Chapter 14

Mongoose

What is in This Chapter ?

This chapter we discuss an add-on to **MongoDB** called **Mongoose**. We first show how it makes connecting to the database a little easier. Then we will discus **Schemas** and how they can make our life a lot easier by performing **validation** on our data to ensure that only valid data is stored in the database. We discuss **schema models** that make coding a lot easier through the chaining together of various query parameters using **JavaScript** syntax. Lastly, we discuss **Mongoose instance methods**, **query helper** methods and **referencing** other documents in the database.

14.1 Mongoose Connections

When we work directly with **MongoDB**, we are talking to the database in a very flexible but low-level way, which often means writing lots of repetitive code to check data types, handle validation (i.e., check that the input makes sense), and build queries. ***Mongoose*** is a tool built on top of **MongoDB** that solves these problems by acting as a "bridge" between the database and our **JavaScript** code.

With ***Mongoose***, we can:

- ✓ define clear rules for our data
- ✓ rely on automatic validation and type handling
- ✓ use simpler, more consistent methods to query and update the database

In short, it makes working with **MongoDB** much more organized, efficient, and reliable.

To use **Mongoose**, we first install it: **npm install mongoose**

(Note: If we do this, we won't need to install the **MongoDB** driver since **Mongoose** already includes it internally)

The easiest way to explain how to connect and disconnect from a database using **Mongoose** is to compare it to the way we made the connection previously in **MongoDB**. Below on the left is our **db.js** code for connecting to the database in **MongoDB** and on the right is how we will do it in **Mongoose**. The differences are highlighted:

MongoDB (db.js)

```
// Get the MongoClient class from the mondodb module
const { MongoClient } = require("mongodb");

const dbName = "websiteDatabase";  // Name of our database
const url = "mongodb://localhost:27017";

// Create a MongoClient instance
const client = new MongoClient(url);
let db = null;   // The database connection (null upon start)

// Connect to the database
async function connectToDatabase() {
    // If not already connected, connect now
    if (!db) {
        try {
            await client.connect();
            db = client.db(dbName);
            console.log("Connected to MongoDB: " + dbName);
        } catch (err) {
            console.error("Error connecting to MongoDB:", err);
            throw err; // re-throw this error
        }
    }
    return db;
}

// Disconnect from the database
async function disconnectFromDatabase() {
    try {
        await client.close();
        db = null;  // Reset for next time
        console.log("Disconnected from MongoDB: " + dbName);
    } catch (err) {
        console.error("Error disconnecting from MongoDB:", err);
    }
}
module.exports = { connectToDatabase, disconnectFromDatabase };
```

Mongoose (mongooseDB.js)

```
// Get the mongoose module
const mongoose = require("mongoose");

const dbName = "websiteDatabase"; // Name of our database
const url = "mongodb://localhost:27017/" + dbName;


// notice that the database name is part is of URL now



// Connect to the database
async function connectToDatabase() {
    // If not already connected, connect now
    if (mongoose.connection.readyState !== 1) { // 1 = connected
        try {
            await mongoose.connect(url);
            console.log("Connected to MongoDB: " + dbName);
        } catch (err) {
            console.error("Error connecting to MongoDB:", err);
            throw err; // re-throw this error
        }
    }
    return mongoose.connection;
}

// Disconnect from the database
async function disconnectFromDatabase() {
    try {
        await mongoose.disconnect();

        console.log("Disconnected from MongoDB: " + dbName);
    } catch (err) {
        console.error("Error disconnecting from MongoDB:", err);
    }
}
module.exports = { connectToDatabase, disconnectFromDatabase };
```

Notice that **Mongoose** connects to a database directly, instead of requiring a **MongoClient** class to be used and then getting the database from the client. We also do not need to use a `db` variable to hold onto the database because **Mongoose** keeps track of its connection status. We can simply ask the `mongoose.connect` for its `readyState` … which is one of these: **0**=disconnected, **1**=connected, **2**=connecting, **3**=disconnecting. When connecting, we return the connection now, instead of the database.

14.2 Data Validation Using Schemas

Before we begin to use Mongoose, we need to first understand what a ***schema*** is.

> ***A schema is a definition that specifies what*** *fields* ***a document can have, what*** *types* ***those fields must be, and any*** *rules* ***or*** *constraints* ***they must follow.***

So, each collection typically has its own schema, and **Mongoose** uses that schema to define the following for documents in the collection:

- **Fields** (e.g., name, price, stock, dimensions)
- **Types** (e.g., String, Number, Number, Object)
- **Validation rules** (e.g., price is required, stock must be ≥ 0)

Mongoose ensures that any data we save through it follows the schema we have defined. However, if we insert data directly into **MongoDB** without using **Mongoose**, those rules won't be applied, and the data might not match the schema.

One nice feature about **Mongoose** is that it automatically handles type conversion and validation every time we save a document, taking care of the boring stuff for us 😊.

