanism that could close that remaining gap.

 [View solution](#)

CHAPTER 2 QUICK REFERENCE

- **Same shapes as the MongoDB sibling** — `title` / `releaseYear` / `notes` plus per-type fields, one shared `items` collection, an `itemType` field distinguishing document shape

- **No built-in enforcement** — Firestore accepts any document shape in any collection with no complaint
- **The gap-closer** — a standalone `validateItem()` function reproducing Mongoose's own `required` / `enum` checks by hand
- **Still open** — where `validateItem()` actually gets called from (client-side before a write, or a backend route) is Chapter 3's own question
- **A free win** — every fetched document already carries all of its own fields; no base-model/subtype-model split to trip over
- **A real cost** — filtering to one item type needs an explicit `where()` clause every time, unlike Mongoose's automatic per-discriminator filtering
- **Next chapter:** Firestore Reads & Writes From React — the Client SDK vs. a Backend API

CHAPTER 3: FIRESTORE READS & WRITES FROM REACT

Personal Catalogue: React & Firebase

Chapter 3 · Firestore Reads & Writes From React: The Client SDK vs. a Backend API

Two chapters have now written directly to Firestore from plain scripts and left one real question hanging both times: does this app ever get a backend server of its own, or does React talk to Firestore directly for good? This chapter answers it, for real, and builds the one piece that answer actually depends on.

A Confession About the Last Two Chapters' Own Code

Chapter 1 said to enable Firestore in **production mode**, specifically because test mode leaves every document wide open. What that also means, honestly: if the `addDoc` / `getDocs` code from Chapters 1 and 2 was actually run against a real production-mode project, every single call would have failed. Firestore's own real production-mode default is a deny-all rule:

```
// firestore.rules – the real default in production mode
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if false;
    }
  }
}
```

Nothing about that is a mistake in either earlier chapter — the code shown was always correct, and this chapter is what actually turns it on.

Resolving the Fork: No Backend Server of This Course's Own

Chapter 1's own finding held up: nothing in this app's real v1 scope needs a secret, and nothing needs a Cloud Function to hold one. Given that, introducing a small Express-style API layer here would buy back exactly nothing this variant doesn't already get from the client SDK

alone — it would just reintroduce the self-run server this course exists specifically to do without. So the real decision: React talks to Firestore **directly**, using the client SDK, for every read and write this app makes. There is no backend process of this course's own at all.

That decision puts real weight somewhere else, though — Chapter 2's own warn-box already named it. With no server standing between the browser and the database, `validateItem()` alone was never going to be enough on its own; it's a function, not a gate anyone is forced to pass through. Something has to actually enforce it. That something is Firestore Security Rules.

Turning Chapter 2's Checks Into Real Rules

Security Rules are evaluated by Firebase's own servers on every single request, regardless of what code sent it — a script, a browser tab, or a stray line typed straight into the Firebase console. Translated directly from Chapter 2's own `validateItem()` logic:

```
// firestore.rules
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {

    function hasValidBaseFields(data) {
      return data.title is string && data.title.size() > 0
        && data.itemType in ['Book', 'Cd', 'Dvd', 'Bluray'];
    }

    function hasTypeRequiredFields(data) {
      return (data.itemType != 'Book' || (data.author is string && data.author.size() > 0)
        && (data.itemType != 'Cd' || (data.artist is string && data.artist.size() > 0)
    }

    function hasValidResolution(data) {
      return data.itemType != 'Bluray'
        || !('resolution' in data)
        || data.resolution in ['1080p', '4K'];
    }

    match /items/{itemId} {
      allow read: if true;
      allow write: if hasValidBaseFields(request.resource.data)
```

```

    && hasTypeRequiredFields(request.resource.data)
    && hasValidResolution(request.resource.data);
  }
}
}

```

Deployed with `firebase deploy --only firestore:rules`, this is the exact same rule set the MongoDB sibling's own discriminator schemas enforce, expressed as declarative rules instead of JavaScript classes — and, critically, applied by Firestore itself rather than by any one function someone has to remember to call.

Proving It — Including on Purpose Skipping `validateItem()`

A valid write, with `validateItem()` called first exactly as Chapter 2 intended:

```

import { collection, addDoc } from "firebase/firestore";
import { db } from "../firebase";
import { validateItem } from "../validateItem";

async function addItem(data) {
  validateItem(data); // throws instantly, no network round trip, if data is malformed
  return addDoc(collection(db, "items"), data);
}

await addItem({ itemType: "Cd", title: "Discovery", artist: "Daft Punk" });
// succeeds – validateItem() passes, then Firestore's own rules pass too

```

Now the deliberate skip — calling `addDoc` directly, exactly the "a different code path forgot to validate" scenario Chapter 2's own warn-box raised:

```

await addDoc(collection(db, "items"), { itemType: "Cd", title: "Discovery" });
// no artist, and validateItem() was never called at all

```

```

// Real output

```

```
FirebaseError: Missing or insufficient permissions.
```

THE GAP CHAPTER 2 LEFT OPEN IS CLOSED

That rejection has nothing to do with `validateItem()` — it never ran. Firestore itself refused the write because `hasTypeRequiredFields()` evaluated to false against the real document being sent. This is the universal guarantee Mongoose discriminators give the MongoDB sibling for free, now genuinely present here too — just enforced by rules instead of by a schema class.

What Security Rules Still Can't Do Here

`allow read: if true` above is a deliberate, honest choice, not an oversight. This app has no Firebase Authentication, no user accounts, and no login of any kind — a decision this course made back in Chapter 1, in keeping with a genuinely personal, single-user tool. Firestore's own `request.auth` variable, the normal way a rule checks *who* is making a request, is always `null` here, for every request, because no one has ever signed in through this app at all.

RULES CHECK SHAPE, NOT IDENTITY — IN THIS APP, BY DESIGN

Every rule written above checks what a document *looks like*, never who's sending it. That's a real, genuine limit of Security Rules without an Authentication layer sitting underneath them — and it's the correct tradeoff for a tool built for exactly one person, deployed somewhere only that person knows the URL for. It would stop being the right tradeoff the moment this app gained real user accounts, or was deployed somewhere its address might reasonably leak — at which point rules gating on `request.auth.uid`, the same mechanism the site's own Food Tracker (React + Firebase) course reaches for once it adds Authentication, would become necessary here too.

The Architecture, Now Fully Resolved

LAYER	MONGODB SIBLING	THIS COURSE
Where a request lands first	An Express route, running on a server this project starts and keeps running	Firestore itself — React calls the client SDK directly, no server of this course's own
Shape validation	Mongoose discriminator schemas, checked in the Express process	Firestore Security Rules, checked by Firebase's own servers

LAYER	MONGODB SIBLING	THIS COURSE
Instant client-side feedback	Whatever the frontend chooses to check before sending the request	<code>validateItem()</code> , called before every write — a UX convenience, not the real gate
The real, universal gate	The Express route + Mongoose, together	Security Rules alone
Who can write	Whoever can reach the Express server (no auth built for this project either)	Anyone who can reach the Firestore project — no <code>request.auth</code> check exists to make

Every remaining chapter in this course builds directly against this: React components calling `validateItem()` then a Firestore SDK function, with the Security Rules deployed here standing permanently behind them.

Hands-On Exercises

EXERCISE 1

Deploy the Security Rules above to a real Firestore project (or the local emulator), then call `addDoc` directly — bypassing `validateItem()` entirely — with a Book document that has no `author`. Confirm the write is rejected and record the real error message.

 [View solution](#)

EXERCISE 2

Explain why `allow read: if true` is a reasonable choice for this specific app today, and name the exact condition under which it would stop being reasonable.

 [View solution](#)

EXERCISE 3

Explain what real value `validateItem()` still adds once Security Rules are deployed, given that the rules alone are now the actual, universal gate — in other words, why wasn't Chapter 2's own work wasted effort now that Chapter 3 exists?

 [View solution](#)

CHAPTER 3 QUICK REFERENCE

- **Resolved** — no backend server of this course's own; React talks to Firestore directly via the client SDK
- **Why** — nothing in v1 scope needs a secret (Chapter 1's own finding), so a server would add cost with no real benefit
- **The real gate** — Firestore Security Rules, translated directly from Chapter 2's own `validateItem()` logic, enforced by Firebase itself regardless of what code is writing
- **Production mode's real default** — deny-all until rules like these are deployed; earlier chapters' own code only starts working now
- `validateItem()` **'s real remaining job** — instant, round-trip-free feedback in the UI; a convenience layered on top of the real gate, not a replacement for it
- **Honest limit** — rules here check document shape only, never who's writing, since this app has no Authentication layer at all
- **Next chapter:** The Add-Item Form — type-specific fields and client-side validation

CHAPTER 4: THE ADD-ITEM FORM

Personal Catalogue: React & Firebase

Chapter 4 · The Add-Item Form: Type-Specific Fields, Client-Side Validation

Chapter 3 finished the plumbing: a `validateItem()` call, then a direct Firestore write, with real Security Rules standing behind both. This chapter builds the one form that actually produces the data going into that pipeline — and finds two places where skipping a step that was purely cosmetic for the MongoDB sibling would leave this app with a genuinely broken document.

