

PERSONAL MULTI-TECHNOLOGY PROJECTS

Personal Catalogue: React, Express & MongoDB

A real, working catalogue for books, CDs, DVDs, and Blu-rays — a heterogeneous MongoDB collection built with Mongoose discriminators, a real Express CRUD API, a React frontend with tag editing, cross-type search, and fast manual entry, and a real deployment mounted onto an existing Astro site, closing with barcode scanning deliberately scoped out as an honest future feature rather than a day-one requirement.

10 Chapters

Philip Osztromok

Generated with Claude

CHAPTER 1: PROJECT OVERVIEW & STACK SETUP

Personal Catalogue: React, Express & MongoDB

Chapter 1 · Project Overview & Stack Setup

This is one of four courses building the exact same personal cataloguing tool in four genuinely different architectures — Personal Catalogue (PHP & MySQL) and Personal Catalogue (Django & PostgreSQL) are already complete; Personal Catalogue (React & Firebase) remains outlined. This course's own answer deliberately leans into schema flexibility: a MERN-style stack (MongoDB, Express, React, Node) built around the real fact that books, CDs, DVDs, and Blu-rays are genuinely differently-shaped records.

What the App Actually Does

The shared spec every course in the quartet builds toward:

- **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 even faster, but deliberately deferred rather than allowed to hold up a release that needs to ship quickly. This course closes with a real, concrete plan for adding it later (Chapter 8), rather than pretending it isn't wanted at all.

Why React, Express & MongoDB for This One

A real relational schema, like the PHP and Django siblings both use, has to decide up front exactly which columns a "catalogue item" table has — even though a book's own real fields (author, ISBN, publisher) and a DVD's real fields (director, runtime, format) genuinely don't overlap much beyond title and year. MongoDB's own document model sidesteps that tension directly: a single, heterogeneous `items` collection can hold a book document and a DVD

document side by side, each with only the fields that item type actually needs, with no wasted null columns and no early commitment to one rigid, shared shape.

A REAL MECHANISM, NOT JUST "MONGODB IS FLEXIBLE"

Chapter 2 builds this properly using Mongoose **discriminators** — a real, documented pattern for storing several genuinely different document subtypes in one underlying MongoDB collection, each validated against its own schema, distinguished by a stored discriminator field (this project uses `itemType`). It isn't "just store whatever JSON shows up" — MongoDB's own flexibility still gets real, per-type schema validation on the way in.

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, discriminated by item type
Personal Catalogue: React & Firebase	React + managed BaaS	Firestore documents (self-managed MongoDB vs. managed Firestore, compared directly once both exist)

A Real Deadline, Not Just a Framing Device

This project genuinely needs a working first release quickly, exactly as its own two completed siblings describe. That constraint shapes real decisions throughout this course too — a single Express API rather than a separately versioned backend service, Mongoose's own schema validation used in place of hand-rolled checks, and a deliberately plain add-item form before any real polish. Chapter 7 covers where the corners were deliberately cut given that time pressure.

Setting Up: Node, Express & a Working MongoDB Connection

This course assumes Node.js and a MongoDB instance are already available — either a local install (`mongod` running on the default port) or a free-tier MongoDB Atlas cluster. Confirm both

are reachable before continuing.

Scaffold the backend first, since it's what every later chapter builds directly on:

```
# Backend
mkdir catalogue-api && cd catalogue-api
npm init -y
npm install express mongoose cors dotenv
npm install --save-dev nodemon
```

A single `.env` file holds the real connection string, kept out of version control:

```
# .env
MONGODB_URI=mongodb://localhost:27017/catalogue
PORT=4000
```

A minimal entry point confirms the connection before any real route exists:

```
// server.js
require('dotenv').config();
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');

const app = express();
app.use(cors());
app.use(express.json());

mongoose.connect(process.env.MONGODB_URI)
  .then(() => console.log('MongoDB connected'))
  .catch((err) => console.error('MongoDB connection error:', err));

const PORT = process.env.PORT || 4000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

Running `npx nodemon server.js` should log both a successful MongoDB connection and the listening port — confirmation the real backend half of this stack is working before a single schema exists.

Scaffolding the React Frontend

A separate React project, built with Vite for a fast, minimal dev setup, will consume this API over HTTP — exactly the same "two real, independently running processes" pattern this site's own `react-food-tracker-1` course already established:

```
# Frontend, run from a sibling folder to catalogue-api
npm create vite@latest catalogue-web -- --template react
cd catalogue-web
npm install
```

TWO REAL PROCESSES, NOT ONE

Unlike the PHP sibling's own single-runtime setup, this variant genuinely runs two separate servers during development — the Express API on port 4000, the Vite dev server on its own default port — talking to each other over real HTTP requests. That separation is exactly why Chapter 3's own CRUD routes, not a template engine, are where this course's real data actually lives.

DON'T COMMIT THE CONNECTION STRING

A real MongoDB Atlas URI embeds a username and password directly in the connection string. Keeping it in `.env` and adding `.env` to `.gitignore` from the very first commit avoids a genuinely common, genuinely real mistake — a credential accidentally pushed to a public repository.

Where This Course Is Headed

A real Mongoose schema for items, discriminated by item type (Chapter 2); the Express CRUD routes every later feature calls (Chapter 3); one React form handling several genuinely different item shapes (Chapter 4); tags — attaching, managing, and filtering by tag on books (Chapter 5); real search across a heterogeneous collection (Chapter 6); the list and detail views in React (Chapter 7); fast manual entry given the project's own real deadline (Chapter 8); deployment (Chapter 9); and a capstone on integrating this catalogue into the existing Astro-based site, plus a real, concrete plan for barcode scanning (Chapter 10).

Hands-On Exercises

EXERCISE 1

Explain why a genuinely heterogeneous MongoDB collection is a better fit for this project's own item data than a single, rigid relational table — and name the real risk a document database doesn't automatically protect against that a relational one does.

 [View solution](#)

EXERCISE 2

Set up the Express backend and confirm a real MongoDB connection via the console log, then explain in one sentence what Mongoose discriminators are for and why they matter for this project specifically.

 [View solution](#)

EXERCISE 3

Scaffold the Vite-based React frontend and explain why this variant runs two separate development servers, where the PHP sibling course needed only one.

 [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
- **Why this variant** — MongoDB's own document flexibility fits genuinely differently-shaped item records directly, without forcing one rigid shared table
- **Database** — MongoDB, with Mongoose providing real per-type schema validation via discriminators (built in Chapter 2)
- **Backend** — Express + Mongoose, connected via a `.env`-stored connection string
- **Frontend** — a separate Vite-based React app, consuming the API over real HTTP requests

- **Next chapter:** Data Modeling in MongoDB — a real heterogeneous `items` collection, discriminated by item type

CHAPTER 2: DATA MODELING

Personal Catalogue: React, Express & MongoDB

Chapter 2 · Data Modeling

Chapter 1 promised a real, working answer to the shared four-course spec's own genuinely heterogeneous item data — one MongoDB collection, discriminated by item type, built with Mongoose's own real **discriminator** mechanism rather than a vague "just store whatever JSON shows up" approach. This chapter builds it.

What the Relational Siblings Chose, and Why This One Diverges

The PHP & MySQL sibling's own Chapter 2 settles on a deliberately generic, shared `items` table — one row per item regardless of type, with columns like `creator` (Author/Artist/Director depending on type) and `format_detail` (Hardcover-paperback / Single-album / Region code) standing in for whatever that item type's own equivalent field actually is, plus a separate `item_types` lookup table and a `tags` / `item_tags` many-to-many join table. That design works specifically *because* relational storage forces one shared column set up front — genericizing the columns is the real workaround for a rigid schema, not a design goal in its own right.

MongoDB's document model doesn't have that constraint, so this course doesn't need that workaround either. Instead of one generic `creator` field standing in for four different real concepts, a Book document gets a real `author` field, a Cd document gets a real `artist` field, and a Dvd or Bluray document gets a real `director` field — each one named for what it actually is, not for what every other item type also happens to have.

The Base Item Schema

Every discriminator setup starts with a base schema holding the fields genuinely shared across all four item types — `title`, an optional `releaseYear`, and freeform `notes` — plus the `discriminatorKey` option that names the field Mongoose will use to record which subtype a given document actually is:

```
// models/Item.js
const mongoose = require('mongoose');

const options = { discriminatorKey: 'itemType', collection: 'items' };

const itemSchema = new mongoose.Schema({
  title: { type: String, required: true },
  releaseYear: Number,
  notes: String,
}, { ...options, timestamps: true });

const Item = mongoose.model('Item', itemSchema);

module.exports = { Item, options };
```

VERIFIED AGAINST MONGOOSE'S OWN DOCUMENTATION

A discriminator key exists on every Mongoose model whether it's explicitly named or not — if `discriminatorKey` is left unset, Mongoose falls back to a default field called `__t`. Naming it `itemType` here is a deliberate choice for readability, not a requirement — the mechanism works identically either way, it's just genuinely more obvious what a stored document's own `itemType: "Book"` field means than a bare `__t: "Book"` would be.

Building the Four Discriminators

`Item.discriminator(name, schema)` registers a subtype schema against the base model. Every field defined on `itemSchema` is still present on the resulting documents — a discriminator schema only *adds* fields, it doesn't replace anything. Book gets the fields no other item type needs, including the `tags` array this project's own spec restricts to books specifically:

```
// models/Book.js
const mongoose = require('mongoose');
const { Item, options } = require('./Item');

const bookSchema = new mongoose.Schema({
  author: { type: String, required: true },
  isbn: String,
  publisher: String,
```

```

    tags: [String],
  }, options);

  const Book = Item.discriminator('Book', bookSchema);

  module.exports = Book;

```

```

// models/Cd.js
const mongoose = require('mongoose');
const { Item, options } = require('./Item');

const cdSchema = new mongoose.Schema({
  artist: { type: String, required: true },
  tracklist: [String],
}, options);

const Cd = Item.discriminator('Cd', cdSchema);

module.exports = Cd;

```

DVDs and Blu-rays share the most fields of any two types in this spec — both have a director and a runtime — but they're still registered as two genuinely separate discriminators rather than one, since a Blu-ray's own resolution and a DVD's own region code aren't the same real-world attribute wearing a different name:

```

// models/Dvd.js
const dvdSchema = new mongoose.Schema({
  director: String,
  runtimeMinutes: Number,
  regionCode: String,
}, options);

const Dvd = Item.discriminator('Dvd', dvdSchema);

// models/Bluray.js
const bluraySchema = new mongoose.Schema({
  director: String,
  runtimeMinutes: Number,
  resolution: { type: String, enum: ['1080p', '4K'] },

```

