Introduction to MongoDB & Mongoose

What is MongoDB?

- A **NoSQL, document-oriented** database.
- Stores data in **flexible ON-like documents** (BSON format).
- No rigid schema (unlike SQL databases).
- Uses **collections** (like tables) and **documents** (like rows).

Example Document:

```
{
  "_id": "123",
  "name": "Ahmed",
  "age": 25,
  "skills": ["JavaScript", "Node."]
}
```

What is Mongoose?

- An **ODM (Object Data Modeling)** library for MongoDB in Node..
- Acts as a **translator** between JavaScript and MongoDB.
- Provides:
    - Schema definitions.
    - Data validation.
    - Query building.
    - Middleware & hooks.

Schemas & Models

Schema

- A **blueprint** for documents in a collection.
- Defines structure, data types, and validation.

Example:

```
const userSchema = new mongoose.Schema
({
```

```
  name: String,
  age: Number,
  email: { type: String, required: true }
});
```

Model

- A **class** that interacts with the MongoDB collection.
- Created from a schema.

Example:

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

- `'User'` → Collection name (automatically pluralized: `users`).

---

4. Data Types & Validation

Common Data Types

- **String, Number, Boolean, Date, Array, ObjectId** (for relationships).

Validation Rules

Rule	Description	Example
required	Field must be provided	`{ type: String, required: true }`
min / max	Min/max value for numbers	`{ age: { type: Number, min: 18 } }`
minLength / maxLength	Min/max string length	`{ name: { type: String, minLength: 3 } }`
match	Regex pattern validation	`{ email: { type: String, match: /@/ } }`
default	Default value if not provided	`{ isActive: { type: Boolean, default: true } }`

Example Schema with Validation:

```
const userSchema = new mongoose.Schema
({
  name: { type: String, required: true, minLength: 3 },
  email: { type: String, match: /.+\@.+\..+/, required: true },
  age: { type: Number, min: 18, max: 60 },
  isActive: { type: Boolean, default: true }
});
```

- A **regex (regular expression)** that enforces a basic email format:
  - `.+` → At least one character before `@`.
  - `\@` → Must contain the `@` symbol.
  - `.+` → At least one character after `@` (domain name).
  - `\.` → Must contain a `.` (dot) for the domain extension (e.g., `.com`).

---

5. CRUD Operations

Create (C)

```
const newUser = new User ({
  name: 'Ali',
  email: 'ali@example.com',
  age: 25
});

newUser.save()
  .then(user => console.log('Saved:', user))
  .catch(err => console.error('Error:', err));


Or


User.create({ name: 'Ali', age: 25 });
```

Read (R)

Find all documents

```
User.find();
```

➤ Retrieves all users.

Find by ID

```
User.findById('id_here');
```

➤ Finds a user by their unique `_id`.

Find one matching document

```
User.findOne({ name: 'Ali' });
```

➤ Returns the first user with the name 'Ali'.

Find with condition

```
User.find({ age: { $gte: 18 } });
```

➤ Finds users aged 18 or older.

Find selected fields

```
User.find().select('name age');
```

➤ Retrieves only name and age fields.

Sort results

```
User.find().sort({ age: -1 });
```

➤ Sorts users by age in descending order.

Limit results

```
User.find().limit(5);
```

➤ Returns only 5 users.

Skip results

```
User.find().skip(5);
```

➤ Skips the first 5 users.

Populate referenced data

```
Post.find().populate('user');
```

➤ Replaces `user` ID with full user document.

Find with multiple conditions (AND by default)

```
User.find(
       { name: 'Ali',
       age: { $gte: 18 }
});
```

➤ Finds users named "Ali" **and** age ≥ 18.

Find with OR condition

```
User.find({
$or:
[
       { age: { $lt: 18 } },
       { role: 'admin' }
]});
```

➤ Finds users who are either under 18 or admins.

Find documents with field existence check

```
User.find({ email: { $exists: true } });
```

➤ Finds users who have the email field.

Find documents and use select() to exclude fields

```
User.find().select('-password');
```

➤ Retrieves all fields **except** password.

Find documents with pagination

```
User.find().skip(10).limit(10);
```

➤ Used for paginating results — skip 10 and get the next 10 users.

Chain Read Operations

You can combine many of these:

```
User.find({ active: true })
  .select('name email')
  .sort({ createdAt: -1 })
  .skip(10)
  .limit(5)
  .lean();
```

➤ Gets 5 active users, newest first, skips 10, selects name & email only.

```
User.find({ age: { $ne: 30 } }) //$ne: Not equal to
User.find({ age: { $gt: 18 } }) //$gt: Greater than
User.find({ age: { $gte: 18 } }) //$gte: Greater than or equal to
User.find({ age: { $lt: 30 } }) $lt: Less than
User.find({ age: { $lte: 30 } }) $lte: Less than or equal to
User.find({ age: { $in: [20, 25, 30] } }) $in
User.find({ age: { $nin: [20, 25, 30] } }) $nin: Not in
```

Sort by one field ascending

```
User.find().sort({ age: 1 });
```

➤ Sorts users by age in **ascending** order (youngest to oldest).

