DATABASES

MongoDB Fundamentals

The Document Model & CRUD · Schema Design

Data Types & BSON · Indexes

Working with MongoDB from Node.js

Philip Osztromok

Generated with Claude

Table of Contents

MongoDB Fundamentals — 8 Chapters

🍃 What is MongoDB? Document Databases vs Relational

This course is framed throughout as the NoSQL counterpart to the site's existing MySQL courses — if you've been through those, you already know tables, rows, and normalization; this chapter builds directly on that, contrasting MongoDB's document model against what you already know rather than starting from zero.

The Document Model

Where a relational database like MySQL stores data as **rows** in **tables** with a fixed set of columns, MongoDB stores data as **documents** — JSON-like structures — grouped into **collections**. The key structural difference: documents in the same collection don't need to share an identical set of fields the way every row in a MySQL table shares the same columns.

Relational Terms vs. MongoDB Terms

MySQL (Relational)

`Table` → `Row` → `Column`, with a `Primary Key` uniquely identifying each row.

MongoDB (Document)

`Collection` → `Document` → `Field`, with an `_id` field uniquely identifying each document.

The Same User, Two Ways

```
// MongoDB: one document, nested data included directly
{
  "_id": ObjectId("64f1a2..."),
  "name": "Alice",
  "email": "alice@example.com",
  "address": { "city": "London", "postcode": "E1 6AN" }
}
```

In MySQL, that same information would typically be normalized across two tables — a `users` table and a separate `addresses` table joined by a foreign key (per `mysql_advanced_01`'s normalization chapter). MongoDB often just nests the address directly inside the user document instead.

BSON: Binary JSON

MongoDB doesn't actually store plain-text JSON — it stores **BSON** (Binary JSON), a binary-encoded format that's more efficient to parse and traverse than text, and supports a few extra types plain JSON doesn't have natively, like real `Date` objects and MongoDB's own `ObjectId` type (seen in the `_id` field above). You interact with it AS IF it were JSON — the binary encoding is an internal storage detail, not something you typically handle directly.

Schema Flexibility: A Blessing and a Trade-off

Because documents in a collection don't need identical fields, adding a new field to some documents doesn't require anything like a MySQL `ALTER TABLE` migration — you just start including it in new documents. That's a genuine speed advantage during early, fast-moving development. The trade-off: without something enforcing structure, nothing stops two documents in the same collection from representing the same kind of thing in inconsistent, incompatible ways — a problem a fixed relational schema prevents by construction. Chapter 6 covers `$jsonSchema` validation, MongoDB's opt-in answer to this trade-off.

When MongoDB Genuinely Fits vs. When MySQL Is Still Right

MongoDB Tends to Fit

Rapidly evolving data shapes, deeply nested/hierarchical data that's naturally read and written as one unit, high write-throughput workloads that don't need complex cross-entity joins.

MySQL Still Wins

Strong relational integrity (foreign key constraints), complex multi-table joins and transactional consistency across many entities, stable well-understood schemas that genuinely benefit from enforced structure.

Relational	MongoDB
Table	Collection
Row	Document
Column	Field
Primary Key	`_id`
JOIN	Embedding, or `$lookup` (Chapter 2's advanced-course counterpart)
Schema enforced by the engine	Schema optional by default, opt-in via `$jsonSchema`

💻 Coding Challenges

Challenge 1: Translate the Terms

For each relational term, give its MongoDB equivalent: (a) table, (b) row, (c) primary key, (d) column.

Goal: Practice the vocabulary mapping this chapter introduced before moving further into the course.

→ Solution

Challenge 2: Model a Blog Post as a Document

A relational schema has a `posts` table and a separate `comments` table (joined by a foreign key). Sketch how a single blog post, with its comments, might look as one MongoDB document instead.

Goal: Practice translating a normalized relational design into a nested document.

→ Solution

Challenge 3: MongoDB or MySQL?

Recommend MongoDB or MySQL for (a) a product catalog where each product category has wildly different attributes, and (b) a banking system tracking transfers between accounts that must always balance exactly. Justify each choice.

Goal: Practice applying this chapter's "when each fits" section to concrete scenarios.