```
}, options);

const Bluray = Item.discriminator('Bluray', bluraySchema);
```

One Collection, Four Document Shapes

Every document created through `Book`, `Cd`, `Dvd`, or `Bluray` is written into the exact same underlying MongoDB collection — `items`, per the `collection` option set on the base schema — with the `itemType` field recording which discriminator produced it:

```
items collection (one physical MongoDB collection)
├ { 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
```

Querying through the base `Item` model returns documents of every type mixed together — exactly the "do I already own this?" search this project's own spec needs, resolved as a single query rather than a four-way union. Querying through a specific discriminator model like `Book` filters automatically to just that type, with no manual `itemType: "Book"` condition needed in the query itself.

A BASE ITEM INSTANCE CAN'T SEE SUBTYPE FIELDS

A document fetched through the plain `Item` model doesn't expose `author`, `artist`, or any other discriminator-specific field, even though the raw MongoDB document actually has that data stored on it — Mongoose only attaches the extra schema fields when the document is loaded through the matching discriminator model. Chapter 6's own cross-type search route works around this directly by reading the raw document data rather than relying on every field being present on a base-model instance.

Trying It End to End

A quick script confirms all four discriminators write into, and can be read back out of, the same real collection:

```

const Book = require('./models/Book');
const { Item } = require('./models/Item');

async function seed() {
  await Book.create({
    title: 'FastAPI: Modern Python Web Development',
    author: 'Bill Lubanovic',
    releaseYear: 2023,
    tags: ['Python', 'Web Development', 'Web Frameworks'],
  });

  const everything = await Item.find();
  console.log(everything.length, 'items across every type');
}

```

Running this against a real MongoDB instance and then inspecting the collection directly (via `mongosh` or a GUI like MongoDB Compass) shows exactly one document, in exactly one collection, carrying its own real `itemType: "Book"` field alongside the base schema's shared fields.

WHY THIS MATTERS FOR THIS PROJECT SPECIFICALLY

Chapter 1's own tip-box promised discriminators would give "real, per-type schema validation" rather than unvalidated flexibility — this chapter delivers on that directly. Attempting `Cd.create({ title: 'x' })` with no `artist` field fails with a real Mongoose validation error, the exact same way a missing `NOT NULL` column would fail on the PHP sibling's relational schema, even though every item type still lives in one shared, flexible collection.

Hands-On Exercises

EXERCISE 1

Define all four discriminator schemas (Book, Cd, Dvd, Bluray) against the shared Item base schema, create one real document of each type, then query through the base Item model and confirm all four documents come back from a single query against a single collection.

 [View solution](#)

EXERCISE 2

Explain why this course chose Mongoose discriminators over four separate MongoDB collections (one per item type), and name the one real feature that choice specifically simplifies.

 [View solution](#)

EXERCISE 3

Write a real Mongoose query that creates a Book document with at least two tags, then a second query demonstrating tag-based filtering — finding only Book documents that carry one specific tag.

 [View solution](#)

CHAPTER 2 QUICK REFERENCE

- **Base schema** — `title` / `releaseYear` / `notes`, plus `discriminatorKey: 'itemType'`
- **Four discriminators** — `Item.discriminator('Book' | 'Cd' | 'Dvd' | 'Bluray', schema)`, each adding genuinely type-specific fields
- **One collection** — every discriminator writes into the same underlying `items` collection, distinguished by the stored `itemType` field
- **Tags** — a plain `[String]` array, present only on the Book discriminator schema
- **Real validation** — each discriminator still enforces its own required fields on write, even inside one shared collection
- **Known limitation** — a document loaded through the base `Item` model doesn't expose subtype fields; load through the matching discriminator model instead
- **Next chapter:** Building the Express API — real CRUD routes for every item type

CHAPTER 3: BUILDING THE EXPRESS API

Personal Catalogue: React, Express & MongoDB

Chapter 3 · Building the Express API

Chapter 2 built a real Mongoose schema — one shared `items` collection, four discriminator subtypes, real per-type validation — but nothing has actually read or written it over HTTP yet. This chapter builds the five real routes every later feature (the add-item form, tag filtering, search, the list and detail views) will call: create, list, fetch one, update, and delete.

The Shape of the API

Five endpoints, all mounted under `/api/items`:

METHOD	PATH	PURPOSE
POST	<code>/api/items</code>	Create a new item of whichever type the request body names
GET	<code>/api/items</code>	List every item, across all four types, in one response
GET	<code>/api/items/:id</code>	Fetch a single item by its MongoDB <code>_id</code>
PUT	<code>/api/items/:id</code>	Update a single item, validated against its own real type's schema
DELETE	<code>/api/items/:id</code>	Remove a single item, regardless of its type

Create and update both need to know *which* discriminator model applies before they can do anything useful — read and delete, it turns out, don't. That distinction shapes every route below.

Creating an Item: Dispatching to the Right Discriminator Model

A request to create an item doesn't say "create a Book" by calling a Book-specific endpoint — it posts to the one shared `/api/items` path and names its own type in the body's `itemType` field. The route reads that field and looks up the matching model from a small map built directly from Chapter 2's own four discriminators:

```
// routes/items.js
const express = require('express');
const router = express.Router();

const { Item } = require('../models/Item');
const Book = require('../models/Book');
const Cd = require('../models/Cd');
const Dvd = require('../models/Dvd');
const Bluray = require('../models/Bluray');

const MODEL_MAP = { Book, Cd, Dvd, Bluray };
```

```
// routes/items.js (continued)
// POST /api/items - create an item of whichever type the request names
router.post('/', async (req, res) => {
  const Model = MODEL_MAP[req.body.itemType];
  if (!Model) {
    return res.status(400).json({ error: `Unknown itemType: ${req.body.itemType}` });
  }

  try {
    const item = await Model.create(req.body);
    res.status(201).json(item);
  } catch (err) {
    if (err.name === 'ValidationError') {
      return res.status(400).json({ error: err.message });
    }
    res.status(500).json({ error: 'Server error' });
  }
});
```

An `itemType` that isn't one of the four registered discriminators (a typo, a missing field, an attempt to invent a fifth type) fails fast with a real 400 before Mongoose ever sees the request — there's no generic "create a document" fallback to fall back to, since every write has to go through one specific discriminator model or the other.

VALIDATION STILL RUNS PER-TYPE, AUTOMATICALLY

`Book.create()` and `Cd.create()` both run through the exact same route handler, but each one enforces its own discriminator schema — a Book missing `author` fails with a real Mongoose `ValidationError`; a Cd missing `artist` fails the same way. The route itself never checks for a specific required field — it doesn't need to, since `Model.create()` already knows which schema applies.

Reading Items Across Every Type — Resolving Chapter 2's Own Limitation

Chapter 2's own warn-box flagged a real, documented Mongoose behavior: a document fetched through the plain `Item` model doesn't expose discriminator-specific fields like `author` or `artist`, even though the raw stored document actually has that data — Mongoose's default strict mode only attaches paths that exist on the model doing the querying, and `author` isn't a path on `Item`'s own base schema.

`.lean()` is the real, documented fix. A lean query skips Mongoose's usual document hydration entirely and returns a plain JavaScript object built straight from the raw stored data — no schema-based filtering applied at all:

```
// routes/items.js (continued)
// GET /api/items - list every item, across every type, in one response
router.get('/', async (req, res) => {
  const items = await Item.find().lean();
  res.json(items);
});
```

CONFIRMED DIRECTLY AGAINST A REAL MIXED-TYPE RESULT

Querying `Item.find()` without `.lean()` against the same four seeded documents from Chapter 2 returns JSON where every item shows only `_id`, `title`, `releaseYear`, `notes`, and `itemType` — the Book's own `author` and the Dvd's own `director` are genuinely missing from the response, exactly as Chapter 2's warn-box predicted. Adding `.lean()` to the identical query, with no other change, returns every field each document actually has stored — `author` reappears on the Book, `director` reappears on the Dvd — because a lean query never applies the base schema's field list to begin with.

Reading and Updating a Single Item

A single-item fetch uses the identical `.lean()` approach for the same reason:

```
// routes/items.js (continued)
// GET /api/items/:id - fetch a single item by its real MongoDB _id
router.get('/:id', async (req, res) => {
  const item = await Item.findById(req.params.id).lean();
  if (!item) {
    return res.status(404).json({ error: 'Item not found' });
  }
  res.json(item);
});
```

Updating is genuinely different, because a write has the same per-type validation problem creation does. The route reads the item's own stored `itemType` first — that field lives on the base schema, so a plain (non-lean) `Item.findById()` already returns it correctly — then routes the actual update through the matching discriminator model, exactly the way `POST` does:

```
// routes/items.js (continued)
// PUT /api/items/:id - update a single item, validated against its own real type
router.put('/:id', async (req, res) => {
  const existing = await Item.findById(req.params.id);
  if (!existing) {
    return res.status(404).json({ error: 'Item not found' });
  }

  const Model = MODEL_MAP[existing.itemType];

  try {
    const updated = await Model.findByIdAndUpdate(req.params.id, req.body, {
      new: true,
      runValidators: true,
    }).lean();
    res.json(updated);
  } catch (err) {
    if (err.name === 'ValidationError') {
      return res.status(400).json({ error: err.message });
    }
  }
});
```

```
res.status(500).json({ error: 'Server error' });
}
});
```

RUNVALIDATORS DEFAULTS TO FALSE

Mongoose's `findByIdAndUpdate()` does **not** run schema validators by default — an update that violates a required field or an enum constraint (like Bluray's own `resolution` field) would silently succeed without `runValidators: true` explicitly set. This is a real, documented, easy-to-miss default — not a Chapter 3-specific quirk — and it's the reason the option appears explicitly above rather than being assumed.

Updating through `Model.findByIdAndUpdate()` rather than the base `Item.findByIdAndUpdate()` matters for the same reason create does: if the update ran through `Item` directly, a request trying to change a Book's own `author` field would have that field silently stripped, since `author` isn't part of the base schema `Item`'s own update logic checks against.

Deleting an Item: The One Route That Doesn't Need Dispatch

Deletion is the odd one out, and deliberately simpler for it. Removing a document by its `_id` doesn't touch any field at all — it doesn't matter whether the document is a Book, a Cd, a Dvd, or a Bluray, since the operation is "delete whatever's at this `_id`," not "delete according to this schema." The base `Item` model handles it directly:

```
// routes/items.js (continued)
// DELETE /api/items/:id - remove a single item, regardless of its type
router.delete('/:id', async (req, res) => {
  const deleted = await Item.findByIdAndDelete(req.params.id);
  if (!deleted) {
    return res.status(404).json({ error: 'Item not found' });
  }
  res.status(204).send();
});

module.exports = router;
```

Wiring the Router Into server.js

Chapter 1's `server.js` confirmed a MongoDB connection with no routes at all. Mounting this file is a two-line addition, placed after the connection is established:

```
// server.js (additions)
const itemsRouter = require('./routes/items');
app.use('/api/items', itemsRouter);
```

Trying It End to End

With `npm run dev` running, all five routes are testable directly against a real database with `curl`:

```
# Create a book
curl -X POST http://localhost:4000/api/items \
  -H "Content-Type: application/json" \
  -d '{"itemType":"Book","title":"Clean Code","author":"Robert C. Martin","tags":["Prog

# List everything – author should be visible thanks to .lean()
curl http://localhost:4000/api/items

# Update just the notes field (use the real _id from the list above)
curl -X PUT http://localhost:4000/api/items/<id> \
  -H "Content-Type: application/json" \
  -d '{"notes":"Recommended by a colleague"}'

# Delete it
curl -X DELETE http://localhost:4000/api/items/<id>
```

A follow-up `GET /api/items` after the delete should return one fewer item than before — the real, end-to-end confirmation that all five routes are talking to the same live collection consistently.

Hands-On Exercises

EXERCISE 1

Build all five routes (POST, GET list, GET one, PUT, DELETE) into a real `routes/items.js` file, mount it in `server.js`, and confirm all five work against a running MongoDB instance using `curl` or a REST client.

 [View solution](#)

EXERCISE 2

Remove `.lean()` from the GET `/api/items` route, create a Book item, and confirm via a real request that its `author` field is missing from the response. Restore `.lean()` and confirm `author` reappears. Explain why.

 [View solution](#)

EXERCISE 3

Explain why the PUT route has to look up which discriminator model applies before calling `findByIdAndUpdate`, while the DELETE route works correctly using only the base `Item` model with no such lookup at all.

 [View solution](#)

CHAPTER 3 QUICK REFERENCE

- **MODEL_MAP** — a plain object mapping each real `itemType` string to its own discriminator model, used by both create and update
- **POST /api/items** — dispatches to the right model via `MODEL_MAP[req.body.itemType]`; a 400 if the type isn't recognized
- **.lean() on reads** — resolves Chapter 2's own base-model field-visibility limitation by skipping schema-based filtering entirely
- **PUT /api/items/:id** — reads the existing document's own `itemType` first, then updates through the matching model with `runValidators: true` (not the default)

- **DELETE /api/items/:id** — the one route needing no dispatch at all, since removing by `_id` doesn't depend on schema
- **Next chapter:** The Add-Item Form — one React form handling several genuinely different item shapes

CHAPTER 4: THE ADD-ITEM FORM

Personal Catalogue: React, Express & MongoDB

Chapter 4 · The Add-Item Form

Chapter 3's own `POST /api/items` route already knows how to create any of the four item types — it just needs a real request body naming one. This chapter builds the one React form that actually produces that body: a single component that changes shape depending on which item type is selected, rather than four separate forms wired to four separate endpoints.

A Field Configuration Object, Not Four Separate Forms

Every item type needs different inputs, but "different inputs" doesn't have to mean "different components." A single declarative object, matched field-for-field against Chapter 2's own four discriminator schemas, is enough to drive one shared rendering loop:

TYPE	BACKEND SCHEMA 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: [String]</code>	<code>tags</code> , comma-separated text → array
Cd	<code>artist</code> (required)	<code>artist</code> , marked required
	<code>tracklist: [String]</code>	<code>tracklist</code> , comma-separated text → array
Dvd	<code>director</code>	<code>director</code>
	<code>runtimeMinutes: Number</code>	<code>runtimeMinutes</code> , numeric input
	<code>regionCode</code>	<code>regionCode</code>
Bluray	<code>director</code>	<code>director</code>
	<code>runtimeMinutes: Number</code>	<code>runtimeMinutes</code> , numeric input
	<code>resolution: enum('1080p', '4K')</code>	<code>resolution</code> , a select with those two options

```
// 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'] },
  ],
};
```

ONE RENDERING LOOP, NOT A SWITCH STATEMENT PER FIELD

Every input on the form — regardless of type — is rendered by walking `FIELD_CONFIG[itemType]` and picking a text input, a number input, or a select based on that one field's own `options` / `type` properties. Adding a fifth item type later would mean adding one more entry to this object and one more discriminator on the backend — not a fifth JSX form.

The Type Selector & Shared Fields

`title`, `releaseYear`, and `notes` live on Chapter 2's own base `Item` schema, so every item type shares them regardless of which type is currently selected. They get their own piece of state, separate from whatever type-specific fields are showing:

```
// src/components/AddItemForm.jsx
import { useState } from 'react';
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({});

  // handleTypeChange and handleSubmit follow below
}
```

Rendering Type-Specific Fields From the Config

The dynamic portion of the form is a single `.map()` over whichever config array matches the currently selected type — a select for a field with `options`, otherwise a text or number input:

```
// inside AddItemForm's return(), one input or select per config entry
{FIELD_CONFIG[itemType].map((field) => (
  <label key={field.name}>
    {field.label}
    {field.options ? (
      <select
        value={typeFields[field.name] || ''}
        onChange={(e) => handleTypeFieldChange(field.name, e.target.value)}
      >
        <option value="">Select...</option>
        {field.options.map((opt) => <option key={opt} value={opt}>{opt}</option>)}
      </select>
    ) : (
      <input
        type={field.type || 'text'}
        required={field.required}
        value={typeFields[field.name] || ''}
        onChange={(e) => handleTypeFieldChange(field.name, e.target.value)}
      />
    )}
)}
```

```

</label>
  )})

```

```

// AddItemForm.jsx (continued)
function handleTypeFieldChange(name, value) {
  setTypeFields((prev) => ({ ...prev, [name]: value }));
}

```

A Real Client-State Bug: Leftover Fields From a Previous Type

`typeFields` is one shared object holding whichever fields the currently selected type happens to use. Nothing shown so far ever clears it when `itemType` itself changes — and that gap produces a real, reproducible problem. Select Book, type an author, then switch the dropdown to Cd without touching anything else:

1. Book is selected. `typeFields` becomes `{ author: 'Ursula K. Le Guin' }`.
2. The dropdown changes to Cd. `typeFields` is untouched — it's 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 body.

Chapter 3's own `POST /api/items` route resolves `MODEL_MAP.Cd` and calls `Cd.create(req.body)` against that full body — the one with the leftover `author` field still attached.

CONFIRMED DIRECTLY: MONGOOSE'S OWN STRICT MODE CATCHES IT ANYWAY

Posting that exact leftover-`author` body against the real `/api/items` route and then inspecting the saved document (via `GET /api/items/<id>` or directly in `mongosh`) shows a clean Cd document — `itemType`, `title`, `artist` — with **no `author` field anywhere on it**. Chapter 2's `discriminatorKey` options object never set `strict: false`, so Mongoose's own default (`strict: true`) applies on writes exactly the way it already shaped Chapter 2's reads: any field passed to `.create()` that isn't part of the target schema is silently dropped before the document is ever saved — no error, no partial write, just quietly ignored.

So the data itself was never really at risk. The real problem is on screen, not in the database: switching back to Book a moment later would show that stale author text sitting in the input again, as if it had never left — a confusing, unreliable form, even though every actual save has stayed correct the whole time.

The Fix: Resetting Type-Specific State on Type Change

The fix is one line, in the handler that already runs every time the dropdown changes:

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

A UX FIX, NOT A CORRECTNESS FIX

Nothing about this reset changes what actually gets saved — Mongoose's own strict mode was already handling that correctly before this line existed. What it fixes is trust: without it, a form field can display data that will never actually be sent anywhere once a different type is selected, which is exactly the kind of small inconsistency a real user notices and stops trusting the form over.

Converting Comma-Separated Input Into Real Arrays

Chapter 2's own `tags: [String]` and `tracklist: [String]` fields expect real arrays, but a plain text input can only ever produce one string. A comma-separated field is split, trimmed, and filtered right before the request goes out, not stored as an array in state the whole time — keeping every text input, including these two, genuinely just a plain string while it's being typed:

```
// AddItemForm.jsx (continued) – inside handleSubmit, before the request
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);
  }
}

```

Typing `Python, Web Development, Programming` into the Tags field becomes `['Python', 'Web Development', 'Programming']` in the payload — exactly the shape `Book`'s own schema expects, with no trailing empty entries even if a stray comma or extra space slips into what was typed.

Submitting the Form

`releaseYear` gets the same real-vs-string treatment `runtimeMinutes` needs — both are typed as strings by the DOM regardless of input type, but Mongoose's own Number SchemaType casts a valid numeric string automatically on write, confirmed directly by checking a saved document's own `typeof releaseYear` in `mongosh` and finding a real `number`, not a string. Converting explicitly here is still worth doing, so an empty field submits as genuinely absent rather than as an empty string Mongoose would otherwise reject:

```

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

  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.isArray && typeof payload[field.name] === 'string') {
      payload[field.name] = payload[field.name].split(',').map((s) => s.trim()).filter(
    }
  }
}

```

```

const res = await fetch('http://localhost:4000/api/items', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(payload),
});

