MongoDB Project: Blog Ka Database (API ke sath)

The whole series in one build: a blog's users, posts and comments — schemas, a small Node API, and one query that joins everything.

Introduction

Eleven lessons of pieces — documents, CRUD, operators, Mongoose, pipelines, lookups, indexes. Time to assemble them into the thing they were for: **a working blog database with a small API on top**. Users register, authors post, readers comment — and one final query pulls a post home with its author and every comment counted. Every technique you meet below has a lesson number behind it; this capstone is where the MongoDB series shakes hands with the Node series (N.1–N.6) and becomes a backend.

(Roman Urdu: Ye project poori series ka imtihan hai — jo seekha, wahi yahan istemal hoga.)

Key points

- **Design first, from the reading side (MG.8's rule):** a blog reads *posts with their author* and *posts with their comments*. So: `users`, `posts`, `comments` — three collections. A post *embeds* its `tags` (small, read together), *references* its author (shared, read alone), and comments *reference* both post and author (unbounded growth — never embedded).
- **Schemas carry MG.5 + MG.9:** required fields and honest types; the `slug` born in a pre-save hook; `timestamps` on everything; a unique index on `users.email` and on `posts.slug` (MG.10's bouncers doing real duty).
- **The API is thin on purpose:** routes receive a request, call a model, answer JSON. `POST` creates (status **201**), `GET` reads (status **200**). All the intelligence lives in the schemas and queries — the routes are just doors.
- **The full-post read is one pipeline:** `$match` the slug → `$lookup` the author → `$lookup` the comments → count them with `$size` (MG.7 + MG.8 in one query). One trip to the database, one complete answer — no waterfall of follow-up queries from the app.
- **Errors are part of the design:** a duplicate email hits the unique index (E11000) and the API answers **409** in human words; a missing slug answers **404**. MG.10's rule holds: database errors are logs, not UI.

Code examples

Example 1 — The design stands up: models + first user

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  username: { type: String, required: true, unique: true },
  email: { type: String, required: true, unique: true }
}, { timestamps: true });

const postSchema = new mongoose.Schema({
  title: { type: String, required: true },
  slug: { type: String, unique: true },
  body: { type: String, required: true },
  author: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
  tags: [String]
}, { timestamps: true });

postSchema.pre('save', function () {
  if (!this.slug) this.slug = this.title.toLowerCase().replaceAll(' ', '-');
});

const commentSchema = new mongoose.Schema({
  text: { type: String, required: true },
  author: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
  post: { type: mongoose.Schema.Types.ObjectId, ref: 'Post', required: true }
}, { timestamps: true });
```

What it does: Three schemas, and every line is a series callback — required fields (MG.5), the slug hook (MG.9's exact trick), references by ObjectId (MG.8), `unique` guards on username/email/slug (MG.10), embedded `tags` (small, travel with the post). Then the first door opens:

```
app.post('/users', async (req, res) => {
  const user = await User.create(req.body);
  res.status(201).json(user);
});
```

What you will see — `POST /users` with `{ "name": "Ali Raza", "username": "ali_codes", "email": "ali@example.com" }`:

```
HTTP 201 Created
{
  "_id": "650000000000000000000301",
  "name": "Ali Raza",
  "username": "ali_codes",
  "email": "ali@example.com",
  "createdAt": "2026-10-24T05:30:00.000Z",
  "updatedAt": "2026-10-24T05:30:00.000Z",
  "__v": 0
}
```

The blog's first author exists — the very Ali of MG.10's users collection, same id `...301`. Sara registers the same way (`sara_writes`, id `...302`), and Bilal too (`bilal_reads`, id `...303`). The doors work; now let them write.

Example 2 — A post is born, and readers answer it

```
app.post('/posts', async (req, res) => {
  const post = await Post.create(req.body);
  res.status(201).json(post);
});

app.post('/comments', async (req, res) => {
  const comment = await Comment.create(req.body);
  res.status(201).json(comment);
});
```

What it does: Ali publishes the post his MG.9 slug was rehearsing — `POST /posts` with his author id, a title, a body, and embedded tags. Then the comments arrive: Sara first (her exact words from MG.9 — the same comment document, id `...501`), then Bilal (`...502`).

What you will see — the post, then Sara's comment:

```
HTTP 201 Created
{
  "_id": "650000000000000000000401",
  "title": "MongoDB Seekhna Asaan Hai",
  "slug": "mongodb-seekhna-asaan-hai",
  "body": "Documents aur collections ki pehli samajh — apne lafzon mein.",
  "author": "650000000000000000000301",
  "tags": ["mongodb", "nosql"],
  "createdAt": "2026-10-24T05:31:00.000Z",
  "__v": 0
}

HTTP 201 Created
{
  "_id": "650000000000000000000501",
  "text": "Bohat aala, samajh aa gaya!",
  "author": "650000000000000000000302",
  "post": "650000000000000000000401",
  "createdAt": "2026-10-24T05:32:00.000Z",
  "__v": 0
}
```

Look at what the *database* contributed unasked: the slug (hook), the timestamps (schema option), the ids (ObjectIds). The route code stayed six lines because the design carries the weight. Bilal's comment lands the same way — *"Aggregation wala hissa sab se best tha."* — and the post now has two comments pointing at it.

Example 3 — The full read: one pipeline brings everything home

