MongoDB Course

Flavio Copes

Updated August 8, 2026

Contents

Learn MongoDB from document modeling and CRUD through indexes, aggregation, Node.js, transactions, change streams, security, backups, and recovery.

What you'll learn: Design, query, secure, observe, and recover a MongoDB-backed application without copying a relational schema mechanically.

Document database foundations

Documents, collections, BSON, local MongoDB, Atlas, and mongosh.

Understand the document model

See how MongoDB stores BSON documents in collections and how that model differs from rows in related SQL tables.

MongoDB stores **documents** inside **collections**. A document looks like a JavaScript object:

```
{
  name: 'Roger',
  age: 8,
  favoriteFoods: ['biscuits', 'carrots']
}
```

That similarity is why MongoDB feels approachable if you already write JavaScript. But MongoDB does not store JSON text. It stores BSON, a binary format that adds types JSON lacks: dates, 128-bit decimals, binary data, and object IDs. When you insert a document, MongoDB generates a unique `_id` of type `ObjectId` unless you provide one.

Documents can contain nested objects and arrays. In a relational database, a dog with three favorite foods needs a `dogs` table, a `foods` table, and a join between them. In MongoDB the whole thing can live in one document, and one read returns it complete.

Flexible schema, not no schema

Documents in the same collection do not need identical fields. One dog can have a `microchipId`, the next can skip it. This is often called schemaless, but **flexible schema** is more accurate: the structure lives in your application code, and you can enforce parts of it in the database later with schema validation.

Flexibility is a tool, not an excuse. If every document in a collection has a different shape, every reader must handle every shape. Aim for one intentional structure per collection, with deliberate variations.

The design consequence

The relational instinct is to normalize everything into separate tables and join at read time. Carrying that instinct into MongoDB produces the worst of both worlds: many tiny collections and no joins to connect them cheaply.

Instead, document boundaries follow how the application reads and writes data. Data you always load together belongs together. Try it on paper: take a small SQL schema you know, pick its most common query, and draw one document that answers that query in a single read. Do not assume every table must become a collection — that assumption is the most common modeling mistake people bring from SQL.

Choose local MongoDB or Atlas

Run a disposable local database or create an Atlas deployment without exposing an unauthenticated server.

You have two sensible ways to get a MongoDB server: run MongoDB Community Edition locally, or create a deployment on **MongoDB Atlas**, the managed cloud service.

A local server is perfect for learning and isolated development. It costs nothing, starts instantly, and you can throw it away. On macOS, install it from the official Homebrew tap:

```
brew tap mongodb/brew
brew install mongodb-community
brew services start mongodb-community
```

Verify it is running before you continue:

```
brew services list | grep mongodb
# mongodb-community started flavio ~/Library/LaunchAgents/...
```

By default a local `mongod` binds to localhost only. Keep it that way. An unauthenticated MongoDB exposed to a network is one of the classic ways databases end up scraped and ransomed. If you ever change the bind address, configure authentication first and follow the official security checklist.

When Atlas is the better choice

Atlas gives you a managed deployment with authentication, network access rules, backups, and monitoring already wired up. The free M0 tier is enough for this course. Use Atlas when the application needs a shared environment your teammates or deployed services can reach, or when you want replica-set features without operating servers yourself.

Atlas connection strings use the `mongodb+srv://` scheme and include credentials:

```
mongodb+srv://app_user:s3cretPass@cluster0.ab1cd.mongodb.net/animals
```

Two Atlas defaults matter. First, nothing can connect until you add your IP to the **IP Access List** — a connection that hangs and then fails with a server-selection timeout is almost always this. Second, you create database users in the Atlas UI; use a dedicated application user with access to one database, not the admin account you registered with.

Record the boundary

Whichever you choose, create one practice database and write down four things before adding data: the connection endpoint, the user you connect as, the networks allowed to reach it, and the cleanup step that deletes it. That habit sounds bureaucratic for a lab. It is exactly the discipline that prevents a ”temporary” open database in production.

Navigate with mongosh

Connect with the current shell, select a database, inspect collections, and understand when MongoDB creates them.

The current MongoDB shell is **mongosh**. Connect, run `db` to inspect the selected database, then use `use animals` to switch context.

MongoDB creates a database and ordinary collection when the first document is written. You can also create a collection explicitly when you need validation, capped behavior, or other options from the start.

Create an `animals` database, insert one document, then list databases and collections. Remove the practice data when you finish.

