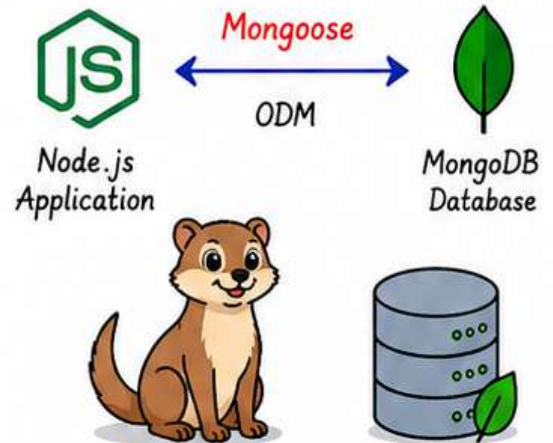


Chapter 16: Mongoose

16.1 Introduction to Mongoose

16.1.1 What is Mongoose?

- * Mongoose is an Object Data Modeling (ODM) library for **MongoDB** and **Node.js**.
- * It provides a straight-forward, **schema-based** solution to model application data.
- * Mongoose allows us to define **schemas**, **validate** data, and interact with MongoDB using **models**.
- * It acts as a **bridge** between our **Node.js** application and **MongoDB** database.



Key Points:

- ✓ Mongoose is an **ODM** (Object Data Modeling) library.
- ✓ It is built on top of the native **MongoDB driver**.
- ✓ It provides **schema validation**, **middleware**, and easy data handling.
- ✓ It makes working with MongoDB in **Node.js** simple and efficient.



16.1.2 Why Use Mongoose?

- * MongoDB is **schema-less**, but in real applications we need structure and validation.
- * Mongoose helps us define schema and **enforce data rules**.
- * It provides methods to perform **CRUD** operations easily.
- * It supports **middleware**, **virtuals**, **relationships** and more advanced features.



Example:

```
// Import mongoose
const mongoose = require('mongoose');
// Connect to MongoDB
mongoose.connect('mongodb://127.0.0.1:27017/mydb')
.then(() => console.log('Connected to MongoDB'))
.catch(err => console.log('Connection Error:', err));
```

Explanation:

- We import **mongoose** library.
- **mongoose.connect()** connects to MongoDB database.
- **.then()** runs when connection is successful.
- **.catch()** handles connection errors.



16.1.3 Benefits of Mongoose

Schema-Based
Define structure and enforce validation.



Easy to Use
Simple API to perform database operations.



Validation
Built-in validation ensures data integrity.



Middleware
Pre and post hooks for custom logic.



Relationships
Supports references and population.



Key Takeaway:



Mongoose simplifies working with **MongoDB** in **Node.js** by providing **schemas**, **validation**, and powerful features that help us build **robust** and **scalable** applications.



16.2 Installation

16.2.1 Prerequisites

Before installing Mongoose, make sure you have:

- * **Node.js** installed on your system.
- * **npm** (comes with Node.js).
- * A **MongoDB** database (local or cloud like MongoDB Atlas).
- * A basic **Node.js** project set up.



Node.js



npm



MongoDB



Node.js
Project

16.2.2 Installing Mongoose

- * We can install Mongoose using **npm**.

Example:

```
# Install Mongoose using npm
npm install mongoose
```

Key Points:

- ✓ This will add **mongoose** to your project.
- ✓ It will be listed in **package.json** under dependencies.
- ✓ You can now **require** and use **mongoose** in your project.



16.2.3 Verifying Installation

We can verify if Mongoose is installed correctly.

Example:

```
# Check installed mongoose version
npm list mongoose
# OR
node -e "console.log(require('mongoose').version)"
```

Explanation:

- **npm list mongoose** shows the installed version.
- The second command prints the **mongoose** version in your terminal.
- If version is displayed, installation is successful.



16.2.4 Adding Mongoose to Project

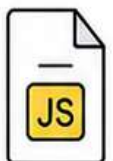
After installation, import mongoose in your project.

Example:

```
// Import mongoose
const mongoose = require('mongoose');
// Now you can use mongoose
```

Key Points:

- ✓ Use **require('mongoose')** to import.
- ✓ Store it in a variable (commonly **mongoose**).
- ✓ Now you can connect to MongoDB and define schemas and models.



Key Takeaway:



Installing Mongoose is simple using **npm**. Ensure you have **Node.js** and **MongoDB** ready, then install, verify and import **mongoose** to start working with your database in **Node.js**.



16.3 MongoDB Connection

16.3.1 What is Connection?

- * Before performing any operation, we need to **connect** our application to MongoDB database.
- * Mongoose provides an easy way to connect to MongoDB using a **connection string (URI)**.
- * Once connected, we can perform CRUD operations on our database.



16.3.2 Connection String (URI)

- * MongoDB connection is made using a **connection string (URI)**.
- * It contains information like protocol, host, port, database name and options.
- * Example of a local MongoDB URI:

```
mongodb://127.0.0.1:27017/mydatabase
```

Key Points:

- ✓ **127.0.0.1** - localhost
- ✓ **27017** - default MongoDB port
- ✓ **mydatabase** - name of the database



16.3.3 Connecting Using Mongoose

- * We use **mongoose.connect()** method to connect to MongoDB.

Example:

```
const mongoose = require('mongoose');

mongoose.connect('mongodb://127.0.0.1:27017/mydatabase')
  .then(() => console.log('Connected to MongoDB'))
  .catch(err => console.log('Connection Error:', err));
```

Explanation:

- **mongoose.connect()** takes the URI as argument.
- **.then()** executes when connection is successful.
- **.catch()** executes if there is any error.



16.3.4 Handling Connection Events

- * Mongoose provides events to handle connection status.

Example:

```
mongoose.connection.on('connected', () => {
  console.log('Mongoose connected to MongoDB');
});
mongoose.connection.on('error', (err) => {
  console.log('Mongoose connection error:', err);
});
mongoose.connection.on('disconnected', () => {
  console.log('Mongoose disconnected');
});
```

Event	Description
connected	Fired when Mongoose successfully connects to MongoDB.
error	Fired when a connection error occurs.
disconnected	Fired when Mongoose disconnects from MongoDB.



Key Takeaway:



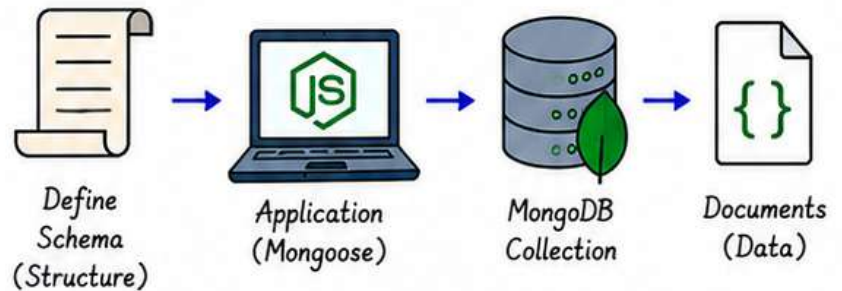
Connecting to MongoDB using Mongoose is **simple** and **powerful**. Use the correct URI, handle connection events and check for errors to build **reliable** applications.



16.4 Schemas

16.4.1 What is a Schema?

- * A schema defines the **structure** of documents in a MongoDB collection.
- * It specifies the **fields**, their **data types**, **default values**, and **validation rules**.
- * Schemas help in organizing data and ensuring data **consistency**.
- * In Mongoose, schemas are defined using `mongoose.Schema()`.



16.4.2 Defining a Schema

- * We define a schema by specifying **field names** and their **properties**.

Example:

```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: { type: Number, min: 0 },
  createdAt: { type: Date, default: Date.now }
});
```

Key Points:

- ✓ `mongoose.Schema()` is used to create a schema.
- ✓ Each field has a name and set of options.
- ✓ `required: true` ⇒ field is mandatory.
- ✓ `unique: true` ⇒ value must be unique.
- ✓ `default` ⇒ sets default value if not provided.
- ✓ `min` ⇒ sets minimum value for numbers.