Before we create and use a collection in the database, we need to define a schema for the documents in that collection. We do this by creating a **Schema** object through use of a constructor:

```
const productSchema = new mongoose.Schema({...});
```

This constructor takes a **JavaScript** object that will list the **name** of each field that each document should have (i.e., product documents in this example) and the **type** of data that each field will store. Here is an example:

```
const productSchema = new mongoose.Schema({
    name: String,
    price: Number,
    stock: Number,
    dimensions: {
        length: Number,
        width: Number,
        height: Number
    },
    weight: Number
});
```

Notice that we can have nested objects (e.g., dimensions). The possible field types are:

Field Type	Description
`String`	text data
`Number`	numeric data
`Boolean`	true or false values
`Date`	date and time values
`Buffer`	binary data (e.g., images, PDFs, files)
`mongoose.Schema.Types.ObjectId`	reference to another document in **MongoDB**
`Array` **or** `[...]`	list of values of any type (e.g., [Number])
`mongoose.Schema.Types.Mixed`	any kind of data; no restrictions

Here is an expanded version of our product schema that makes use of these other types:

```
const reviewSchema = new mongoose.Schema({
    user: mongoose.Schema.Types.ObjectId,  // an object id (pertaining to a user)
    rating: Number,
    comment: String,
    date: Date
});

const productSchema = new mongoose.Schema({
    name: String,
    price: Number,
    stock: Number,
    dimensions: {
        length: Number,
        width: Number,
        height: Number
    },
    weight: Number,
    category: String,
    releaseDate: Date,                    // a Date object (includes the time)
    image: Buffer,                        // small image stored directly
    manual: Buffer,                       // PDF or file
    reviews: [reviewSchema],              // another Schema object (see above)
    tags: [String],                       // an array of Strings
    metadata: mongoose.Schema.Types.Mixed  // unpredictable attributes
});
```

Notice that we use a second schema called `reviewSchema` and that the `productSchema` simply includes an array of `reviewSchema` objects. That `reviewSchema` contains a `mongoose.Schema.Types.ObjectId` type … which means that it is an **_id** for another document in the database … likely a user ID from a **users** collection within the same database.

Notice that the `releaseDate` stores a **Date** object (which includes the time of day). This is better than storing it as a string because we can perform operations and calculations on that **Date** object. The `Buffer` type is used for `image` (e.g., **.jpg**) and `manual` (e.g., **.pdf**). The `tags` field is an array of strings.

Finally, the metadata field is set to a `mongoose.Schema.Types.Mixed` type which indicates that there could be any type of thing here. We should only use `Mixed` if our data is unpredictable. If we know the structure, it's better to model it with a proper schema. Sometimes people use `Mixed` for things like dynamic settings or config objects.

Once we define a schema, we compile it into a **model**:

> ***A model is a class-like constructor created from a schema that lets us interact with a specific** MongoDB **collection.***

A **model** is like a *class* built from a schema, where each instance of that class represents a single document that can be created, saved, updated, queried, or deleted. **Mongoose** enforces the schema rules and provides methods to interact with the database, making the model the bridge between your schema and the **MongoDB** collection.

We compile a schema into a model/class using the `model()` function, which takes two parameters … the name of the type of document we are modeling and the schema itself. For example, we could create a **Product** model using the **productSchema** we defined earlier as follows:

```
const Product = mongoose.model("Product", productSchema);
```

The code will cause **Mongoose** to register this model and automatically associate it with a **products** collection in **MongoDB**. Notice, that by default, it pluralizes and lowercases the model name to determine the collection name, so "Product" becomes "products".

Now that we have a schema and a model, we can create a new **Product** by using the model as a constructor:

```
// Create a new product document
const newProduct = new Product({
  name: "Toy Car",
  price: 9.99,
  stock: 100,
  dimensions: { length: 10, width: 5, height: 3 },
  weight: 0.5
});
```

The code is similar to what we would do if we wanted to make an instance of the **Product** class in Java. Now, we can see how the model is like a class 😉.

Once the product has been created, we can save it to the database by calling the `save()` function for this product … and it is always safer to wrap saves with **try**/**catch** blocks:

```
try {
    await newProduct.save();
    console.log("Product saved:", newProduct);
} catch (err) {
    console.error("Error saving product:", err);
}
```

By having the product created first and then saved afterwards, this allows us to run code (or even modify the document instance) *before* saving, if we need to. However, we can do the creating and saving in one step by using the `create()` function instead:

```
// Create an object with the data in it
const newProductData = {
    name: "Toy Car",
    price: 9.99,
    stock: 100,
    dimensions: { length: 10, width: 5, height: 3 },
    weight: 0.5
};

try {
    const newProduct = await Product.create(newProductData);
    console.log("Product created:", newProduct);
} catch (err) {
    console.error("Error creating product:", err);
}
```

By comparison, this simpler code:

```
const newProduct = await Product.create(newProductData);
```

does the *rough* equivalent to this **MongoDB** code:

```
const result = await db.collection("products").insertOne(newProductData);
```

However, there will be more going on behind the scenes because **Mongoose** lets us specify schema rules. For example, we can specify things such as:

- the `name` field is **required**.
- the `price` field must be a number **greater than zero**.
- the `category` field can only be one **of a predefined set of allowed values**.
- the `createdAt` field can automatically get a **default value** (e.g., the current date/time).