Create a disposable database and inspect what MongoDB creates:

```
use animals
db.animals.insertOne({ name: 'Luna', age: 4 })
show collections
```

```
db.animals.find()
```

Run db.getName() before each destructive command. Notice that selecting animals did not persist anything until the insert. Remove the practice database only after confirming you are still connected to the local lab rather than Atlas or another shared server.

Check your understanding: document database foundations

Check the key models, decisions, commands, security boundaries, and failure cases from document database foundations.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

CRUD and querying

Insert, find, update, delete, project, sort, paginate, and validate documents.

Insert and read documents

Use insertOne, insertMany, find, filters, projections, and ObjectId values without treating shell expressions as SQL.

Use `insertOne()` for one document and `insertMany()` for a batch. MongoDB adds an `_id` when you do not supply one, and the response tells you what it generated:

```
db.animals.insertOne({ name: 'Roger', species: 'dog', favoriteFoods: ['biscuits'] })
// { acknowledged: true, insertedId: ObjectId('688f4a2e9d1c2a7b3e4f5a60') }

db.animals.insertMany([
  { name: 'Buck', species: 'dog', favoriteFoods: ['carrots'] },
  { name: 'Luna', species: 'cat', favoriteFoods: ['tuna', 'chicken'] },
  { name: 'Togo', species: 'dog', favoriteFoods: ['biscuits', 'carrots'] }
])
// { acknowledged: true, insertedIds: { '0': ObjectId('...'), '1': ObjectId('...'), '2': Obje
```

The collection is created on the first insert. The old `insert()` method still exists in some drivers but is not the right API for new code.

The Node.js driver uses the same `insertMany()` method on a collection:

```
const animals = (await getDb()).collection('animals')

const result = await animals.insertMany([
  { name: 'Buck', species: 'dog' },
  { name: 'Luna', species: 'cat' }
])

console.log(result.insertedCount) //2
```

Await the result so you can handle write errors and inspect how many documents were inserted.

By default, `insertMany()` is ordered. MongoDB stops at the first document that fails, for example

because it violates a unique index. Documents inserted before the failure remain in the collection, while later documents are skipped.

Pass `{ ordered: false }` when the documents are independent and MongoDB should attempt the rest of the batch:

```
await animals.insertMany(documents, { ordered: false })
```

An unordered batch can still report a bulk write error, so handle the error and inspect its result instead of assuming the whole array was inserted.

Reading with filters and projections

Read with `find()` and pass a filter document. An empty filter matches everything:

```
db.animals.find({ species: 'dog' })
db.animals.findOne({ name: 'Roger' })
```

A filter is a document, not a SQL expression. Equality is `{ species: 'dog' }`, comparisons use operators like `{ age: { $gte: 5 } }`. If `findOne()` matches several documents it returns the first one according to the query plan, so add a unique field to the filter when the exact record matters.

A **projection** limits the fields that come back — the second argument to `find()`:

```
db.animals.find({ name: 'Roger' }, { name: 1, favoriteFoods: 1, _id: 0 })
// [ { name: 'Roger', favoriteFoods: [ 'biscuits' ] } ]
```

Returning less data matters once documents grow beyond toy size.

The ObjectId trap

Here is the mistake everyone makes once. You copy an `_id` from earlier output and query with it as a string:

```
db.animals.find({ _id: '688f4a2e9d1c2a7b3e4f5a60' })
// (no results)

db.animals.find({ _id: ObjectId('688f4a2e9d1c2a7b3e4f5a60') })
// [ { _id: ObjectId('688f4a2e9d1c2a7b3e4f5a60'), name: 'Roger', ... } ]
```

Nothing errors. The string form of an ObjectId simply does not equal the ObjectId value, so the first query matches zero documents. In application code this usually appears as "the document exists in the shell but my API returns 404": the route handler received the ID as a string and never converted it. Wrap it in `ObjectId()` (or `new ObjectId(id)` in the Node driver) before filtering.

Update and delete deliberately

Use update operators, upserts, single-document methods, and guarded deletions without replacing or removing too much data.

MongoDB update methods take two documents: a filter that selects, and an update document built from operators such as `$set`, `$inc`, and `$push`. Keeping them separate in your head prevents the two classic accidents: updating the wrong documents, and replacing a document when you meant to change one field.

```
db.animals.updateOne(
```

```
  { name: 'Roger' },
  { $inc: { visits: 1 }, $push: { favoriteFoods: 'cheese' } }
)
// { acknowledged: true, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 }
```

`$inc` increments a number, `$push` appends to an array, `$set` assigns a value, `$unset` removes a field. Use `updateOne()` when one match is the intended boundary and `updateMany()` when every matching document should change.