→ Solution

⚠️ Gotcha: Schema Flexibility Doesn't Mean No Planning Needed

It's tempting to treat MongoDB's lack of an enforced schema as permission to skip designing your data's shape at all. In practice, a collection full of documents that represent "the same kind of thing" in wildly inconsistent ways (one document has `email`, another has `emailAddress`, a third nests it under `contact.email`) becomes just as painful to query reliably as a badly designed relational schema — the difference is MongoDB won't stop you from creating that mess in the first place. Schema flexibility means the database doesn't enforce consistency for you by default; it doesn't mean consistency stops mattering.

🎯 What's Next

The next chapter gets hands-on: **Installing MongoDB & mongosh** — getting a real MongoDB instance running, connecting with the `mongosh` shell, and a first look at MongoDB Compass.

⚙️ Installing MongoDB & mongosh

Chapter 1 covered the document model conceptually. This chapter gets a real MongoDB instance running locally, so every chapter from here on can be genuinely hands-on.

Installing MongoDB Community Server

MongoDB Community Server is free and installs on Windows, macOS, and Linux — on Debian/Ubuntu, it's available via `apt` once MongoDB's own package repository is added, the same general package-manager pattern already familiar from the Linux course's software-installation chapter (`linux_lesson_10`). If you've already been through the **Docker** course, running MongoDB in a container is an even quicker way to get started without a native install at all:

Running MongoDB in Docker

```
docker run -d --name mongo -p 27017:27017 mongo
```

This pulls the official MongoDB image and exposes it on its default port, `27017` — the same port a native install listens on.

mongosh: The MongoDB Shell

mongosh is MongoDB's modern interactive shell (the successor to an older, now-legacy `mongo` shell) — a JavaScript-based command line for running commands directly against a MongoDB instance. Launching it with no arguments connects to a MongoDB instance running on `localhost`, port `27017`, by default:

Launching mongosh and Running a First Command

```
$ mongosh
// Connecting to: mongodb://127.0.0.1:27017/
// Using MongoDB: 7.0.x

test> db.getName()
test
```

`test` is the default database mongosh starts you in — nothing special has to exist yet for the shell itself to connect.

Your First Commands

Command	What It Does
`show dbs`	List all databases that currently have data
`use <name>`	Switch to (or prepare to create) a database
`show collections`	List collections in the current database
`db.stats()`	Show storage statistics for the current database

MongoDB Compass: The GUI

Compass is MongoDB's official graphical client — browse databases, collections, and documents visually, build queries with a form-based interface, and inspect indexes, all without typing shell commands. It's entirely optional: everything Compass does, `mongosh` can also do — Compass is a convenience layer for visual exploration, not a separate capability.

mongosh vs. Compass

mongosh (CLI)

Scriptable, fast for repeated commands, works well over SSH to a remote server, the natural fit once you're writing real application code later in this course.

Compass (GUI)

Visual browsing of documents and schemas, easier for exploring unfamiliar data, handy for building and testing a query before writing it into code.

Connecting from a Connection String

Anything that connects to MongoDB — `mongosh`, Compass, or a driver in application code — ultimately uses a **connection string** in the form `mongodb://<host>:<port>/<database>`:

A Local Connection String

```
mongodb://localhost:27017/shop
```

(MongoDB Atlas, the managed cloud version, uses a `mongodb+srv://` variant instead — not covered in depth here, since this course focuses on a self-hosted instance, but worth knowing the prefix differs if you ever connect to a cloud-hosted cluster.) Chapter 8 uses this exact connection-string format to connect from real Node.js code.

💻 Coding Challenges

Challenge 1: Create and Confirm a Database

Write the mongosh commands to switch to a new database called `shop`, then confirm what you'd need to do for it to actually appear in `show dbs`.

Goal: Practice the basic mongosh workflow, including this chapter's tip-box gotcha about when a database actually gets created.

→ Solution

Challenge 2: Write a Connection String

Write the connection string for a MongoDB instance running on host `db.internal`, port `27018`, targeting a database named `inventory`.

Goal: Practice the connection-string format that every MongoDB client ultimately relies on.

→ Solution

Challenge 3: mongosh or Compass?

Recommend mongosh or Compass for (a) exploring an unfamiliar database's document shapes for the first time, and (b) writing a deploy script that seeds some initial data automatically. Justify each.

Goal: Practice matching the right tool to the task, not defaulting to one for everything.