16.4.3 Common Schema Field Types

Field Type	Description	Example
String	Represents text data.	<code>{ type: String }</code>
Number	Represents numeric data.	<code>{ type: Number }</code>
Date	Represents date and time.	<code>{ type: Date }</code>
Boolean	Represents true or false.	<code>{ type: Boolean }</code>
Array	Represents an array of values.	<code>{ type: [String] }</code>
Object	Represents nested data.	<code>{ type: Object }</code>



Example (Different Types):

```
const productSchema = new mongoose.Schema({
  title: { type: String, required: true },
  price: { type: Number, required: true },
  inStock: { type: Boolean, default: true },
  tags: { type: [String] },
  info: {
    description: { type: String },
    rating: { type: Number, min: 0, max: 5 }
  }
});
```

16.4.4 Creating a Model from Schema

- * A model is a wrapper around the schema. It allows us to interact with the collection.

Example:

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true }
});
const User = mongoose.model('User', userSchema);
// 'User' is the model name and also the collection name (pluralized)
```

Key Points:

- ✓ `mongoose.model(modelName, schema)` creates a model.
- ✓ The model is used to perform CRUD operations.
- ✓ The collection name is automatically created (usually plural of model name).



Key Takeaway:



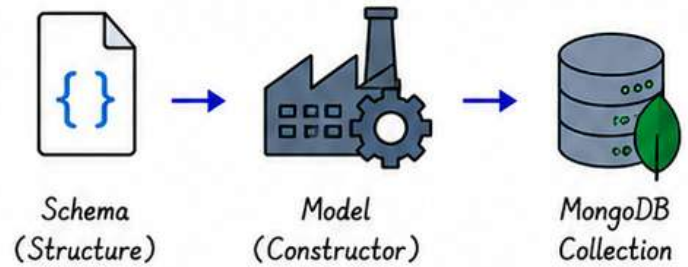
Schemas in Mongoose define the **structure** of our data. They help enforce **validation**, set **defaults**, and **organize** data effectively in MongoDB collections.



16.5 Models

16.5.1 What is a Model?

- * A model is a **constructor** compiled from a **schema**.
- * Each model maps to a **MongoDB collection**.
- * Models provide an interface to perform **CRUD** operations on the collection.
- * In Mongoose, we create a model using **mongoose.model()**.



16.5.2 Creating a Model

- * We pass the **schema** and a **collection name** to create a model.

Example:

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: { type: Number, min: 0 }
}, { timestamps: true });

const User = mongoose.model('User', userSchema, 'users');
// 'users' is the collection name in MongoDB
```

Key Points:

- ✓ **mongoose.model(modelName, schema, collectionName)** creates a model.
- ✓ **modelName** is used to interact with the collection.
- ✓ **collectionName** is optional; if not provided, Mongoose creates a collection name automatically (pluralized and lowercased form of model name).
- ✓ The model is the entry point for database operations.



16.5.3 Models and CRUD Operations

- * Models provide methods to perform **CRUD** operations on the collection.

Operation	Description	Example Method	Example Usage
Create	Insert new documents	<code>Model.create()</code>	<code>User.create({ name: 'John', 'john@gmail.com' })</code>
Read	Find documents	<code>Model.find()</code> , <code>Model.findById()</code>	<code>User.find({})</code> , <code>User.findById(id)</code>
Update	Update existing documents	<code>Model.updateOne()</code> , <code>Model.findByIdAndUpdate()</code>	<code>User.updateOne({ name: 'John' }, { age: 25 })</code>
Delete	Delete documents	<code>Model.deleteOne()</code> , <code>Model.findByIdAndDelete()</code>	<code>User.deleteOne({ name: 'John' })</code>



16.5.4 Example: Using Model

Example:

```
// Create a new user
const newUser = new User({ name: 'Alice', email: 'alice@gmail.com', age: 22 });
newUser.save()
  .then(() => console.log('User saved successfully'))
  .catch(err => console.log('Error:', err));

// Find all users
User.find({})
  .then(users => console.log(users))
  .catch(err => console.log('Error:', err));
```

Explanation:

- **new User()** creates a new document.
- **.save()** inserts it into the collection.
- **find()** retrieves all documents.
- All methods return a **promise**.



Key Takeaway:



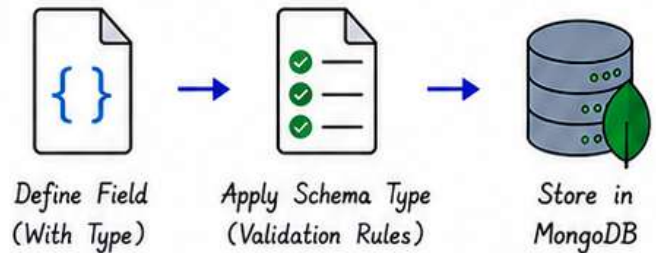
Models are the heart of Mongoose. They are built from **schemas** and provide powerful methods to perform **CRUD** operations, making interaction with MongoDB **simple** and **efficient**.



16.6 Schema Types

16.6.1 What are Schema Types?

- * Schema types define the **type of data** that a field can hold.
- * Mongoose provides **built-in schema types** for common JavaScript values.
- * We can also create **custom types** if needed.
- * Using the right data type helps in **data validation**, **storage** and **retrieval**.



16.6.2 Built-in Schema Types

- * Mongoose provides several built-in schema types:

Type	Description	Example	Default Value
String	Represents text data.	{ type: String }	'' (empty string)
Number	Represents numeric data.	{ type: Number }	0
Boolean	Represents true or false.	{ type: Boolean }	false
Date	Represents date and time.	{ type: Date }	Date.now
Buffer	Represents binary data.	{ type: Buffer }	undefined
ObjectId	Represents MongoDB ObjectId.	{ type: mongoose.Schema.Types.ObjectId }	Auto-generated
Array	Represents an array of values.	{ type: [String] }	[] (empty array)
Mixed	Allows any type of data.	{ type: mongoose.Schema.Types.Mixed }	undefined
Map	Represents key-value pairs.	{ type: Map, of: String }	{ } (empty map)
Decimal128	Represents high precision decimal.	{ type: mongoose.Schema.Types.Decimal128 }	null

Key Points:

- ✓ Choose the correct type based on the data you want to store.
- ✓ Using proper types ensures data integrity and validation.
- ✓ Default values are used when no value is provided.
- ✓ ObjectId is mainly used for references and relationships.

16.6.3 Example: Using Different Types

Example:

```
const productSchema = new mongoose.Schema({
  name: { type: String, required: true },
  price: { type: Number, required: true },
  inStock: { type: Boolean, default: true },
  createdAt: { type: Date, default: Date.now },
  tags: { type: [String] },
  meta: { type: mongoose.Schema.Types.Mixed },
  owner: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }
});
```

Explanation:

- **String** - used for text fields.
- **Number** - used for numbers.
- **Boolean** - used for true/false values.
- **Date** - stores date and time.
- **Array** - stores multiple values.
- **Mixed** - allows any data type.
- **ObjectId** - used to reference another document.

16.6.4 Custom Schema Types

- * We can create custom schema types by extending built-in types.
- * Custom types help in adding custom validation and behavior.

Example:

```
const emailSchemaType = new mongoose.SchemaType({
  type: String,
  validate: {
    validator: function(v) { return /.+@.+.+/.test(v); },
    message: props => `${props.value} is not a valid email!`
  }
});
```

Key Points:

- ✓ Built-in types handle most scenarios.
- ✓ Custom types are useful for complex validation and business rules.
- ✓ Always use correct types to make your application robust and efficient.

Key Takeaway:



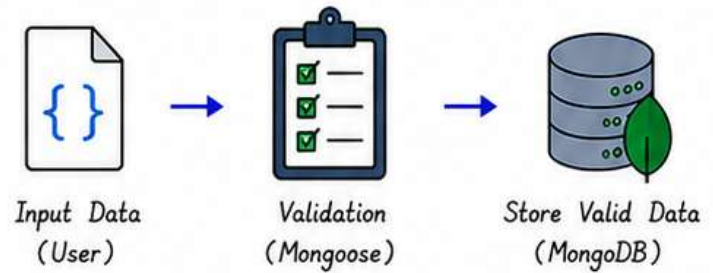
Schema types define the **structure** and **type of data** in Mongoose. Using the right types ensures **data integrity**, **validation** and **efficient** database operations.