Always read the response. `matchedCount: 0` means your filter found nothing and the update did no work — no error is raised. A misspelled name or a string passed where an `ObjectId` was needed fails silently this way.

If you pass a plain document without operators to `replaceOne()`, the entire document is replaced except `_id`. That is occasionally what you want. Reaching for it by habit deletes every field you forgot to include.

An **upsert** inserts the document when the filter matches nothing:

```
db.counters.updateOne(
  { _id: 'pageviews' },
  { $inc: { total: 1 } },
  { upsert: true }
)
```

Deleting with a guard

Deletion follows the same filter logic:

```
db.animals.deleteOne({ _id: ObjectId('688f4a2e9d1c2a7b3e4f5a60') })
// { acknowledged: true, deletedCount: 1 }
```

An empty filter matches every document. That makes `deleteMany({})` useful for wiping a disposable test collection and catastrophic against a real one. There is no confirmation prompt and no undo.

Reject an empty filter when it is built from user input or optional parameters:

```
if (Object.keys(filter).length === 0) {
  throw new Error('Refusing to delete with an empty filter')
}

const result = await animals.deleteMany(filter)
```

`deleteMany({})` removes the documents but keeps the collection and its indexes. `drop()` removes the collection and its indexes too. Use `drop()` only when that is the result you intend.

My advice is to preview every broad mutation before running it. Count what the filter matches first:

```
db.animals.countDocuments({ species: 'hamster' })
// 2  ← expected? then delete
db.animals.deleteMany({ species: 'hamster' })
```

Test the non-matching case too: run a delete with a filter you know matches nothing and confirm `deletedCount: 0`. Code that treats ”zero deleted” as success hides typos in production filters.

Add schema validation

Use JSON Schema validation for stable invariants while preserving deliberate flexibility between document shapes.

MongoDB has a flexible schema, not an absent schema. While you prototype, that flexibility is a feature. Once the application stabilizes, some fields become invariants: every animal has a name, age is always a number. **Schema validation** lets the database enforce exactly those rules while leaving the rest of the document free.

Why put checks in the database when the application already validates? Because the application is not the only writer. Imports, migration scripts, admin tooling, and the teammate poking around in `mongosh` all bypass application code. The application produces friendly error messages; the database guarantees the invariant.

Validation is expressed as a `$jsonSchema` and attached when creating a collection:

```
db.createCollection('animals', {
  validator: {
    $jsonSchema: {
      required: ['name', 'age'],
      properties: {
        name: { bsonType: 'string' },
        age: { bsonType: 'int', minimum: 0 }
      }
    }
  }
})
```

Now try one valid and one invalid insert:

```
db.animals.insertOne({ name: 'Roger', age: 8 })
// { acknowledged: true, insertedId: ObjectId('...') }

db.animals.insertOne({ name: 'Buck', age: 'four' })
// MongoServerError: Document failed validation
```

The error includes a `details` object that names the failing rule — here `age` failed the `bsonType` check. Read it instead of guessing; with several rules it tells you exactly which one rejected the write.

Tightening a live collection

Adding validation to an existing collection uses `collMod`, and this is where planning matters. Existing documents are not checked when you add the validator — only future writes are. Before tightening, find out what would fail:

```
db.animals.countDocuments({ age: { $not: { $type: 'int' } } })
```

If old documents violate the new rule, an innocent update to an unrelated field on one of them can suddenly fail validation. That is the classic surprise: writes that worked yesterday start erroring after someone added a validator. Either fix the old documents first, or set `validationLevel: 'moderate'` so existing invalid documents can still be updated while new inserts must comply.

Keep the validator small. Enforce the invariants every valid write must respect, and leave deliber-

ately flexible fields out of it.

Check your understanding: crud and querying

Check the key models, decisions, commands, security boundaries, and failure cases from crud and querying.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Modeling and indexes

Access patterns, embedding, references, document growth, indexes, and explain plans.

Model from access patterns

Start schema design from the reads, writes, ownership, growth, and consistency requirements the application actually has.

In a relational database you model the data, then write whatever queries you need. MongoDB works better the other way around: model from **access patterns**. List the questions the application asks, the fields it changes together, and how each collection grows. The schema falls out of that list.