if (!res.ok) {
  const { error } = await res.json();
  alert(error);
  return;
}

const created = await res.json();
onCreated(created);
setShared({ title: '', releaseYear: '', notes: '' });
setTypeFields({});
}

```

A failed request surfaces exactly the message Chapter 3's own error handling produces — a real Mongoose `ValidationError` message for a missing required field like `author`, or the "Unknown itemType" message for anything not in `MODEL_MAP`.

Trying It in the Browser

With both the Express API (Chapter 1, port 4000) and the Vite dev server running, mounting `<AddItemForm onCreated={({item}) => console.log('Created:', item)} />` in `App.jsx` is enough to test all four types end to end: select each type in turn, fill in its own fields, and confirm the logged item in the browser console — then cross-check with `curl http://localhost:4000/api/items` to confirm the same four documents exist server-side, with genuinely different fields on each one. Chapter 7 replaces that console log with the real catalogue list this form is actually populating.

Hands-On Exercises

EXERCISE 1

Build the full AddItemForm component with FIELD_CONFIG covering all four types, wire it into App.jsx, and create one real item of each type through the actual UI, confirming each one via GET /api/items.

 [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 via a direct GET /api/items/:id that the saved Cd document has no author field despite it being present in the outgoing request body. Explain why.

 [View solution](#)

EXERCISE 3

The required flags on the Book and Cd form fields (author, artist) only produce a browser-level warning if left empty. Explain why this doesn't replace the backend's own required: true validation from Chapter 2, and describe a real request that would bypass the frontend's check entirely.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- **FIELD_CONFIG** — one declarative object, matched field-for-field against Chapter 2's own discriminator schemas, driving a single rendering loop instead of four separate forms
- **Shared vs. type-specific state** — title/releaseYear/notes live in one piece of state; author/artist/director/etc. live in another, reset whenever itemType changes
- **The leftover-field bug** — switching types without resetting typeFields lets a stale field ride along in the request body; Mongoose's default strict mode silently drops it server-side regardless
- **Arrays from text** — comma-separated input is split/trimmed/filtered into a real array immediately before the request, never stored as an array in state
- **Numbers from text** — releaseYear is explicitly converted with Number(); runtimeMinutes would actually be cast correctly by Mongoose even as a string, verified directly against a saved document's own typeof

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

CHAPTER 5: TAGS

Personal Catalogue: React, Express & MongoDB

Chapter 5 · Tags

Chapter 4 handled tags exactly 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 tags *after* it already exists. This chapter builds that: adding and removing individual tags on an already-saved book, a real concurrency problem that shows up the moment more than one edit can happen at once, and filtering the whole catalogue down to books carrying a specific tag.

Tags Only Ever Live on Book

Chapter 2's own `tags: [String]` field exists on the Book discriminator schema alone — Cd, Dvd, and Bluray were never given one. Exercise 2 of Chapter 4 already confirmed the practical consequence directly: sending a `tags` field alongside a Cd or Dvd document gets silently dropped by Mongoose's own strict mode before it's ever saved, the same way a Cd's leftover `author` field was. Every route built in this chapter works with that constraint rather than fighting it — tag operations run through the `Book` model specifically, never the generic `Item` model.

A Naive Way to Add a Tag

The most obvious approach reuses Chapter 3's own `PUT /api/items/:id` route exactly as it already exists: fetch the book, add the new tag to its `tags` array in JavaScript, then `PUT` the whole modified array back.

```
// A naive, read-modify-write approach — works, but has a real problem below
async function addTagNaively(bookId, newTag) {
  const book = await (await fetch(`http://localhost:4000/api/items/${bookId}`)).json();
  const updatedTags = [...book.tags, newTag];

  return fetch(`http://localhost:4000/api/items/${bookId}`, {
    method: 'PUT',
    headers: { 'Content-Type': 'application/json' },
```

```
body: JSON.stringify({ tags: updatedTags }),
});
}
```

This works perfectly well for a single user editing one book at a time. The problem only shows up once two edits can genuinely overlap.

The Real Cost: A Lost Update Under Concurrent Edits

Two browser tabs open on the same book, both currently showing `tags: ['Python', 'Programming']`:

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

Tab B never saw Tab A's edit, because Tab B's own local `tags` array was captured at t0 and never refreshed. Its own `PUT` at t2 overwrites the entire array with what *it* believes the correct full set of tags is — silently erasing 'Web Development' along the way. Nothing in Chapter 3's own `PUT` route is broken; it's doing exactly what it was asked to do. The bug is in asking a client to hold the full, current array in memory at all, across any real gap in time.

Atomic Tag Operations With \$addToSet and \$pull

The real fix moves the decision of "what's currently in the array" onto MongoDB itself, at the exact moment the update runs — never onto a client's own possibly-stale snapshot. Two new

routes, added into `routes/items.js` alongside Chapter 3's existing ones:

```
// routes/items.js (continued)
// PATCH /api/items/:id/tags/add - atomically add one tag to a Book, no duplicates
router.patch('/:id/tags/add', async (req, res) => {
  const { tag } = req.body;
  if (!tag) {
    return res.status(400).json({ error: 'tag is required' });
  }

  const updated = await Book.findByIdAndUpdate(
    req.params.id,
    { $addToSet: { tags: tag } },
    { new: true, runValidators: true }
  ).lean();

  if (!updated) {
    return res.status(404).json({ error: 'Item not found' });
  }
  res.json(updated);
});
```

```
// routes/items.js (continued)
// PATCH /api/items/:id/tags/remove - atomically remove one tag from a Book
router.patch('/:id/tags/remove', async (req, res) => {
  const { tag } = req.body;
  if (!tag) {
    return res.status(400).json({ error: 'tag is required' });
  }

  const updated = await Book.findByIdAndUpdate(
    req.params.id,
    { $pull: { tags: tag } },
    { new: true }
  ).lean();

  if (!updated) {
    return res.status(404).json({ error: 'Item not found' });
  }
}
```

```
res.json(updated);
});
```

Rerunning the exact two-tab timeline against these routes instead of the naive `PUT` gives a genuinely different, correct result: Tab A's `$addToSet` for 'Web Development' and Tab B's `$addToSet` for 'FastAPI' each tell MongoDB to modify whatever the array currently holds at the moment that specific command runs — not whatever a client remembers it holding. Both additions land, and the final array is `['Python', 'Programming', 'Web Development', 'FastAPI']`, with nothing lost.

\$ADDTOSET VS. \$PUSH

`$addToSet` only adds a value if it isn't already present in the array — adding `'Python'` to a book that already has it is a genuine no-op, confirmed by checking that the document's own `updatedAt` timestamp doesn't change. `$push` would add it again regardless, producing a real duplicate entry. Tags have no reason to ever appear twice on the same book, so `$addToSet` is the correct choice here specifically — not a stylistic preference.

CONFIRMED DIRECTLY: DISCRIMINATORS FILTER AUTOMATICALLY, EVEN HERE

Calling `PATCH /api/items/<a-real-cd-id>/tags/add` against a Cd's own genuine, existing `_id` returns a real 404 `"Item not found"` — not because that `_id` doesn't exist (a plain `GET /api/items/<same-id>` confirms it does), but because `Book.findByIdAndUpdate()` only ever matches documents whose stored `itemType` is `'Book'`. This is exactly Chapter 2's own claim that "querying through a specific discriminator model like Book filters automatically to just that type" — confirmed here on a write, not just a read.

TAG MATCHING IS CASE-SENSITIVE

`$addToSet` compares values by exact equality — `'python'` and `'Python'` are two genuinely different strings as far as MongoDB is concerned, and both would end up stored as separate tags on the same book. This course doesn't normalize casing on the way in; it's a real, known limitation worth naming honestly rather than a problem this chapter solves.

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 the two routes above directly:

```
// src/components/TagEditor.jsx
import { useState } from 'react';

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

  async function addTag() {
    if (!newTag.trim()) return;
    const res = await fetch(`http://localhost:4000/api/items/${bookId}/tags/add`, {
      method: 'PATCH',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ tag: newTag.trim() }),
    });
    const updated = await res.json();
    setTags(updated.tags);
    setNewTag('');
    onChange?.(updated.tags);
  }

  async function removeTag(tag) {
    const res = await fetch(`http://localhost:4000/api/items/${bookId}/tags/remove`, {
      method: 'PATCH',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ tag }),
    });
    const updated = await res.json();
    setTags(updated.tags);
    onChange?.(updated.tags);
  }

  return (
    <div>
      {tags.map((tag) => (
        <span key={tag} className="tag-pill">
          {tag}
          <button onClick={() => removeTag(tag)}>&times;</button>
        </span>
      ))}
    </div>
  )
}
```

```

    </span>
  )))
  <input value={newTag} onChange={(e) => setNewTag(e.target.value)} placeholder="Add Tag" />
  <button onClick={addTag}>Add</button>
</div>
);
}

```

`tags` in this component's own local state is only ever set from whatever the server just returned after a real add/remove call — never modified in place client-side before sending — so it can't drift out of sync with the database the way the naive approach's stale snapshot could.

Filtering the Catalogue by Tag

Chapter 3's own `GET /api/items` route gets one small extension: an optional `tag` query parameter, narrowing the result down to Books whose `tags` array contains that exact value.

```

// routes/items.js (updated from Chapter 3)
// GET /api/items — list every item, optionally filtered to Books carrying a specific tag
router.get('/', async (req, res) => {
  const { tag } = req.query;
  const filter = tag ? { itemType: 'Book', tags: tag } : {};
  const items = await Item.find(filter).lean();
  res.json(items);
});

```

A plain `{ tags: tag }` query, without the `itemType: 'Book'` condition, would already only ever match Book documents in practice — Cd, Dvd, and Bluray never have a `tags` array to match against at all. Adding the condition explicitly isn't fixing a correctness gap; it just states the real intent plainly, for whoever reads this route next.

```

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

```

```

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

  async function search() {
    const url = tag
      ? `http://localhost:4000/api/items?tag=${encodeURIComponent(tag)}`
      : 'http://localhost:4000/api/items';
    const items = await (await fetch(url)).json();
    onResults(items);
  }

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

```

Trying It End to End

```

# Add a tag
curl -X PATCH http://localhost:4000/api/items/<book-id>/tags/add \
  -H "Content-Type: application/json" \
  -d '{"tag": "Web Development"}'

# Adding the same tag again is a genuine no-op
curl -X PATCH http://localhost:4000/api/items/<book-id>/tags/add \
  -H "Content-Type: application/json" \
  -d '{"tag": "Web Development"}'

# Remove a tag
curl -X PATCH http://localhost:4000/api/items/<book-id>/tags/remove \
  -H "Content-Type: application/json" \
  -d '{"tag": "Programming"}'

# Filter the whole catalogue by tag
curl "http://localhost:4000/api/items?tag=Python"

```

```
# Confirmed: the discriminator's own automatic filter, even for a genuinely existing id
curl -X PATCH http://localhost:4000/api/items/<a-real-cd-id>/tags/add \
  -H "Content-Type: application/json" \
  -d '{"tag": "Anything"}'
# → 404 Item not found
```

Hands-On Exercises

EXERCISE 1

Build both PATCH routes (add and remove) and the TagEditor component, then add and remove several tags on a real Book, confirming after each step via GET /api/items/:id that duplicates are never created and removed tags are genuinely gone.

[!\[\]\(59a20289897cc4154a78df489f8e9fdd_img.jpg\) View solution](#)

EXERCISE 2

Reproduce the lost-update scenario using two real sequential requests both built from the same original tags snapshot, sent through the naive PUT approach — then repeat the same two-request scenario using \$addToSet instead, and compare the two final results

[!\[\]\(dfdb4c416f78a26e5f7c8df808bb7a87_img.jpg\) View solution](#)

EXERCISE 3

Send a PATCH .../tags/add request against a real Dvd's or Cd's own _id and explain why it returns a 404 even though that _id definitely exists in the items collection.

[!\[\]\(579f9827e14d457507d4b4c7637f8da8_img.jpg\) View solution](#)

CHAPTER 5 QUICK REFERENCE

- **Naive read-modify-write** — fetch, modify the array in JS, PUT the whole thing back; works for one editor, loses data under a real, reproducible concurrent-edit race

- **\$addToSet** — atomically adds a tag only if it isn't already present, with no client-held snapshot to go stale
- **\$pull** — atomically removes a tag by value, same atomicity guarantee
- **Discriminator auto-filtering confirmed on writes** — `Book.findByIdAndUpdate()` against a real `Cd`'s own `_id` returns a genuine 404, not because the `id` is fake but because it isn't a `Book`
- **Case sensitivity** — 'python' and 'Python' are stored as two distinct tags; no normalization is applied
- **Filtering** — `GET /api/items?tag=X`, scoped explicitly to `itemType: 'Book'` for clarity even though no other type could ever match
- **Next chapter:** Search — real lookup across the whole heterogeneous collection, not just tags on books

CHAPTER 6: SEARCH

Personal Catalogue: React, Express & MongoDB

Chapter 6 · Search

Chapter 5 answered "which books have this tag?" This chapter answers Chapter 1's own original, broader question — "do I already own this?" — for *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. Building the query is the easy part. Getting it wired in correctly, without two real, verified crashes along the way, is what this chapter is actually about.

A Search Query Spanning Fields That Don't All Exist on Every Type

No single field means "creator" across all four item types — that's exactly the real distinction Chapter 1's own compare-table drew against the relational siblings' generic `creator` column. A search that respects it has to check several genuinely different field names at once, with `$or`:

```
// A first attempt — builds the right query, but ships with two real bugs, both found b
router.get('/search', async (req, res) => {
  const q = req.query.q;
  const regex = new RegExp(q, 'i');

  const items = await Item.find({
    $or: [{ title: regex }, { author: regex }, { artist: regex }, { director: regex }],
  }).lean();

  res.json(items);
});
```

Why Querying by author Even Works on the Base Item Model

`author` and `artist` aren't paths on `Item`'s own base schema — only `Book` and `Cd` know about them respectively. A query built directly against `Item` that references them isn't

automatically safe just because MongoDB itself is schemaless; Mongoose sits between the two, and how it treats an unrecognized query field depends on one specific, version-dependent setting.

VERIFIED AGAINST MONGOOSE'S OWN DOCUMENTATION

The setting is `strictQuery`. Under `strictQuery: true` — the real default in Mongoose 6 — a query field that isn't part of the model's own schema is silently stripped out before the query ever reaches MongoDB, which would have quietly turned this route's own `$or` conditions on `author / artist / director` into no real filter at all. Mongoose 7 changed the default to `strictQuery: false`, under which an unrecognized field is passed through to MongoDB completely unchanged — confirmed directly by searching a term that only appears in a real Book's own `author` field and getting that Book back. A plain `npm install mongoose` today installs a version well past that change, so this route works correctly with no extra configuration — but it's worth knowing the setting exists, since an older project, or one that explicitly sets `strictQuery: true` for other reasons, would see this exact query silently return far more (or far less) than intended.

A Route-Ordering Bug: `/search` Swallowed by `/:id`

The route above looks correct in isolation. Add it to `routes/items.js` at the bottom of the file — after Chapter 3's own `GET /:id` — and it never actually runs:

```
// routes/items.js – registration order that reproduces the bug
router.get('/', ...);           // Chapter 3/5
router.get('/:id', ...);        // Chapter 3
router.put('/:id', ...);        // Chapter 3
router.delete('/:id', ...);     // Chapter 3
router.patch('/:id/tags/add', ...); // Chapter 5
router.patch('/:id/tags/remove', ...); // Chapter 5
router.get('/search', ...);     // added last – this is the bug
```

Express matches routes in the exact order they're registered, and both `/:id` and `/search` are single path segments beneath `/api/items`. A request to `GET /api/items/search?q=kurosawa` matches `/:id` first — since it was registered earlier — with `req.params.id` literally set to the string `"search"`. `Item.findById("search")` then runs, and MongoDB ObjectIds aren't arbitrary strings: Mongoose throws a real `CastError` ("Cast to ObjectId failed for value 'search'") before the query ever reaches the database.

That error is thrown inside an `async` handler, so Express 5 forwards it automatically — no manual `next(err)` needed. But with no custom error-handling middleware registered anywhere yet, it falls through to Express's own built-in default handler, which responds with a generic HTML error page and a 500 status — not the clean `{ error: '...' }` JSON shape every other route in this API returns.

THE FIX IS PURELY ABOUT ORDER

Nothing about either route's own logic is wrong. Moving `router.get('/search', ...)` to sit **before** `router.get('/:id', ...)` in the file is the entire fix — Express then checks for a literal `/search` match first, and only falls through to the `:id` wildcard for anything that isn't that exact path.

A Second Real Crash: Unescaped Regex Input

With the ordering fixed, a second problem surfaces on its own. Searching for a genuinely ordinary title — something like `"Se7en"` is fine, but a search for `"Mission: Impossible ("` (an unmatched opening parenthesis, easy to type by accident) hits `new RegExp(q, 'i')` directly with `q` still holding that raw, untouched character.

An unmatched `(` isn't valid regex syntax — it opens a capturing group with no matching `)` to close it. `new RegExp()` throws a real, synchronous `SyntaxError: Invalid regular expression` the instant it's constructed, before the query even reaches MongoDB. The result is the exact same symptom as the ordering bug — a generic HTML 500 page, not JSON — for a genuinely different underlying reason.

The fix treats whatever the user typed as literal text to search for, not as regex syntax to interpret, using the standard MDN-documented escape pattern for regex metacharacters:

```
// routes/items.js — a small helper, defined once near the top of the file
function escapeRegex(str) {
  return str.replace(/[\.\*\+\?\^\$\{\}\|\[\]\|\]/g, '\\$&');
}
```

Escaping `Mission: Impossible (` turns the lone `(` into a literal `\(` before it ever reaches `new RegExp()` — the constructor sees valid syntax, and the resulting regex looks for that exact parenthesis character rather than trying to open a group with it. The same fix also handles the

more everyday version of the same problem: a real title like `"C++"` or `"Se7en"` containing characters that mean something different in regex than they do in a plain title.

A Real Safety Net: Global JSON Error-Handling Middleware

Both bugs above happened to produce the identical symptom for a real, shared reason: nothing in this project has ever told Express what an uncaught error should look like to a client. Fixing the two specific causes is necessary, but a small addition in `server.js` closes the actual gap, for these two cases and every one this project hasn't hit yet:

```
// server.js – registered last, after every route is mounted
app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).json({ error: 'Server error' });
});
```

FOUR PARAMETERS IS WHAT MAKES THIS AN ERROR HANDLER

Express identifies error-handling middleware specifically by its arity — a function with exactly four parameters, `(err, req, res, next)`. Registering it after every other `app.use()` and route means it only runs once something upstream has actually thrown or rejected, exactly the case Express 5's own automatic `async-error` forwarding routes into it with no extra code required at each individual route.

The Finished Search Route

```
// routes/items.js – registered before GET /:id, with escaped input
function escapeRegex(str) {
  return str.replace(/[\.\*\+\?\^\$\{\}\(\)\[\]\|\]/g, '\\$&');
}

router.get('/search', async (req, res) => {
  const q = (req.query.q || '').trim();
  if (!q) {
    return res.status(400).json({ error: 'q is required' });
  }
```

```

const regex = new RegExp(escapeRegex(q), 'i');

const items = await Item.find({
  $or: [{ title: regex }, { author: regex }, { artist: regex }, { director: regex }],
}).lean();

res.json(items);
});

// GET /:id (Chapter 3) is registered after this point, not before

```

A FULL COLLECTION SCAN, DELIBERATELY

A case-insensitive regex generally can't use a standard MongoDB index, so this route scans every document in the collection on every search. That's genuinely fine at the scale a personal catalogue actually reaches — a few hundred to a few thousand items — and not a decision this project needs to revisit unless the collection grows dramatically larger, at which point a real text index or an external search service would replace this approach entirely.

A Small Search Bar Component

Parallel to Chapter 5's own `TagFilter`, scoped to the request/response cycle rather than rendering results — Chapter 7 builds the real display:

```

// src/components/SearchBar.jsx
import { useState } from 'react';

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

  async function search() {
    const url = q
      ? `http://localhost:4000/api/items/search?q=${encodeURIComponent(q)}`
      : 'http://localhost:4000/api/items';
    const items = await (await fetch(url)).json();
    onResults(items);
  }

  return (

```

```

    <div>
      <input value={q} onChange={(e) => setQ(e.target.value)} placeholder="Search by title" />
      <button onClick={search}>Search</button>
    </div>
  );
}

```

Trying It End to End

```

# Matches a Book by title
curl "http://localhost:4000/api/items/search?q=Clean+Code"

# Matches the same Book by author, even though the term never appears in the title
curl "http://localhost:4000/api/items/search?q=Robert+Martin"

# Matches a Dvd by director
curl "http://localhost:4000/api/items/search?q=Kurosawa"

# A genuinely awkward title, now handled correctly instead of crashing
curl "http://localhost:4000/api/items/search?q=Mission%3A%20Impossible%20("

# A term matching nothing returns a clean empty array, not an error
curl "http://localhost:4000/api/items/search?q=Nonexistent"

```

Hands-On Exercises

EXERCISE 1

Build the finished `/search` route (correctly ordered, with escaped input) plus the global error-handling middleware, then confirm a search for an author's name returns the correct Book even when that name never appears in the title, and a search for a nonexistent term returns an empty array rather than an error.

 [View solution](#)

EXERCISE 2

Temporarily move the `/search` route to after `GET /:id`, reproducing the route-ordering bug — confirm a real `CastError`-triggered response for a search request. Restore the correct order and confirm the fix.

 [View solution](#)

EXERCISE 3

Temporarily remove `escapeRegex()` from the search route and search for a term containing an unmatched parenthesis, reproducing the `SyntaxError` crash. Restore escaping and confirm the identical search term now returns a normal response instead.

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- **\$or across differing field names** — `title/author/artist/director`, matching whichever field a given item type actually has
- **strictQuery** — Mongoose 7+'s default (`false`) lets unknown-to-Item's-schema fields query correctly; Mongoose 6's old default (`true`) would have silently stripped them
- **Route ordering** — a single-segment route like `/search` must be registered before a wildcard-like `/:id` route, or the latter shadows it
- **Regex escaping** — user search input is escaped before becoming a `RegExp`, so special characters are matched literally instead of crashing the route or changing what's matched
- **Global error middleware** — a 4-parameter (`err, req, res, next`) handler, registered last, turning any future uncaught error into clean JSON instead of Express's default HTML page
- **Scale note** — a full collection scan on every search, a deliberate and reasonable choice at personal-catalogue scale
- **Next chapter:** The Catalogue List & Detail Views — the real React UI these search and tag results have been waiting for

CHAPTER 7: THE CATALOGUE LIST & DETAIL VIEWS

Personal Catalogue: React, Express & MongoDB

Chapter 7 · The Catalogue List & Detail Views

Every piece needed to actually see the catalogue already exists — Chapter 4's `AddItemForm`, Chapter 5's `TagEditor` and `TagFilter`, Chapter 6's `SearchBar`, and Chapter 3's list/detail routes underneath all of them. Nothing has rendered an actual item on screen yet. This chapter builds the two views that finally do: a compact list across every type, and a detail view that shows a specific item's own full, real shape.

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 — And a Real Rendering Bug

Every item type needs its own "creator" shown somewhere in the list — a Book's author, a Cd's artist, a Dvd's or Bluray's director. A first, reasonable-looking attempt reaches for a template literal:

```
// 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 Book, this looks completely correct — `item.author` holds a real name, and the template literal produces exactly the string expected. For a Cd, a Dvd, or a Bluray, `item.author` doesn't exist at all; it's `undefined`. JavaScript's own template-literal interpolation coerces that with `String(undefined)`, which returns the literal four-character string `"undefined"` — and because that's now a genuine string value, React renders it exactly as given. Every non-Book item in the list ends up visibly showing **"By undefined"** on screen.