16.7 Validation

16.7.1 What is Validation?

- * Validation is the process of ensuring that the data stored in the database is **correct** and **meaningful**.
- * Mongoose allows us to define **validation rules** in schemas.
- * If the data violates the rules, Mongoose throws a **validation error**.
- * Validation helps maintain **data integrity** and prevents invalid data from being stored.



16.7.2 Built-in Validation Rules

- * Mongoose provides several built-in validation rules.

Rule / Validator	Description	Example	Example Usage
required	Field is mandatory.	required: true	name: { type: String, required: true }
min	Sets minimum value for numbers.	min: 18	age: { type: Number, min: 18 }
max	Sets maximum value for numbers.	max: 100	age: { type: Number, max: 100 }
minLength	Minimum length for strings.	minLength: 3	name: { type: String, minLength: 3 }
maxLength	Maximum length for strings.	maxLength: 50	name: { type: String, maxLength: 50 }
enum	Allows only values from a list.	enum: ['M', 'F']	gender: { type: String, enum: ['M', 'F'] }
match	Matches a value with a regex.	match: /regex/	email: { type: String, match: /.+@.+/ }
validate	Custom validation function.	validate: fn	age: { type: Number, validate: fn }

Key Points:

- ✓ Validation runs before saving data.
- ✓ If validation fails, an error is returned.
- ✓ Helps prevent invalid or unexpected data.
- ✓ Built-in validators cover most common scenarios.



16.7.3 Example: Schema with Validation

Example:

```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
  name: { type: String, required: true, minLength: 3, maxLength: 30 },
  email: { type: String, required: true, match: /.+@.+.+/ },
  age: { type: Number, min: 18, max: 100 },
  gender: { type: String, enum: ['M', 'F'], required: true }
});
const User = mongoose.model('User', userSchema);
```

Explanation:

- **name** is required and length between 3 and 30.
- **email** must be in valid email format.
- **age** must be between 18 and 100.
- **gender** accepts only 'M' or 'F'.
- If any rule is violated, validation error will be returned.



16.7.4 Custom Validation

- * We can create custom validation rules using the **validate** property.

Example:

```
const productSchema = new mongoose.Schema({
  price: { type: Number, required: true, min: 0 },
  stock: { type: Number, required: true, validate: {
    validator: function(v) { return v % 1 === 0; },
    message: props => '{props.value} must be an integer'
  }
  }
});
```

Key Points:

- ✓ Custom validators provide flexibility for complex conditions.
- ✓ **validator** is a function that returns **true** or **false**.
- ✓ **message** is shown when validation fails.
- ✓ Use custom validation when built-in rules are not enough.



Key Takeaway:



Validation in Mongoose ensures that only **correct** and **meaningful** data is stored in MongoDB. Use built-in validators for common rules and custom validators for advanced requirements to maintain **data integrity** and build **reliable applications**.



16.8 Default Values

16.8.1 What are Default Values?

- * Default values are used to provide a value to a field when **no value** is specified.
- * If a field has a **default value**, Mongoose automatically sets that value when a new document is created.
- * They help in reducing the amount of data we need to provide and ensure **consistency**.
- * Defaults can be **fixed values** or **dynamic values** (using functions).



16.8.2 Using Default Values in Schema

- * We use the **default** property in a schema to set default values.

Example:

```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true },
  role: { type: String, default: 'user' },
  isActive: { type: Boolean, default: true },
  createdAt: { type: Date, default: Date.now }
});
const User = mongoose.model('User', userSchema);
```

Key Points:

- ✓ **default** sets a value if none is provided.
- ✓ It can be a **static value** (string, number, boolean, etc.) or a function (for **dynamic values**).
- ✓ Defaults are applied only when creating **new** documents.
- ✓ If a value is provided, the default value is ignored.



16.8.3 Types of Default Values

Default Type	Description	Example in Schema	Result if Not Provided
Static Value	A fixed value (string, number, boolean, etc.).	<code>age: { type: Number, default: 18 }</code>	18 will be stored.
Boolean Value	Default true or false.	<code>isActive: { type: Boolean, default: true }</code>	true will be stored.
Date Value	Default to current date/time.	<code>createdAt: { type: Date, default: Date.now }</code>	Current date & time will be stored.
Function	Function returns a value (dynamic default).	<code>id: { type: String, default: () => generateId() }</code>	Result of function will be stored.



16.8.4 Example: Creating Documents with Defaults

Example:

```
const User = mongoose.model('User', userSchema);
// Creating a user without optional fields
const u1 = new User({
  name: 'Alice',
  email: 'alice@gmail.com'
});
u1.save().then(() => console.log('User saved!'))
.catch(err => console.log('Error:', err));
```

Stored Document in MongoDB:

```
{
  "_id": ObjectId("665abc1234567890abcdef00"),
  "name": "Alice",
  "email": "alice@gmail.com",
  "role": "user",
  "isActive": true,
  "createdAt": "2024-05-15T10:20:30.123Z",
  "__v": 0
}
```



Defaults Applied Automatically!

Key Takeaway:



Default values in Mongoose make our application **easier** by automatically filling in missing data. They ensure **consistency**, **save time** and **reduce errors** in data entry.



16.9 Required Fields

16.9.1 What are Required Fields?

- * A required field is a field that must have a value when creating or updating a document.
- * If a required field is missing, Mongoose throws a **validation error**.
- * Required fields help ensure that **important data** is always provided.
- * We can make a field required by setting **required: true** in the schema.



16.9.2 Declaring Required Fields in Schema

- * We use **required: true** to mark a field as required.

Example:

```
const mongoose = require('mongoose');  
const productSchema = new mongoose.Schema({  
  name: { type: String, required: true },  
  price: { type: Number, required: true },  
  category: { type: String, required: true },  
  description: { type: String },  
  createdAt: { type: Date, default: Date.now }  
});  
const Product = mongoose.model('Product', productSchema);
```

Key Points:

- ✓ Set **required: true** for fields that must have a value.
- ✓ Mongoose validates required fields before saving the document.
- ✓ If any required field is missing, a validation error is thrown.
- ✓ Only fields with **required: true** are checked.

Name [____] *
Price [____] *
Category [____] *
Description [____]

* = Required Field

16.9.3 Behavior of Required Fields

Scenario	Provided Data	Result	Explanation
All required fields provided	All fields with required: true have values	Document saved successfully	All required fields are present, validation passes.
One or more required fields missing	Some fields with required: true are empty	Validation error thrown	Mongoose does not allow saving document with missing required fields.
Required field is null or undefined	Required fields set to null or undefined	Validation error thrown	Mongoose treats null/undefined as missing value.



Validation Error

16.9.4 Example: Missing Required Field

Example:

```
// Missing the 'price' field (which is required)  
const p1 = new Product({  
  name: 'Laptop',  
  category: 'Electronics'  
});  
p1.save().then(() => console.log('Saved!'))  
  .catch(err => console.log('Error:', err.message));
```

Output (Error):

```
Error: Product validation failed: price: Path 'price' is required.  
  
// Mongoose does not save the document  
// because 'price' is a required field and  
// no value was provided.
```



Document Not Saved!

Key Takeaway:



Required fields in Mongoose ensure that critical data is always provided. They help maintain **data integrity** and prevent incomplete or invalid data from being stored.



16.10 Custom Validation

16.10.1 What is Custom Validation?

- * Built-in validations (like `required`, `min`, `max`) cover common cases.
- * Custom validation allows us to define our own rules using the `validate` property.
- * It gives more flexibility to enforce complex or business-specific constraints.
- * Mongoose supports custom validation using functions (synchronous or asynchronous).