Take a small publishing app. Before designing anything, write the workload down:

```
Reads (most frequent first)
  R1: article page - article + author name + comments     every page view
  R2: homepage - latest 20 article summaries              every visit
  R3: author profile - author + their article titles      occasional

Writes
  W1: add a comment to an article                         frequent
  W2: publish an article                                  a few per day
  W3: update an author bio                                rare
```

Now design documents that serve R1 and R2 in one read each. An `articles` document can carry its title, body, a copy of the author's display name, and recent comments. The homepage query then needs a projection of the same collection, nothing else.

```
{
  _id: ObjectId('...'),
  title: 'Modeling in MongoDB',
  authorId: ObjectId('...'),
  authorName: 'Flavio',
  comments: [ { user: 'anna', text: 'Great post' } ],
  publishedAt: ISODate('2026-08-03T09:00:00Z')
}
```

What a flexible schema will not rescue

Three shapes become expensive later no matter how flexible the schema is.

Unbounded arrays. Comments on a popular article grow forever. A document has a 16 MB hard limit, and every read carries the whole array. If growth has no natural bound, cap it (keep the latest 10 embedded) or move the rest to its own collection.

Duplicated values with unclear ownership. Copying `authorName` into articles is fine — until someone renames an author and half the copies never get updated. Duplicate deliberately, and name which collection holds the authoritative value and what updates the copies.

Mixed lifecycles. A document that combines a user's profile, their settings, and their activity log mixes data that changes at completely different rates. Writes contend, and every reader pays for data it did not ask for.

Do the exercise for your own app: three most common reads, three most common writes, then documents. Collections come last, not first.

Choose embedding or references

Embed data read and changed together, and reference data with independent ownership, growth, or many-to-many relationships.

Every relationship in a MongoDB schema gets one of two treatments. **Embedding** puts the related data inside the parent document. **Referencing** stores it in its own collection and keeps only an ID in the parent.

Here are authors and posts modeled both ways. Embedded:

```
// posts collection - author details inside
{
  _id: ObjectId('...'),
  title: 'My first post',
  author: { name: 'Flavio', twitter: '@flaviocopes' },
  body: '...'
}
```

Referenced:

```
// posts collection - only a pointer
{ _id: ObjectId('...'), title: 'My first post', authorId: ObjectId('64ff...'), body: '...' }

// authors collection - the authoritative record
{ _id: ObjectId('64ff...'), name: 'Flavio', twitter: '@flaviocopes' }
```

Embedding wins on reads: the post page needs one query instead of two. It also wins on atomicity — a write to a single document is atomic, so the post and its embedded author details can never disagree mid-update.

Referencing wins when the related entity has a life of its own. Ask three questions:

Does it change independently? If the author renames themselves, the referenced version updates one document. The embedded version leaves stale copies in every post until you update them all.

Does it grow without bound? An author's list of posts embedded inside the author document grows forever. Referenced posts do not.

Is it many-to-many? Tags shared across thousands of posts should not be copied into each one with no plan for updates.

Duplication is allowed — with a name attached

The two options combine. A common production pattern keeps the reference *and* embeds the one or two fields the read path needs:

```
{ title: 'My first post', authorId: ObjectId('64ff...'), authorName: 'Flavio' }
```

This is deliberate duplication. It is fine as long as you name the authoritative copy (the `authors` collection) and the update plan (a rename triggers an `updateMany` on posts). Duplication without that plan is how data quietly rots: the profile page shows the new name, old posts show the old one, and nobody can say which is correct.

Work through the exercise: model authors and posts both ways, then list which changes are cheap and which are painful in each version. Renames, deletes, and "show all posts by author" make the trade-offs concrete fast.

Index from real queries

Create compound indexes from filters and sorts, then read executionStats instead of assuming an index improved the workload.

Without a useful index, MongoDB may scan the collection. Indexes speed supported reads but add storage and write work.

Use `explain('executionStats')` on a representative query. Compare returned documents, examined documents, examined keys, and the execution stages. Compound field order must match the important equality, range, and sort patterns.

Load enough sample documents to make a scan visible. Add one justified index and compare the plan before and after.