Then, whenever we try to save a document, **Mongoose** will check all of these rules automatically. If anything doesn't match the rules, it throws an error instead of saving invalid data. So, we don't have to write a bunch of manual checks in our code (although we will still want to check for this kind of stuff at the user interface level).

To specify these kind of field-validation requirements, we supply an object for each field (as opposed to a simple string or number). Within this object we specify the required fields, validity checks and default values.

Here is a comparison:

Without Validation Rules

```
const productSchema = new mongoose.Schema({
    name: String,
    price: Number,
    stock: Number,
    dimensions: {
        length: Number,
        width: Number,
        height: Number
    },
    weight: Number
});
```

With Validation Rules

```
const productSchema = new mongoose.Schema({
  name: { type: String, required: true },
  price: { type: Number, required: true, min: 0 },
  stock: { type: Number, default: 0 },
  dimensions: {
    length: { type: Number, min: 0 },
    width: { type: Number, min: 0 },
    height: { type: Number, min: 0 }
  },
  weight: { type: Number, min: 0 }
});
```

Notice that each value is now an object with braces **{ }**. We keep the type that we had before, but also now specify that it is a type by literally writing **type:** before it. This helps to distinguish the type from the other object keys pertaining to validation checks. Notice a few other attributes set above (i.e., **required**, **min** and **default**). Each one is a key/value pair, separated by commas. Here is a table showing some of the more common validation checks that we can do, along with some basic examples:

Key	Applies To…	Description / Example
required	All	Makes the field mandatory `name: { type: String, required: true }`
default	All	Sets a default value, if no value was provided `stock: { type: Number, default: 0 }`
min **max**	Number/ Date	Enforces a minimum/maximum value (can use one or both) `price: { type: Number, min: 1, max: 5 }`
minLength **maxLength**	String	Enforces a length limit (can use one or both) `rating: { type: Number, minLength: 1, maxLength: 5 }`
enum	String/ Number	Restricts values to be only those from a list `category: { type: String, enum: ["Toy", "Book", "Game"] }`
match	String	Validates against a RegExp `email: { type: String, match: /.+\@.+\..+/ }`
validate	All	Custom validator function `age: { type: Number, validate: v => v % 2 === 0 }`
unique	All	Creates a unique index (not true validation, but ensures uniqueness) `username: { type: String, unique: true }`
trim	String	Removes leading/trailing spaces `name: { type: String, trim: true }`
lowercase	String	Converts string to lowercase `email: { type: String, lowercase: true }`

`uppercase`	String	Converts string to uppercase `code: { type: String, uppercase: true }`
`immutable`	All	Prevents field from being changed after creation `createdAt: { type: Date, default: Date.now, immutable: true }`

Remember … adding validation is crucial because it acts as a “safety net” between our application and the database. Without it, bad or inconsistent data can slip in (e.g., negative prices, missing required fields, or impossible dates). Good validation helps prevent errors early and protects the integrity of our system.

Here is an expanded example of a product schema with some realistic rules applied:

```
const reviewSchema = new mongoose.Schema({
    user: { type: mongoose.Schema.Types.ObjectId, required: true, ref: 'User' },
    rating: { type: Number, required: true, min: 1, max: 5 },
    comment: { type: String, maxlength: 1000 },
    date: { type: Date, default: Date.now }
});

const productSchema = new mongoose.Schema({
    name: { type: String, required: true, trim: true, minlength: 2, maxlength: 100 },
    price: { type: Number, required: true, min: 0 },
    stock: { type: Number, default: 0, min: 0 },
    dimensions: {
        length: { type: Number, min: 0 },
        width: { type: Number, min: 0 },
        height: { type: Number, min: 0 } },
    weight: { type: Number, min: 0 },
    category: { type: String, enum: ['Toys', 'Electronics', 'Books', 'Other'], required: true },
    releaseDate: { type: Date,
        validate: {
            validator: v => !v || v <= new Date(),
            message: 'Release date cannot be in the future.'
        }
    },
    image: Buffer,
    manual: Buffer,
    reviews: [reviewSchema],
    tags: { type: [String],
        validate: {
            validator: arr => arr.length <= 10,
            message: 'A product can have at most 10 tags.'
        }
    },
     metadata: { type: mongoose.Schema.Types.Mixed }
});
```

Notice …

- for `user` we specify that it is required. We also indicate `ref`**:** **'User'** here. That tells **Mongoose** that the ID will be an `_id` from the **User** collection in the database … more on this later …
- for `ratings` we specify a valid range (i.e., 1 to 5).
- for the reviews `date` we specified a `default` date which is the current day/time.
- for the product `name`, we used `trim` to get rid of extra spaces.
- most number fields have a `min` of **0**, to prevent negative numbers.
- the `category` has a fixed set of values that are allowed.
- the validation check for the `releaseDate` ensures that there is a date provided and that it is not some date in the future and the validation check for the `tags` ensures that there are at most 10 tags.