The Same Field Configuration, Reused Directly

Personal Catalogue (React, Express & MongoDB)'s own Chapter 4 built a declarative `FIELD_CONFIG` object, matched field-for-field against its Mongoose discriminator schemas, to drive one shared rendering loop instead of four separate forms. Nothing about that idea is MongoDB-specific — it's reused here verbatim, matched instead against this course's own Chapter 2 field list:

TYPE	FIRESTORE FIELD (CH.2)	FRONTEND CONFIG FIELD
Book	<code>author</code> (required)	<code>author</code> , marked required
	<code>isbn</code>	<code>isbn</code>
	<code>publisher</code>	<code>publisher</code>
	<code>tags</code> (array)	<code>tags</code> , comma-separated text → array
Cd	<code>artist</code> (required)	<code>artist</code> , marked required
	<code>tracklist</code> (array)	<code>tracklist</code> , comma-separated text → array
Dvd	<code>director</code>	<code>director</code>
	<code>runtimeMinutes</code> (number)	<code>runtimeMinutes</code> , numeric input
	<code>regionCode</code>	<code>regionCode</code>
Bluray	<code>director</code>	<code>director</code>
	<code>runtimeMinutes</code> (number)	<code>runtimeMinutes</code> , numeric input

TYPE	FIRESTORE FIELD (CH.2)	FRONTEND CONFIG FIELD
	<code>resolution</code> ('1080p'/'4K')	<code>resolution</code> , a select with those two options
<pre> // src/fieldConfig.js export const ITEM_TYPES = ['Book', 'Cd', 'Dvd', 'Bluray']; export const FIELD_CONFIG = { Book: [{ name: 'author', label: 'Author', required: true }, { name: 'isbn', label: 'ISBN' }, { name: 'publisher', label: 'Publisher' }, { name: 'tags', label: 'Tags (comma-separated)', isArray: true },], Cd: [{ name: 'artist', label: 'Artist', required: true }, { name: 'tracklist', label: 'Tracklist (comma-separated)', isArray: true },], Dvd: [{ name: 'director', label: 'Director' }, { name: 'runtimeMinutes', label: 'Runtime (minutes)', type: 'number' }, { name: 'regionCode', label: 'Region Code' },], Bluray: [{ name: 'director', label: 'Director' }, { name: 'runtimeMinutes', label: 'Runtime (minutes)', type: 'number' }, { name: 'resolution', label: 'Resolution', options: ['1080p', '4K'] },], }; </pre>		
Shared State, Type-Specific State <code>title</code>, <code>releaseYear</code>, and <code>notes</code> are the base fields every item type shares, so they get their own piece of state, kept separate from whichever type-specific fields the <code>FIELD_CONFIG</code> lookup happens to be rendering right now: <pre> // src/components/AddItemForm.jsx import { useState } from 'react'; </pre>		

```
import { collection, addDoc } from "firebase/firestore";
import { db } from "../firebase";
import { validateItem } from "../validateItem";
import { ITEM_TYPES, FIELD_CONFIG } from '../fieldConfig';

export default function AddItemForm({ onCreate }) {
  const [itemType, setItemType] = useState('Book');
  const [shared, setShared] = useState({ title: '', releaseYear: '', notes: '' });
  const [typeFields, setTypeFields] = useState({});
  const [error, setError] = useState(null);

  // handleTypeChange, handleTypeFieldChange, and handleSubmit follow below
}
```

The type-specific inputs render the same way the MongoDB sibling's own form does — one `.map()` over `FIELD_CONFIG[itemType]`, choosing a select for a field with `options` and a text or number input otherwise. Nothing about that rendering logic is backend-specific, so it isn't repeated here.

The Leftover-Field Bug, Reproduced — and Not Auto-Corrected

`typeFields` is one shared object holding whichever fields the current type happens to use, and nothing clears it when `itemType` itself changes. Select Book, type an author, then switch the dropdown to Cd:

1. Book is selected. `typeFields` becomes `{ author: 'Ursula K. Le Guin' }`.
2. The dropdown changes to Cd. `typeFields` is untouched — still `{ author: 'Ursula K. Le Guin' }`, even though no author field is rendered anymore.
3. An artist value is typed. `typeFields` is now `{ author: 'Ursula K. Le Guin', artist: 'Kraftwerk' }`.
4. The form submits with `itemType: 'Cd'` and **both** fields in the outgoing document.

The MongoDB sibling's own version of this exact bug turned out to be harmless in practice — Mongoose's default strict mode silently drops any field a document's own schema doesn't declare, so the leftover `author` never actually reached storage. Checking the equivalent here, directly:

```
await addDoc(collection(db, "items"), {
  itemType: "Cd",
  title: "Discovery",
  artist: "Daft Punk",
  author: "Ursula K. Le Guin", // leftover from Book, never cleared
});
```

CONFIRMED DIRECTLY: FIRESTORE KEEPS IT

Fetching that document back out shows `itemType`, `title`, `artist`, and **a real, permanently stored `author` field** that has no business being on a Cd at all. Neither `validateItem()` nor the Chapter 3 Security Rules ever check for the *absence* of unexpected fields — they only check that required fields are present. Firestore has no concept of a document's own schema rejecting a field it doesn't recognize, because it has no concept of a document schema at all. The stale field isn't a cosmetic display glitch here — it's a real, saved data-quality bug.

The Fix — Now a Correctness Fix, Not Just a UX One

The fix itself is identical to the MongoDB sibling's own one-liner:

```
// AddItemForm.jsx (continued)
function handleChange(e) {
  setItemType(e.target.value);
  setTypeFields({}); // clear leftover fields from whatever type was selected before
}
```

THE SAME LINE, A DIFFERENT REASON TO NEED IT

The MongoDB sibling's own version of this fix was framed honestly as a UX correction, not a correctness one — Mongoose's strict mode was already protecting the actual saved data regardless. Here, that framing doesn't hold: without this reset, a real, permanently stored field ends up on the wrong kind of document, because nothing in this app's own stack — not `validateItem()`, not the Security Rules — was ever built to reject fields it doesn't recognize. This one line is doing real correctness work, not just tidying up a confusing input.

Closing this specific gap more thoroughly — rejecting a write outright if it carries a field no valid item type actually uses — is a real option worth naming rather than building here:

Firestore Security Rules support exactly that check, via

`request.resource.data.keys().hasOnly([...])`. Chapter 3's own rules don't do this today; the one-line client-side reset below is what actually keeps this app's real data clean until that stronger rule, if it's ever added, exists as a second line of defense.

Arrays From Text, and Why Numbers Need Real Conversion Here

Comma-separated text becomes a real array immediately before the write, exactly as the MongoDB sibling's own form does it — split, trimmed, and filtered, never stored as an array in state while it's still being typed:

```
// AddItemForm.jsx (continued) – inside handleSubmit, before the write
const payload = { itemType, ...shared, ...typeFields };

for (const field of FIELD_CONFIG[itemType]) {
  if (field.isArray && typeof payload[field.name] === 'string') {
    payload[field.name] = payload[field.name]
      .split(',')
      .map((s) => s.trim())
      .filter(Boolean);
  }
}
```

`releaseYear` and `runtimeMinutes` need the same real-vs-string treatment the MongoDB sibling's own form gives them — but the reason to bother is genuinely stronger here. Every DOM input produces a string regardless of its declared `type`, and Mongoose's own `Number` SchemaType casts a valid numeric string to a real number automatically on write, whether the client remembers to convert it or not. Firestore does no such thing. Whatever JavaScript type a value has when it's handed to `addDoc()` is exactly the type Firestore stores — a string stays a string, permanently:

```
await addDoc(collection(db, "items"), {
  itemType: "Dvd", title: "Spirited Away", runtimeMinutes: "125", // a string, not conv
```

```
});
// stored exactly as sent – runtimeMinutes is a real Firestore string field, not a number
```

WHY A STORED STRING INSTEAD OF A NUMBER ACTUALLY MATTERS

Firestore's own inequality queries (`where("runtimeMinutes", ">", 120)`), the kind a future "movies

`runtimeMinutes` is stored as the string `"125"` is simply invisible to a query filtering on the number `120`, with no error raised anywhere to explain why it never shows up. The conversion below isn't optional tidiness here; skipping it produces a real document that quietly breaks a real future query.

Submitting the Form

```
// AddItemForm.jsx (continued)
async function handleSubmit(e) {
  e.preventDefault();
  setError(null);

  const payload = {
    itemType,
    title: shared.title,
    releaseYear: shared.releaseYear ? Number(shared.releaseYear) : undefined,
    notes: shared.notes || undefined,
    ...typeFields,
  };

  for (const field of FIELD_CONFIG[itemType]) {
    if (field.type === 'number' && payload[field.name] !== undefined) {
      payload[field.name] = Number(payload[field.name]);
    }
    if (field.isArray && typeof payload[field.name] === 'string') {
      payload[field.name] = payload[field.name].split(',').map((s) => s.trim()).filter((s) => s);
    }
  }

  try {
    validateItem(payload); // throws instantly, no round trip, if data is malformed
  } catch (err) {
    setError(err.message);
  }
}
```

```

    return;
  }

  try {
    const ref = await addDoc(collection(db, "items"), payload);
    onCreate({ id: ref.id, ...payload });
    setShared({ title: '', releaseYear: '', notes: '' });
    setTypeFields({});
  } catch (err) {
    setError("Save failed – this data was rejected by the server."); // a real Security
  }
}

```

Two separate `catch` blocks are deliberate. The first catches `validateItem()`'s own synchronous throw — a specific, immediate message, no network request ever sent. The second catches whatever `addDoc()` itself might reject, which in practice should only ever be a real Security Rules denial, since anything `validateItem()` would have caught already stopped the function one line earlier.

Trying It in the Browser

With the Firestore project's rules deployed from Chapter 3, mounting `<AddItemForm onCreate={{(item) => console.log('Created:', item)}} />` is enough to test all four types end to end: select each type, fill in its own fields, and confirm the logged item — then cross-check directly in the Firebase console's own Firestore data viewer that all four documents exist in the `items` collection, each with genuinely different fields, and none of them carrying a stray field left over from whichever type was selected before it. Chapter 7 replaces this console log with the real catalogue list this form is actually populating.

Hands-On Exercises

EXERCISE 1

app, and create one real item of each type through the actual UI, confirming each one via the Firebase console's Firestore data viewer.

 [View solution](#)

EXERCISE 2

Temporarily remove the `setTypeFields({})` reset from `handleTypeChange`, reproduce the leftover-field bug (fill a Book's author, switch to Cd, fill artist, submit), and confirm directly that the saved Cd document really does carry a stale author field this time — unlike the MongoDB sibling's own version of the same bug. Explain why the outcome differs.