Measure the query before and after one index:

```
db.animals.find({ species: 'cat' }).sort({ age: -1 }).explain('executionStats')
db.animals.createIndex({ species: 1, age: -1 })
db.animals.find({ species: 'cat' }).sort({ age: -1 }).explain('executionStats')
```

Compare execution stages, totalDocsExamined, totalKeysExamined, and nReturned. Keep the index only when it supports an important query and its write cost is acceptable.

Check your understanding: modeling and indexes

Check the key models, decisions, commands, security boundaries, and failure cases from modeling and indexes.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Applications and events

Node.js drivers, aggregation pipelines, transactions, change streams, retries, and idempotency.

Connect from Node.js

Create one shared MongoClient, keep credentials on the server, select collections explicitly, and close resources in scripts and tests.

Install the official driver and put the client in one infrastructure module:

```
npm install mongodb
```

```
// db.js
import { MongoClient } from 'mongodb'

const client = new MongoClient(process.env.MONGODB_URI)

export async function getDb() {
  await client.connect()
  return client.db('animals')
}

export async function closeDb() {
  await client.close()
}
```

Two decisions are baked in here. The connection string lives in an environment variable, so credentials never land in the repository — and never, ever in browser JavaScript. And there is exactly one `MongoClient` for the whole process. The client manages a connection pool internally; creating a new client per request creates a new pool per request, and under load you exhaust the server's connections. Calling `connect()` twice on the same client is safe, it returns the existing connection.

A long-running API server connects once and reuses the client forever. Short scripts and tests are different: they must close the client, or the open pool keeps the Node.js event loop alive and the process never exits. A script that "hangs at the end" almost always forgot this. Close in a `finally` block so it happens on failure too:

```
import { getDb, closeDb } from './db.js'

try {
  const animals = (await getDb()).collection('animals')

  await animals.insertOne({ name: 'Roger', species: 'dog' })
  const roger = await animals.findOne({ name: 'Roger' })
  console.log(roger)
  // { _id: new ObjectId('688f4a2e...'), name: 'Roger', species: 'dog' }
} finally {
  await closeDb()
}
```

Run it and verify both things: the document prints, and the process exits on its own. That second check is the point of the exercise.

Pass collections in, not the client around

Business logic that imports the database module directly is painful to test. Pass the collection as an argument instead:

```
export async function registerVisit(animals, name) {
  return animals.updateOne({ name }, { $inc: { visits: 1 } })
}
```

Tests can now call `registerVisit()` with a collection from an isolated test database and assert on real driver behavior. The function does not know or care which database it talks to — that decision stays in one place, at the edge of the application.

Build an aggregation pipeline

Transform documents in small stages with match, unwind, group, project, and sort while keeping early filtering visible.

An **aggregation pipeline** passes documents through ordered stages. Each stage filters, reshapes, groups, joins, or calculates data for the next stage. It is how you answer questions that `find()` cannot, like "what are the five most popular favorite foods among dogs?"

Build it one stage at a time, inspecting output after each addition. Start by filtering:

```
db.animals.aggregate([
  { $match: { species: 'dog' } }
])
// full dog documents, unchanged
```

Each dog has a `favoriteFoods` array. `$unwind` turns one document with three foods into three documents with one food each — that gives `$group` something to count:

```
db.animals.aggregate([
  { $match: { species: 'dog' } },
  { $unwind: '$favoriteFoods' },
  { $group: { _id: '$favoriteFoods', count: { $sum: 1 } } }
])
// [ { _id: 'biscuits', count: 12 }, { _id: 'carrots', count: 9 }, ... ]
```

Note the `'$favoriteFoods'` syntax: inside a pipeline, a string starting with `$` refers to a field's value. Finish with ordering and a bound:

```
db.animals.aggregate([
  { $match: { species: 'dog' } },
  { $unwind: '$favoriteFoods' },
  { $group: { _id: '$favoriteFoods', count: { $sum: 1 } } },
  { $sort: { count: -1 } },
  { $limit: 5 }
])
```

That is the report: top five foods, most popular first.

Stage order is performance

Place selective `$match` stages as early as possible. A `$match` at the start of the pipeline can use an index on the collection. The same condition placed after a `$group` cannot — by then the original documents are gone and MongoDB has already processed the entire collection to build the groups.

The failure mode looks like this: the pipeline returns correct results in development, then takes thirty seconds on the production collection with two million documents. Running the same pipeline with `.explain()` shows a `COLLSCAN` feeding the first stage. The fix is usually moving a filter earlier, or adding the index the early `$match` needs.