→ Solution

⚠️ Gotcha: A New Database Doesn't Really Exist Until It Has Data

Running `use shop` in mongosh switches your session's context to a database called `shop` — but if `shop` doesn't exist yet, MongoDB doesn't actually create it at that moment. Run `show dbs` right after, and `shop` won't appear in the list at all. MongoDB only actually creates the database (and persists it) once you insert at least one document into a collection within it. Until then, `use shop` is really just telling the shell "point future commands at this name," not "create this database now" — a subtle difference from a MySQL `CREATE DATABASE` statement, which takes effect immediately.

🎯 What's Next

With a working instance and shell, the next chapter starts writing real data: **CRUD Basics: Inserting and Finding Documents** — `insertOne`/`insertMany`, `find`/`findOne`, and the query operators used to filter results.

📝 CRUD Basics: Inserting and Finding Documents

With a working instance and shell from Chapter 2, this chapter starts writing and reading real data — the Create and Read of CRUD. Chapter 4 covers Update and Delete.

Inserting Documents

`insertOne` adds a single document; `insertMany` adds several at once. If you don't supply your own `_id`, MongoDB generates one automatically.

insertOne

```
db.products.insertOne({ name: "Notebook", price: 3.50, category: "stationery" })

// { acknowledged: true, insertedId: ObjectId("64f3...") }
```

insertMany

```
db.products.insertMany([
 { name: "Laptop", price: 899, category: "electronics" },
 { name: "Novel", price: 12, category: "books" }
])

// { acknowledged: true, insertedIds: { '0': ObjectId("..."), '1': ObjectId("...") } }
```

Finding Documents: find and findOne

`find()` matches potentially *many* documents; `findOne()` returns just the first match, or `null` if nothing matches.

Basic Queries

```
db.products.find({})
// every document in the products collection

db.products.findOne({ name: "Notebook" })
// the single matching document, or null
```

Query Operators: Filtering Beyond Exact Match

A plain field/value pair means "equals" implicitly. MongoDB's query operators (always prefixed with `$`) express everything else:

Operator	Meaning
`$eq`	Equals (usually left implicit)
`$ne`	Not equal
`$gt` / `$gte`	Greater than / greater than or equal
`$lt` / `$lte`	Less than / less than or equal
`$in`	Matches any value in a given list
`$nin`	Matches none of the values in a given list

Combining Comparison Operators

```
db.products.find({ price: { $gt: 10, $lt: 50 } })
// products priced strictly between 10 and 50
```

Using $in

```
db.products.find({ category: { $in: ["electronics", "books"] } })
// products in EITHER category
```

Projection: Choosing Which Fields Come Back

A second argument to `find()`/`findOne()` controls which fields are returned — `1` to include, `0` to exclude (a document's `_id` is included by default unless explicitly excluded). This is conceptually similar to the client-specifies-what-it-wants idea the GraphQL course covers at the whole-API level — here it's just scoped to a single query:

A Projected Query

```
db.products.find({}, { name: 1, price: 1, _id: 0 })
// returns only name and price for every product, no _id
```

Combining Conditions: Implicit AND, and $or

Multiple fields inside one query object are implicitly ANDed together. For OR logic, use the explicit `$or` operator with an array of alternative conditions:

Implicit AND vs. Explicit $or

```
// implicit AND: category is "books" AND price is under 15
db.products.find({ category: "books", price: { $lt: 15 } })

// explicit $or: category is "books" OR price is under 15
db.products.find({ $or: [ { category: "books" }, { price: { $lt: 15 } } ] })
```

💻 Coding Challenges

Challenge 1: Insert Three Documents at Once

Write an `insertMany` call adding three "task" documents to a `tasks` collection, each with a `title` and a `done` boolean field.

Goal: Practice the insertMany syntax with a realistic multi-document array.

→ Solution

Challenge 2: Query with Comparison Operators

Write a `find` query on a `products` collection returning products priced 20 or more, in either the `"electronics"` or `"home"` category.

Goal: Practice combining `$gte` and `$in` in one query.

→ Solution

Challenge 3: Write a Projected Query

Write a query on a `users` collection that returns only each user's `email` field (no `_id`, no other fields).

Goal: Practice projection syntax for both inclusion and explicit `_id` exclusion.

→ Solution

⚠️ Gotcha: find() Returns a Cursor, Not an Array

In the interactive `mongosh` shell, running `db.products.find({})` appears to just print a list of documents — which makes it easy to assume `find()` hands back a plain array. It doesn't: it returns a **cursor**, an object that lazily fetches matching documents in batches as you iterate it. The shell happens to auto-iterate and print the first batch for convenience, quietly hiding this detail. It matters once you start writing real driver code (Chapter 8), where you'll typically need to explicitly iterate the cursor or call something like `.toArray()` to get an actual array of documents — code that treats a raw `find()` result as if it were already an array will not behave the way the shell's output made it look.

