**ScopeTreasure Courses**

MONGODB COURSES · LESSON MG.9 · CODE SCOPE SERIES

# Mongoose Deep — Schema Ke Aage Ki Duniya

The tools real apps run on: virtuals that compute themselves, middleware hooks that act on their own, and populate that fills references.

## Introduction

MG.5's schema guarded the door. But real apps ask more of Mongoose: a name displayed whole while stored in two parts; a blog slug that must be born with every post, never forgotten; a comment whose `author` field holds only an ObjectId, while the screen needs the author's *name*. Mongoose's deeper toolkit answers all three: **virtuals** (fields that compute themselves), **middleware** (hooks that run on their own at save-time), and **populate** (references filled in at read time). These three are most of what separates a tutorial app from a working one.

*(Roman Urdu: Virtual = hisaab par banna, database mein nahi. Middleware = kaam se pehle khud chal parne wala chowkidar. Populate = number se naam tak ka safar.)*

## Key points

- **A virtual is a field that is never stored:** `studentSchema.virtual('fullName').get(...)` computes from stored fields at read time. The database document holds only `firstName` / `lastName`; `fullName` exists whenever you ask. One source of truth, zero duplication — change the first name and the full name is never stale.
- **Virtuals need one switch to travel:** by default `doc.fullName` works in code, but `toJSON()` / `toObject()` output (what your API sends) leaves virtuals out — pass `{ virtuals: true }` in the schema options to include them. Forgetting this switch is the classic "works in console, missing in API" bug.
- **Middleware = hooks around operations:** `schema.pre('save', fn)` runs *before* every save; `post('save', fn)` runs after. Slugs, `updatedAt` stamps, password hashing (Phase 7's auth lessons), cleanup after delete — if it must *always* happen with an operation, it belongs in a hook, not in every route's memory.
- **Hooks must call the next step:** a pre-hook receives control and must hand it on (finish the async function / call `next()`), or the operation waits forever — a save that hangs with no error is a hook that forgot.
- **Populate fills a reference at read time:** a field declared `{ type: mongoose.Schema.Types.ObjectId, ref: 'User' }` stores only the id. `.populate('author', 'name')` swaps the id for the actual document (only the `name` field, if you say so) in the *results*. The stored data never changes — populate is a read-time `$lookup` wearing a friendlier coat (MG.8's cousin).
- **Validation grows teeth here too:** `enum` (only these values), `match` (a regex the text must pass — emails), custom validators with your own messages. MG.5 named the rule; this level writes rules that read like the business itself ("status can only be draft or published").

## Code examples

### Example 1 — Virtuals: fullName that is never stored

```
const studentSchema = new mongoose.Schema(
  {
    firstName: { type: String, required: true },
    lastName: { type: String, required: true },
    marks: { type: Number, min: 0, max: 100 }
  },
  { toJSON: { virtuals: true } }
);

studentSchema.virtual('fullName').get(function () {
  return this.firstName + ' ' + this.lastName;
});

const Student = mongoose.model('Student', studentSchema);
const sara = await Student.create({ firstName: 'Sara', lastName: 'Khan', marks: 92 });
console.log(sara.fullName);
console.log(JSON.stringify(sara));
```

What it does: The schema stores name in two parts (forms collect them separately) and switches virtuals on for JSON. The virtual getter joins them on demand. `create` saves only the real fields — then both reads use the computed one.

**What you will see:**

```
Sara Khan
{"firstName":"Sara","lastName":"Khan","marks":92,"_id":"650000000000000000000002","__v":0,"ful
lName":"Sara Khan","id":"650000000000000000000002"}
```

The JSON wears `fullName` like a real field — and also `id`, Mongoose's built-in virtual (the `_id` as text). But open the raw document in mongosh and neither exists there: `fullName` is *computed*, never stored. Edit `firstName` tomorrow and every screen shows the new full name instantly — a stored copy could never promise that.

**Example 2 — Middleware: a slug is born with every post**

```
const postSchema = new mongoose.Schema(
  {
    title: { type: String, required: true },
    slug: { type: String, unique: true },
    body: String
  },
  { timestamps: true }
);

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

const Post = mongoose.model('Post', postSchema);
const post = await Post.create({ title: 'Mera Pehla Blog', body: 'Assalam o Alaikum, ye mera pehla blog hai.' });
console.log(post.slug, '|', post.createdAt.toISOString());
```

What it does: The `pre('save')` hook runs before *every* save of a post — if the slug is missing, it builds one from the title (lowercase, spaces to dashes: `mera-pehla-blog`). The `timestamps: true` option adds Mongoose-managed `createdAt` / `updatedAt` automatically. No route code remembered anything; the *schema* guarantees it.