Keep each stage small enough to inspect on its own. And keep expectations honest: aggregation is powerful, but it does not make an unbounded data model or a missing index disappear. A pipeline that compensates for a bad document shape on every read is a signal to fix the shape.

Choose transactions or change streams

Use transactions for atomic multi-document state changes and change streams for observing committed changes without confusing the two jobs.

A single MongoDB document update is atomic. That is not a limitation to work around — it is the reason embedding related data in one document is the first tool for consistency. Reach for a **transaction** only when one business invariant truly spans multiple documents or collections.

The textbook case is a transfer between two account documents. Either both writes happen or neither does:

```
const session = db.getMongo().startSession()
try {
  session.startTransaction()
  const accounts = session.getDatabase('bank').accounts

  accounts.updateOne({ _id: 'alice' }, { $inc: { balance: -100 } })
  accounts.updateOne({ _id: 'bob' }, { $inc: { balance: 100 } })

  session.commitTransaction()
} catch (error) {
  session.abortTransaction()
  throw error
}
```

Transactions require a replica set or sharded cluster — a plain standalone `mongod` refuses to start one. They also add coordination cost, hold resources while open, and can abort under contention, so your code must be ready to retry. If every operation in your application runs in a transaction, that is not caution. It is a document model that put data which belongs together into separate collections.

Change streams observe, they do not change

A **change stream** is a different job entirely: it lets you subscribe to committed changes on a collection, database, or cluster.

```
const stream = db.orders.watch([
  { $match: { operationType: 'insert' } }
```

```
])

stream.on('change', event => {
  console.log(event.fullDocument._id, 'created')
})
```

This fits search-index updates, cache invalidation, and analytics feeds: react to what happened, after it happened. Like transactions, change streams need a replica set, because they read from the replication machinery.

Consumers still carry responsibility. Your process will restart, so persist the stream's **resume token** and pass it back to `watch()` to continue where you left off. Expect to see some events more than once after a resume, and make handlers idempotent.

Classify before you build

Try these three: an account transfer, a "notify analytics when an order is created" requirement, and a search-index update. The transfer changes two documents under one invariant — transaction. The other two observe committed changes and react — change streams. If a case seems to need both, split it: change the state atomically first, let observers react afterward.

Check your understanding: applications and events

Check the key models, decisions, commands, security boundaries, and failure cases from applications and events.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Security and operations

Authentication, network controls, replication, backups, monitoring, scaling, and recovery.

Secure the database boundary

Require authentication, narrow network access, encrypt connections, separate users, and rotate credentials without exposing an open mongod.

A MongoDB server should not be an anonymous public endpoint. That sentence sounds obvious, yet exposed unauthenticated MongoDB instances have fueled years of automated data theft: scanners find the open port, dump the data, replace it with a ransom note. Every one of those incidents started with default settings on a public address.

The boundary has four layers, and you want all of them.

Authentication. A fresh self-managed `mongod` does not require login. Enable it in the config file:

```
# /etc/mongod.conf
security:
  authorization: enabled
net:
  bindIp: 127.0.0.1
```

Create the administrator first, then a separate, narrowly privileged user for the application:

```
use animals
db.createUser({
  user: 'app_user',
  pwd: passwordPrompt(),
  roles: [ { role: 'readWrite', db: 'animals' } ]
})
```

`readWrite` on one database is enough for a typical application. The runtime account should not be able to create users, drop other databases, or reconfigure the server — that is what your separate admin account is for, used by humans, rarely.

Network. `bindIp: 127.0.0.1` keeps the server on localhost. When the application lives on another host, bind to a private interface and firewall port 27017 to exactly the machines that need it. On Atlas the same idea is the IP Access List. `0.0.0.0/0` "temporarily" is how permanent holes are made.

Encryption. Use TLS on any connection that crosses a network, and prefer connection strings that require it (`tls=true`; Atlas `mongodb+srv://` strings do this for you).

Credential hygiene. The URI embeds the password, so treat the whole URI as a secret: store it in a secret manager or deployment configuration, and make sure logs and error messages never print it. Verify that deliberately — trigger a failed connection and read what your logging actually captured:

```
node app.js 2>&1 | grep -c 's3cretPass'
# 0   ← must be zero
```

Rehearse rotation before you need it: create a second user, deploy with the new credentials, remove the old user. If that requires downtime, fix it now, not during an incident.

Finish by reviewing your practice environment against the official MongoDB security checklist, and close every access path you cannot justify out loud.