🎯 What's Next

The next chapter covers the other half of CRUD: **Updating and Deleting Documents** — `updateOne`/`updateMany`, update operators like `$set`/`$inc`/`$push`, `deleteOne`/`deleteMany`, and upserts.

✏️ Updating and Deleting Documents

Chapter 3 covered Create and Read. This chapter finishes off CRUD with Update and Delete — and one behavior worth knowing about carefully before you hit it by accident.

updateOne and updateMany

Both take a **filter** (which document(s) to change) and an **update** (what to change). `updateOne` affects only the first match; `updateMany` affects every matching document.

Update Operators: $set, $inc, $push

Operator	Meaning
`$set`	Set a field to a specific value (the most common update operator)
`$inc`	Increment (or decrement, with a negative number) a numeric field
`$push`	Append a value onto an array field

$set — Change a Field

```
db.tasks.updateOne(
  { title: "Write chapter draft" },
  { $set: { done: true } }
)
```

$inc — Adjust a Counter

```
db.products.updateOne(
  { name: "Notebook" },
  { $inc: { stock: -1 } }
)
// decrements stock by 1 — one sold
```

$push — Append to an Array

```
db.posts.updateOne(
  { title: "Why I Switched to MongoDB" },
  { $push: { comments: { author: "Dana", text: "Nice writeup!" } } }
)
// adds one new comment to the embedded comments array from Chapter 1's challenge
```

Upserts: Update or Insert

Adding `{ upsert: true }` as a third argument tells MongoDB: if no document matches the filter, *insert* a new one instead of doing nothing.

An Upsert for a Page-View Counter

```
db.pageViews.updateOne(
  { page: "/pricing" },
  { $inc: { views: 1 } },
  { upsert: true }
)
// first time this page is viewed: creates { page: "/pricing", views: 1 }
// every time after: increments the existing views field
```

Deleting Documents: deleteOne and deleteMany

Deleting

```
db.tasks.deleteOne({ title: "Write chapter draft" })
// removes the first matching document

db.tasks.deleteMany({ done: true })
// removes every completed task
```

💻 Coding Challenges

Challenge 1: Update a Single Field

Write an `updateOne` call that sets a product named `"Laptop"`'s `price` field to `799`.

Goal: Practice basic `$set` syntax.

→ Solution

Challenge 2: Append to an Array

Write an `updateOne` call adding a new tag `"sale"` to a product's `tags` array field, for the product named `"Notebook"`.

Goal: Practice `$push` on an array field.

→ Solution

Challenge 3: Write an Upsert

Write an `updateOne` call that increments a visitor counter for a user identified by `userId: "u42"`, creating the counter document with `{ userId: "u42", visits: 1 }` if it doesn't exist yet.

Goal: Practice combining `$inc` with `upsert: true`.

→ Solution

⚠️ Gotcha: Forgetting $set Silently Replaces the Whole Document

If the second argument to `updateOne` has **no operator** like `$set` at all — just plain fields, e.g. `db.users.updateOne({ name: "Alice" }, { email: "new@example.com" })` — MongoDB doesn't merge that field into the existing document. It treats the whole thing as a **replacement** document, wiping out every other field the original document had. This is exactly the REST PUT-vs-PATCH distinction from the API Types & Design course's HTTP chapter, applied to MongoDB: a bare update document behaves like a full `PUT` replacement, while wrapping it in `$set` behaves like a `PATCH` partial update. Always use `$set` (or another update operator) unless you genuinely intend to replace the entire document.

🎯 What's Next

The next chapter goes deeper into design itself: **The Document Data Model & Schema Design** — embedding vs. referencing, denormalization philosophy, and modeling one-to-many and many-to-many relationships as documents.

🧩 The Document Data Model & Schema Design

Chapters 3–4 covered CRUD mechanically. This chapter covers the decision that matters most in MongoDB: how to actually shape your documents — contrasted throughout with the normalization-first approach the MySQL course's `mysql_advanced_01` chapter builds around.