**What you will see:**

```
mera-pehla-blog | 2026-10-24T05:30:00.000Z
```

That slug is the post's future URL (`/posts/mera-pehla-blog`) — clean, automatic, and impossible to forget, because forgetting is exactly what hooks exist to defeat. The same pattern guards passwords in the auth lessons: a pre-save hook hashes whatever arrives, so no code path can store a plain password by accident.

**Example 3 — Populate: the comment meets its author**

```
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 }
});
const Comment = mongoose.model('Comment', commentSchema);

const comments = await Comment.find().populate('author', 'name');
console.log(comments[0]);
```

What it does: A comment stores only ids (MG.8's referencing, in Mongoose clothes) — efficient to save, useless to display. `populate('author', 'name')` follows the `ref` and returns the author document in place of the id, trimmed to just the name. The blog's comment section can now print *"Sara Khan: Bohat aala..."* in one query.

**What you will see:**

```
{
  text: 'Bohat aala, samajh aa gaya!',
  author: { _id: new ObjectId('650000000000000000000302'), name: 'Sara Khan' },
  post: new ObjectId('650000000000000000000401'),
  _id: new ObjectId('650000000000000000000501'),
  __v: 0
}
```

The `author` slot holds a small document now, while `post` — not populated — stays an honest ObjectId. Nothing in the database changed; populate is purely how the answer is *dressed*. Populate exactly the fields the screen needs: every populated path is a lookup the database must run.

## Quick recap

- **Virtuals** compute at read time (`fullName` from parts) — never stored, never stale; `{ toJSON: { virtuals: true } }` lets them travel into API output.
- **Middleware hooks** (`pre`/`post` around save, delete, find…) make guarantees automatic: slugs, timestamps, hashing. A hook that doesn't hand control onward hangs the operation.
- **Populate** fills an ObjectId reference with its document at read time — Mongoose's friendly `$lookup`. Stored data is untouched.
- Deeper validation (`enum`, `match`, custom messages) turns the schema into the business's own rulebook.

## Common mistakes

- **"My virtual vanished from the API":** it works with `doc.fullName`, then the JSON response lacks it. The `virtuals: true` switch in `toJSON` options was off — the number-one virtual bug, and it is a one-line fix.
- **A hook that never finishes:** an `async` pre-hook that throws silently, or an old-style hook that forgets `next()` — saves hang forever with no error anywhere. When a save "just stops", read your hooks first.
  *(Roman Urdu: Save ruk jaye aur error bhi na aaye — pehle shak middleware par karo.)*
- **Populating everything, everywhere:** populate is a database lookup per path per query. Populating author *and* post *and* likes on a 10,000-comment list is 30,000 lookups wearing a trench coat. Populate what the screen shows — MG.8's `$lookup` inside an aggregation is often the better tool for big reports.
- **Storing what a virtual could compute:** a saved `fullName` string drifts the first time someone edits `firstName` in another tool. If a value is *always* derivable from stored fields, derive it — store sources, compute views.

## Practice

1. Add a virtual `result` to Example 1's schema: "Pass" when marks ≥ 40, else "Fail" — as a getter. Why must this be a virtual and not a stored field? (Hint: whose marks change in MG.3?)
2. Write a `pre('save')` hook for the Student schema that trims spaces from `firstName` / `lastName` and capitalises their first letters. Which MG.3-era silent bug does this hook make impossible?
3. A comment list must show each comment's text, its author's name, and its post's *title*. Write the two populate calls (paths + field selections) — and say why populating `post` with all fields would be wasteful.

## Practice quiz

**1. What is a virtual, and why is it never stale?**

- A field stored twice for speed
- A field that is never stored: studentSchema.virtual('fullName') computes from stored fields at read time, so change the first name and the full name is never stale
- A field that exists only in tests
- A deleted field

**Answer:** A field that is never stored: studentSchema.virtual('fullName') computes from stored fields at read time, so change the first name and the full name is never stale

**2. A virtual works as doc.fullName in code but is missing from the API's JSON. What switch was forgotten?**

- The virtual was never defined
- { virtuals: true } in the schema's toJSON options, without which toJSON and toObject output leave virtuals out
- The model name is wrong

- Virtuals cannot travel into JSON at all

**Answer:** { virtuals: true } in the schema's toJSON options, without which toJSON and toObject output leave virtuals out

**3. What does .populate('author', 'name') do, and what does it change in storage?**

- It writes the author document into the post permanently
- It fills an ObjectId reference with its actual document at read time, only the name field if you say so; the stored data never changes
- It deletes the reference
- It only works on arrays

**Answer:** It fills an ObjectId reference with its actual document at read time, only the name field if you say so; the stored data never changes

---

**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-mongoose-deep · More free tools and notes at https://scopetreasure.com