16.10.2 Defining Custom Validation in Schema

- * Use the `validate` property in a field to apply custom validation.

Example:

```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  age: {
    type: Number,
    validate: {
      validator: function (v) { return v >= 18; },
      message: props => `S{props.value} is not an adult! Age must be 18 or above.`
    }
  },
  email: {
    type: String,
    validate: {
      validator: function (v) {
        return /.+@.+\..+/.test(v);
      },
      message: props => `S{props.value} is not a valid email!`
    }
  }
});
const User = mongoose.model('User', userSchema);
```

Key Points:

- ✓ `validator` is a function that returns `true` or `false`.
- ✓ If `validator` returns `false`, validation fails.
- ✓ `message` is shown when validation fails.
- ✓ Can be synchronous (returns `true/false`) or asynchronous (uses callback or returns a Promise).



Types of Custom Validation

Type	Description	Example
Synchronous (Inline Function)	Validator function returns <code>true</code> or <code>false</code> .	<code>validate: v => v >= 18</code>
Asynchronous (Callback)	Uses callback to perform async validation.	<code>validate: { validator: function(v, done) { ... }, message: '...' }</code>
Asynchronous (Promise)	Validator returns a Promise that resolves to <code>true/false</code> .	<code>validate: async function(v) { ... }</code>

16.10.3 Example: Using Different Custom Validations

Example:

```
const productSchema = new mongoose.Schema({
  name: { type: String, required: true },
  price: {
    type: Number,
    min: 0,
    validate: {
      validator: function (v) { return v > 0; },
      message: 'Price must be greater than 0!'
    }
  },
  stock: {
    type: Number,
    validate: {
      validator: v => Number.isInteger(v) && v >= 0,
      message: 'Stock must be a non-negative integer!'
    }
  },
  email: {
    type: String,
    validate: {
      validator: async function (v) {
        const exists = await User.exists({ email: v });
        return !exists; // email must be unique
      },
      message: 'Email is already registered!'
    }
  }
});
```

Explanation:

- Price must be `> 0` (custom rule).
- Stock must be a non-negative integer.
- Email must be unique (async validation).

16.10.4 Output (When Validation Fails)

Example Output:

```
Error: Product validation failed
{
  "errors": {
    "price": {
      "message": "Price must be greater than 0!",
      "kind": "user defined",
      "path": "price",
      "value": -10
    },
    "stock": {
      "message": "Stock must be a non-negative integer!",
      "kind": "user defined",
      "path": "stock",
      "value": 3.5
    }
  }
}
```



Validation
Error!

Key Takeaway:



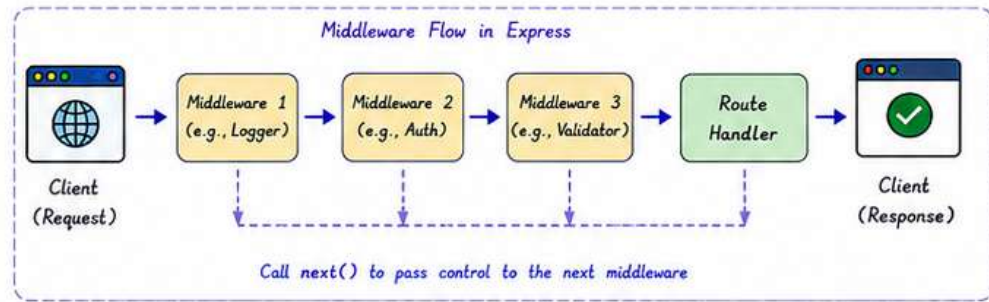
Custom validation in Mongoose helps enforce complex and business-specific rules. Use `validate` with `validator` and `message` to create powerful validations. Choose `synchronous`, `callback`, or `Promise-based` validation based on your needs.



16.11 Middleware

16.11.1 What is Middleware?

- * Middleware functions have access to the request object (**req**), the response object (**res**) and the **next()** function in the application's request-response cycle.
- * Middleware functions can execute any code, make changes to the request and response objects, end the request-response cycle, or call the **next** middleware function in the stack.
- * Express has a built-in middleware function **express.json()** and **express.urlencoded()**.



16.11.2 Types of Middleware

Type	Description
1. Application-level Middleware	Bound to an instance of app object using <code>app.use()</code> .
2. Router-level Middleware	Bound to an instance of <code>express.Router()</code> object.
3. Built-in Middleware	Built-in middleware functions provided by Express (e.g., <code>express.json()</code> , <code>express.urlencoded()</code>).
4. Third-party Middleware	Middleware functions created by third-party modules (e.g., <code>morgan</code> , <code>cors</code>).
5. Error-handling Middleware	Middleware functions that handle errors. (Have 4 arguments: <code>err</code> , <code>req</code> , <code>res</code> , <code>next</code>).

16.11.3 Example: Simple Middleware

Example: Logger Middleware

```
const express = require('express');
const app = express();

// Middleware to log request details
function logger(req, res, next) {
  console.log(`${req.method} ${req.url} - ${new Date().toLocaleTimeString()}`);
  next(); // pass control to the next middleware
}

app.use(logger);

app.get('/', (req, res) => {
  res.send('Hello from Express!');
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

Middleware (Logger)



Route Handler



16.11.4 Built-in Middleware in Express

Middleware	Purpose
<code>express.json()</code>	Parses incoming requests with JSON payloads. Adds <code>req.body</code> .
<code>express.urlencoded({ extended: true })</code>	Parses incoming requests with URL-encoded payloads. Adds <code>req.body</code> .
<code>express.static(path)</code>	Serves static files from the specified directory.
<code>express.raw()</code>	Parses incoming requests with Buffer payloads.
<code>express.text()</code>	Parses incoming requests with text payloads.

16.11.5 Example: Using Built-in Middleware

```
const express = require('express');
const app = express();

app.use(express.json()); // for JSON data
app.use(express.urlencoded({ extended: true })); // for form data
app.use(express.static('public')); // serve static files

app.post('/data', (req, res) => {
  console.log(req.body);
  res.send('Data received');
});

app.listen(3000, () => console.log('Server running...'));
```



Built-in Middleware

16.11.6 Error-handling Middleware

- * It is used to handle errors in the application.
- * It must have four arguments: (**err**, **req**, **res**, **next**).

Example:

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something went wrong!');
});
```



Error Handler

16.11.8 Key Points

- ✓ Middleware functions are executed in the order they are defined.
- ✓ Always call **next()** to pass control to the next middleware.
- ✓ Use middleware for logging, authentication, validation, parsing, etc.
- ✓ Error-handling middleware should be defined at the end.

Key Takeaway:

16.11.7 Example: Complete Flow

```
const express = require('express');
const app = express();

// 1. Logger Middleware
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});

// 2. Built-in Middleware
app.use(express.json());

// 3. Route
app.get('/hello', (req, res) => {
  res.send('Hello Middleware!');
});

// 4. Error-handling Middleware
app.use((err, req, res, next) => {
  res.status(500).send('Internal Server Error');
});

app.listen(3000, () => console.log('Server running on port 3000'));
```



Middleware Flow



Server Running



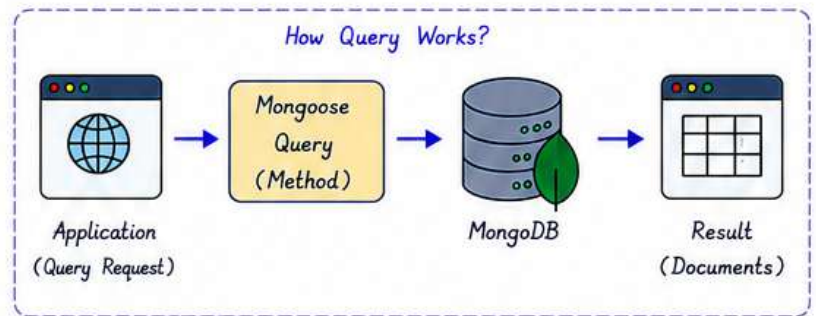
Middleware in Express helps us process requests, perform tasks like **logging**, **parsing**, **authentication**, **validation** and handle **errors** efficiently before reaching the route handlers.