Embedding vs. Referencing: The Core Decision

Every relationship between two pieces of data can be represented one of two ways: **embedding** the related data directly inside the parent document (as Chapter 1's blog post + comments example did), or **referencing** it — storing just an `_id` pointing to a separate document in another collection, conceptually similar to a MySQL foreign key.

Embedding vs. Referencing

Embedding

Related data lives directly inside the parent document. One read gets everything — no second query needed — but the data can only grow so far before that becomes a liability.

Referencing

The parent stores just an id; the related data lives in its own collection, looked up separately. More like a relational foreign key, at the cost of a second query when you need both.

When to Embed

Embed when the related data is always read together with its parent, doesn't grow without bound, and conceptually belongs to one single "thing" — an address embedded in a user, or a handful of recent comments embedded in a post, exactly as Chapter 1 modeled it.

When to Reference

Reference when the related data is large or effectively unbounded, shared and reused across many different parent documents, or frequently queried independently of whatever "owns" it.

Referencing a Shared Category

```
// products collection — references a category, doesn't embed it
{ name: "Laptop", categoryId: ObjectId("64f9...") }

// categories collection — one shared document per category
{ _id: ObjectId("64f9..."), name: "Electronics", taxRate: 0.2 }
```

If "Electronics" instead got embedded directly into every product document, renaming the category or changing its tax rate would mean updating potentially thousands of product documents at once — referencing means updating exactly one document.

Denormalization: A Deliberate Choice, Not Sloppiness

The MySQL course's normalization chapter treats duplicated data as something to design away. MongoDB documents often duplicate data *on purpose*, trading some update complexity for read performance — for example, embedding a post's author **name** directly on the post, not just an `authorId` reference, so displaying a post never needs a second lookup just to show who wrote it. The trade-off is real: if that author later changes their display name, every post that embedded the old name needs updating (or the staleness has to be accepted temporarily) — precisely the "update anomaly" relational normalization exists to prevent. MongoDB doesn't eliminate that trade-off; it just makes it a deliberate, chapter-by-chapter design choice rather than something the database forces on you either way.

One-to-Many Relationships

Relationship Shape	Example	Recommended Pattern
One-to-few	A user with a few addresses	Embed as an array
One-to-many (bounded)	A post with a manageable number of comments	Embed, with an eye on growth
One-to-squillions	A device generating millions of log entries	Always reference — never embed

Many-to-Many Relationships

Take students and courses — a student takes many courses, and a course has many students. The most direct pattern: store an array of course `_id`s on the student document (or vice versa). For a genuinely many-to-many relationship with its own attributes (like an enrollment date or a grade), a separate **join collection** often works better — the exact document-model counterpart to a MySQL join table:

A Join Collection for Enrollments

```
// enrollments collection — one document per student-course pairing
{ studentId: ObjectId("..."), courseId: ObjectId("..."), enrolledAt: ISODate("2026-01-10") }
```

💻 Coding Challenges

Challenge 1: Embed or Reference?

For each, recommend embedding or referencing: (a) a user's shipping addresses (typically 1-3 per user), (b) a YouTube-style video's comments (potentially tens of thousands), (c) a "country" field shared by millions of user documents.

Goal: Practice applying the growth/sharing criteria this chapter introduced.

→ Solution

Challenge 2: Design a Many-to-Many Relationship

Design the document(s) for a many-to-many relationship between "authors" and "books" (an author can write several books; a book can have several co-authors), including a way to store each author's specific royalty percentage on a given book.

Goal: Practice the join-collection pattern for a many-to-many relationship that carries its own data.

→ Solution

Challenge 3: Identify a Denormalization Trade-off

A product document embeds its supplier's name directly (not just a supplierId). The supplier later rebrands and changes their name. Explain what happens to existing product documents, and how you might address it.

Goal: Practice reasoning through the real consequence of a denormalization decision made earlier.

→ Solution

⚠️ Gotcha: The 16MB Document Size Limit

Every MongoDB document has a hard maximum size of **16MB** — not a soft guideline, an enforced limit. This is exactly why "one-to-squillions" relationships must be referenced rather than embedded: a viral post that keeps embedding new comments directly into its own document will eventually hit this ceiling, and further inserts/updates to that document will simply start failing. It's easy to overlook this while a post only has a handful of comments in testing — the failure only shows up once real growth happens, which is precisely why the growth pattern matters more than the current size when choosing between embedding and referencing.