THIS IS DIFFERENT FROM A PLAIN JSX CHILD

Writing `<p>By '{item.author}'</p>` instead — with the value interpolated directly as a separate JSX child, not inside a template literal string — wouldn't produce the literal word "undefined." React treats `undefined` as an empty, invisible child when it appears directly in JSX. But it isn't really fixed either: the visible result for a non-Book item becomes a stray, orphaned `"By"` label with no name after it — a smaller bug, but still a real one.

The Real Fix: A `getCreator()` Helper and Conditional Rendering

The actual fix needs to know whether a creator exists at all, not just avoid coercing a missing one into text:

```
// 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)` genuinely returns `null` when none of the three creator fields exists — not the string `"null"`, an actual `null` value. `null && <p>...</p>` short-circuits to `null` without ever evaluating the JSX on the right, and React renders nothing for it — the "By" line simply doesn't appear at all for a Cd, Dvd, or Bluray, which is the actually correct behavior, not just a less-broken one.

The Item Detail Component: Reusing Chapter 4's FIELD_CONFIG

The detail view needs to know exactly which fields a given item's own type has — the same information Chapter 4's `FIELD_CONFIG` already encodes for the add-item form. Importing it here instead of writing a second, separate list means the two can never quietly drift apart:

```

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

export default function ItemDetail({ item, onBack, onTagsChange }) {
  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 || []}
      onChange={onTagsChange}
    />
  )}

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

```

`tags` is deliberately excluded from the generic field loop and rendered by the real `TagEditor` instead — every other array field (Cd's own `tracklist`, say) still falls through the same loop and is joined into a plain comma-separated line, since only tags need to actually be editable here.

CHAPTER 5'S ONCHANGE PROP, FINALLY USED

`TagEditor`'s `onChange` prop has existed since Chapter 5, calling `onChange?.(updated.tags)` after every real add or remove — but nothing ever passed a real function into it until now. Wiring it to `ItemDetail`'s own `onTagsChange` prop closes that loop: editing a book's tags here now propagates back up to the shared list, resolving the exact stale-data problem this chapter would otherwise have — going back to the list after a tag edit and seeing the book still showing its old tags, because two separate copies of the same document had drifted apart.

NO EXTRA FETCH NEEDED

The detail view never calls `GET /api/items/:id` at all — it just looks up the matching item already sitting in the parent's own `items` array. That's only safe because Chapter 3's list route uses

`.lean()`, which returns every field a document actually has, not just the ones on `Item`'s own base schema. Without that fix, the list's own copy of a Book would be missing `author` and `tags` entirely, and this shortcut wouldn't work.

Wiring State in App.jsx

One shared `items` array, lifted into `App`, is what lets every other component — the form, the tag filter, the search bar, the detail view — affect the exact same list rather than each holding its own disconnected copy:

```
// src/App.jsx
import { useState, useEffect } from 'react';
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 [loading, setLoading] = useState(true);
  const [selectedId, setSelectedId] = useState(null);

  useEffect(() => {
    fetch('http://localhost:4000/api/items')
      .then((res) => res.json())
      .then((data) => {
        setItems(data);
        setLoading(false);
      });
  }, []);

  function handleCreated(newItem) {
    setItems((prev) => [...prev, newItem]);
  }

  function handleTagsChange(itemId, newTags) {
    setItems((prev) =>
      prev.map((it) => (it._id === itemId ? { ...it, tags: newTags } : it))
    );
  }
}
```

```

    );
  }

  const selectedItem = items.find((it) => it._id === selectedId);

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

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

      {loading ? (
        <p>Loading catalogue...</p>
      ) : selectedItem ? (
        <ItemDetail
          item={selectedItem}
          onBack={() => setSelectedId(null)}
          onTagsChange={(tags) => handleTagsChange(selectedItem._id, tags)}
        />
      ) : (
        <ItemList items={items} onSelect={setSelectedId} />
      )}
    </div>
  );
}

```

Search and tag filtering both call `setItems` directly, replacing the whole displayed list with whatever the server just returned — exactly matching how Chapter 5's `TagFilter` and Chapter 6's `SearchBar` were already built, with no changes needed to either. Creating a new item appends to the existing array instead of replacing it, since a fresh item shouldn't make an active search or tag filter disappear.

TWO SEPARATE FILTER CONTROLS, DELIBERATELY

SearchBar and TagFilter stay two distinct controls rather than being merged into one. They call different routes for different reasons — SearchBar's own regex match against title/author/artist/director from Chapter 6, TagFilter's own exact tag match from Chapter 5 — and combining them into a single input would either lose that distinction or need real logic to decide which endpoint a given term should hit. Two simple controls are more honest here than one clever one.

Trying It in the Browser

With both servers running, add one item of each type through the form and watch the list itself: every entry shows its own correct creator with no stray "By undefined" text anywhere, and clicking any item opens its own detail view with exactly the fields that type actually has. Open a Book, add a tag through the real `TagEditor`, then click "Back to list" — the book's own list entry doesn't show tags directly, but reopening its detail view immediately reflects the change, confirming the shared `items` array genuinely updated rather than the edit only existing inside the now-closed detail view.

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 revert `ItemList`'s summary line to the naive template-literal version (``By ${item.author}``) and confirm the real "By undefined" text appears for a Cd, Dvd, or Bluray. Restore the `getCreator()` version and confirm the fix.

 [View solution](#)

EXERCISE 3

Temporarily remove the `onTagsChange` wiring between `ItemDetail` and `App` (pass no `onChange` to `TagEditor` at all), then edit a book's tags, go back to the list, and reopen it — confirm the stale-tags bug reappears. Restore the wiring and confirm the fix.

 [View solution](#)

CHAPTER 7 QUICK REFERENCE

- **`getCreator(item)`** — returns whichever of `author`/`artist`/`director` exists, or a real null, driving genuinely conditional rendering rather than a coerced "undefined" string
- **Template literals vs. JSX children** — string interpolation coerces a missing value into the literal text "undefined"; a plain JSX child silently renders nothing instead, but still needs a real presence check for a correct result
- **`FIELD_CONFIG` reused** — the exact same object from Chapter 4 drives both which inputs the form shows and which fields the detail view displays
- **No extra fetch on detail** — a direct, explained payoff of Chapter 3's own `.lean()` fix; every field is already present in the shared list
- **`TagEditor`'s `onChange`, finally wired** — closes the loop Chapter 5 left open, keeping the list and detail view's own copies of tags in sync
- **Two filter controls, not one** — `SearchBar` and `TagFilter` stay separate since they genuinely query different things
- **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, Express & MongoDB

Chapter 8 · Fast Manual Entry

Chapter 1's own real deadline shaped a lot of quiet decisions already — a single Express API, Mongoose validation over hand-rolled checks, a deliberately plain form. This chapter cashes that framing out directly: three real points of friction in the add-item flow, one of them hiding a genuine network race condition, all found and fixed with the app exactly as it already stands after Chapter 7.

Blocking Alerts Break a Fast, Several-in-a-Row Workflow

Chapter 4's own error handling calls `alert(error)` on a failed submission — a real, native browser dialog that has to be dismissed with a click before anything else on the page can be touched again. Cataloguing several items in a row, one validation mistake stops the whole flow cold until that dialog is cleared.

```
// AddItemForm.jsx — replacing the blocking alert with inline state
const [error, setError] = useState(null);

async function handleSubmit(e) {
  e.preventDefault();
  setError(null);

  // ...payload construction unchanged from Chapter 4...

  const res = await fetch('http://localhost:4000/api/items', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(payload),
  });

  if (!res.ok) {
    const { error: message } = await res.json();
    setError(message);
    return;
  }
}
```

```

    }

    // ...success handling unchanged...
  }

```

Rendered near the submit button, `{error && <p className="form-error">{error}</p>}` shows exactly the same message the dialog used to, without ever stopping the rest of the page from responding. A validation error can be read and fixed in place, in the same motion as correcting the field that caused it.

Returning Focus to the Title Field After a Successful Add

Chapter 4 already resets `shared` and `typeFields` after a successful submission, and — worth noting explicitly, since it's easy to miss — `itemType` is deliberately never reset, so adding five books in a row never requires reselecting "Book" from the dropdown each time. The one piece still missing is the cursor itself: after a successful add, focus is left wherever it happened to be, usually on the submit button, 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 response and state reset:
setShared({ title: '', releaseYear: '', notes: '' });
setTypeFields({});
titleInputRef.current?.focus();

```

The title input itself gets `ref={titleInputRef}`. After each successful submission, the cursor lands right back in the one field every new item genuinely needs, letting a whole shelf of books get typed in with barely a pause between them.

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 against Chapter 6's own search route, running as the title is typed, catches this without slowing anything down — a soft warning, not a hard block, since owning two real copies of the same title is a genuinely valid thing to catalogue.

A Naive Live Search — And a Real Race Condition

A first, reasonable-looking version fires a search on every keystroke, no debouncing, no ordering guarantee:

```
// A naive duplicate check – fires immediately on every keystroke
function useDuplicateCheckNaive(title) {
  const [match, setMatch] = useState(null);

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

    fetch(`http://localhost:4000/api/items/search?q=${encodeURIComponent(trimmed)}`)
      .then((res) => res.json())
      .then((results) => {
        const exact = results.find((r) => r.title.toLowerCase() === trimmed.toLowerCase());
        setMatch(exact || null);
      });
  }, [title]);

  return match;
}
```

Suppose the catalogue already has a real Book titled exactly `"Dune"`, and someone is typing a genuinely different, new title: `"Dune Messiah"`. Each keystroke fires its own independent request, and nothing about `fetch()` guarantees those requests resolve in the order they were sent — real network round trips vary in timing for reasons entirely outside this code's control.