 [View solution](#)

EXERCISE 3

Explain why omitting the explicit `Number()` conversion on `runtimeMinutes` would be a genuinely more serious mistake in this course than in the MongoDB sibling, and describe a concrete, later scenario where the resulting bug would surface.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- **FIELD_CONFIG** — the identical declarative object the MongoDB sibling introduced, reused verbatim against this course's own Chapter 2 fields
- **Shared vs. type-specific state** — `title/releaseYear/notes` live in one piece of state; type-specific fields live in another, reset whenever `itemType` changes
- **The leftover-field bug, this time uncaught** — Firestore has no schema to silently drop an unexpected field the way Mongoose's strict mode does; the one-line reset is a real correctness fix here, not just a UX one
- **Arrays from text** — comma-separated input split/trimmed/filtered into a real array immediately before the write
- **Numbers need real conversion** — Firestore stores exactly the JS type it's handed, with no automatic casting; skipping `Number()` leaves a string field a future numeric query would silently never match
- **Two error paths** — `validateItem()`'s synchronous throw (fast, specific) vs. a Security Rules rejection surfaced through `addDoc()` (a real network round trip)

- **Next chapter:** Tags — storing, attaching, and filtering by tag on books specifically

CHAPTER 5: TAGS

Personal Catalogue: React & Firebase

Chapter 5 · Tags: Storing, Attaching & Filtering by Tag on Books

Chapter 4 only ever wrote tags once — typed as comma-separated text, converted to an array, sent along with a brand-new Book. Nothing since then has touched a book's own tags after it already exists. This chapter builds that: adding and removing individual tags safely under real concurrent edits, a genuine gap this app has that the MongoDB sibling's own discriminator models close automatically, and filtering the catalogue by tag.

Firestore Already Has an Atomic Answer

The MongoDB sibling's own Chapter 5 built a real lost-update race first, then fixed it with `$addToSet` / `$pull` — atomic array operators that let MongoDB itself decide what's currently in an array at the exact moment an update runs, rather than trusting a client's own possibly-stale snapshot. Firestore's client SDK ships the identical idea, directly usable from React with no backend route at all: `arrayUnion()` and `arrayRemove()`.

The naive approach still fails the same way first, though, and it's worth seeing why before reaching for the fix:

```
// A naive, read-modify-write approach — works, but has a real problem below
import { doc, getDoc, updateDoc } from "firebase/firestore";

async function addTagNaively(bookId, newTag) {
  const ref = doc(db, "items", bookId);
  const snap = await getDoc(ref);
  const updatedTags = [...snap.data().tags, newTag];
  await updateDoc(ref, { tags: updatedTags });
}
```

TIME	TAB A	TAB B	TAGS ACTUALLY SAVED IN FIRESTORE
t0	Loads the book, sees ['Python', 'Programming']	Loads the book, sees ['Python', 'Programming']	['Python', 'Programming']
t1	Adds 'Web Development' locally, writes ['Python', 'Programming', 'Web Development']	— (still holding its own t0 snapshot)	['Python', 'Programming', 'Web Development']
t2	—	Adds 'FastAPI' to its own <i>stale</i> local copy, writes ['Python', 'Programming', 'FastAPI']	['Python', 'Programming', 'FastAPI'] — Tab A's addition is gone, with no error anywhere

Exactly the same failure the MongoDB sibling reproduced, for exactly the same reason: nothing about Firestore made this naive code race-safe on its own, because the race lives in the client's own logic, not in either database. The fix is equally direct:

```
// tagOperations.js
import { doc, updateDoc, arrayUnion, arrayRemove } from "firebase/firestore";
import { db } from "../firebase";

export async function addTag(bookId, tag) {
  return updateDoc(doc(db, "items", bookId), { tags: arrayUnion(tag) });
}

export async function removeTag(bookId, tag) {
  return updateDoc(doc(db, "items", bookId), { tags: arrayRemove(tag) });
}
```

Rerunning the exact two-tab timeline through `arrayUnion()` instead gives the same correct result the MongoDB sibling found: Firestore itself resolves each union against whatever the array holds the instant that specific write is applied, not against either tab's own memory of it. Both additions land — the final array is `['Python', 'Programming', 'Web Development', 'FastAPI']`.

ARRAYUNION() ALSO DEDUPES

Calling `addTag(bookId, 'Python')` on a book that already has that tag is a genuine no-op — Firestore's own `arrayUnion()` only adds a value if an equal one isn't already present in the array, the exact same behavior `$addToSet` gives the MongoDB sibling. A plain array overwrite with a manually appended value would add it again regardless, producing a real duplicate.

The Real Gap: Nothing Scopes This to Books

The MongoDB sibling's own Chapter 5 confirmed something worth revisiting here directly: calling `Book.findByIdAndUpdate()` against a real Cd's own `_id` returns a genuine 404, because that discriminator model only ever matches documents whose stored `itemType` is `'Book'`. `addTag()` above has no equivalent protection at all — it's a plain `updateDoc()` call against whatever document ID it's given, with no concept of "the Book model" to route through in the first place, because Firestore has no discriminator models of any kind.

Calling it against a real Cd's own document ID proves this directly:

```
await addTag(realCdDocumentId, "Anything");
// succeeds – no error, no rejection
// the Cd document now has a real tags field it was never supposed to have
```

THE CURRENT SECURITY RULES DON'T CATCH THIS EITHER

Checking this write against Chapter 3's own rules confirms why it wasn't stopped:

`hasValidBaseFields()` still passes (the Cd's title and itemType are unchanged), `hasTypeRequiredFields()` still passes (the Cd's own required `artist` field is still present), and `hasValidResolution()` was never about tags at all. None of the three rules built so far check for a field that shouldn't be there — exactly the same category of gap Chapter 4 named for a leftover `author` field, now showing up a second time on a completely different field.

Closing It: A Real Rule Addition

The fix is a fourth rule function, added to `firestore.rules` alongside the three from Chapter 3, checking specifically that a `tags` field is never present unless the document's own `itemType` is `'Book'`:

```
// firestore.rules (updated from Chapter 3)
function tagsOnlyOnBooks(data) {
  return !('tags' in data) || data.itemType == 'Book';
}

match /items/{itemId} {
  allow read: if true;
  allow write: if hasValidBaseFields(request.resource.data)
    && hasTypeRequiredFields(request.resource.data)
    && hasValidResolution(request.resource.data)
    && tagsOnlyOnBooks(request.resource.data);
}
```

Deployed via `firebase deploy --only firestore:rules`, the identical `addTag()` call against the same Cd document now fails the way the MongoDB sibling's own version always did:

```
await addTag(realCdDocumentId, "Anything");
// FirebaseError: Missing or insufficient permissions.
```

A RULE GREW BECAUSE A REAL BUG WAS FOUND, NOT IN ADVANCE

This is the honest shape this course keeps taking: Firestore's own lack of a schema layer doesn't announce its gaps up front the way a missing Mongoose field would — it takes an actual reproduced bug, on a specific field, to know a specific rule needs adding. `tagsOnlyOnBooks()` exists because this chapter went looking for the MongoDB sibling's own discriminator-scoping guarantee and found it genuinely missing, not because it was anticipated back in Chapter 3.

A Small Tag Editor Component

A focused component — not the full book detail page Chapter 7 builds — renders each current tag as a removable pill and offers a small input for adding a new one, calling

`addTag` / `removeTag` directly, with no fetch call or backend route anywhere in between:

```
// src/components/TagEditor.jsx
import { useState } from 'react';
import { addTag, removeTag } from '../tagOperations';
```

```

export default function TagEditor({ bookId, initialTags, onChange }) {
  const [tags, setTags] = useState(initialTags);
  const [newTag, setNewTag] = useState('');

  async function handleAdd() {
    if (!newTag.trim()) return;
    const tag = newTag.trim();
    await addTag(bookId, tag);
    const updated = [...new Set([...tags, tag])];
    setTags(updated);
    setNewTag('');
    onChange?.(updated);
  }

  async function handleRemove(tag) {
    await removeTag(bookId, tag);
    const updated = tags.filter((t) => t !== tag);
    setTags(updated);
    onChange?.(updated);
  }

  return (
    <div>
      {tags.map((tag) => (
        <span key={tag} className="tag-pill">
          {tag}
          <button onClick={() => handleRemove(tag)}>&times;</button>
        </span>
      ))}
      <input value={newTag} onChange={(e) => setNewTag(e.target.value)} placeholder="Add tag" />
      <button onClick={handleAdd}>Add</button>
    </div>
  );
}

```

Unlike the MongoDB sibling's own version, which reads its updated `tags` array straight back from the server's JSON response after each request, this component updates its local state

exactly what `arrayUnion` / `arrayRemove` just did, using a real `Set` to guarantee the same no-duplicate guarantee locally that Firestore already enforces server-side.

Filtering the Catalogue by Tag

Firestore's own `array-contains` operator does exactly what the MongoDB sibling's own plain `{ tags: tag }` filter does — matching any document whose array field contains a given value — combined here with an explicit `itemType` filter, for the same reason the sibling course states its own equivalent condition plainly: no other item type ever has a `tags` field to match against, but saying so directly keeps the intent obvious to whoever reads this query next.

```
import { collection, query, where, getDocs } from "firebase/firestore";

export async function findBooksByTag(tag) {
  const q = query(
    collection(db, "items"),
    where("itemType", "==", "Book"),
    where("tags", "array-contains", tag)
  );
  const snapshot = await getDocs(q);
  return snapshot.docs.map((d) => ({ id: d.id, ...d.data() }));
}
```

THE FIRST REAL RUN OF THIS QUERY MAY ASK FOR AN INDEX

Combining an equality filter (`itemType`) with an `array-contains` filter is a real compound query, and Firestore often needs a composite index to run one efficiently. The first time this exact combination runs against a real project, expect a real console error containing a direct link to auto-create the missing index — not a bug in the query itself, just Firestore's own real, one-time setup step for this specific filter combination.

Case Sensitivity — the Same Real Limitation

`arrayUnion` / `arrayRemove` and `array-contains` all compare by exact value equality, the same as MongoDB's own `$addToSet`. `'python'` and `'Python'` are two genuinely different strings here too, and both would end up stored as separate tags on the same book. This course

doesn't normalize casing on the way in either — a real, shared limitation across both variants, not something specific to either database.

Hands-On Exercises

EXERCISE 1

Build `addTag/removeTag` using `arrayUnion/arrayRemove` and the `TagEditor` component, then add and remove several tags on a real `Book`, confirming after each step via the Firebase console that duplicates are never created and removed tags are genuinely gone.

