**Is it good practice to perform three layers of validation?**  
Yes — it’s a best practice known as **defense in depth**:

* **Client (React):** Perform UX-level checks (e.g., required fields, minimum length) to catch obvious errors early and improve the user experience.
* **API layer (Express):** Trust nothing from the client. Validate request bodies, parameters, and queries before executing business logic.
* **Database layer (MongoDB/Mongoose):** Act as the final safeguard. Enforce schema rules and constraints to maintain data integrity, even if earlier validations fail.

Doing all three ensures better security, data quality, and resilience against bad or malicious inputs.

1. Mongoose Schema Validation (DB layer)
   1. Common built-ins

import mongoose from 'mongoose';

/\*\*

\* IMPORTANT NOTES ON VALIDATION:

\* - Mongoose runs validators before writing to MongoDB on operations like .save(), .create(), and

\* any update operations that have { runValidators: true }.

\* - If a validator fails, Mongoose throws a ValidationError and aborts the write.

\* - Some options are "setters" (e.g., lowercase, trim) that transform the value before validation.

\* - `unique: true` is NOT a validator; it asks MongoDB to create a unique index (DB-level constraint).

\*/

const userSchema = new mongoose.Schema({

// ---- name ----

name: {

type: String, // Must be a string; type-cast attempts happen before validation.

required: [true, 'Name is required'],

// required → validation: fails if the key is missing or value is null/undefined/empty string.

// Custom message 'Name is required' is used in ValidationError.

trim: true, // SETTER: removes leading/trailing spaces \*before\* other validators.

minlength: [2, 'Name too short'], // validation: length >= 2

maxlength: [50, 'Name too long'] // validation: length <= 50

},

// ---- email ----

email: {

type: String,

required: true, // validation: must be present (truthy); message defaults from Mongoose if not array form.

lowercase: true, // SETTER: converts to lowercase before validation/save.

unique: true, // DB INDEX, not a validator → ensures no duplicate values at the database level.

// If duplicates exist before index creation, inserts may still fail at the DB with E11000 duplicate key error.

match: [/^\S+@\S+\.\S+$/, 'Invalid email']

// validation: value must match the regex. If it doesn't, ValidationError with 'Invalid email'.

},

// ---- age ----

age: {

type: Number, // type-cast to Number if possible; else CastError before validators.

min: [13, 'Must be 13+'], // validation: numeric value must be >= 13

max: [120, 'Age unrealistic'] // validation: numeric value must be <= 120

},

// ---- role ----

role: {

type: String,

enum: { // validation: value must be one of the listed allowed values.

values: ['user', 'admin'],

message: '{VALUE} not allowed' // if 'guest' → "guest not allowed"

},

default: 'user' // if not provided, set to 'user' before validation/save.

},

// ---- passwordHash ----

passwordHash: {

type: String,

required: true, // validation: must be present; typically a bcrypt hash string.

select: false // QUERY BEHAVIOR: excluded by default from query results.

// Not a validator; prevents accidental exposure:

// User.find() → no passwordHash

// User.find().select('+passwordHash') → explicitly include it

},

// ---- tags ----

tags: {

type: [String], // Array of strings. Each element will be cast to String.

validate: {

// Custom validator runs on the whole array. If the function returns false, it fails.

validator: (arr) => Array.isArray(arr) && arr.length <= 10,

message: 'Max 10 tags' // ValidationError message when array length > 10 (or not an array).

}

// Note: You could add per-item validators via "validate" on the array or use a sub-schema.

}

}, { timestamps: true }); // createdAt/updatedAt auto-managed by Mongoose (set before save/update).

/\*\*

\* HOW VALIDATION TRIGGERS IN PRACTICE

\*

\* 1) On document save/create:

\* try {

\* await User.create({

\* name: ' Jo ', // trim -> 'Jo', passes minlength (2)

\* email: 'User@Example.COM', // lowercase -> 'user@example.com', passes regex

\* age: 25, // within [13, 120]

\* role: 'user', // in enum

\* passwordHash: '...bcrypt...', // required present

\* tags: ['node', 'mongo'] // array length 2 <= 10

\* });

\* } catch (err) {

\* // err is ValidationError if any validator failed.

\* // err.errors.<path>.message contains the message you defined (e.g., 'Name is required').

\* }

\*

\* 2) On updates (findOneAndUpdate/updateOne/etc.):

\* - Validators DO NOT run by default.

\* - Use { runValidators: true, context: 'query' } to enforce them:

\* await User.updateOne({ \_id }, { age: 10 }, { runValidators: true });

\* // → fails with 'Must be 13+'

\*

\* 3) Unique index collisions:

\* - `unique: true` creates an index; duplicates throw a MongoServerError (E11000), not a ValidationError.

\* - Handle separately:

\* try { ... } catch (err) {

\* if (err.code === 11000) { /\* duplicate key on unique index \*/ }

\* }

\*

\* 4) Casting vs Validation:

\* - Casting (type conversion) happens first. If casting fails (e.g., age: 'abc'), you get a CastError.

\* - If casting succeeds, then validators run; failing them yields a ValidationError.