We may also supply a function as a value instead of supplying a fixed value. For example, consider this simple `stock` validation:

```
stock: { type: Number, default: 0, min: 0, required: true }
```

In reality, if we had downloadable software products or music mp3 files, we would not need a stock validation since we don't need to keep count of digital copies … so we could do something like this:

```
stock: {
    type: Number,
    default: 0,
    min: 0,
    required: function () {
        return this.category !== 'Software' && this.category !== 'Music';
    }
}
```

Notice the use of the word `this` in the above code. `this` represents the current document/object that we are trying to validate. It gives us access to all fields of the document, even ones defined later in the schema. So, here we only make the field required if it is not music or software.

For another example, consider how we used the dimensions and weight in our schema:

```
dimensions: {
    length: { type: Number, min: 0 },
    width: { type: Number, min: 0 },
    height: { type: Number, min: 0 } },
weight: { type: Number, min: 0 },
```

The `dimensions` are not required, but we can adjust the `weight` to be required if there are `dimensions` specified … by doing something like this:

```
weight: {
    type: Number,
    min: 0,
    required: function () {
        // Require weight only if all dimensions are specified
        return this.dimensions && (
               this.dimensions.length > 0 &&
               this.dimensions.width > 0 &&
               this.dimensions.height > 0);
    }
}
```

We can even make things more complicated by adding a validator to the weight so that if a weight exists, then all three dimensions must also exist, otherwise it is an error. Here is how we can do this:

```
weight: {
    type: Number,
    min: 0,
    validate: {
        validator: function (v) {
            // If no weight, skip validation
            if (v == null) return true;

            // If weight exists, all three dimensions must be > 0
            return this.dimensions &&
                   this.dimensions.length > 0 &&
                   this.dimensions.width > 0 &&
                   this.dimensions.height > 0;
        },
        message: 'Weight requires all three dimensions (length, width, height).'
    }
}
```

As we can see, there is a lot of flexibility when writing validation code for our schemas.

Notice, in the example above, that we can supply a custom message for the validating. We can even provide custom error messages for each of our validation rules in **Mongoose**. To do this, we use array notation, where the first element is the validator value, and the second element is the error message:

```
  price: {
       type: Number,
       required: [true, "A price is required for this product."],
       min: [0, "Price must be positive."]
    }
```

This works for built-in validators such as `required`, `min`, `max`, `minlength`, `maxlength`, and `enum`, whereas for the custom validator functions, we use the `message` property (as we saw earlier).

So, what happens when any one of these validation checks fails? Well, first of all … when doing a `save()` or `create()` operation, the data will NOT written to the database. With `update()` operations, however, the behavior is a little different. By default, updates skip validation, so the

changes will still be applied even if they violate the schema rules. To enforce validation during an update, we need to include `{ runValidators: true }` in the options … we will see this later. In that case, **Mongoose** will run the same validation checks, and any failures will prevent the database from being updated.

Next, a `mongoose.Error.ValidationError` is thrown (or passes it to the callback). This error object contains detailed information about the validation failures. Each invalid field has its own entry inside the error text, so if multiple fields fail validation, we will see one error entry per field.

The `ValidationError` object has two very useful fields:

- `message` - a string which is a concatenated message of all validation errors.
- `errors` - an object whose keys are invalid field names and whose values are individual `ValidationError` objects.

We can iterate through each of the errors in the `errors` object and ask for these properties:

- `message` - a human-readable message (e.g., "Path \age` (12) is less than minimum allowed value (18).`")
- `kind` - the type of validation that failed (e.g., "required", "min", "max", "enum", etc.).
- `path` - the field name (e.g., "age").
- `value` - the invalid value that was passed in (12).

Recall that we use a **try**/**catch** block to handle any errors on a `save()`, `create()` or `update()`:

```
// Save it to the database
try {
    await newProduct.save();
    console.log("Product saved:", newProduct);
} catch (err) {
    console.error("Error saving product:", err);
}
```

We can write more code in the **catch** block to handle each of these errors and decide what to do. Here is an example of iterating through the errors and just displaying the information:

```
// Save it to the database
try {
    await newProduct.save();
    console.log("Product saved:", newProduct);
} catch (err) {
    console.error("Error saving product:", err);
    for (const e in err.errors) {
        console.log("Kind:", err.errors[e].kind);       // "max"
        console.log("Message:", err.errors[e].message); // "Path `rating` (6) is more than …"
        console.log("Path:", err.errors[e].path);       // "rating"
        console.log("Value:", err.errors[e].value);     // 6
    }
}
```

Clearly, validation in **Mongoose** is very useful because it lets us automatically enforce rules on our data without having to write extensive custom checks. If typecasting fails or a value doesn't meet the defined criteria, **Mongoose** will throw an error that we can catch and handle, making our code cleaner, more reliable, and easier to maintain.

However, having many complex schemas directly in our server code can make it messy. To keep things organized, we can define each schema in its own module and export the mode.