 [View solution](#)

EXERCISE 2

Reproduce the lost-update scenario using two real sequential calls both built from the same original tags snapshot, sent through the naive read-modify-write approach — then repeat the same two-call scenario using `arrayUnion` instead, and compare the two final results.

 [View solution](#)

EXERCISE 3

Call `addTag()` against a real `Cd's` or `Dvd's` own document ID both before and after deploying the `tagsOnlyOnBooks` rule, and explain precisely why the outcome differs — including why neither of the two earlier rule functions from Chapter 3 already caught this.

 [View solution](#)

CHAPTER 5 QUICK REFERENCE

- **`arrayUnion()` / `arrayRemove()`** — Firestore's own direct equivalent to `$addToSet/$pull`, atomic and dedupe-aware, callable straight from React with no backend route
- **The naive read-modify-write race** — reproduced identically to the MongoDB sibling; fixed the same way, by never trusting a client's own held snapshot

- **The real gap** — no discriminator model means nothing stops a tag operation from landing on a Cd or Dvd document; confirmed by actually adding one
- **The fix** — a fourth Security Rules function, `tagsOnlyOnBooks()`, added to the same rules file Chapter 3 started
- **Filtering** — `where("itemType","=","Book") + where("tags","array-contains",tag)`; may need a one-time composite index
- **Shared limitation** — tag matching is case-sensitive on both variants; neither normalizes casing
- **Next chapter: Search** — Firestore's own real query limitations, compared directly against MongoDB's richer query language

CHAPTER 6: SEARCH

Personal Catalogue: React & Firebase

Chapter 6 · Search: Firestore's Own Real Query Limitations & Working Around Them

Chapter 1's own original question — "do I already own this?" — needs an answer across *any* item, matched by title or by whichever identifying field that item's own type actually has: author for a Book, artist for a Cd, director for a Dvd or Bluray. The MongoDB sibling answered this with one regex-powered `$or` query and two real crashes along the way. This chapter answers the same question with a genuinely different tool, because Firestore doesn't have the one the MongoDB sibling reached for at all.

The Real Limitation: No Substring Search, Anywhere

Firestore's own query engine supports equality, range comparisons (`<`, `<=`, `>`, `>=`) against an ordered field, `array-contains`, and `in`/`not-in` — and nothing else. There is no regex, no `LIKE`, and no built-in way to ask "does this field contain this substring anywhere." The MongoDB sibling's own `$or: [{ title: regex }, { author: regex }, ...]` has no Firestore equivalent to translate to — not a harder version of the same query, a genuinely different capability Firestore's query language doesn't have.

The Closest Native Tool: Prefix Range Queries

Firestore does support one real, commonly-used trick for "starts with" matching, exploiting the fact that strings are compared byte by byte: querying for everything between a prefix and that same prefix followed by a very high Unicode character finds every string starting with it.

```
import { collection, query, where, orderBy, getDocs } from "firebase/firestore";

async function searchByTitlePrefix(prefix) {
  const q = query(
    collection(db, "items"),
    orderBy("title"),
    where("title", ">=", prefix),
    where("title", "<=", prefix + "␣")
  );
```

```
);
const snapshot = await getDocs(q);
return snapshot.docs.map((d) => ({ id: d.id, ...d.data() }));
}
```

Tried against a real book titled "Clean Code": `searchByTitlePrefix("Clean")` finds it — `"Clean"` is genuinely a prefix. `searchByTitlePrefix("Code")` finds nothing at all, even though "Code" is right there in the title, because it doesn't sit at the *start* of the string.

THREE REAL LIMITS, NOT ONE

This trick is prefix-only (a mid-string term never matches), case-sensitive (Unicode ordering places lowercase and uppercase letters at different code points, so `"clean"` wouldn't correctly bound a query meant to match `"Clean Code"` without also storing a lowercase mirror field), and scoped to one field at a time — even combining four of these prefix filters across title/author/artist/director with Firestore's own newer `or()` query composition would still only catch a term sitting at the very start of one of those fields, never a term like "Martin" appearing partway through "Robert Martin."

The Real Answer at This App's Scale: Fetch, Then Filter

The MongoDB sibling's own honest scale note already applies here just as directly: at a few hundred to a few thousand items, a personal catalogue doesn't need a real search engine — it needs a search that works correctly. Since Chapter 3 already committed this course to no backend server of its own, the practical answer is the same idea the MongoDB sibling calls a "full collection scan," just run in the browser instead of on a database server: fetch every item, then filter with plain JavaScript string matching:

```
// search.js
import { collection, getDocs } from "firebase/firestore";
import { db } from "../firebase";

export async function searchItems(rawQuery) {
  const term = rawQuery.trim().toLowerCase();
  const snapshot = await getDocs(collection(db, "items"));
  const all = snapshot.docs.map((d) => ({ id: d.id, ...d.data() }));

  if (!term) return all;
```

```

return all.filter((item) => {
  const haystacks = [item.title, item.author, item.artist, item.director];
  return haystacks.some((field) => field && field.toLowerCase().includes(term));
});
}

```

This one function does everything the MongoDB sibling's own `$or` query does — title, author, artist, or director, matched anywhere in the string, case-insensitively — genuinely more flexible than the prefix trick above, at the real cost of doing the matching after every document has already been fetched.

THIS COURSE NEVER REPRODUCES THE MONGODB SIBLING'S OWN TWO CRASHES — STRUCTURALLY

The MongoDB sibling's own Chapter 6 found and fixed a route-ordering bug (a single-segment `/search` route shadowed by an earlier-registered `/:id` route) and an unescaped-regex crash (a raw parenthesis breaking `new RegExp()`). Neither can happen here — not because they were fixed, but because neither ingredient exists in this app's own architecture. Chapter 3 already decided this course has no Express-style router with path segments to order incorrectly, and `searchItems()` above never constructs a regex from user input at all, so there's no special-character syntax to ever need escaping. `"Mission: Impossible ("` is just a plain string being compared with `.includes()` — the parenthesis means nothing special to it.

The Real Cost the MongoDB Sibling Never Has to Think About

Fetching every document to search it isn't just slower at scale — it has a real, direct cost that a self-hosted MongoDB collection simply doesn't. Firestore bills by the number of documents actually read, and `getDocs(collection(db, "items"))` reads and counts **every single document in the collection**, on every search, regardless of how few actually match. A self-run MongoDB instance has no equivalent per-document billing at all — its own "full collection scan" cost is CPU time, not a metered line item.

A REAL, PRACTICAL MITIGATION: DEBOUNCE THE INPUT

Searching on every single keystroke would multiply that real read cost by however many characters get typed. A short debounce — waiting for a brief pause in typing before actually calling `searchItems()` — is a genuine, worthwhile mitigation at this app's scale, built directly into the search component below.

A Small, Debounced Search Bar

```
// src/components/SearchBar.jsx
import { useState, useEffect } from 'react';
import { searchItems } from '../search';

export default function SearchBar({ onResults }) {
  const [q, setQ] = useState('');

  useEffect(() => {
    const timer = setTimeout(async () => {
      onResults(await searchItems(q));
    }, 300);
    return () => clearTimeout(timer); // cancel the pending search if q changes again
  }, [q]);

  return (
    <input
      value={q}
      onChange={(e) => setQ(e.target.value)}
      placeholder="Search by title, author, artist, or director"
    />
  );
}
```

Each keystroke resets the 300ms timer rather than immediately triggering a real Firestore read — only once typing genuinely pauses does `searchItems()` actually run, keeping the real, metered read cost tied to how often someone stops to look at results, not to how many characters they typed.

Trying It End to End

```
// Matches a Book by title
await searchItems("Clean Code");

// Matches the same Book by author, even though the term never appears in the title
await searchItems("Robert Martin");
```

```
// Matches a Dvd by director
await searchItems("Kurosawa");

// A genuinely awkward title – no crash, no escaping needed, it's just a substring check
await searchItems("Mission: Impossible (");

// A term matching nothing returns a clean empty array, not an error
await searchItems("Nonexistent");
```

Hands-On Exercises

EXERCISE 1

Build `searchItems()` and confirm it returns the correct `Book` for a search matching only its `author` field, and returns a clean empty array (not an error) for a term matching nothing at all.

 [View solution](#)

EXERCISE 2

Build `searchByTitlePrefix()` and run it twice against a real book titled "Clean Code" — once with the query "Clean" and once with "Code" — recording both real results and explaining precisely why one succeeds and the other returns nothing.

 [View solution](#)

EXERCISE 3

Explain why this course's own search implementation is structurally immune to both real crashes the MongoDB sibling's own Chapter 6 found and fixed — the route-ordering bug and the unescaped-regex `SyntaxError` — tying your answer directly back to the architecture decision made in Chapter 3.

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- **The real limitation** — Firestore has no substring/regex search of any kind; only equality, range, array-contains, and in/not-in
- **The native prefix trick** — orderBy + a >= / <= range bounded by "□"; prefix-only, case-sensitive, single-field
- **The real answer at this scale** — fetch every document, filter client-side with plain string .includes(), matching the MongoDB sibling's own "full collection scan is fine here" logic one layer up
- **Structural immunity** — no Express router to mis-order, no regex ever constructed from user input, so neither of the MongoDB sibling's own two real crashes can occur here at all
- **The real hidden cost** — Firestore bills per document read; every search reads the entire collection regardless of match count, mitigated here with a 300ms debounce
- **Ceiling** — a genuine, larger-scale catalogue would need a real third-party search service (an Algolia-style Firestore extension), the same honest ceiling the MongoDB sibling names for its own full-collection regex scan
- **Next chapter:** The Catalogue List & Detail Views in React

CHAPTER 7: THE CATALOGUE LIST & DETAIL VIEWS

Personal Catalogue: React & Firebase

Chapter 7 · The Catalogue List & Detail Views in React

Every piece needed to actually see the catalogue already exists — Chapter 4's `AddItemForm`, Chapter 5's tag operations, Chapter 6's `searchItems()`. Nothing has rendered an actual item on screen yet. This chapter builds the two views that finally do — and finds that Firestore's own real-time capabilities quietly remove a whole category of manual wiring the MongoDB sibling's own equivalent chapter had to build by hand.