\*/

/\*\* OPTIONAL: MODEL \*/

const User = mongoose.model('User', userSchema);

export default User;

/\*\*

\* QUICK PASS/FAIL EXAMPLES

\*

\* ✅ Pass:

\* { name: 'Ana', email: 'ana@example.com', age: 30, role: 'admin', passwordHash: '...', tags: [] }

\*

\* ❌ Fail (missing required):

\* { email: 'a@b.com', passwordHash: '...' } // name required → ValidationError

\*

\* ❌ Fail (format):

\* { name: 'John', email: 'not-an-email', passwordHash: '...' } // match fails → 'Invalid email'

\*

\* ❌ Fail (bounds):

\* { name: 'John', email: 'j@e.com', age: 9, passwordHash: '...' }// min fails → 'Must be 13+'

\*

\* ❌ Fail (enum):

\* { name: 'John', email: 'j@e.com', role: 'guest', passwordHash:'...' } // 'guest not allowed'

\*

\* ❌ Fail (custom validator):

\* { name: 'John', email: 'j@e.com', passwordHash:'...', tags: new Array(11).fill('x') }

\* // → 'Max 10 tags'

\*/

Custom Validation –

**1️⃣ Clean version of your example**

js

CopyEdit

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

age: {

type: Number,

required: true,

validate: {

validator: function (value) {

return value >= 18 && value <= 65; // custom check

},

message: 'Age must be between 18 and 65'

}

}

});

const User = mongoose.model('User', userSchema);

**Key points:**

* Use validate: (lowercase), not Validate:.
* validator can be a normal or arrow function. If you need to use this (document), use a normal function.
* message is shown when the check fails.

**2️⃣ Testing the validator**

js

CopyEdit

(async () => {

try {

await User.create({ age: 17 }); // fails

} catch (err) {

console.log(err.message); // "User validation failed: age: Age must be between 18 and 65"

}

try {

await User.create({ age: 30 }); // passes

console.log('Saved successfully!');

} catch (err) {

console.log(err.message);

}

})();

**3️⃣ Why use custom validators**

* When **built-in rules** (min, max, enum, match) are not enough.
* When you need **multiple conditions** in one check.
* When you need **cross-field validation** (this gives access to other fields).
* Async field validation

## 4️⃣ Upgrade ideas to practice

### **Multiple rules on the same field**

js

CopyEdit

age: {

type: Number,

validate: [

{ validator: v => Number.isInteger(v), message: 'Age must be an integer' },

{ validator: v => v >= 18 && v <= 65, message: 'Age must be 18–65' }

]

}

### **Cross-field**

js

CopyEdit

discount: {

type: Number,

validate: {

validator: function (v) {

return v < this.price; // compare with another field of t

he same document or record or object .dont use arrow function else this wont point to the same document.

},

message: 'Discount must be less than price'

}

}

### **Async check**

js

CopyEdit

email: {

type: String,

validate: {

validator: async function (value) {

const count = await mongoose.model('User').countDocuments({ email: value });

return count === 0; // true = valid

},

message: 'Email already exists'

}

}

Async custom validation –

const mongoose = require('mongoose');

const { MongoMemoryServer } = require('mongodb-memory-server');

let mongod;

async function main() {

// Start in-memory MongoDB and connect

mongod = await MongoMemoryServer.create();

await mongoose.connect(mongod.getUri());

// Schema with async custom validator

const userSchema = new mongoose.Schema({

email: {

type: String,

required: true,

validate: {

validator: async function (value) {

const exists = await mongoose.model('User').exists({

email: value,

\_id: { $ne: this.\_id } // ignore self

});

return !exists; // valid if not found

},

message: 'Email already exists'

}

}

});

const User = mongoose.model('User', userSchema);

// Test: First insert -> pass

await User.create({ email: 'a@site.com' });

console.log('First insert passed ✅');

// Test: Duplicate insert -> fail

try {

await User.create({ email: 'a@site.com' });

} catch (err) {

console.log('Duplicate insert failed as expected ❌:', err.message);

}

// Clean up

await mongoose.disconnect();

await mongod.stop();

}

main();

/\*

validate: {

validator: async function (value) {

const exists = await mongoose.model('User').exists({

email: value,

\_id: { $ne: this.\_id } // ignore self

});

return !exists; // valid if not found

},

message: 'Email already exists'

}

1. validator: async function (value) { ... }

This is your custom validator function for the email field in a Mongoose schema.

Mongoose automatically calls this function before saving or updating a document.

value is the actual value of the field being validated — here, it’s the email you’re trying to save.

It’s marked async so you can do database queries inside it (which return Promises).

2. const exists = await mongoose.model('User').exists({ ... })

This queries the database for any User document that matches the given filter.

.exists() is a fast way to check if at least one matching document exists —

It returns:

null if nothing matches (no duplicate)

{ \_id: ... } if something matches (duplicate found)

Here the filter is:

{

email: value,

\_id: { $ne: this.\_id }

}

email: value → match documents with the same email.

\_id: { $ne: this.\_id } → $ne means “not equal to”.

This ignores the current document itself when validating on updates.

Without it, updating your own record without changing the email would wrongly trigger a "duplicate" error.

3. return !exists;

exists will be either null or a document.

!exists:

true → valid (no duplicate found)

false → invalid (duplicate found)

Mongoose expects the validator to return:

true → passes validation

false → fails validation

4. message: 'Email already exists'

If the validator returns false, Mongoose throws a ValidationError.

This message is what gets attached to the error for the email field.

\*/

# What is API-level validation?

Checks done **in Express before your controller/business logic runs**.  
You validate req.body, req.params, and req.query so only **safe, well-formed** data hits your code or DB.

API-level validation + sanitisation-

**>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>**