For example, we could make a file called **ProductModel.js** that looks like this:

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

let productSchema = Schema({
    name: { type: String, required: true, trim: true, minlength: 2, maxlength: 100 },
    price: { type: Number, required: true, min: 0 },
    stock: { type: Number, default: 0, min: 0 },
    dimensions: {
        length: { type: Number, min: 0 },
        width: { type: Number, min: 0 },
        height: { type: Number, min: 0 } },
    weight: { type: Number, min: 0 }
});
module.exports = mongoose.model("Product", productSchema);
```

This way, the schema and model setup is encapsulated, keeping the server code clean and easier to maintain. Then we could use `require()` in our code to make use of it. Here is some code to do this (i.e., **test-product-model.js**):

```
const mongoose = require("mongoose");
const db = require("./mongooseDB.js"); // Our connection code shown earlier

const Product = require("./ProductModel");

async function main() {
    let connection;
    try {
        // Connect to database
        connection = await db.connectToDatabase();

        // Fetch all products
        const products = await Product.find();
        console.log(products);
    } catch (err) {
        console.error("Error fetching products:", err);
    } finally {
        // Disconnect
        await db.disconnectFromDatabase();
    }
}

main();
```

Of course, the code assumes that we have a product database all set up.

14.3 Server Queries

Let's discuss the execution of queries in **Mongoose**. In the code that we just looked at, we performed a `find()` query that returned a collection of products:

```
const products = await Product.find();
```

`Product.find()` produces a **Mongoose Query** object. The query isn't executed immediately; it only runs when we use `await` on it, call `.then()`, or use `.exec()` (note: we will not discuss the use of **.then** and **.exec**, but will use **await**). By deferring the execution like this, we can chain additional constraints or modifiers onto the query before sending it to the database.

In contrast, in the native **MongoDB** driver, calling `.find()` returns a **Cursor** object, not the documents themselves. A cursor is an iterator that lets us iterate through the results one document at a time, which is useful because it avoids loading the entire result set into memory at once. This is good for streaming and when dealing with massive data sets. **Mongoose**, however, is more flexible when it comes to building up the query, as we will see.

Assume that we want to build up the query before executing it. We would start as follows:

```
const query = Product.find();
```

Now we can chain together a number of methods to create more specific queries. The `.where()` method allows us to focus on a particular field or condition and chain additional constraints. For example, the following two queries are equivalent:

```
    const query = Product.find({rating: 5 });
    const query = Product.find().where({rating: 5 });
```

Both return all products with a rating of **5**. But using `.where()` is very useful when we want to chain multiple conditions or comparisons in a readable way by using additional methods. Here are some that we can use:

Method	Example	What it does
`.equals()`	`.where("category").equals("toys")`	match if equal
`.gt() .gte()`	`.where("price").gt(10)`	match if > or >=
`.lt() .lte()`	`.where("price").lt(100)`	match if < or <=
`.in()`	`.where("tags").in(["red","blue"])`	match if field value is in the given array
`.nin()`	`.where("tags").nin(["green","white"])`	match if field value is not in the array
`.ne()`	`.where("stock").ne(0)`	match if not equal
`.regex()`	`.where("name").regex(/car/i)`	match using a regular expression

We can chain many together as follows:

```
// Find 5-star-rated products in the range from $10-$100
const query = Product.find().where({ rating: 5 }).where("price").gte(10).where("price").lte(100);
```

Notice how multiple `.where()` calls are chained together with additional constraints. All of these will build up the query which can then be executed. If we are done building up the query with just these constraints, then we can actually just write `await` in front, to execute it. But then the result will be the collection of products, not the query:

```
// Find 5-star-rated products in the range from $10-$100
const products = await Product.find()
    .where({ rating: 5 })
    .where("price").gte(10)
    .where("price").lte(100);
```

As a comparison, here is what "equivalent" code would look like in native **MongoDB**:

```
const collection = db.collection("products");
const products = await collection.find({rating: 5, price: {$gte: 10, $lte: 100}}).toArray();
```

While this works perfectly, many developers find the **Mongoose** version more readable and easier to build step-by-step, especially when chaining multiple conditions or modifiers. So, if we had to perform computations in between (perhaps over time based on user input), we could build the query step-by-step and then execute it as follows:

```
let query = Product.find();

query = query.where({ rating: 5 });

// ... do something to determine the min/max price to search ...
const minPrice = 10;
const maxPrice = 100;
// ...
query = query.where("price").gte(minPrice);
query = query.where("price").lte(maxPrice);

const products = await query.exec();
```

There are a lot of things that we can say about the `.regex()` method since there are many options … making it powerful for pattern matching in text fields. Without getting into too many details, here are a few examples along with explanations:

Example using .where("name")	Description
`.regex(/car/i)`	name contains "car", case-insensitive
`.regex(/.*ener.*/)`	name contains substring "ener" anywhere
`.regex(/^car/)`	name starts with "car"
`.regex(/car$/)`	name ends with "car"
`.regex(/^car$/i)`	name is exactly "car", ignoring case
`.regex(/[aeiou]/i)`	name contains at least one vowel, case-insensitive
`.regex(/^[A-Z]/)`	name starts with an uppercase letter
`.regex(/^(toy|car)/i)`	name starts with either "toy" or "car", case-insensitive
`.regex(/[^a-z]/)`	name contains any character that is not a lowercase letter

Recall how we could use *projections* in **MongoDB** to specify a subset of fields to be returned. The 2nd parameter here is a projection, requesting only the **name** and **price**:

```
db.products.find({}, {name: 1, price:1, _id:0})
```

We ended up with this kind of result:

```
{ name: 'Laptop', price: 1200 },
{ name: 'Pizza', price: 9.99 },
{ name: 'Coke', price: 0.99 },
etc..
```

In Mongoose, we can achieve projection by using the `.select()` query method in our chain. We simply list the fields in a string separated by spaces:

```
const products = await Product.find()
    .where({ rating: 5 })
    .where("price").gte(10)
    .where("price").lte(100)
    .select("name price rating");  // only name, price, rating and _id fields are returned
```

By default, the **_id** field is automatically returned as well. Alternatively, we can use `.select()` to return all fields except specific ones by including a **-** in front of the fieldname:

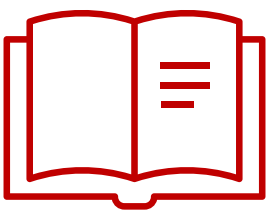
```
const products = await Product.find()
    .where({ rating: 5 })
    .where("price").gte(10)
    .where("price").lte(100)
    .select("-reviews -createdAt") // all except the review and createdAt fields
```

We cannot combine inclusion and exclusion in the same query except when it comes to **_id**. By default, the **_id** is always returned unless we explicitly exclude it as follows:

```
const products = await Product.find()
    .where({ rating: 5 })
    .where("price").gte(10)
    .where("price").lte(100)
    .select("name price rating -_id");  // name, price, rating fields, but no _id this time
```

There are also methods that are quite useful for pagination. The `.skip(n)` query method allows us to skip the first ***n*** results and the `.limit(n)` query method limits the number of results returned. Let's see how to use them. Consider building an online store page that only shows **10** products at a time. We would need to keep track of the page number and the page size of **10**. We can use `.skip()` and `.limit()` to return the correct products per page:

```
const page = 2;      // This will change as user goes to prev/next page
const pageSize = 10; // This is often adjustable on the webpage as well
const products = await Product.find()
    .where("price").gte(10).lte(100)
    .where("stock").gt(0)
    .skip((page - 1) * pageSize)   // skip first 10
    .limit(pageSize)               // only get 10
    .select("name price rating");  // or whatever we want to see
```



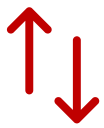
When doing pagination, it is also important to know how many documents there are so that we know how many pages there will be. This will allow us to know if we need to show a **next page** button or to hide it. Also, sometimes we want the user to know how many are there by showing them something like "Showing 11–20 of 85 results" or "Page 2 of 9". We need to the total number of products to be able to do that. The `.countDocuments()` method is what we need:

```
const totalProducts = await Product.countDocuments()
    .where("price").gte(10).lte(100)
    .where("stock").gt(0);
```

Notice that we still need the same query constraints that we used when figuring out the skip and limit values so that we have the same price range and stock value.

The `.sort()` method is also valuable because it lets us order results by one or more fields (i.e., primary, secondary, tertiary, etc.). We can supply multiple `.sort()` methods in the chain:

```
const products = await Product.find()
    .sort("price")     // primary sort: lowest to highest price
    .sort("-stock")    // secondary sort: highest to lowest stock
```



The use of the **-** in front of the field name indicates to sort in descending order. It is important to note that the order of the keys matters. The **first key** is the primary sort. If two products have the same price, the secondary key (i.e., stock) is used to break the tie. If there was a third `.sort()`, it would act as the next level of comparison. Here are two additional ways to do the same thing:

```
const products = await Product.find()
    .sort({ price: 1, stock: -1 });

const products = await Product.find()
    .sort("price -stock");
```

Model Methods

Mongoose provides model methods that are similar to **MongoDB** operations, but with the advantage that it understands our schema and enforces validation rules. These methods allow us to create, read, update, and delete documents using a consistent, chainable syntax. Here are the common **Mongoose** model methods:

Read:

- `find({…})` - return all matching documents.
- `findOne({…})` - return the first matching document.
- `findById(anId)` - find document with this `id` (same as `find({ _id: anId })`).

Update:

- `updateOne({…}, update)` - update the first match.
- `updateMany({…}, update)` - update all matches.
- `findByIdAndUpdate(_id, update, options)` - update by `_id`

- `findOneAndUpdate({…}, update, options)` - update the first match
- `replaceOne({…}, newDoc)` - completely replace a document.

Delete:

- `deleteOne({…})` - delete the first match.
- `deleteMany({…})` - delete all matches.
- `findByIdAndDelete(_id)` - delete by `_id`.
- `findOneAndDelete({…})` - delete the first match.

Create:

- `create(doc)` - create and save a new document.
- `insertMany([docs])` - insert multiple at once.

As a reminder, by default: methods like `update()`, `updateOne()`, `updateMany()`, and `findByIdAndUpdate()` bypass **Mongoose** schema validation unless we explicitly enable it by using the option `{runValidators: true}`. The best practice for updating a document involves finding the document, making the necessary changes and then saving it again:

```
const product = await Product.findById(someId);
product.price = product.price - 10;
await product.save();   // ensures schema validation
```

Keep in mind, however, that this code is not atomic. That is, if multiple clients update the same document at once, there could be race conditions where the above lines of code can be interrupted and corruption can occur. For simple cases, this is fine, however for high-concurrency scenarios it is better to write atomic code like this:

```
await Product.findByIdAndUpdate(someId, { $inc: { price: -10 } }, { runValidators: true });
```

Toys Example:

Let's do an example by putting everything all together. Assume that we have some similar toy data that we used in our testing for our **MongoDB** code a few chapters ago. Let's create a schema that has some realistic validation checking. We will create a **ToyProductModel.js** file that will contain the schema and export the model. We will put in some realistic validation checks. Make sure that you understand the code:

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

const toyProductSchema = new mongoose.Schema({
    name: {
        type: String,
        required: [true, "Product name is required"],
        trim: true,
        minlength: [2, "Name must be at least 2 characters long"],
        maxlength: [50, "Name cannot exceed 50 characters"]
    },
```

```
    category: {
        type: String,
        required: true,
        enum: ["Building","Dolls","Vehicles","Outdoor","Arts & Crafts","Puzzles","Remote Control"]
    },
    price: {
        type: Number,
        required: true,
        min: [0.01, "Price must be greater than zero"],
        max: [1000, "Price must be less than $1000"]
    },
    stock: {
        type: Number,
        required: true,
        min: [0, "Stock cannot be negative"],
        validate: {
            validator: Number.isInteger,
            message: "Stock must be an integer"
        }
    },
    ageRange: {
        type: String,
        required: true,
        match: [/^\d+\+$/, "Age range must be in the format 'N+' (e.g., '3+')"]
    }
});

module.exports = mongoose.model("ToyProduct", toyProductSchema);
```

There should be nothing shocking in the above code. We will also make some data (for testing purposes) that we will store in a separate file. In a real system, an employee would enter this data manually into the system through some kind of user interface via an *Admin Dashboard* or *Web Form* of some sort. We will put our toy data in a file called **toysData.js** which will be similar to what we did in **MongoDB**, but this time we will add a bunch of toys that will not pass some of our validation checks. Here is the file:

```
// Here are some toys that we will "try to" insert
const toys = [
    { name: "Hot Wheels Track Set", category: "Vehicles", price: 49.99, stock: 75, ageRange: "5+" },

    // Negative stock
    { name: "Ken Doll", category: "Dolls", price: 14.99, stock: -10, ageRange: "3+" },

    { name: "Play-Doh Party Pack", category: "Arts & Crafts", price: 12.99, stock: 200, ageRange: "3+" },
    { name: "Puzzle Globe", category: "Puzzles", price: 22.50, stock: 60, ageRange: "6+" },

    // Missing price
    { name: "Frisbee", category: "Outdoor", price: 0, stock: 50, ageRange: "6+" },

    { name: "Wooden Train Set", category: "Vehicles", price: 34.99, stock: 80, ageRange: "3+" },
    { name: "Barbie Dreamhouse", category: "Dolls", price: 199.99, stock: 25, ageRange: "3+" },

    // Price too high
    { name: "Luxury Dollhouse", category: "Dolls", price: 5000, stock: 5, ageRange: "3+" },

    { name: "RC Helicopter", category: "Remote Control", price: 89.99, stock: 15, ageRange: "10+" },
    { name: "Rubik's Cube", category: "Puzzles", price: 9.99, stock: 150, ageRange: "6+" },
```

```
    // Missing name
    { name: "", category: "Dolls", price: 29.99, stock: 10, ageRange: "4+" },
    // Missing category
    { name: "Superball Playset", price: 24.99, stock: 15, ageRange: "4+" },

    { name: "NERF Blaster Elite", category: "Outdoor", price: 39.99, stock: 60, ageRange: "8+" },

    // Invalid category
    { name: "Mystery Maze", category: "Electronics", price: 49.99, stock: 20, ageRange: "5+" },

    // Invalid fractional stock
    { name: "Crash Up Derby", category: "Vehicles", price: 19.99, stock: 4.5, ageRange: "5+" },

    { name: "LEGO Classic Bricks", category: "Building", price: 29.99, stock: 120, ageRange: "4+" },

    // Name too long
    { name: "Ultimate LEGO Creator Expert Modular City Expansion Set",
      category: "Outdoor", price: 39.99, stock: 25, ageRange: "8+" },

    { name: "Craft Bead Kit", category: "Arts & Crafts", price: 18.99, stock: 100, ageRange: "5+" },
    { name: "Foam Football", category: "Outdoor", price: 14.99, stock: 200, ageRange: "6+" },

    // Improper age format
    { name: "Family Fun Pack", category: "Puzzles", price: 9.99, stock: 30, ageRange: "three+" }
];

module.exports = toys;
```