Two Views, Two Real Responsibilities

ASPECT	LIST VIEW	DETAIL VIEW
Data shown	Title, type, creator, release year — a compact summary	Every field the item's own type actually has, via Chapter 4's <code>FIELD_CONFIG</code>
Tag editing	Not available	The full <code>TagEditor</code> from Chapter 5, Book only
Navigation	Click an item to open its detail view	"Back to list" returns
Data source	The shared items array in App state	The same shared array — no extra fetch

A Naive Summary Card — the Same Rendering Bug, Backend-Agnostic

Every item type needs its own "creator" shown somewhere in the list. This particular bug has nothing to do with which database is behind it — it's a plain JavaScript coercion problem the MongoDB sibling's own Chapter 7 already found, and it would bite this course exactly as hard if it weren't caught here too:

```
// A naive summary line – looks fine for a Book, breaks for everything else
<li key={item.id}>
  <strong>{item.title}</strong>
  <p>{`By ${item.author}`}</p>
```

```
<span>{item.releaseYear}</span>
</li>
```

For a Cd, Dvd, or Bluray, `item.author` is `undefined` — the template literal coerces it with `String(undefined)`, producing the literal text **"By undefined"**, which React then renders exactly as given. The fix is the same real one, unchanged from the sibling course:

```
// src/components/ItemList.jsx
function getCreator(item) {
  return item.author || item.artist || item.director || null;
}

export default function ItemList({ items, onSelect }) {
  if (items.length === 0) {
    return <p>No items yet – add one above.</p>;
  }

  return (
    <ul>
      {items.map((item) => (
        <li key={item.id} onClick={() => onSelect(item.id)}>
          <strong>{item.title}</strong>{ ' '}
          <span className="type-badge">{item.itemType}</span>
          {getCreator(item) && <p>By {getCreator(item)}</p>}
          {item.releaseYear && <span>{item.releaseYear}</span>}
        </li>
      ))}
    </ul>
  );
}
```

`getCreator(item)` returns a real `null` when no creator field exists, and `null && <p>...</p>` short-circuits without rendering anything at all — the correct outcome, not just a less-broken one.

The Item Detail Component: Reusing Chapter 4's FIELD_CONFIG

```
// src/components/ItemDetail.jsx
import TagEditor from './TagEditor';
import { FIELD_CONFIG } from '../fieldConfig';

export default function ItemDetail({ item, onBack }) {
  const fields = FIELD_CONFIG[item.itemType] || [];

  return (
    <div>
      <button onClick={onBack}>&larr; Back to list</button>
      <h2>{item.title}</h2>
      <p className="type-badge">{item.itemType}</p>
      {item.releaseYear && <p>Released: {item.releaseYear}</p>}

      {fields
        .filter((field) => field.name !== 'tags')
        .map((field) => {
          const value = item[field.name];
          if (!value || (Array.isArray(value) && value.length === 0)) return null;
          return (
            <p key={field.name}>
              {field.label}: {Array.isArray(value) ? value.join(', ') : value}
            </p>
          );
        })}

      {item.itemType === 'Book' && (
        <TagEditor bookId={item.id} initialTags={item.tags || []} />
      )}

      {item.notes && <p>Notes: {item.notes}</p>}
    </div>
  );
}
```

NO EXTRA FETCH NEEDED — FOR AN EVEN MORE STRUCTURAL REASON THAN THE MONGODB SIBLING

The MongoDB sibling's own detail view skips a separate `GET /api/items/:id` call only because its list route uses `.lean()`, returning every field a document has rather than only the base `Item` schema's own fields. This course never needed an equivalent fix at all: Chapter 2 already established that a Firestore document fetched any way at all always carries every field it was written with — there

was never a base-model-vs-subtype-model split here for a "lean" option to work around in the first place.

The Naive Data Flow: One Fetch, Then Drift

A first, reasonable approach loads the catalogue once on mount, the same shape every prior chapter's own standalone examples have used:

```
// A naive one-time fetch – works, but has the same real gap the MongoDB sibling found
useEffect(() => {
  getDocs(collection(db, "items")).then((snapshot) => {
    setItems(snapshot.docs.map((d) => ({ id: d.id, ...d.data() })));
  });
}, []);
```

Open a Book's detail view, add a tag through `TagEditor`, click "Back to list," then reopen the same book: its detail view shows the *old* tags again. The write to Firestore genuinely succeeded — the document itself is correct — but the `items` array sitting in `App`'s own React state was only ever populated once, at mount, and nothing told it a document underneath it had changed since. This is exactly the stale-data gap the MongoDB sibling's own Chapter 5 `onChange` prop exists to close.

The Real Fix: a Live Firestore Listener

Firestore's client SDK offers something the MongoDB sibling's own REST API has no equivalent of at all: a subscription that keeps delivering fresh data automatically, for as long as it stays open, with no manual "please refetch" call needed anywhere in the app.

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

useEffect(() => {
  const unsubscribe = onSnapshot(collection(db, "items"), (snapshot) => {
    setItems(snapshot.docs.map((d) => ({ id: d.id, ...d.data() })));
  });
});
```

```
return () => unsubscribe(); // stop listening when this component unmounts
}, []);
```

`onSnapshot()` fires immediately with the collection's own current state, exactly like the naive `getDocs()` version — but then fires *again*, automatically, every single time any document in the `items` collection changes for any reason: a new item created by `AddItemForm`, a tag added or removed by `TagEditor`, anything at all. `setItems()` is called with a genuinely fresh snapshot every time, with nothing in any other component needing to know this is happening.

CHAPTER 5'S ONCHANGE PROP IS NO LONGER LOAD-BEARING

The MongoDB sibling's own Chapter 7 treats wiring `TagEditor`'s `onChange` prop as the real fix for the stale-list problem — a genuine, necessary piece of manual state-lifting. Here, the exact same stale-list problem is already solved the moment `onSnapshot()` replaces the one-time fetch, with no prop wiring required at all: the tag write itself is what triggers a fresh snapshot, and the fresh snapshot is what refreshes `items`, and `selectedItem` (derived fresh from `items` on every render) reflects the change automatically. Chapter 5's own `onChange` prop still exists and still works, but it's no longer the thing actually keeping the list and detail view in sync — that job now belongs to the listener, not to any one component remembering to call a callback.

The same mechanism quietly removes a second piece of manual wiring, too: `AddItemForm`'s own `onCreated` callback, previously needed to append a freshly created item into the shared array by hand, is no longer required for the list to show a new item — the listener already picks it up the moment `addDoc()` succeeds.

NOT FREE — JUST A DIFFERENT, USUALLY SMALLER COST

A live listener still counts against Firestore's own per-document-read billing from Chapter 6: the initial snapshot reads every document once, and each later update reads only the documents that actually changed — typically far cheaper than Chapter 6's own repeated full-collection re-fetch on every search, but not literally free. Forgetting to call the returned `unsubscribe()` function when a component unmounts leaves the listener running in the background indefinitely, a real, avoidable source of both wasted reads and a memory leak.

Wiring State in App.jsx

```

// src/App.jsx
import { useState, useEffect } from 'react';
import { collection, onSnapshot } from "firebase/firestore";
import { db } from './firebase';
import AddItemForm from './components/AddItemForm';
import SearchBar from './components/SearchBar';
import TagFilter from './components/TagFilter';
import ItemList from './components/ItemList';
import ItemDetail from './components/ItemDetail';

export default function App() {
  const [items, setItems] = useState([]);
  const [filtered, setFiltered] = useState(null); // null means "no active filter, show
  const [selectedId, setSelectedId] = useState(null);

  useEffect(() => {
    const unsubscribe = onSnapshot(collection(db, "items"), (snapshot) => {
      setItems(snapshot.docs.map((d) => ({ id: d.id, ...d.data() })));
    });
    return () => unsubscribe();
  }, []);

  const selectedItem = items.find((it) => it.id === selectedId);
  const displayed = filtered ?? items;

  return (
    <div>
      <h1>Personal Catalogue</h1>

      {!selectedItem && (
        <>
          <SearchBar onResults={setFiltered} />
          <TagFilter onResults={setFiltered} />
          <AddItemForm onCreated={() => setFiltered(null)} />
        </>
      )}

      {selectedItem ? (
        <ItemDetail item={selectedItem} onBack={() => setSelectedId(null)} />
      ) : (
        <ItemList items={displayed} onSelect={setSelectedId} />
      )}
    </div>
  );
}

```

```

    })
  </div>
);
}

```

`items` is now the live, listener-driven source of truth; `filtered` holds whatever `SearchBar` or `TagFilter` most recently returned, and clearing it after a new item is created is enough to bring the full, current list back into view — since `items` itself is already correct the instant the listener's own next snapshot arrives.

`TagFilter` is a thin wrapper around Chapter 5's own `findBooksByTag()` function, exactly the way `SearchBar` already wraps Chapter 6's `searchItems()`:

```

// src/components/TagFilter.jsx
import { useState } from 'react';
import { findBooksByTag } from '../tagOperations';

export default function TagFilter({ onResults }) {
  const [tag, setTag] = useState('');

  async function search() {
    onResults(tag ? await findBooksByTag(tag) : null);
  }

  return (
    <div>
      <input value={tag} onChange={(e) => setTag(e.target.value)} placeholder="Filter by tag" />
      <button onClick={search}>Search</button>
    </div>
  );
}

```

TWO SEPARATE FILTER CONTROLS, DELIBERATELY

`SearchBar` and `TagFilter` stay two distinct controls, for the same reason the MongoDB sibling keeps them separate — they run genuinely different real operations. `SearchBar`'s own Chapter 6 logic fetches everything and filters client-side; `TagFilter`'s own Chapter 5 logic sends a real, targeted

`array-contains` query to Firestore. Merging them into one input would hide that real difference rather than resolve it.

Trying It in the Browser

Add one item of each type through the form and watch the list update itself with no reload — the listener already has it. Open a Book, add a tag through the real `TagEditor`, click "Back to list," then reopen the same book: the new tag is already there, with no `onChange` prop, no manual state update, and no page refresh involved anywhere in making that happen.

Hands-On Exercises

EXERCISE 1

Build `ItemList`, `ItemDetail`, and the updated `App.jsx` exactly as this chapter describes, then create one item of each type and confirm every list entry shows a correct creator line (or none at all for a type with no creator field set) with no "By undefined" text anywhere.

 [View solution](#)

EXERCISE 2

Temporarily replace the `onSnapshot` listener in `App.jsx` with the earlier one-time `getDocs()` fetch, reproduce the stale-tags bug (edit a book's tags, go back, reopen it), and confirm the old tags are shown. Restore the listener and confirm the fix — with no other code changed anywhere.

 [View solution](#)

EXERCISE 3

Explain precisely why switching to `onSnapshot()` fixes the stale-tags bug without needing any change to `TagEditor`, `ItemDetail`, or `AddItemForm` — and name the one real, ongoing cost that switch introduces which a one-time `getDocs()` call never had.

 [View solution](#)

CHAPTER 7 QUICK REFERENCE

- **getCreator(item)** — the same backend-agnostic "By undefined" rendering bug the MongoDB sibling found, fixed the identical way
- **FIELD_CONFIG reused** — the same object from Chapter 4 drives both the add-item form and the detail view
- **No extra fetch on detail** — true here for an even more fundamental reason than the MongoDB sibling's `.lean()` fix: every Firestore document always carries every field it was written with
- **onSnapshot()** — a live listener replacing a one-time fetch; fires on mount and again on every future change, anywhere
- **Manual sync wiring eliminated** — Chapter 5's `onChange` prop and `AddItemForm`'s `onCreated` callback are both no longer required to keep the list and detail view in sync
- **Not free** — a live listener still counts real Firestore reads and must be unsubscribed on unmount
- **Next chapter:** Fast Manual Entry — reducing friction in the add-item flow given the project's own real deadline

CHAPTER 8: FAST MANUAL ENTRY

Personal Catalogue: React & Firebase

Chapter 8 · Fast Manual Entry: Reducing Friction Given the Real Deadline

Chapter 1's own real deadline has shaped quiet decisions throughout — no server to write and deploy, a live-listener list instead of manual refresh wiring. This chapter cashes that framing out directly against the add-item flow itself: real friction points, one of them hiding a genuine race condition that turns out to have nothing to do with which database sits underneath it.

One Piece of Friction This Course Never Actually Had

The MongoDB sibling's own Chapter 8 opens by replacing a blocking `alert()` call with inline error state. This course's own Chapter 4 was built with inline `error` state from the very first version of `handleSubmit` — there was never an `alert()` here to remove. Worth stating plainly rather than silently skipping: not every friction point named for one variant necessarily exists in another.

Returning Focus to the Title Field After a Successful Add

Chapter 4 already resets `shared` and `typeFields` after a successful write, and `itemType` is deliberately never reset — adding five books in a row never needs reselecting "Book" each time. The one piece still missing is the cursor: after a successful add, focus is left wherever it happened to be, so typing the next title means reaching for the mouse first.

```
// AddItemForm.jsx — a ref on the title input, focused after a successful add
import { useState, useRef } from 'react';

const titleInputRef = useRef(null);

// inside handleSubmit, after a successful addDoc() and state reset:
setShared({ title: '', releaseYear: '', notes: '' });
setTypeFields({});
titleInputRef.current?.focus();
```

The title input itself gets `ref={titleInputRef}`. A whole shelf of books can now be typed in with barely a pause between submissions.

A New Feature: Warning About Likely Duplicates While Typing

Fast entry has its own real risk: cataloguing quickly makes it easy to add something that's already there without noticing. A live check reusing Chapter 6's own `searchItems()` — built for a different purpose there, general search — catches this as the title is typed, as a soft warning rather than a hard block, since owning two real copies of the same title is a genuinely valid thing to catalogue.

A Naive Live Check — the Same Real Race Condition

A first, reasonable-looking version calls `searchItems()` on every keystroke, with no debounce and no ordering guarantee:

```
// A naive duplicate check – fires immediately on every keystroke
import { useState, useEffect } from 'react';
import { searchItems } from '../search';

function useDuplicateCheckNaive(title) {
  const [match, setMatch] = useState(null);

  useEffect(() => {
    const trimmed = title.trim();
    if (trimmed.length < 3) {
      setMatch(null);
      return;
    }

    searchItems(trimmed).then((results) => {
      const exact = results.find((r) => r.title.toLowerCase() === trimmed.toLowerCase());
      setMatch(exact || null);
    });
  }, [title]);
```

```
return match;
}
```

With a real Book titled exactly "Dune" already saved, typing "Dune Messiah" reproduces the identical race the MongoDB sibling found — nothing about `getDocs()` guarantees two independent Firestore reads resolve in the order they were sent, any more than two independent `fetch()` calls would:

TIME	INPUT SO FAR	CALL FIRED	RESPONSE ARRIVES	DISPLAYED WARNING
t0	"Dune"	<code>searchItems("Dune")</code> — will find the real, existing "Dune" book	—	none yet
t1	"Dune Messiah"	<code>searchItems("Dune Messiah")</code> — no exact match exists	The "Dune Messiah" call resolves first: no match	none shown — correct, for the moment
t2	(still "Dune Messiah")	—	The <i>earlier</i> "Dune" call finally resolves, late	⚠ "Dune" already exists — shown even though the field now reads "Dune Messiah"

NOT A MONGODB-SPECIFIC BUG — A GENERIC ASYNC JAVASCRIPT ONE

Nothing about this race depends on Express, MongoDB, or REST specifically. Two independent asynchronous calls — a `fetch()` to an API, or a `getDocs()` call straight to Firestore, makes no difference — carry no guarantee about which one's response lands first. The MongoDB sibling's own version of this bug and this one share the exact same root cause and need the exact same fix, because the cause was never about either database in the first place.

The Fix: Debouncing and Ignoring Stale Responses

```
// src/hooks/useDuplicateCheck.js
import { useState, useEffect, useRef } from 'react';
import { searchItems } from '../search';
```

```

export function useDuplicateCheck(title) {
  const [match, setMatch] = useState(null);
  const latestQuery = useRef('');

  useEffect(() => {
    const trimmed = title.trim();
    latestQuery.current = trimmed;

    if (trimmed.length < 3) {
      setMatch(null);
      return;
    }

    const timeoutId = setTimeout(async () => {
      const results = await searchItems(trimmed);

      // Ignore this response if the title has moved on since this call was made
      if (latestQuery.current !== trimmed) return;

      const exact = results.find((r) => r.title.toLowerCase() === trimmed.toLowerCase());
      setMatch(exact || null);
    }, 400);

    return () => clearTimeout(timeoutId);
  }, [title]);

  return match;
}

```

Retracing the exact same "Dune" / "Dune Messiah" timeline: the debounce means neither call fires until typing actually pauses, already making the race far less likely. The real guarantee is `latestQuery` — when the stale "Dune" response does eventually arrive, its own captured `trimmed` value no longer matches `latestQuery.current` (by then "Dune Messiah"), so it's discarded before it can ever reach `setMatch()`.

THE DEBOUNCE IS DOING DOUBLE DUTY HERE

In the MongoDB sibling, the 400ms delay exists purely to reduce how often the race can occur. Here it does that too, but it's also directly closing the real, metered-read concern Chapter 6 already raised: without it, every keystroke would trigger its own full `getDocs()` call reading the entire collection, a

real Firestore read cost multiplied by however many characters get typed. The same one delay is fixing a correctness problem and a cost problem at once.

Wiring the Duplicate Warning Into the Form

```
// AddItemForm.jsx (continued)
import { useDuplicateCheck } from '../hooks/useDuplicateCheck';

// inside the component, alongside the other state:
const duplicateMatch = useDuplicateCheck(shared.title);