16.12 Query Methods

16.12.1 What are Query Methods?

- * Query methods are used to **find**, **filter** and **retrieve** documents from a MongoDB collection.
- * They allow us to **read data** based on conditions.
- * Mongoose provides **powerful** methods to perform queries easily.
- * Queries return **results** in the form of documents or arrays of documents.



16.12.2 Common Query Methods in Mongoose

Method	Description	Returns	Example
<code>find()</code>	Finds all documents that match a condition.	Array of documents	<code>Model.find({ age: { \$gt: 18 } })</code>
<code>findOne()</code>	Finds the first document that matches a condition.	Single document	<code>Model.findOne({ email: 'abc@gmail.com' })</code>
<code>findById()</code>	Finds a document by its <code>_id</code> .	Single document	<code>Model.findById('665a1b2c3d4e5f6789012345')</code>
<code>findByIdAndUpdate()</code>	Finds a document by id and updates it.	Updated document (by default)	<code>Model.findByIdAndUpdate(id, { name: 'John' }, { new: true })</code>
<code>findByIdAndDelete()</code>	Finds a document by id and deletes it.	Deleted document (by default)	<code>Model.findByIdAndDelete(id)</code>
<code>countDocuments()</code>	Counts the number of documents matching a condition.	Number	<code>Model.countDocuments({ age: { \$gte: 18 } })</code>
<code>sort()</code>	Sorts the documents.	Array of documents	<code>Model.find().sort({ age: -1 })</code> // Descending
<code>limit()</code>	Limits the number of results.	Array of documents	<code>Model.find().limit(10)</code>
<code>skip()</code>	Skips a number of results.	Array of documents	<code>Model.find().skip(20)</code>
<code>select()</code>	Selects specific fields to return.	Array of documents	<code>Model.find().select('name email age')</code>

16.12.3 Example: Using Query Methods

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

// 1. Find all users older than 18
User.find({ age: { $gt: 18 } }).then(users => {
  console.log('Users:', users);
});

// 2. Find one user by email
User.findOne({ email: 'alice@gmail.com' }).then(user => {
  console.log('User:', user);
});

// 3. Find user by ID
User.findById('665a1b2c3d4e5f6789012345').then(user => {
  console.log('User by ID:', user);
});

// 4. Count users
User.countDocuments({ age: { $gte: 18 } }).then(count => {
  console.log('Total Users:', count);
});

// 5. Find users and sort by age (descending), Limit 5
User.find().sort({ age: -1 }).limit(5).then(users => {
  console.log('Top 5 oldest users:', users);
});
```

Key Points:

- ✓ `find()` returns all matching documents.
- ✓ `findOne()` returns the first matching document.
- ✓ `findById()` is used to find a document using its `_id`.
- ✓ `countDocuments()` returns the count of matching documents.
- ✓ `sort()`, `limit()`, `skip()` are used for pagination and sorting.
- ✓ `select()` helps to include only required fields.



16.12.4 Example: Pagination (skip + limit)

Example:

```
const page = 2; // Page number
const size = 5; // Documents per page

User.find()
  .skip((page - 1) * size) // Skip documents
  .limit(size) // Limit documents
  .sort({ name: 1 }) // Sort by name (A-Z)
  .then(users => {
    console.log('Users on page', page, ':', users);
  });
```



Pagination
(skip + limit)

16.12.5 Logical Operators in Queries

Operator	Meaning	Example Query	Matches
<code>\$and</code>	All conditions must be true	<code>{ \$and: [{ age: { \$gte: 18 } }, { status: 'active' }] }</code>	Age $\geq$ 18 AND status is active
<code>\$or</code>	At least one condition must be true	<code>{ \$or: [{ role: 'admin' }, { role: 'user' }] }</code>	Role is admin OR user
<code>\$not</code>	Negates a condition	<code>{ age: { \$not: { \$lt: 18 } } }</code>	Age is NOT less than 18
<code>\$nor</code>	None of the conditions can be true	<code>{ \$nor: [{ status: 'banned' }, { age: { \$lt: 18 } }] }</code>	Status is not banned AND age is not less than 18

Example: \$and Query

```
User.find({ $and: [
  { age: { $gte: 18 } },
  { status: 'active' }
] }).then(users => {
  console.log(users);
});
```



Key Takeaway:



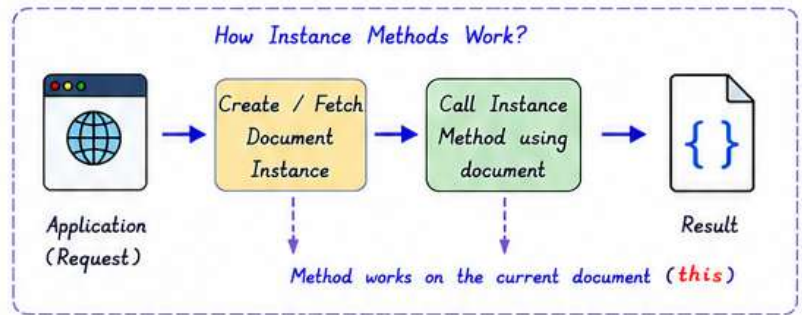
Query methods in Mongoose help us retrieve the right data from the database efficiently. They support **filtering**, **sorting**, **pagination**, **field selection** and **logical operations**.



16.13 Instance Methods

16.13.1 What are Instance Methods?

- Instance methods are functions defined on a schema that can be called on **document instances**.
- They have access to the document (using **this** keyword) on which they are called.
- They are useful for operations related to a single document.
- Instance methods are different from **static methods**, which are called on the Model.



16.13.2 Common Instance Methods in Mongoose

Method	Description	Arguments	Returns	Example
<code>save()</code>	Saves the current document.	None	Promise (saved document)	<code>user.save();</code>
<code>validate()</code>	Validates the document.	None	Promise	<code>user.validate();</code>
<code>remove()</code>	Removes the current document from the collection.	None	Promise	<code>user.remove();</code> // deletes the document
<code>isModified(path)</code>	Checks if a field (or any path) is modified.	(String) path (optional)	Boolean	<code>user.isModified('email');</code>
<code>isNew</code>	Checks if the document is new (not saved to DB yet).	None	Boolean	<code>if (user.isNew) { ... }</code>
<code>markModified(path)</code>	Marks a path as modified.	(String) path	None	<code>user.markModified('name');</code>
<code>toObject()</code>	Converts the document to a plain JavaScript object.	(options)	Object	<code>user.toObject();</code>
<code>toJSON()</code>	Converts the document to JSON.	(options)	Object	<code>user.toJSON();</code>
<code>populate()</code>	Populates referenced fields for the document.	(path, select, ...)	Query (thenable)	<code>user.populate('role');</code>

16.13.3 Example: Defining Instance Methods

Example:

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  age: Number,
  isActive: { type: Boolean, default: true }
});

// Instance method to get user's info
userSchema.methods.getInfo = function() {
  return `S${this.name} ( {this.email} )`;
};

// Instance method to check if user is adult
userSchema.methods.isAdult = function() {
  return this.age >= 18;
};

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

Key Points:

- Instance methods use the function keyword (not arrow function) because they rely on **this**.
- They operate on a single document instance.
- Defined using `schema.methods.methodName`.
- Accessed using `document.methodName()`.
- Useful for business logic related to a document.



16.13.4 Example: Using Instance Methods

Example:

```
async function run() {
  const user = new User({ name: 'Alice', email: 'alice@email.com', age: 20 });

  console.log(user.getInfo()); // Alice ( alice@email.com )
  console.log(user.isAdult()); // true

  await user.save();

  user.age = 16;
  console.log('Is modified (age)?', user.isModified('age')); // true

  const obj = user.toObject();
  console.log(obj);
}

run();
```

Note:

- We cannot use arrow functions for instance methods because arrow functions **do not** have their own **this**. They inherit **this** from the surrounding scope.

16.13.5 Instance Methods vs Static Methods

Feature	Instance Methods	Static Methods
Called on	Document instances	Model
Access	Use this to access document	Use Model name
Defined using	<code>schema.methods.methodName</code>	<code>schema.statics.methodName</code>
Purpose	Work on a single document	Work with the collection
Example	<code>user.getInfo();</code> <code>user.isAdult();</code>	<code>User.findByEmail();</code> <code>User.countActive();</code>

Example of Static Method (for comparison)