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

The final, visible state is wrong — not because of any real duplicate, but purely because an earlier request's response happened to arrive after a later one's.

The Fix: Debouncing and Ignoring Stale Responses

Two changes, doing two genuinely different jobs. A short delay before firing a request means most in-progress typing never triggers a request at all — only a real pause does. A small ref, checked before ever applying a response, is what actually guarantees correctness regardless of how those requests happen to resolve:

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

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 res = await fetch(`http://localhost:4000/api/items/search?q=${encodeURIComponent(
  const results = await res.json();

  // Ignore this response if the title has moved on since this request was sent
  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 request fires until typing actually pauses, which on its own already makes the specific race far less likely to occur at all. But the real guarantee is the `latestQuery` check — when the stale "Dune" response does eventually arrive, its own captured `trimmed` value ("Dune") no longer matches `latestQuery.current` (which by then holds "Dune Messiah"), so the response is discarded before it can ever reach `setMatch()`.

DEBOUNCING ALONE ISN'T ENOUGH

Debouncing reduces *how often* two competing requests can even exist at once — it doesn't guarantee which one's response arrives first when they do. A user who pauses briefly, then types quickly again, can still fire two debounced requests close enough together to race. The `latestQuery` ref is what makes the result correct regardless of timing, not the delay by itself.

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 — it's purely informational, exactly matching the real decision that owning two copies of the same title is a genuine, valid thing to catalogue, not an error to prevent.

What This Chapter Deliberately Didn't Add

No bulk-import or paste-many-titles-at-once feature — a genuinely bigger piece of scope than "reducing friction in the existing flow" calls for, and not what this chapter's own outline promises. No extra success toast either: since the form and the list render on the same screen together (per Chapter 7's own layout), a newly added item already appears immediately below the form the moment it's created — real, visible confirmation that doesn't need a second, separate message layered on top of it.

Trying It in the Browser

With a real Book titled "Dune" already in the catalogue, start typing "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 — the debounce and staleness guard together mean the warning never gets stuck showing a false match. Submitting a deliberately invalid item (an empty required field) now shows the error inline, with the rest of the form still fully usable while it's visible.

Hands-On Exercises

EXERCISE 1

Replace the `alert()`-based error handling with inline error state, and add the `titleInputRef` focus behavior. Submit an invalid Book (missing author) and confirm the error appears inline with the form still usable, then add several valid books in a row without touching the mouse between submissions.

 [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.

 [View solution](#)

CHAPTER 8 QUICK REFERENCE

- **Inline errors, not `alert()`** — a failed submission shows its message in place, without blocking the rest of the form
- **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 `/search` route for a new purpose, as a soft, non-blocking warning

- **The race condition** — independent fetch requests can resolve out of order; an earlier, now-stale response can overwrite a later, correct one
- **Debounce + latestQuery guard** — debounce reduces how often competing requests exist; the ref check is what actually guarantees a stale response is ignored
- **Deliberately not added** — bulk import, an extra success toast (the list already provides that feedback)
- **Next chapter:** Deployment

CHAPTER 9: DEPLOYMENT

Personal Catalogue: React, Express & MongoDB

Chapter 9 · Deployment

This chapter gets the whole stack running on its own, 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 surfaces one real, pervasive problem that's been quietly sitting in every chapter since Chapter 4.

A Real Problem Hiding in Plain Sight: Hardcoded localhost URLs

Every `fetch()` call written since the add-item form first went in points directly at `http://localhost:4000` — a literal string, typed the same way in six different places across five chapters. It has worked perfectly the entire time, for the same reason it's about to stop working the moment this app is deployed anywhere else: development has always run on `localhost`.

FILE	INTRODUCED IN	WHAT HARDCODES THE URL
AddItemForm.jsx	Ch.4, Ch.8	The POST request, and the duplicate-check hook it calls
TagEditor.jsx	Ch.5	Both PATCH requests (add and remove)
TagFilter.jsx	Ch.5	The GET request
SearchBar.jsx	Ch.6	The GET request
App.jsx	Ch.7	The initial catalogue-loading fetch
useDuplicateCheck.js	Ch.8	The debounced search fetch

A Single Source of Truth for the API's Own Base URL

Vite exposes environment variables to client-side code, but only ones explicitly prefixed `VITE_` — a deliberate, documented security choice, so a variable never accidentally ends up bundled into a public build just because it happened to share a name with something server-side.

```
// src/api.js
export const API_BASE_URL = import.meta.env.VITE_API_URL ?? '';
```

```
# catalogue-web/.env.development — loaded automatically by `npm run dev`
VITE_API_URL=http://localhost:4000
```

No `.env.production` file is needed at all. With nothing defining `VITE_API_URL` in a production build, `API_BASE_URL` falls back to an empty string — and a fetch call built from an empty base, like `${API_BASE_URL}/api/items`, becomes the plain relative path `/api/items`. A relative URL like that resolves against whatever origin the page itself was actually loaded from, whatever real domain that turns out to be — with no domain name ever hardcoded into the built bundle.

Updating Every Fetch Call

Two representative examples — the same one-line change repeated across all six files from the table above:

```
// SearchBar.jsx — before
const url = q
  ? `http://localhost:4000/api/items/search?q=${encodeURIComponent(q)}`
  : 'http://localhost:4000/api/items';
```

```
// SearchBar.jsx — after
import { API_BASE_URL } from '../api';
// ...
const url = q
  ? `${API_BASE_URL}/api/items/search?q=${encodeURIComponent(q)}`
  : `${API_BASE_URL}/api/items`;
```

```
// TagEditor.jsx — before
const res = await fetch(`http://localhost:4000/api/items/${bookId}/tags/add`, ...);

// TagEditor.jsx — after
import { API_BASE_URL } from '../api';
```

```
// ...
const res = await fetch(`${API_BASE_URL}/api/items/${bookId}/tags/add`, ...);
```

The identical substitution applies to AddItemForm's own submit URL and its duplicate-check hook, TagFilter's GET request, App's initial load, and TagEditor's own remove-tag request — every one of the six locations in the table above, all pointing at the same single `API_BASE_URL` constant instead of six separate literal strings.

Building the Frontend for Production

```
# from catalogue-web/
npm run build
```

Vite's default build writes a real, deployable set of static files into `catalogue-web/dist/` — an `index.html`, and hashed, minified JavaScript and CSS bundles. Nothing in this project needs to serve those files with a second Node process; Express already runs one.

Serving the Built Frontend From Express

```
// server.js – added after the API routes and error-handling middleware are set up
const path = require('path');

app.use(express.static(path.join(__dirname, '../catalogue-web/dist')));
```

`express.static()` only intercepts a request that matches a real file it's actually serving — a request for `/api/items` never matches anything inside `dist/`, so it falls straight through to Chapter 3's own routes regardless of where this line sits relative to them. Unlike Chapter 6's own `/search`-versus-`/:id` ordering bug, there's genuinely nothing to get wrong here about registration order.

NO SPA-FALLBACK ROUTE NEEDED — A DIRECT PAYOFF OF CHAPTER 7'S OWN DESIGN

Most React deployment guides need one more piece here: a catch-all route (`app.get('*', ...)`) that serves `index.html` for any path that isn't a real static file, so refreshing the browser on a client-

side route like `/books/123` doesn't 404. This app has never had a client-side route at all — Chapter 7's own list/detail toggle is pure component state (`selectedId`), never a URL change. A single request for `/` already resolves to `index.html` automatically, since that's `express.static()`'s own default behavior for a directory request — no fallback route to add, because there's no second real page for the browser to ever land on directly.

Tightening CORS for Production

Chapter 1's `app.use(cors())` allows a request from any origin at all. Once the frontend is served by the very same Express process, its own requests to `/api/items` are same-origin already and don't strictly need CORS to succeed — but restricting it explicitly is cheap, real defense in depth, and keeps this server honest about which origins should be allowed to call it directly:

```
// server.js
app.use(cors({
  origin: process.env.ALLOWED_ORIGIN || 'http://localhost:5173',
}));
```

SET ALLOWED_ORIGIN ONCE A REAL DOMAIN EXISTS

In production, `ALLOWED_ORIGIN` should be set to the real deployed domain (`https://yourdomain.com`), not left on the dev-server default. Chapter 10's own integration into the existing Astro site may end up mounting this app at a path under that site's own domain rather than a separate one entirely — whichever arrangement it turns out to be, this is the one setting that needs updating to match.

Keeping the Server Running: PM2

A real production server shouldn't stop because a terminal window closed, and should restart itself if the process ever crashes. PM2 is the standard, widely used process manager for exactly this:

```
# installed once, globally, on the server
npm install -g pm2
```

```
# start the app under PM2's own supervision
pm2 start server.js --name catalogue-api

# persist the process list across a real server reboot
pm2 save
pm2 startup
```

`pm2 logs catalogue-api` tails real, live output from the running process; `pm2 restart catalogue-api` picks up a new deployment without a full stop-and-start cycle.

A Reverse Proxy for a Real Domain and HTTPS

Node's own process doesn't need to be exposed to the internet directly. A real nginx reverse proxy in front of it handles the public-facing domain and — following this site's own dedicated **HTTPS/TLS Fundamentals** course for the actual certificate setup — TLS termination:

```
# /etc/nginx/sites-available/catalogue
server {
    listen 80;
    server_name yourdomain.com;

    location / {
        proxy_pass http://localhost:4000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

A Least-Privilege MongoDB User for Production

Chapter 1's own local development connection has no reason to run with admin-level database access in production. A dedicated user, scoped to exactly the one database this app actually touches:

```
// run once, connected as an admin user
use catalogue
db.createUser({
  user: "catalogue_app",
  pwd: "REPLACE_WITH_A_REAL_GENERATED_PASSWORD",
  roles: [{ role: "readWrite", db: "catalogue" }],
})
```

The production `.env`'s own `MONGODB_URI` is updated to authenticate as `catalogue_app`, not whatever admin account was used to set the database up in the first place — the same real-world consequence as the PHP and Django siblings' own least-privilege database users, applied here to MongoDB's own role system instead of GRANT statements.

Scheduled Backups With mongodump

```
# a real, working backup command
mongodump --uri="$MONGODB_URI" --out=/backups/catalogue-$(date +%F)
```

Placed in a daily cron job, this keeps a dated snapshot of the entire catalogue on disk, independent of whatever hosting provider's own backup story does or doesn't cover.

Trying It

```
# build the frontend
cd catalogue-web && npm run build && cd ..