Sort by one field descending

```
User.find().sort({ age: -1 });
```

➤ Sorts users by age in **descending** order (oldest to youngest).

Sort by multiple fields

```
User.find().sort({ age: 1, name: -1 });
```

➤ Sorts by age ascending; if same age, then by name descending.

Sort by createdAt (most recent first)

```
User.find().sort({ createdAt: -1 });
```

➤ Shows newest users first based on creation time.

Sort by updatedAt (oldest first)

```
User.find().sort({ updatedAt: 1 });
```

➤ Shows oldest updated users first.

Sort with chaining and limit

```
User.find().sort({ score: -1 }).limit(3);
```

➤ Gets top 3 users with highest scores.

Update (U)

◆ Update one document

```
User.updateOne({ name: "Ali" }, { $set: { age: 25 } });
```

➤ Updates age to 25 for the first user named "Ali".

◆ Update multiple documents

```
User.updateMany({ role: "user" }, { $set: { active: true } });
```

➤ Activates all users with the role "user".

◆ Update by ID

```
User.findByIdAndUpdate("id_here", { $set: { name: "Ahmed" } });
```

➤ Updates the name of the user with the given ID.

◆ Replace a document completely

```
User.replaceOne({ _id: "id_here" }, { name: "Zain", age: 30 });
```

➤ Replaces the entire document with new data.

◆ Update with increment

```
User.updateOne({ name: "Ali" }, { $inc: { age: 1 } });
```

➤ Increases age by 1 for user "Ali".

◆ Find and update, then return the new document

```
User.findOneAndUpdate(
       { name: "Ali" },
       { $set: { age: 30 } }, { new: true }
);
```

➤ Finds Ali, updates age, and returns the updated document.

◆ Add to array field

```
User.updateOne({ name: "Ali" }, { $push: { hobbies: "reading" } });
```

➤ Adds "reading" to the hobbies array for Ali.

Delete (D)

◆ Delete one document

```
User.deleteOne({ name: "Ali" });
```

➤ Deletes the first user with the name "Ali".

◆ Delete multiple documents

```
User.deleteMany({ active: false });
```

➤ Deletes all users who are not active.

◆ Delete by ID

```
User.findByIdAndDelete("id_here");
```

➤ Deletes the user with the given ID.

◆ Find and delete a matching document

```
User.findOneAndDelete({ name: "Ali" });
```

➤ Finds and deletes the first user named "Ali".

---

6. Relationships

1:1 (User ↔ Profile)

```
// User Schema
const userSchema = new mongoose.Schema({
  name: String,
  profile: { type: mongoose.Schema.Types.ObjectId, ref: 'Profile' }
});

// Profile Schema
const profileSchema = new mongoose.Schema({
  bio: String,
  user: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }
});
```

1:Many (User ↔ Posts)

```
// Post Schema
const postSchema = new mongoose.Schema({
  title: String,
  user: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }
});
```

Many:Many (Students ↔ Courses)

```
// Student Schema
const studentSchema = new mongoose.Schema({
  name: String,
  courses: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Course' }]
});

// Course Schema
const courseSchema = new mongoose.Schema({
  title: String,
  students: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Student' }]});
```

7. Advanced Querying

◆ Basic populate

```
Post.find().populate('user');
```

➤ Replaces `user` ID in each post with full user document.

◆ Populate specific fields

```
Post.find().populate('user', 'name email');
```

➤ Populates only `name` and `email` fields of the user.

◆ Populate nested references

```
Comment.find().populate({
  path: 'post',
  populate: { path: 'user', select: 'name' }
});
```

➤ Populates post, and within post, populates the user's name.

◆ Populate with conditions

```
Post.find().populate({
  path: 'comments',
  match: { approved: true }
});
```

➤ Populates only approved comments in each post.

◆ Populate multiple fields

```
Order.find().populate('customer').populate('products');
```

➤ Populates both `customer` and `products` fields.

Timestamps in Mongoose

Mongoose can automatically track **createdAt** and **updatedAt**.

◆ Enable timestamps in schema

```
const userSchema = new mongoose.Schema({
  name: String,
  age: Number
}, { timestamps: true });
```

➤ Adds automatic `createdAt` and `updatedAt` fields to each document.

◆ Query with timestamp condition

```
User.find({ createdAt: { $gte: new Date('2024-01-01') } });
```

➤ Finds users created on or after Jan 1, 2024.

◆ Sort by latest created

```
Post.find().sort({ createdAt: -1 });
```

➤ Gets posts sorted from newest to oldest.