```
userSchema.statics.findByEmail = function(email) {
  return this.findOne({ email: email });
};

// Usage
const user = await User.findByEmail('alice@email.com');
```

Key Takeaway:



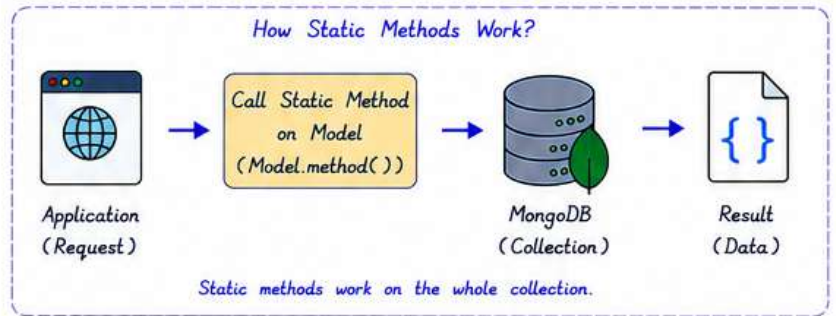
Instance methods in Mongoose allow us to define functions that work on **document instances**. They provide a clean way to encapsulate document-specific logic and operations.



16.14 Static Methods

16.14.1 What are Static Methods?

- * Static methods are called on the **Model** (not on document instances).
- * They operate on the **entire collection**.
- * They are used for common operations like **counting, creating, deleting** many documents, etc.
- * We use the **model name** followed by the method name to access static methods.



16.14.2 Common Static Methods in Mongoose

Method	Description	Arguments	Returns	Example
<code>countDocuments()</code>	Counts the number of documents matching a condition.	(filter)	Promise (Number)	<code>User.countDocuments({ age: { \$gte: 18 } })</code>
<code>estimatedDocumentCount()</code>	Counts all documents in a collection (ignores filter).	None	Promise (Number)	<code>User.estimatedDocumentCount()</code>
<code>exists()</code>	Checks if at least one document matches the given filter.	(filter)	Promise (Object or null)	<code>User.exists({ email: 'abc@gmail.com' })</code>
<code>create()</code>	Creates one or more documents.	(doc or [docs])	Promise (Document or Array)	<code>User.create({ name: 'John' })</code> <code>User.create([{ name: 'A' }, { name: 'B' }])</code>
<code>insertMany()</code>	Inserts multiple documents.	(Array of docs)	Promise (Array of Documents)	<code>User.insertMany({ name: 'A' }, { name: 'B' })</code>
<code>deleteOne()</code>	Deletes the first document that matches filter.	(filter)	Promise (Delete Result)	<code>User.deleteOne({ email: 'abc@gmail.com' })</code>
<code>deleteMany()</code>	Deletes all documents that match the filter.	(filter)	Promise (Delete Result)	<code>User.deleteMany({ age: { \$lt: 18 } })</code>
<code>updateOne()</code>	Updates the first document matching filter.	(filter, update, options)	Promise (Update Result)	<code>User.updateOne({ name: 'John' }, { \$set: { age: 25 } })</code>
<code>updateMany()</code>	Updates all documents matching filter.	(filter, update, options)	Promise (Update Result)	<code>User.updateMany({ active: false }, { \$set: { active: true } })</code>
<code>find()</code>	Finds all documents matching filter.	(filter, projection, options)	Query (thenable)	<code>User.find({ age: { \$gte: 18 } })</code>
<code>findOne()</code>	Finds the first document matching filter.	(filter, projection, options)	Query (thenable)	<code>User.findOne({ email: 'abc@gmail.com' })</code>

16.14.3 Example: Using Static Methods

Example:

```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  age: Number,
  active: { type: Boolean, default: true }
});
const User = mongoose.model('User', userSchema);

async function run() {
  // 1. Create a new user
  await User.create({ name: 'Alice', email: 'alice@email.com', age: 20 });

  // 2. Insert many users
  await User.insertMany([
    { name: 'Bob', email: 'bob@email.com', age: 22 },
    { name: 'Charlie', email: 'charlie@email.com', age: 19 }
  ]);

  // 3. Count users greater than or equal to 18
  const count = await User.countDocuments({ age: { $gte: 18 } });
  console.log('Users >= 18: ', count);

  // 4. Check if a user exists
  const exists = await User.exists({ email: 'bob@email.com' });
  console.log('Bob exists?', exists ? 'Yes' : 'No');

  // 5. Update many users
  await User.updateMany({ active: true }, { $set: { active: false } });

  // 6. Delete one user
  await User.deleteOne({ email: 'charlie@email.com' });
}
run();
```

Key Points:

- ✓ Static methods are called on the **Model**.
- ✓ They operate on the **whole collection**.
- ✓ They are used for common operations like create, count, update, delete, etc.
- ✓ They return Promises.
- ✓ Use them for bulk or collection-level actions.



16.14.4 Example: Counting and Pagination

Example:

```
async function getUsers(page = 1, limit = 5) {
  const skip = (page - 1) * limit;
  const total = await User.countDocuments(); // total users
  const users = await User.find().skip(skip).limit(limit).sort({ name: 1 });

  console.log('Total Users: ${total}');
  console.log('Page ${page} Users:', users);
}

getUsers(1, 3); // first page with 3 users
```



Pagination
(Skip + Limit)

16.14.6 Example: Using exists()

Example:

```
async function checkEmail(email) {
  const user = await User.exists({ email: email });
  if (user) {
    console.log('Email already registered.');
```



16.14.5 Static Methods vs Instance Methods

Feature	Static Methods	Instance Methods
Called on	Model	Document instance
Access	<code>Model.methodName()</code>	<code>document.methodName()</code>
Scope	Whole collection	Single document
Purpose	Bulk / collection operations	Document-specific operations
Example	<code>User.countDocuments()</code>	<code>user.save()</code>
Keyword	No this	Uses this keyword

Key Takeaway:



Static methods in Mongoose are powerful **collection-level** operations.

They help us perform tasks like **counting, creating, updating, deleting** and querying data easily.

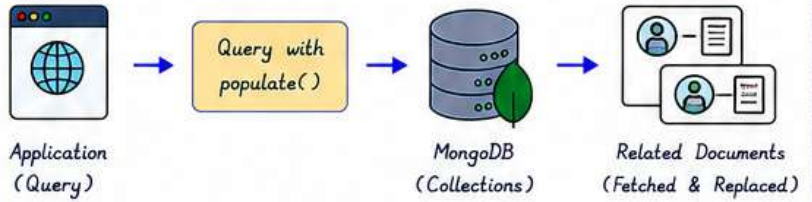


16.15 Population

16.15.1 What is Population?

- * Population is a Mongoose feature that **replaces** the specified paths in the document with the **actual documents** from other collections.
- * It is used to **reference** documents in other collections.
- * We use **populate()** method with query functions.
- * It helps in working with **relational data** in MongoDB.

How Population Works?



Population replaces the referenced ids with the actual documents.

16.15.2 Example Scenario

We have two collections:

1. users collection

_id	name
u1	Alice
u2	Bob
u3	Charlie



2. posts collection (has reference to users)

_id	title	user
p1	Introduction to MongoDB	u1
p2	Mongoose Basics	u2
p3	Population in Mongoose	u1



Key Idea:

The field 'user' in posts collection stores the **_id** of a user. Using **populate()**, we can replace this id with the actual user document.

16.15.3 Using populate()

Example:

```
const mongoose = require('mongoose');

// User Schema
const userSchema = new mongoose.Schema({
  name: String
});
const User = mongoose.model('User', userSchema);