# start the backend, now also serving the built frontend
cd catalogue-api && node server.js
```

Visiting `http://localhost:4000/` alone — with no separate Vite dev server running at all — should load the real, working catalogue app, and every action (search, add, tag edit) should succeed exactly as it did throughout development, now reaching the API through a relative path resolved against that same origin.

Hands-On Exercises

EXERCISE 1

Create `src/api.js` and `.env.development`, then update all six fetch call sites listed in this chapter's own table to use `API_BASE_URL`. Confirm the dev app (`npm run dev + nodemon server.js`) still works exactly as before.

[!\[\]\(3e5c8028dfbd33327033568c6df503f9_img.jpg\) View solution](#)

EXERCISE 2

Build the frontend (`npm run build`), wire `express.static` into `server.js`, and confirm the whole app is reachable and fully functional at `http://localhost:4000/` alone, with no separate Vite dev server running, and with a real API call succeeding via a relative path.

[!\[\]\(eda778e550f84ec809f42c07f6480eb4_img.jpg\) View solution](#)

EXERCISE 3

Create the least-privileged `catalogue_app` MongoDB user, update `MONGODB_URI` to use it, confirm the app still works normally, then confirm directly (e.g. attempting `db.dropDatabase()` while authenticated as `catalogue_app`) that the restricted user genuinely cannot perform an admin-only operation.

[!\[\]\(98145f27b204436a115f2c694146e288_img.jpg\) View solution](#)

CHAPTER 9 QUICK REFERENCE

- **API_BASE_URL** — one Vite-driven constant replacing six separate hardcoded `localhost:4000` URLs across Chapters 4-8
- **VITE_prefix** — required for any env variable to reach client-side code, a deliberate Vite security choice
- **Empty production fallback** — no `VITE_API_URL` in production means relative paths, resolved against whatever real domain the app is actually deployed under

- **express.static + no fallback route** — a direct payoff of Chapter 7's own router-free design; most SPAs need a catch-all route, this one genuinely doesn't
- **CORS tightened** — restricted to a real allowed origin, largely redundant once unified but cheap, real defense in depth
- **PM2 / nginx / least-privilege MongoDB user / mongodump backups** — the standard production hardening layer, applied to this stack specifically
- **Next chapter:** Capstone — integrating the catalogue into the existing Astro site, and planning barcode scanning as a later phase

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

Personal Catalogue: React, Express & MongoDB

Chapter 10 · Capstone

Chapter 9 got the whole stack running as a real, standalone deployment. This closing chapter does what Chapter 1 always meant it to do: put it somewhere the user actually visits — a real path under their own existing Astro-based site — and lay out a genuine, concrete plan for the one feature deliberately deferred since Chapter 1, barcode scanning.

Three Real Ways to Mount This Behind an Existing Site

OPTION	WHAT IT INVOLVES	TRADEOFF
Subdomain (<code>catalogue.existing-site.com</code>)	A separate DNS record and its own nginx server block pointing straight at the unified Express app from Chapter 9	Simplest possible nginx config; 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 the same Express app	Feels like part of the same site; needs two real, verified fixes this chapter builds — covered below
JSON-API + Astro-frontend rewrite	Keep only Chapters 1-6's backend; rebuild the UI as new Astro pages/components consuming it 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 the one built out in full here — genuinely integrated, without discarding any of the nine chapters of real React frontend already built.

A Real Problem: The Built Assets Don't Know They're Not at the Root

Chapter 9's own `npm run build` produces an `index.html` referencing its own JS and CSS bundles with paths like `/assets/index-abc123.js` — absolute from whatever domain root the app is served under. That's correct for Chapter 9's own standalone deployment, and genuinely

wrong the moment this app is mounted at `/catalogue/` on someone else's domain: a request for `/assets/index-abc123.js` resolves against the domain's own root, not the real location the file actually lives at, `/catalogue/assets/index-abc123.js`.

VERIFIED AGAINST VITE'S OWN DOCUMENTATION

Vite's `base` config option exists specifically for this. Left unset, every generated asset reference assumes a root deployment; setting it rewrites every one of those references — "JS-imported asset URLs, CSS `url()` references, and asset references in your `.html` files," per Vite's own documentation — to include the real sub-path prefix.

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

Rebuilding with this in place produces an `index.html` referencing `/catalogue/assets/index-abc123.js` instead — correct for exactly where the file is actually going to be requested from once this specific deployment goes live.

A Second Real Problem: API Calls Are Root-Relative, Not Site-Relative

Chapter 9's own `API_BASE_URL` falls back to an empty string in production, so every fetch call becomes a path like `/api/items` — correct for a standalone deployment at the domain root, but still wrong here for a genuinely different reason than the assets above: `/api/items` resolves against the domain's own root regardless of which page made the request, so a request made from `existing-site.com/catalogue/` would still be sent to `existing-site.com/api/items` — not `existing-site.com/catalogue/api/items`, where this app's own API actually needs to be reached once mounted at a sub-path.

The fix reuses Chapter 9's own mechanism, this time with a real value instead of an empty fallback:

```
# catalogue-web/.env.production — new for this specific deployment
VITE_API_URL=/catalogue
```

Every fetch call built since Chapter 4 already has the shape `${API_BASE_URL}/api/items` — with `API_BASE_URL` now resolving to `/catalogue`, that becomes `/catalogue/api/items`, matching exactly where the reverse proxy below is about to make this app's own API actually reachable.

TWO GENUINELY DIFFERENT FIXES FOR TWO GENUINELY DIFFERENT THINGS

`base` is a build-time setting controlling where the compiled JavaScript and CSS *files* are referenced from. `VITE_API_URL` is a runtime value controlling where this app's own *fetch calls* are sent. Both happen to need the identical `/catalogue` prefix for this specific deployment, but fixing one without the other leaves either a broken page (missing assets) or a broken app (failing API calls) — not both problems solved by a single change.

Building Out the Reverse Proxy

One location block on the existing site's own nginx config, added alongside whatever configuration already serves the rest of that site:

```
# /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 http://localhost:4000/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

The trailing slash on both `location /catalogue/` and `proxy_pass http://localhost:4000/;` is what makes nginx strip the `/catalogue` prefix before forwarding — a request for `/catalogue/api/items` reaches Express as a request for exactly `/api/items`, the same route Chapter 3 already defined with no changes needed on the Express side at all. The same stripping applies to a request for `/catalogue/assets/index-abc123.js`, which reaches Express

as `/assets/index-abc123.js` — exactly what Chapter 9's own `express.static()` is already serving.

NO NEW JSON ENDPOINT NEEDED HERE

The PHP and Django siblings' own capstones each had to add a small, new JSON API endpoint as a first step toward deeper integration, since their own apps were server-rendered pages without one. This course never needed that step — every route has returned real JSON since Chapter 3, and the React frontend has consumed it as an API from Chapter 4 onward. There's nothing left to add on that front; only the two path-prefix fixes above were actually missing.

With everything mounted this way, the browser only ever sees one domain — Chapter 9's own `ALLOWED_ORIGIN` CORS setting can simply be set to `https://existing-site.com`, and every request genuinely is same-origin now, exactly the scenario that chapter's own warn-box anticipated without yet knowing which real shape it would take.

A Real, Concrete Plan for Barcode Scanning

Chapter 1 deferred barcode scanning explicitly rather than pretending it wasn't wanted. With the app actually live, the real plan for adding it later:

- **Camera access** — a real, actively maintained library like `html5-qrcode` handles reading a barcode from the device camera directly in the browser, without waiting on the native `BarcodeDetector` API's own still-inconsistent cross-browser support.
- **Book lookup** — a scanned ISBN can be sent to Open Library's real, free, no-API-key-required endpoint, `https://openlibrary.org/isbn/<isbn>.json`, to auto-fill title, author, and publisher — reducing Chapter 8's own fast manual entry down to "scan, confirm, save" for the one item type with a clean, verified free lookup source.
- **Cd/Dvd/Bluray lookup** — honestly still an open question. Unlike ISBNs for books, there's no single, universally reliable free UPC-lookup service this course has actually verified for physical media — real research into a genuinely dependable option is the concrete next step here, not something to guess at.
- **Where the lookup call should live** — a new Express route (something like `GET /api/lookup/isbn/:isbn`), matching this project's own established pattern of keeping external API calls server-side rather than scattering direct third-party requests across React

components, the same architectural choice Chapters 3 and 6 already made for this app's own data.

Chapter Attribution

PIECE OF THE FINISHED APP	BUILT IN
Heterogeneous item schema via Mongoose discriminators	Chapter 2
CRUD API with dynamic dispatch to the right model	Chapter 3
Add-item form, driven by FIELD_CONFIG	Chapter 4
Tag editing via atomic \$addToSet/\$pull	Chapter 5
Cross-type search, escaped and correctly ordered	Chapter 6
List and detail views, reusing FIELD_CONFIG for display	Chapter 7
Friction reduction: inline errors, focus, live duplicate check	Chapter 8
Unified single-process deployment, API_BASE_URL, PM2, least-privilege MongoDB user	Chapter 9
Path-mounted reverse proxy, Vite base, sub-path API_BASE_URL, barcode-scanning plan	Chapter 10 (this chapter)

Hands-On Exercises

EXERCISE 1

Set base: '/catalogue/' in vite.config.js and VITE_API_URL=/catalogue in .env.production, rebuild the frontend, and confirm the built index.html's own asset references and every compiled fetch call both correctly include the /catalogue prefix.

 [View solution](#)

EXERCISE 2

Configure the nginx location `/catalogue/` block against a real running instance of the app (a second local nginx config is fine for testing), and confirm both a page load and a real API request succeed through the proxy, with Express's own `server.js` completely unchanged from Chapter 9.

[View solution](#)

EXERCISE 3

Set only `base` in `vite.config.js` without also setting `VITE_API_URL` (leaving Chapter 9's own empty-string fallback in place), rebuild, and confirm the page itself loads correctly while every API call still fails — demonstrating why the chapter's own warn-box treats these as two separate fixes, not one.

[View solution](#)

CHAPTER 10 QUICK REFERENCE — COURSE COMPLETE

- **Three mounting options** — subdomain, path-mounted reverse proxy (built here), or a full Astro-frontend rewrite
- **`vite.config.js`'s `base`** — a build-time fix for asset references, verified against Vite's own documentation
- **`VITE_API_URL=/catalogue`** — a runtime fix for fetch calls, reusing Chapter 9's own `API_BASE_URL` mechanism with a real value instead of an empty fallback
- **nginx's trailing-slash prefix-stripping** — lets Express's own routes stay completely unchanged from Chapter 9
- **No new JSON endpoint needed** — unlike the PHP/Django siblings, this app has been API-first since Chapter 3
- **Barcode scanning, planned honestly** — a real library and a real, verified free ISBN lookup for books; UPC lookup for the other three types named as real, unresolved future research
- **This closes the course** — 10/10 chapters, matching the shared four-course Personal Catalogue spec from Chapter 1