// AddItemForm.jsx (continued) – rendered just below the title input
{duplicateMatch && (
  <p className="duplicate-warning">
    You may already have this – "{duplicateMatch.title}" is already in your catalogue.
  </p>
)}
```

Nothing about the warning blocks the submit button — purely informational, matching the real decision that owning two copies of the same title is a genuinely valid thing to catalogue, not an error to prevent.

What This Chapter Deliberately Didn't Add

No bulk-import feature — a genuinely bigger piece of scope than "reducing friction in the existing flow" calls for. No extra success toast either, and the reason is arguably even stronger here than for the MongoDB sibling: Chapter 7's own `onSnapshot()` listener already shows a newly created item in the list the instant Firestore confirms the write, with no manual state update needed to make that happen — a second, separate confirmation message would just repeat what's already visibly true on screen.

Trying It in the Browser

With a real Book titled "Dune" already in the catalogue, type "Dune Messiah" into the title field at a normal pace and watch the warning appear briefly on "Dune" and then correctly clear itself once the full, different title has settled. Submitting a deliberately invalid item now shows the error inline, with the rest of the form still fully usable while it's visible — and adding several valid books in a row never needs the mouse between submissions.

Hands-On Exercises

EXERCISE 1

Add the `titleInputRef` focus behavior to `AddItemForm`, then add several valid books in a row without touching the mouse between submissions, confirming the cursor is already in the title field each time.

 [View solution](#)

EXERCISE 2

Build the naive `useDuplicateCheckNaive` hook (no debounce, no staleness guard) and reproduce the race condition described in the chapter — with a real "Dune" book already saved, type "Dune Messiah" and observe a false "Dune already exists" warning appearing after the full title has been typed.

 [View solution](#)

EXERCISE 3

Replace the naive hook with the fixed `useDuplicateCheck` (debounced, with the `latestQuery` staleness guard) and repeat the exact same typing sequence from Exercise 2, confirming the false warning no longer appears — then explain why this same fix also reduces this chapter's own real Firestore read cost, not just the race condition.

 [View solution](#)

CHAPTER 8 QUICK REFERENCE

- **No alert() to remove** — this course's own Chapter 4 always used inline error state; one friction point the MongoDB sibling had that this course never did
- **Focus returned to title** — a ref-based .focus() call after every successful add, on top of itemType already persisting since Chapter 4
- **Live duplicate check** — reuses Chapter 6's own searchItems() for a new purpose, as a soft, non-blocking warning
- **The race condition** — confirmed to be a generic async-JavaScript problem, not a MongoDB-specific one; the identical bug and the identical fix apply here
- **Debounce + latestQuery guard** — debounce reduces how often competing calls exist AND cuts real Firestore read costs; the ref check is what actually guarantees a stale response is ignored
- **Deliberately not added** — bulk import, an extra success toast (Chapter 7's own live listener already provides that feedback automatically)
- **Next chapter:** Deployment

CHAPTER 9: DEPLOYMENT**Personal Catalogue: React & Firebase**

Chapter 9 · Deployment

This chapter gets the whole app running as a real standalone deployment — Chapter 10's own capstone handles the more specific question of mounting it inside the user's existing Astro-based site. Getting there is dramatically shorter than any of this project's three siblings, for one direct, cumulative reason: this course never built a server of its own to deploy in the first place.

A Genuinely Simpler Deployment — By Design, Not by Luck

PRODUCTION CONCERN	THE MONGODB SIBLING NEEDS	THIS COURSE NEEDS
A running backend process	Express, kept alive with PM2	Nothing — Firestore is already a running, managed service
Cross-origin request handling	CORS middleware, tightened for production	Nothing — the Firestore client SDK isn't a cross-origin request in the CORS sense at all
A public-facing reverse proxy	nginx, for a real domain and TLS termination	Nothing — Firebase Hosting provides both directly
A scoped, least-privilege database account	A dedicated MongoDB user with a readWrite-only role	Nothing new — already done, as Security Rules, back in Chapters 3 and 5
Serving the built frontend	express.static(), from the same process as the API	Firebase Hosting — a dedicated static-file CDN

None of this is a shortcut this course is taking — it's the direct, cumulative consequence of Chapter 3's own real decision that this app never needed a backend server at all.

Isolating Dev From Production: a Different Reason for Env Variables Than the MongoDB Sibling's

The MongoDB sibling's own Chapter 9 moves six hardcoded `localhost:4000` URLs into one environment-driven constant, because a real production domain has to replace a literal development address. This course has no API URLs to centralize — there's no backend of its own to point at. What genuinely is worth moving into environment variables here is something different: the Firebase project config itself, so a production build can point at a completely separate Firebase project than local development uses, keeping real production data untouched by anything built or tested locally.

```
// src/firebase.js
import { initializeApp } from "firebase/app";
import { getFirestore } from "firebase/firestore";

const firebaseConfig = {
  apiKey: import.meta.env.VITE_FIREBASE_API_KEY,
  authDomain: import.meta.env.VITE_FIREBASE_AUTH_DOMAIN,
  projectId: import.meta.env.VITE_FIREBASE_PROJECT_ID,
};

const app = initializeApp(firebaseConfig);
export const db = getFirestore(app);
```

```
# .env.development — a real, separate development Firebase project
VITE_FIREBASE_API_KEY=AIzaSy...dev
VITE_FIREBASE_AUTH_DOMAIN=catalogue-dev.firebaseio.com
VITE_FIREBASE_PROJECT_ID=catalogue-dev

# .env.production — the real production project
VITE_FIREBASE_API_KEY=AIzaSy...prod
VITE_FIREBASE_AUTH_DOMAIN=catalogue-prod.firebaseio.com
VITE_FIREBASE_PROJECT_ID=catalogue-prod
```

ISOLATION, NOT SECRECY — A DIRECT CALLBACK TO CHAPTER 1

Chapter 1's own warn-box already established that this config object was never a secret in the first place — it identifies a project, it doesn't grant access. Moving it into environment variables here isn't about hiding anything; it's purely so `npm run dev` and a real production build can point at two genuinely separate Firebase projects, so testing locally never risks touching or corrupting real catalogue data.

Building & Deploying to Firebase Hosting

```
# from the project root
npm run build
```

Vite's default build writes a real, deployable set of static files into `dist/`. Firebase Hosting is configured once, per project, to serve exactly that folder:

```
// firebase.json
{
  "hosting": {
    "public": "dist",
    "rewrites": [{ "source": "**", "destination": "/index.html" }]
  }
}
```

THE REWRITE RULE ABOVE IS A NO-OP HERE — THE SAME REASON AS CHAPTER 7

Firebase Hosting's own `rewrites` option exists for the same reason most SPA-hosting setups need one: a real client-side route like `/books/123` would otherwise 404 on a hard refresh, since no static file at that exact path exists. This app never had a client-side route at all — Chapter 7's own list/detail toggle is pure component state, never a URL change — so this rule, while harmless to include, never actually has anything to rewrite in practice.

```
# deploy the built frontend
firebase deploy --only hosting

# or deploy hosting AND any pending Security Rules changes together
firebase deploy
```

What This Course Never Needed to Build

No CORS configuration exists anywhere in this app, in any chapter — the Firestore client SDK talks to Firebase's own servers using its own protocol, not a same-origin-restricted browser fetch, so there was never a cross-origin policy of this app's own to write in the first place. No

PM2, because there is no long-running process of this course's own to keep alive; Firestore and Firebase Hosting are already running, managed services maintained by Firebase itself. No nginx reverse proxy, because Firebase Hosting already terminates TLS and serves the app directly.

THE "LEAST-PRIVILEGE ACCESS" WORK ALREADY HAPPENED — EARLIER

The MongoDB sibling creates a dedicated, restricted database user in this exact chapter. This course has no equivalent user account to create at all — Firestore has no login-based database-user model the way MySQL or MongoDB does. The real equivalent work already happened, earlier, as Security Rules: Chapter 3's own `hasValidBaseFields()` / `hasTypeRequiredFields()` / `hasValidResolution()` and Chapter 5's own `tagsOnlyOnBooks()` are this app's real access-control layer, and nothing new needs to be written here to reach production — only confirming those rules are actually deployed to the real production project, not just the development one used throughout this course so far.

Automatic HTTPS, for Free

Firebase Hosting provisions and renews a free SSL certificate automatically for both its own `*.web.app` subdomain and any real custom domain connected to the project — no certificate request, no renewal cron job, and no dedicated reverse proxy layer required to get it, unlike the MongoDB sibling's own nginx-plus-certificate setup.

Scheduled Backups, the Managed Way

The MongoDB sibling schedules a real `mongodump` cron job to keep a dated snapshot on disk, independent of whatever a hosting provider's own backup story does or doesn't cover. Firestore offers a real, equivalent managed feature — scheduled backups, configured directly through Google Cloud rather than a self-written script:

```
# a real, one-time setup command — Google Cloud manages the schedule from here
gcloud firestore backups schedules create \
  --database='(default)' \
  --recurrence=daily \
  --retention=7d
```

The same underlying need — a dated snapshot existing independently of the live database, in case something ever needs to be restored — solved here by configuring a managed service once, rather than maintaining a script and a cron entry indefinitely.

Trying It

```
npm run build
firebase deploy --only hosting
```

Visiting the real <https://catalogue-prod.web.app> URL Firebase prints after deployment should load the working catalogue app directly, with every action — search, add, tag edit — succeeding exactly as it did throughout development, now reading from and writing to the real production Firestore project instead of the development one.

Hands-On Exercises

EXERCISE 1

Move the Firebase config in `firebase.js` into `VITE_`-prefixed environment variables, create separate `.env.development` and `.env.production` files pointing at two different real Firebase projects, and confirm `npm run dev` connects to the development project while a production build connects to the other.

 [View solution](#)

EXERCISE 2

Build the app, configure `firebase.json` for Hosting, and deploy it with `firebase deploy --only hosting`. Confirm the real, live URL Firebase provides loads the working app and that a real `add-item` action succeeds against the deployed site.

 [View solution](#)

EXERCISE 3

Explain, for each of CORS, PM2, and a reverse proxy, precisely why this course never needed to build it — tying each answer back to a specific earlier architectural decision rather than a general "Firebase handles it" claim.

 [View solution](#)

CHAPTER 9 QUICK REFERENCE

- **No backend to deploy** — the direct, cumulative payoff of Chapter 3's own no-server decision
- **Env-driven Firebase config** — for real dev/prod project isolation, not for secrecy (Chapter 1 already settled that question)
- **Firebase Hosting** — `npm run build`, then `firebase deploy --only hosting`
- **The SPA rewrite rule is a no-op here too** — the same real reason as Chapter 7's own missing fallback route: no client-side routing exists in this app at all
- **Never needed** — CORS, PM2, an nginx reverse proxy, a dedicated database user (Security Rules already cover that job, done back in Chapters 3 and 5)
- **Free, automatic HTTPS** — provisioned and renewed by Firebase Hosting itself
- **Managed backups** — Firestore's own scheduled backup feature, configured once rather than maintained as a cron script
- **Next chapter:** Capstone — integrating the catalogue into the existing Astro site, and comparing self-managed MongoDB against managed Firestore across the whole project

CHAPTER 10: CAPSTONE — INTEGRATING THE CATALOGUE INTO THE EXISTING SITE

Personal Catalogue: React & Firebase

Chapter 10 · Capstone

Chapter 9 got the whole app running as a real, standalone Firebase Hosting deployment. This closing chapter puts it somewhere the user actually visits — a real path under their own existing Astro-based site — lays out a genuine, concrete plan for the one feature deferred since Chapter 1, and closes the whole course with an honest, itemized scorecard against its own MongoDB sibling.

Three Real Ways to Mount This — With Firebase Hosting Changing the Menu Slightly

OPTION	WHAT IT INVOLVES	TRADEOFF
Subdomain (<code>catalogue.existing-site.com</code>)	Point a DNS record straight at Firebase Hosting via its own custom-domain feature — no reverse proxy needed at all	Simplest possible setup; feels like a separate site rather than one integrated experience
Path-mounted reverse proxy (<code>existing-site.com/catalogue/</code>)	One nginx location block on the existing site's own domain, proxying that one path to Firebase Hosting's real URL	Feels like part of the same site; needs the real fixes this chapter builds — covered below
A deeper Astro-native rewrite	Keep only Firestore's own schema and Security Rules; rebuild the UI as new Astro pages/components calling the Firestore SDK directly	The deepest real integration and the most rework — appropriate only if Astro should become the one true frontend long-term

The path-mounted option is built out in full here, genuinely integrated without discarding any of the nine chapters of real React frontend already built.

The Real Asset-Path Fix — Identical to the MongoDB Sibling, for the Identical Reason

Chapter 9's own `npm run build` produces an `index.html` referencing its own JS and CSS bundles with absolute paths like `/assets/index-abc123.js`. That's correct for a standalone deployment at a domain root, and genuinely wrong once mounted at `/catalogue/`: the browser's own subsequent request for that absolute path resolves against the domain's own root, not the real sub-path this app actually lives under — a request for `/assets/index-abc123.js` never even reaches nginx's own `/catalogue/` location block, since the request itself doesn't start with that prefix at all. This is a pure client-side, browser-resolves-absolute-paths problem, with nothing about Firebase Hosting or Firestore changing it in any way. The fix is the identical one the MongoDB sibling verified against Vite's own documentation:

```
// vite.config.js
export default defineConfig({
  plugins: [react()],
  base: '/catalogue/',
});
```

Rebuilt with this in place, `index.html` references `/catalogue/assets/index-abc123.js` instead — a path nginx's own `/catalogue/` location block genuinely matches, and one Firebase Hosting itself never needs to know anything special about, since it just serves whatever file that path resolves to inside its own `dist/` upload.

The Runtime Fix the MongoDB Sibling Needed — and This Course Genuinely Doesn't

The MongoDB sibling's own second fix updates `VITE_API_URL` to a real sub-path value, because its own fetch calls are built as relative paths that resolve against whatever origin and path the current page happens to be served from — the browser has no way to know an API call meant for `/catalogue/api/items` shouldn't just go to `/api/items` unless it's told so explicitly.

THIS APP HAS NO EQUIVALENT RUNTIME FIX TO MAKE

The Firestore client SDK never builds a relative, current-page-based URL for anything. Every read and write goes directly to Firebase's own real, fixed backend endpoints (keyed by the `projectId` set back in Chapter 9's own environment variables), completely independent of whatever domain or path the React app itself happens to be served from at that moment. Mounting this app at `/catalogue/`,

at a bare subdomain, or at a completely different domain entirely changes nothing about how Firestore calls resolve — there was never a same-origin-relative API call here for a sub-path to break in the first place, a direct, concrete extension of Chapter 9's own "no CORS needed" finding.

Reverse-Proxying to an External, CDN-Fronted Backend — a Genuinely Different Nginx Problem

The MongoDB, PHP, and Django siblings all proxy to a local process on the same machine — `proxy_pass http://localhost:PORT/;`, with no real concern about which domain name the request claims to be for, since a local process doesn't route based on that. Firebase Hosting is a shared, multi-tenant CDN serving many different Firebase projects from the same infrastructure, and it decides which project's own content to serve based on the `Host` header (and, for HTTPS, SNI) presented on the request:

```
# /etc/nginx/sites-available/existing-site (excerpt)
server {
    listen 443 ssl;
    server_name existing-site.com;

    # ...the existing Astro site's own location blocks, unchanged...

    location /catalogue/ {
        proxy_pass https://catalogue-prod.web.app/;
        proxy_set_header Host catalogue-prod.web.app;
        proxy_ssl_server_name on;
    }
}
```

TWO LINES THAT GENUINELY AREN'T OPTIONAL HERE

Naively passing through the original `Host: existing-site.com` header (nginx's own real default when none is set explicitly) would confuse Firebase's own routing entirely — it has no project registered under that hostname, so the request would fail or resolve to the wrong content. Explicitly rewriting `Host` to the real Firebase Hosting domain fixes that. `proxy_ssl_server_name on;` is the second, easy-to-miss piece: it makes nginx present the correct SNI hostname during the TLS handshake itself, which the same shared, multi-tenant infrastructure also depends on to route the connection to the right backend before a single byte of the actual HTTP request is ever read.

With both in place, the trailing slash on `location /catalogue/` and `proxy_pass https://catalogue-prod.web.app/;` strips the `/catalogue` prefix before forwarding, exactly the same mechanism the MongoDB sibling relies on — a request for `/catalogue/assets/index-abc123.js` reaches Firebase Hosting as a request for exactly `/assets/index-abc123.js`, which its own CDN already serves correctly with no changes needed on the Firebase Hosting side at all.

Barcode Scanning — Where This Course's Own "No Backend" Streak Finally Ends

Chapter 1 deferred barcode scanning explicitly, and flagged an honest, unresolved question rather than asserting an answer: whether this course could get away with never needing a Cloud Function at all. With the app actually live, that question has a real answer now — no, not once this specific feature is added.

- **Camera access** — a real, actively maintained library like `html5-qrcode` reads a barcode from the device camera directly in the browser, matching the MongoDB sibling's own recommendation exactly, since this piece is pure frontend and has nothing to do with either backend.
- **Book lookup, and the real reason it needs a Cloud Function** — a scanned ISBN can be sent to Open Library's real, free, no-API-key endpoint to auto-fill title, author, and publisher. Calling it directly from the browser would work today, but the moment this app ever needs a real, secret-holding API key for any lookup service, that call has to move server-side — and since this course has no server of its own at all, its first-ever `onCall` Cloud Function is what that server-side call would actually become, the exact scenario Chapter 1's own honest open question was waiting to be resolved by.
- **Cd/Dvd/Bluray lookup** — honestly still open, the same real gap the MongoDB sibling names: no single, universally reliable free UPC-lookup service has actually been verified for physical media here. Genuine research is the concrete next step, not something to guess at.

Chapter Attribution

PIECE OF THE FINISHED APP	BUILT IN
Firestore document shapes, matched to the MongoDB sibling's own schema	Chapter 2
The client-SDK-only architecture decision, plus real Security Rules	Chapter 3
Add-item form, driven by the shared FIELD_CONFIG	Chapter 4
Tag editing via arrayUnion/arrayRemove, plus the tagsOnlyOnBooks rule	Chapter 5
Fetch-and-filter search, replacing regex entirely	Chapter 6
List and detail views, driven by a live onSnapshot listener	Chapter 7
Friction reduction: focus return, debounced duplicate check	Chapter 8
Firebase Hosting deployment, env-driven project config	Chapter 9
Path-mounted reverse proxy, Vite base, barcode-scanning plan	Chapter 10 (this chapter)

Self-Managed MongoDB vs. Managed Firestore: the Real Scorecard

AREA	REAL OUTCOME ACROSS THIS COURSE
Backend of any kind	Firestore wins — no server to write, run, or deploy at all (Ch.3, Ch.9)
Per-type schema validation	MongoDB wins — Mongoose discriminators enforce it for free; this course hand-built validateItem() and Security Rules to match (Ch.2, Ch.3)
Atomic array updates (tags)	A genuine tie — arrayUnion/arrayRemove match \$addToSet/\$pull exactly, with no backend route needed on either side to get there (Ch.5)
Preventing a field on the wrong document type	MongoDB wins — automatic via discriminator models; this course found the gap the hard way and closed it with a hand-written rule (Ch.5)
Text search	MongoDB wins clearly — real regex/\$or search vs. no substring search capability at all here, worked around with a client-side fetch-and-filter (Ch.6)
Keeping a UI in sync with the database	Firestore wins — onSnapshot() eliminates the manual state-lifting wiring this course's own sibling needs (Ch.7)
Production deployment complexity	Firestore wins decisively — no CORS, PM2, or reverse proxy for a backend of its own (Ch.9)

AREA	REAL OUTCOME ACROSS THIS COURSE
Ongoing operating cost model	MongoDB wins on predictability — a self-run instance has no per-document-read billing; every Firestore read is metered, made concrete in Chapter 6's own full-collection search cost
Sub-path deployment fixes needed	Firestore needs fewer — only the shared Vite asset-path fix, not a second runtime API-path fix, since Firestore never routes through the app's own current origin (Ch.10)

Neither column wins outright. What genuinely changed, chapter after chapter, wasn't whether real problems existed — it's *where* the work to solve them landed: inside a schema class and an Express route for the MongoDB sibling, or inside a hand-written validator and a declarative rules file here. Managed infrastructure removed real operational work this course never had to think about; it also removed guarantees this course had to go rebuild by hand, one genuine gap at a time.

Hands-On Exercises

EXERCISE 1

Set base: `'/catalogue/'` in `vite.config.js`, rebuild, and confirm the built `index.html`'s own asset references correctly include the `/catalogue` prefix — then explain in one sentence why no equivalent runtime environment-variable fix is needed for Firestore calls, unlike the MongoDB sibling's own second fix.

 [View solution](#)

EXERCISE 2

Configure the nginx location `/catalogue/` block against a real, deployed Firebase Hosting site (a second local nginx config is fine for testing), including the Host header override and `proxy_ssl_server_name`. Confirm the page loads correctly, then temporarily remove just the Host header line and observe what happens.

 [View solution](#)

EXERCISE 3

Using this chapter's own scorecard, pick the single real finding from the whole course you think matters most for a genuinely larger catalogue (tens of thousands of items rather than a few hundred), and explain why — citing the specific chapter it came from.

 [View solution](#)

CHAPTER 10 QUICK REFERENCE — COURSE COMPLETE

- **Three mounting options** — a Firebase Hosting custom domain, a path-mounted reverse proxy (built here), or a deeper Astro-native rewrite
- **The asset-path fix** — identical to the MongoDB sibling's own vite.config base fix, for the identical browser-side reason
- **The runtime fix that isn't needed** — Firestore never routes through the app's own current origin, so there's no second, API-path-specific fix to make
- **A genuinely new nginx problem** — proxying to a CDN-fronted external service needs an explicit Host header override and proxy_ssl_server_name, unlike proxying to a local process
- **Barcode scanning** — the real point where this course's own "no backend" streak ends, needing its first Cloud Function
- **The scorecard** — real, itemized wins and costs across both variants; no clean overall winner, just different places the real work landed
- **This closes the course** — 10/10 chapters, matching the shared four-course Personal Catalogue spec from Chapter 1