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

// Find posts and populate user field
Post.find().populate('user').then(posts => {
  console.log(posts);
});
```

Result (After Population):

```
[
  {
    _id: 'p1',
    title: 'Introduction to MongoDB',
    user: { _id: 'u1', name: 'Alice' } // populated user document
  },
  {
    _id: 'p2',
    title: 'Mongoose Basics',
    user: { _id: 'u2', name: 'Bob' }
  },
  {
    _id: 'p3',
    title: 'Population in Mongoose',
    user: { _id: 'u1', name: 'Alice' }
  }
]
```



populate('user') replaces the user id with the actual user document.

16.15.4 Advanced Population

1. Populate Select Specific Fields

Example:

```
Post.find().populate('user', 'name').then(posts => {
  console.log(posts);
});

// Only 'name' field from user will be returned.
```

2. Populate with Multiple Paths

Example:

```
Post.find().populate('user').populate('comments.user').then(results => {
  console.log(results);
});

// If comments schema has a user field, it will also be populated.
```



Tip:

You can chain multiple **populate()** methods for different paths.

16.15.5 Population vs Normal Query

Feature	Normal Query	With Population
Returned Data	Returns ids only	Returns full referenced documents
Collections	Single collection data	Data from multiple collections
Complexity	Simple	More powerful and flexible
Use Case	When reference data is not needed	When related data is required

16.15.6 Example: Without Population vs With Population

Without Population:

```
Post.find().then(posts => {
  console.log(posts);
});

/* Output */
[
  { _id: 'p1', title: 'Intro', user: 'u1' },
  { _id: 'p2', title: 'Basics', user: 'u2' }
]
```

With Population:

```
Post.find().populate('user').then(posts => {
  console.log(posts);
});

/* Output */
[
  { _id: 'p1', title: 'Intro', user: { _id: 'u1',
    name: 'Alice' } },
  { _id: 'p2', title: 'Basics', user: { _id: 'u2',
    name: 'Bob' } }
]
```

16.15.7 Important Points

- ✓ Use **populate()** when you want related documents instead of ids.
- ✓ Populate works only on fields defined with **ref** in the schema.
- ✓ It performs a separate query behind the scenes and 'joins' the data.
- ✓ Over-populating can affect performance, so **populate** only when needed.

Key Takeaway:



Population in Mongoose allows us to replace referenced ids with actual documents from other collections. It makes our data more **meaningful** and **easier to work with**.



16.15.8 populate() Options

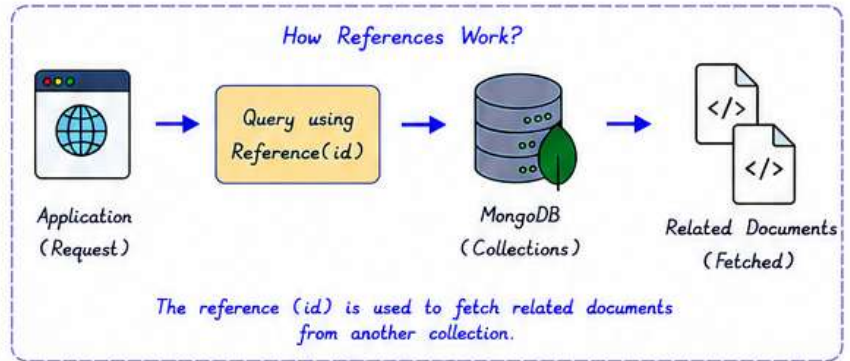
Option	Description	Example
select	Select specific fields from populated document.	<code>.populate('user', 'name email')</code>
match	Filter documents while populating.	<code>.populate({ path: 'user', match: { active: true } })</code>
limit	Limit number of populated documents.	<code>.populate({ path: 'comments', limit: 5 })</code>
sort	Sort populated documents.	<code>.populate({ path: 'comments', sort: { createdAt: -1 } })</code>



16.16 References

16.16.1 What are References?

- * References are used to represent relationships between documents in **different collections**.
- * Instead of embedding large or frequently changing documents, we store a reference (id) to the related document.
- * References are useful for **one-to-one**, **one-to-many** and **many-to-many** relationships.
- * We use **DBRef** or a simple foreign key (**_id**) as a reference.



16.16.2 Types of References

Type	Description
DBRef	MongoDB provides a DBRef object with \$ref, \$id and \$db fields.
Manual Reference (Foreign Key)	Store the _id of the referenced document in a field.
One-to-One	A document in one collection refers to a single document in another collection.
One-to-Many	A document refers to multiple documents in another collection (array of ids).
Many-to-Many	Both collections store references to each other.

16.16.3 Example Scenario

We have two collections:

1. **authors** collection

_id	name
a1	Alice Walker
a2	Robert Frost
a3	Rabindranath Tagore

2. **books** collection (references authors)

_id	title	authorId
b1	The Color Purple	a1
b2	The Road Not Taken	a2
b3	Gitanjali	a3
b4	Another Poem	a2

Here, the field "**authorId**" in books collection stores **reference** to the **_id** of a document in **authors** collection.

16.16.4 Example: Using DBRef

Example:

```
// Insert author
db.authors.insertOne({
  _id: ObjectId(),
  name: "Alice Walker"
});

// Insert book with DBRef to author
db.books.insertOne({
  title: "The Color Purple",
  author: { $ref: "authors", $id: ObjectId("664a1f2b3c4d5e6789012345"), $db: "myLibrary" }
});
```

Query and Retrieve:

```
db.books.find().populate('author');

// Output
{
  _id: ObjectId("..."),
  title: "The Color Purple",
  author: {
    _id: ObjectId("664a1f2b3c4d5e6789012345"),
    name: "Alice Walker"
  }
}
```

populate('author') replaces the DBRef with the actual author document.

16.16.5 Example: Using Manual Reference (Foreign Key)

Example:

```
// Insert author
db.authors.insertOne({
  _id: ObjectId("664a1f2b3c4d5e6789012345"),
  name: "Robert Frost"
});

// Insert book with manual reference
db.books.insertOne({
  title: "The Road Not Taken",
  authorId: ObjectId("664a1f2b3c4d5e6789012345") // reference to author _id
});
```

Query and Retrieve:

```
// Step 1: Find books
const books = db.books.find();

// Step 2: Fetch author details
books.forEach(book => {
  const author = db.authors.findOne({ _id: book.authorId });
  book.author = author; // attach author details
  printjson(book);
});
```

Manual reference is lightweight and easy to understand.

16.16.6 DBRef vs Manual Reference

Feature	DBRef	Manual Reference
Structure	Uses \$ref, \$id, \$db	Uses only _id
Size	Slightly larger	Smaller
Querying	populate() can be used directly	Need manual lookup
Performance	Slightly slower	Faster
Flexibility	More flexible (cross-DB)	Less flexible (same DB)
Usage	Rarely used now	Commonly used

16.16.7 Types of Relationships with References

1. **One-to-One**
User profile references one user document.
2. **One-to-Many**
One author has many books.
3. **Many-to-Many**
Students enroll in many courses and courses have many students.



16.16.8 Advantages

- ✓ Keeps documents small and organized.
- ✓ Avoids data duplication.
- ✓ Makes updates easier.
- ✓ Supports flexible relationships.

16.16.9 Limitations

- ✗ Requires extra queries (joins).
- ✗ Complex queries may affect performance.
- ✗ Populate can be slower for large datasets.

16.16.10 Best Practices

- ✓ Use manual references for most cases.
- ✓ Use DBRef only when cross-database reference is required.
- ✓ Index reference fields for better performance.

Key Takeaway:



References in MongoDB help us build relationships between collections without duplicating data. They provide **flexibility**, **scalability** and maintain data **consistency** in real-world applications.

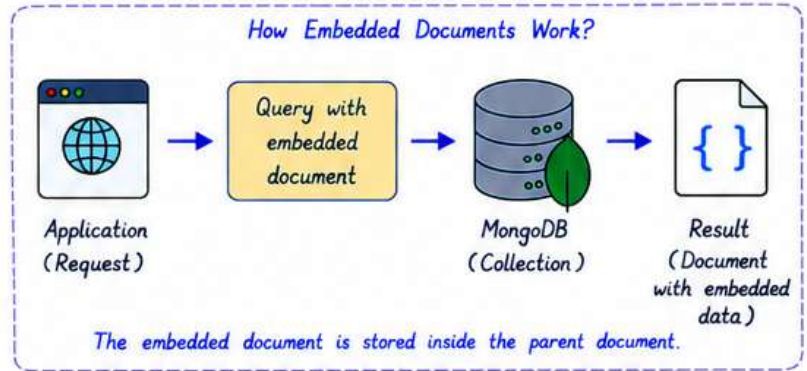


16.17 Embedded Documents

16.17.1 What are Embedded Documents?

- * Embedded documents are sub-documents stored **within a document**.
- * They allow you to store **related data** together in a single document.
- * Used when the data is accessed and updated **together as a unit**.
- * They can be **nested** (documents inside documents) and can contain arrays.

How Embedded Documents Work?



16.17.2 Example Scenario

We have a collection of users, and each user has an address stored as an embedded document.

_id	name	address (embedded document)
u1	Alice	{ street: "123 Main St", city: "New York", zip: "10001" }
u2	Bob	{ street: "456 Oak Ave", city: "Boston", zip: "02110" }
u3	Charlie	{ street: "789 Pine Rd", city: "Chicago", zip: "60601" }



Key Idea:

- ✓ The **address** field is an embedded document.
- ✓ It stores related information with the user.
- ✓ This keeps the data together and easy to access.

16.17.3 Inserting an Embedded Document

Example:

```
db.users.insertOne({
  name: "Alice",
  age: 28,
  address: {
    street: "123 Main St",
    city: "New York",
    zip: "10001"
  }
});
```

16.17.4 Querying Embedded Documents

Example:

```
// Find users living in New York
db.users.find({
  "address.city": "New York"
});

// Find a specific street
db.users.find({
  "address.street": "123 Main St"
});
```

16.17.5 Updating Embedded Documents

Example:

```
// Update the city in the embedded address
db.users.updateOne(
  { name: "Alice" },
  { $set: { "address.city": "San Francisco" } }
);

// Add a new field to embedded document
db.users.updateOne(
  { name: "Bob" },
  { $set: { "address.country": "USA" } }
);
```

16.17.6 Nested Embedded Documents

Example:

```
db.users.insertOne({
  name: "Charlie",
  contact: {
    email: "charlie@example.com",
    phone: { countryCode: "+1", number: "555-1234" }
  }
});
```

16.17.7 Arrays of Embedded Documents

Example:

```
db.users.insertOne({
  name: "David",
  orders: [
    { item: "Laptop", price: 1200 },
    { item: "Mouse", price: 25 },
    { item: "Keyboard", price: 75 }
  ]
});
```



Tip:

Embedded documents can also be used inside arrays (for example, orders, comments, items, etc.).

16.17.8 Advantages and Disadvantages

Advantages

- ✓ Keeps related data together.
- ✓ Faster read performance (single document).
- ✓ Simpler data model.
- ✓ Useful for one-to-one and one-to-many relationships.

Disadvantages

- ✗ May lead to large documents (16MB limit).
- ✗ Harder to update if deeply nested.
- ✗ Not ideal for many-to-many relationships.

16.17.9 Best Practices

- ✓ Use embedded documents for related data that is accessed together.
- ✓ Avoid very large embedded documents (keep under 16MB).
- ✓ Use arrays of embedded documents for one-to-many relationships.
- ✓ Consider referencing (separate collection) for many-to-many relationships.



Key Takeaway: Embedded documents help you organize **related data** in a **single document**, making your application faster, simpler and easier to manage.



16.18 Mongoose Transactions

16.18.1 What are Transactions?

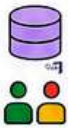
- Transactions allow us to group multiple operations as a **single unit of work**.
- All operations in a transaction succeed, or none do.
- If any operation fails, all changes are **rolled back**.
- Transactions are supported in MongoDB **replica sets** and **sharded clusters**.
- Mongoose provides a simple API to work with MongoDB transactions using **sessions**.

16.18.2 Example Scenario

We have two collections: **accounts** and **transactions**.
We want to transfer money from one account to another.
Both operations must succeed together.

accounts Collection

_id	name	balance
1	Alice	5000
2	Bob	2000



transactions Collection

_id	from	to	amount
(empty)			



16.18.4 Example: Money Transfer Using Transaction

Example:

```
const mongoose = require('mongoose');
const Account = require('./models/Account');
const Transaction = require('./models/Transaction');

async function transferMoney(fromId, toId, amount) {
  const session = await mongoose.startSession();
  session.startTransaction();
  try {
    // 1. Deduct from sender
    const sender = await Account.findById(fromId).session(session);
    if (!sender || sender.balance < amount) {
      throw new Error('Insufficient balance or sender not found');
    }
    sender.balance -= amount;
    await sender.save({ session });

    // 2. Add to receiver
    const receiver = await Account.findById(toId).session(session);
    if (!receiver) {
      throw new Error('Receiver not found');
    }
    receiver.balance += amount;
    await receiver.save({ session });

    // 3. Record transaction
    await Transaction.create([
      {
        from: fromId,
        to: toId,
        amount: amount,
        date: new Date()
      }
    ], { session });

    // 4. Commit the transaction
    await session.commitTransaction();
    console.log('Transaction committed successfully');
  } catch (err) {
    // 5. Abort on any error
    await session.abortTransaction();
    console.error('Transaction aborted:', err.message);
  } finally {
    session.endSession();
  }
}
```



How Mongoose Transactions Work?

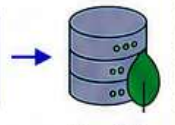


Application
(Start Session)

Start
Transaction

Perform Operations
(Multiple Queries)

Commit
or
Abort



MongoDB
(Apply or Rollback)

Commit → Saves all changes

Abort → Discards all changes (Rollback)

16.18.3 Transaction Flow

1. Start a session.
2. Start a transaction.
3. Perform multiple operations using the session.
4. Commit the transaction if all operations succeed.
5. Abort the transaction if any operation fails.



16.18.5 Common Session / Transaction Methods

Method	Description	Used For
<code>mongoose.startSession()</code>	Starts a new session.	Begin session
<code>session.startTransaction()</code>	Starts a transaction on the session.	Begin transaction
<code>Model.find().session(s)</code>	Executes query in the session.	Run queries
<code>session.commitTransaction()</code>	Commits all operations.	Save changes
<code>session.abortTransaction()</code>	Aborts all operations.	Rollback changes
<code>session.endSession()</code>	Ends the session.	Finish session

16.18.6 Example: What Happens on Failure?

If any operation fails (e.g., receiver does not exist), the transaction is aborted and no changes are saved. The sender's balance remains unchanged.



16.18.7 Important Points

- ✓ Transactions require MongoDB replica set or sharded cluster.
- ✓ All operations in the transaction must use the same session.
- ✓ Keep transactions short to avoid locking and performance issues.
- ✓ Use `try...catch` to handle errors and abort the transaction.
- ✓ Transactions are atomic (all or nothing).
- ✓ Do not forget to end the session.



16.18.8 Do & Don't

Do	Don't
✓ Use transactions for related operations.	✗ Don't run long transactions.
✓ Always handle errors and abort.	✗ Don't forget to end the session.
✓ Use sessions for all queries in transaction.	✗ Don't use transactions for simple independent operations.
✓ Test transactions in replica set environment.	✗ Don't mix session and non-session queries in same transaction.

16.18.10 Best Practices

- ✓ Keep transactions small and focused.
- ✓ Avoid nested transactions.
- ✓ Monitor and log transaction failures.
- ✓ Use transactions only when data integrity is critical.



16.18.9 Transaction Lifecycle



Key Takeaway:



Mongoose transactions help maintain data **consistency** by ensuring all related operations succeed together or fail together, keeping your application data **reliable** and **accurate**.