🎯 What's Next

The next chapter covers **Data Types & BSON** — `ObjectId`, dates, arrays, and nested documents in more depth, plus schema validation with `$jsonSchema` for when you want to opt back into enforced structure.

🔢 Data Types & BSON

Chapter 1 introduced BSON briefly. This chapter goes deeper into the actual types it supports, and covers `$jsonSchema` validation — MongoDB's opt-in answer to Chapter 5's point that schema flexibility is a deliberate trade-off, not a lack of options.

BSON Types in Depth

Type	Notes
`ObjectId`	MongoDB's default unique identifier type
`String`	UTF-8 text
`Int32` / `Int64`	Whole numbers
`Double`	Standard floating-point — the default for decimal literals
`Decimal128`	Exact decimal representation — see this chapter's money warning below
`Boolean`	`true` / `false`
`Date`	A real timestamp type — see the string-vs-Date gotcha below
`Array`	An ordered list, can hold any mix of types, including nested arrays/objects
`Object`	A nested (embedded) document
`Null`	An explicit absence of a value

ObjectId: MongoDB's Default Unique Identifier

An `ObjectId` is a 12-byte value made of a 4-byte creation timestamp, a 5-byte random value, and a 3-byte incrementing counter. One useful consequence: ObjectIds are roughly sortable by creation time, and the creation timestamp can be pulled directly out of an ObjectId with no separate stored field needed:

Extracting a Timestamp from an ObjectId

```
const id = ObjectId("64f1a2b3c4d5e6f7a8b9c0d1");
id.getTimestamp()
// ISODate("2023-09-01T14:22:11.000Z") — no createdAt field required
```

Dates: A Common Gotcha

MongoDB has a real `Date` BSON type (stored internally as milliseconds since the Unix epoch). A common mistake is storing a date as a plain **string** instead — `"2026-07-06"` — rather than an actual Date value.

Date-as-String vs. a Real Date

Stored as a String

`{ createdAt: "2026-07-06" }`

Range queries (`$gt`/`$lt`) compare strings *lexicographically*, character by character — which happens to work for consistently-formatted `YYYY-MM-DD` strings, but breaks the moment formats mix or times are involved.

Stored as a Real Date

`{ createdAt: ISODate("2026-07-06") }`

Range queries compare actual points in time correctly, regardless of formatting, time zones, or precision.

Arrays and Nested Documents

Arrays can hold any mix of types, including further nested arrays or objects — Chapter 5's embedding examples (addresses, comments) are really just arrays of nested documents. Nesting can go arbitrarily deep, but very deep nesting tends to hurt both query readability and performance — a practical reason to keep documents reasonably flat where possible, on top of Chapter 5's growth-based reasons for choosing embedding vs. referencing in the first place.

Decimal128: Precise Numbers for Money

The default numeric type for a decimal literal in `mongosh` is `Double` — ordinary binary floating-point, the same representation used in most programming languages, which cannot represent every decimal value exactly. `Decimal128` stores an exact decimal value instead, avoiding the rounding errors floating-point math can introduce — worth using deliberately for anything involving money.

Schema Validation with $jsonSchema

MongoDB's opt-in mechanism for enforcing structure on a collection — required fields, expected types, and value constraints — directly addressing the "nothing stops inconsistent documents" trade-off from Chapter 1 and Chapter 5:

A Validator Requiring Structure

```
db.createCollection("users", {
 validator: {
  $jsonSchema: {
   bsonType: "object",
   required: ["email", "age"],
   properties: {
    email: { bsonType: "string" },
    age: { bsonType: "int", minimum: 0 }
   }
  }
 }
})
// inserts violating this shape are now rejected, restoring the enforcement MySQL provides by default
```

💻 Coding Challenges

Challenge 1: Pick the Right BSON Type

Recommend a BSON type for each: (a) a customer's account balance in dollars, (b) an order's placement timestamp, (c) a product's list of tags.

Goal: Practice matching a real field to the type that avoids known pitfalls (money, dates) covered in this chapter.

→ Solution

Challenge 2: Spot the Date-as-String Bug

A query `db.orders.find({ placedAt: { $gt: "2026-02-01" } })` is meant to find orders placed after February 1st, but returns some orders from January and misses some from March. Explain the likely root cause.

Goal: Practice diagnosing the string-vs-Date comparison problem in a realistic symptom.

→ Solution

Challenge 3: Write a $jsonSchema Validator

Write a `$jsonSchema` validator for a `products` collection requiring a `name` (string) and a `price` (number, minimum 0).