Understand replication and scaling

Separate high availability from sharding, and understand elections, read and write concerns, shard keys, and failure behavior.

MongoDB has two scaling mechanisms that beginners constantly mix up. A **replica set** copies the same data to several servers for availability. **Sharding** splits a collection across several servers for capacity. They solve different problems, and adding the wrong one is expensive.

Replica sets: surviving a dead server

A replica set is typically three nodes. One is the **primary** and takes all writes; the others replicate its operation log. When the primary dies, the remaining members hold an election and promote a new primary, usually within seconds. Every Atlas deployment is a replica set — this is the baseline for anything in production.

Replication is asynchronous, which forces two explicit choices. **Write concern** decides when a write counts as done:

```
db.orders.insertOne(
```

```
  { item: 'book', qty: 1 },
  { writeConcern: { w: 'majority' } }
)
```

`w: 'majority'` waits until most members have the write. `w: 1` returns after the primary alone — faster, but if that primary dies before replicating, the acknowledged write is rolled back. For data you cannot lose, use majority.

Read preference decides where reads go. Reading from secondaries spreads load but can return slightly stale data. The classic bug: a user saves a change, the next page reads from a lagging secondary, and their edit "disappears" for a few seconds. If that is unacceptable, read from the primary or use appropriate read concern.

Sharding: outgrowing one machine

Sharding distributes a collection across shards, routed by a **shard key**. The key must spread writes evenly and appear in your common queries, so the router can target one shard instead of broadcasting to all of them. A monotonically increasing key (like a timestamp) funnels every insert to one shard; a key missing from queries makes every read a scatter-gather. Shard keys are hard to change later, so this decision deserves real care.

Do not add sharding to fix one slow unindexed query. Sharding adds routers, config servers, and operational weight — an index costs none of that.

Close with the drill from the exercise: draw the failure of a replica-set primary (evidence: brief write errors, election in the logs; recovery: automatic failover) and of one overloaded shard (evidence: latency on a subset of keys, uneven shard sizes; recovery: rebalancing or a better shard key). If you cannot name the evidence, you are not ready to operate the topology.

Back up, observe, and recover

Monitor workload signals, create backups with a known consistency point, test restores, and keep a runbook for corruption or operator mistakes.

A backup is useful only when you can restore it. Plenty of teams discover at the worst moment that their backups were empty, partial, or three weeks old. So this lesson works backwards: define what recovery must achieve, then pick tools, then prove they work.

Start with two numbers. How much data can you afford to lose (recovery point)? How long can you be down (recovery time)? "One hour of data, back within thirty minutes" leads to very different tooling than "yesterday's snapshot is fine."

On Atlas, enable scheduled backups and, if your recovery point is tight, continuous backup with point-in-time restore. Self-managed, the starting tool is `mongodump`:

```
mongodump --uri="mongodb://backup_user:s3cret@localhost:27017/animals" \
  --out=/backups/animals-2026-08-03
# writes BSON + metadata per collection
```

For anything beyond small databases, prefer filesystem snapshots or Atlas backups — `mongodump` reads everything through the server and gets slow at scale. Whatever you pick, know the backup's **consistency point**: a dump of a live server captures collections at slightly different moments unless you use oplog options or snapshot-based tooling. A backup whose consistency point you cannot state is a guess.

Remember replication is not backup. A replica set faithfully replicates your accidental `deleteMany({})` to every member within milliseconds. Only a backup holds the state from before the mistake.

Watch the signals that predict trouble

Monitor connections, operation rates, replication lag, storage growth, cache pressure, slow queries, and failed backups. Atlas exposes these as metrics and alerts; self-managed setups can scrape them from `db.serverStatus()`. Keep alerts tied to user-visible risk: "replication lag above 60 seconds" and "backup job failed" deserve a page. A dashboard where every moving number alerts trains everyone to ignore it.

The drill

Restore a disposable backup into a separate environment — never over the production database:

```
mongorestore --uri="mongodb://localhost:27018" --drop /backups/animals-2026-08-03
# 1240 document(s) restored successfully. 0 document(s) failed.
```

Then verify like you mean it: compare `db.animals.countDocuments({})` against the source, and run one real application workflow against the restored data. Write down the steps and how long they took — that document is your runbook, and the recovery time you measured is the only one that counts.

Check your understanding: security and operations

Check the key models, decisions, commands, security boundaries, and failure cases from security and operations.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/mongodb/.