```
app.get('/posts/:slug', async (req, res) => {
  const result = await Post.aggregate([
    { $match: { slug: req.params.slug } },
    { $lookup: { from: 'users', localField: 'author', foreignField: '_id', as: 'author' } },
    { $unwind: '$author' },
    { $lookup: { from: 'comments', localField: '_id', foreignField: 'post', as: 'comments' }
},
    { $addFields: { commentsCount: { $size: '$comments' } } },
    { $project: { __v: 0, 'author.email': 0, 'author.__v': 0, 'comments.__v': 0 } }
  ]);
  if (!result.length) return res.status(404).json({ error: 'Post not found' });
  res.json(result[0]);
});
```

What it does: The blog's post page needs *everything at once* — and gets it in one pipeline: match the slug (MG.7), look up the author and unwind him into place (MG.8), look up the comments, count them with `$size`,

and project away the internals (the author's email stays private — projection as manners). A wrong slug matches nothing and earns an honest 404.

What you will see — `GET /posts/mongodb-seekhna-asaan-hai`:

```
HTTP 200 OK
{
  "_id": "650000000000000000000401",
  "title": "MongoDB Seekhna Asaan Hai",
  "slug": "mongodb-seekhna-asaan-hai",
  "body": "Documents aur collections ki pehli samajh — apne lafzon mein.",
  "tags": ["mongodb", "nosql"],
  "author": { "_id": "650000000000000000000301", "name": "Ali Raza", "username": "ali_codes"
},
  "comments": [
    { "_id": "650000000000000000000501", "text": "Bohat aala, samajh aa gaya!", "author":
"650000000000000000000302", "post": "650000000000000000000401" },
    { "_id": "650000000000000000000502", "text": "Aggregation wala hissa sab se best tha.",
"author": "650000000000000000000303", "post": "650000000000000000000401" }
  ],
  "commentsCount": 2,
  "createdAt": "2026-10-24T05:31:00.000Z"
}
```

One request, one pipeline, one complete post page — author dressed, both comments present, count agreeing with the list (2 = 2; check it, always). This single response is MG.7, MG.8 and MG.10 working together without knowing each other's names. *That* is what the whole series was building toward.

Quick recap

- Design from the reading side: **users / posts / comments**; tags embedded, authors and comments referenced.
- Schemas stack the series: required + types (MG.5), slug hook + timestamps (MG.9), unique indexes (MG.10).
- Thin API doors: POST → **201**, GET → **200**, duplicates → **409** in human words, missing → **404**.
- The full-post read is **one aggregation**: `$match` → `$lookup` author → `$lookup` comments → `$size` count → `$project` the manners.

Common mistakes

- **Embedding the comments in the post:** the first version everyone writes. Ten comments feel fine; the post document then grows forever, every list view drags every comment, and the 16 MB ceiling starts its countdown (MG.8's loudest warning). Comments are born referenced.

- **The query waterfall:** fetch the post, *then* the author, *then* the comments, *then* each commenter's name — five trips where one pipeline (or one populate, MG.9) suffices. Count your round trips like money; users pay for them in waiting.

  (Roman Urdu: Ek sawal, ek safar — database ke paanch chakkar lagana code ki susti hai.)

- **Trusting the request body blindly:** `Post.create(req.body)` lets a caller smuggle fields you never offered (an `author` they are not, a forged `createdAt`). Real APIs pick the allowed fields explicitly. The schema guards the shape; the route must guard the guest list.
- **Leaking internals in responses:** raw documents carry `__v`, password hashes (later series), and emails into public JSON. Project on purpose — Example 3 removed the author's email deliberately. Every field in a response should be there *because you chose it*.

Practice

1. Add the missing door: `GET /users/:id/posts` — one user's posts, newest first. Which pipeline (or populate) do you use, and what exactly does the response hold? Write the route and its sample output.
2. A fourth comment arrives, then a fifth. At what comment count does the Example 3 response become a problem for a phone on slow internet — and what is the fix (hint: the comments array is a list; lists get the MG.4 treatment)?
3. Break it on purpose, on paper: register a second user with `ali@example.com`. Trace the request through the schema, the unique index, the error, and the API's reply — status code and human message — naming the lesson each step comes from.

Practice quiz

1. In the blog design, what is embedded and what is referenced?

- Everything is embedded in posts
- A post embeds its tags (small, read together) and references its author (shared, read alone); comments reference both post and author, because they grow without limit and are never embedded
- Everything is referenced
- Tags are referenced, comments are embedded

Answer: A post embeds its tags (small, read together) and references its author (shared, read alone); comments reference both post and author, because they grow without limit and are never embedded

2. In the blog API, which status answers a duplicate email, a missing slug, a created post and a read?

- 200 for all of them
- Duplicate email hits the unique index and answers 409 in human words, missing slug answers 404, POST creates with 201, GET reads with 200
- 500 for duplicates, 200 for missing
- 409 for created, 201 for missing

Answer: Duplicate email hits the unique index and answers 409 in human words, missing slug answers 404, POST creates with 201, GET reads with 200

3. How is a full post read in the blog project?

- By fetching the post, then the author, then the comments, then each commenter's name, five trips
- In one aggregation pipeline: $match the slug, $lookup the author, $lookup the comments, count them with $size, one trip, one complete answer
- By reading the posts collection only
- It cannot be read in one query

Answer: In one aggregation pipeline: $match the slug, $lookup the author, $lookup the comments, count them with $size, one trip, one complete answer

---

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/mongodb-project-blog · More free tools and notes at https://scopetreasure.com