Goal: Practice the validator syntax for required fields and value constraints.

→ Solution

⚠️ Gotcha: Money Stored as Double Can Silently Lose Precision

Typing a decimal number in `mongosh` — `{ price: 19.99 }` — stores it as a `Double` by default, ordinary binary floating-point. Floating-point can't represent every decimal value exactly, the same well-known issue behind `0.1 + 0.2` not exactly equaling `0.3` in most programming languages. For a display price this rarely matters, but for anything involving repeated financial arithmetic — running totals, tax calculations, currency conversion — those tiny errors can accumulate into a genuinely wrong result. Use `NumberDecimal("19.99")` to store an exact `Decimal128` value instead whenever a field represents real money.

🎯 What's Next

The next chapter covers **Indexes** — single-field and compound indexes, how MongoDB actually chooses which index to use via `explain()`, and what a missing index really costs.

⚡ Indexes

Every chapter so far covered how to model and store data. This chapter covers how to make querying it fast — and the tool that shows you whether a query is actually using an index at all.

Why Indexes Exist: The Collection Scan Problem

Without an index, a query has to check **every single document** in a collection one by one to find matches — a **collection scan**. That's fine on a few hundred documents, and disastrous on a few million. An **index** is a separate, sorted data structure that lets MongoDB jump straight to matching documents instead of examining everything.

Creating a Single-Field Index

Indexing a Field

```
db.products.createIndex({ category: 1 })
// 1 = ascending, -1 = descending
```

Compound Indexes

A **compound index** spans more than one field at once — useful when queries regularly filter and sort on multiple fields together:

An Index Across Two Fields

```
db.products.createIndex({ category: 1, price: -1 })
// speeds up queries filtering by category AND sorting by price descending
```

Field Order Matters: Equality, Sort, Range

The order fields are listed in a compound index isn't arbitrary — a widely used heuristic is **Equality, Sort, Range (ESR)**: put fields queried with an exact match first, fields used for sorting next, and fields queried with a range (`$gt`/`$lt`) last. A compound index built in the wrong order for how it's actually queried may not get used as efficiently as one built to match the query's real shape.

explain(): Seeing What the Database Actually Did

Appending `.explain("executionStats")` to a query reveals whether MongoDB used an index (`IXSCAN`) or fell back to scanning the whole collection (`COLLSCAN`), and how many documents were actually examined versus returned:

COLLSCAN vs. IXSCAN

```
// No index on category — full collection scan
db.products.find({ category: "books" }).explain("executionStats")
// stage: "COLLSCAN", totalDocsExamined: 100000, nReturned: 250
// With an index on category — direct lookup
db.products.find({ category: "books" }).explain("executionStats")
// stage: "IXSCAN", totalDocsExamined: 250, nReturned: 250
```

The second run examined exactly as many documents as it returned — the index let MongoDB go straight to the matches instead of checking all 100,000 documents to find 250 of them.

What a Missing Index Actually Costs

A query relying on a collection scan gets **linearly slower** as the collection grows — a query taking 10ms against 10,000 test documents might take several seconds against 10 million real ones, even though the query itself never changed. This is exactly why the problem is often invisible during development, where test collections are small, and only shows up once real data volume hits production.

💻 Coding Challenges

Challenge 1: Create a Compound Index

Queries on an `orders` collection regularly filter by `status` and sort by `createdAt` descending. Write the `createIndex` call that would speed this up.

Goal: Practice basic compound index syntax matching a real query pattern.

→ Solution

Challenge 2: Read an explain() Output

An `explain("executionStats")` result shows `stage: "COLLSCAN"`, `totalDocsExamined: 500000`, `nReturned: 12`. What does this tell you, and what would you do about it?

Goal: Practice reading explain() output as an actionable diagnosis, not just numbers.

→ Solution

Challenge 3: Order a Compound Index Correctly

A query filters by `status: "active"` (exact match), sorts by `priority`, and filters `createdAt` with a range (`$gt`). Using the Equality-Sort-Range rule, what order should the compound index's fields be in?

Goal: Practice applying the ESR heuristic to a concrete multi-clause query.

→ Solution

⚠️ Gotcha: Indexes Aren't Free — They Slow Down Writes

It's tempting to index every field "just in case" once you've seen how much faster a query with an index can be. But every index has to be updated whenever a document is inserted, updated, or deleted in a way that touches an indexed field — more indexes means more work on every write, and more storage used to hold the index structures themselves. Indexes should be created deliberately, based on the query patterns your application actually runs (exactly what `explain()` helps you verify), not applied blanket across every field on the assumption that more indexing is always better.

🎯 What's Next

The next chapter connects everything to real application code: **Working with MongoDB from Node.js** — the native driver vs. the Mongoose ODM, and finally seeing this course's queries called from a real app instead of typed into `mongosh` by hand.

🟢 Working with MongoDB from Node.js

Every prior chapter used `mongosh` directly. This final chapter connects it all to real application code — exactly the kind of connection the Node.js course's database-integration chapter (`node2-5`) covered for MySQL, now applied to MongoDB.

The Native Driver: mongodb

Installing `npm install mongodb` gives you `MongoClient`, which connects using the exact same connection-string format Chapter 2 introduced:

Connecting from Node.js

```
const { MongoClient } = require("mongodb");

const client = new MongoClient("mongodb://localhost:27017");
await client.connect();
const db = client.db("shop");
```

CRUD from the Native Driver

The driver's methods map almost directly onto the `mongosh` commands from Chapters 3–4 — same method names, called from JavaScript instead of typed interactively:

Insert and Find, in Real Code

```
const products = db.collection("products");

await products.insertOne({ name: "Laptop", price: 899 });

const cheapItems = await products.find({ price: { $lt: 50 } }).toArray();
// .toArray() — the exact step Chapter 3's cursor gotcha said you'd need here
```

Mongoose: An ODM Layer on Top

Mongoose is an Object Document Mapper — it adds JavaScript-level schemas, models, and validation on top of the native driver. Where Chapter 6's `$jsonSchema` enforces structure at the

database level, Mongoose enforces it at the application level, alongside convenience features like default values and pre/post hooks:

Defining a Mongoose Schema and Model

```
const mongoose = require("mongoose");

const productSchema = new mongoose.Schema({
  name: { type: String, required: true },
  price: { type: Number, min: 0 }
});
const Product = mongoose.model("Product", productSchema);

await Product.create({ name: "Notebook", price: 3.50 });
```

Native Driver vs. Mongoose

> **Native Driver**
>
> Closer to raw MongoDB, less abstraction, more manual — a good fit for smaller apps or when you want full control.

> **Mongoose**
>
> Application-level schema validation, models, and hooks — convenient for larger apps with many collections needing consistent structure.

Connection Pooling: Connect Once, Reuse

A `MongoClient` instance already manages an internal connection pool — the same underlying idea as the connection pooling covered for MySQL in `node2-5`. Create the client **once** when the application starts, and reuse it for every request; don't create a new client inside each individual route handler.

💻 Coding Challenges

Challenge 1: Connect and Insert

Write the Node.js code (using the native driver) to connect to a local MongoDB instance, select a database called `library`, and insert one document into a `books` collection.

Goal: Practice the connect → select database → operate sequence.

→ Solution

Challenge 2: Write a Mongoose Schema

Write a Mongoose schema and model for a `Task` with a required `title` (String) and a `done` field (Boolean, defaulting to `false`).

Goal: Practice Mongoose's schema syntax, including a default value.

→ Solution

Challenge 3: Fix a Connection-Per-Request Anti-Pattern

An Express route handler creates a new `MongoClient` and calls `.connect()` inside every single request. Explain what's wrong with this and how to fix it.

Goal: Practice applying this chapter's connection-pooling guidance to a concrete anti-pattern.

→ Solution

⚠️ Gotcha: Reconnecting on Every Request

It's an easy mistake to write `new MongoClient(...)` and `await client.connect()` directly inside a route handler, since that's the simplest thing that works in a quick test. Under real traffic, this means every single request pays the full cost of establishing a brand-new connection (and its internal pool) from scratch, and can quickly exhaust available connections on the MongoDB server under any real load. Create the client once — typically at application startup — and reuse that same client instance across every request; the driver's internal connection pool is designed to be shared, not recreated per request.

🎉 Course Complete

That's the full **MongoDB Fundamentals** course — from the document model through CRUD, schema design, data types, indexing, and finally real application code. Next up: **MongoDB Intermediate/Advanced**, covering the aggregation framework, schema design patterns at scale, transactions, replication, sharding, and a capstone data-modeling project.