Now for our test code. We will make use of the **mongooseDB.js** code (that we discussed at the start of the chapter) to make our connection to the database. The testing code will follow the same kind of test that we did for **MongoDB** in that we will add the products, display them, display products under $30 and then update the price of the "Barbie Dreamhouse" and verify that it changed and then finally replace the "NERF Blaster Elite" with a "RC Monster Truck" toy. We will name the file **test-mongoose-queries.js** and it will begin by requiring the needed files:

```
// Get the database connect/disconnect functions
const { connectToDatabase, disconnectFromDatabase } = require("./mongooseDB");
const Product = require("./ToyProductModel");  // our schema model
const toys = require("./toysData");            // the toys to insert
```

Of course, the main part of the code will take on the format as shown below … take note of the added code in the catch block to describe the errors. Also, take note of the `deleteMany()` at the start. This is called to clear the database each time we run the code (to avoid duplicates). Of course, we would only do this for testing purposes and never in a real system.

```
// We made a main() function because we are required to use "async"
async function main() {

    try {
        const db = await connectToDatabase();

        // Clear the collection first each time we run, since this is just a test
        await Product.deleteMany({});

        ... we will insert more code here ...
```

```
    } catch (err) {
        console.error("Error ... something went wrong:", err);
        for (const e in err.errors) {
            console.log(err.errors[e].kind);
            console.log(err.errors[e].message);
            console.log(err.errors[e].path);
            console.log(err.errors[e].value);
        }

    } finally {
        await disconnectFromDatabase();
    }
}
main();
```

Now, we will add the items one at a time so that we can wrap them in a **try**/**catch** that will show any errors. We need to do a **try**/**catch** block in a loop so that if one failed insert happens, the others can continue to be added. This code goes in the main outer **try** block where the yellow highlight is shown above:

```
        // Insert the toys in the above-defined array
        let count = 0;
        for (const t of toys) {
            try {
                let result = await Product.create(t);
                console.log("Added " + result.name);
                count++;
            }
            catch(err) {
                console.log("*** Error: Did not add " + t.name);
                for (const e in err.errors) {
                    console.log("   Path:", err.errors[e].path);
                    console.log("   Kind:", err.errors[e].kind);
                    console.log("   Value:", err.errors[e].value);
                    console.log("   Message:", err.errors[e].message);
                }
            }
        }
        console.log(count + " products were added.");
```

The code is straight forward. We use `create()` to add each toy product in the loop and keep count of how many were added successfully … ensuring to log any errors along the way.

The remainder of the code is similar to what we did in **MongoDB**, except that we no longer need an `await` in front of the FOR loops because we are not using a cursor object anymore. Instead, the functions give us back the lists of products asked for:

```
        // Display all added products
        let prods = await Product.find();
        console.log("Here are the products that were added:");
        for (const p of prods) {
            console.log(p);
        }
```

```
        // Display all products with a price under $30
        let cheap = await Product.find({price:{$lt:30}}, {name: 1, price:1});
        console.log("Here are the products under $30:");
        for (const p of cheap) {
            console.log(p);
        }

        // Update the price of the Barbie Dreamhouse
        result = await Product.updateOne({name:"Barbie Dreamhouse"},
                                         {$set: {price: 179.99}});
        if (result.acknowledged && result.modifiedCount > 0)
            console.log("Price has been updated for Barbie Dreamhouse");
        else
            console.log("Price has NOT been updated for Barbie Dreamhouse");

        // Verify the price of the Barbie Dreamhouse
        let product = await Product.findOne({name:"Barbie Dreamhouse"});
        console.log("Price is now at $" + product.price);

        // Replace the NERF Blaster Elite with a new version of the product
        result = await Product.replaceOne({name: "NERF Blaster Elite"},
            {name: "RC Monster Truck", category: "Remote Control",
             price: 59.99, stock: 40, ageRange: "6+"});
        if (result.acknowledged)
            console.log("NERF Blaster Elite has been replaced by RC Monster Truck");
        else
            console.log("NERF Blaster Elite has NOT been replaced");
```

So, what did we gain from all this **Mongoose** stuff so far? Mainly validation-checking. Also, it is a little easier to work with collections of objects rather than cursor objects.

14.4 Instance/Helper Methods and Populating

Mongoose documents have built-in methods such as `find()`, `save()`, etc., but we can also define our own ***instance methods***. This lets us give meaningful names to operations we perform frequently and keeps our code organized. For example, we can add a `getVolume()` method to `productSchema` to calculate the volume of the product instance … which can be useful for shipping or storage calculations.

Each schema has a property called **methods** that maintains a collection of functions that we can add to.

Here is how we can add a **getVolume()** function to `productSchema`:

```
productSchema.methods.getVolume = function() {
    return this.dimensions.length * this.dimensions.width * this.dimensions.height;
};
```

Similarly, we can add an `isRecent()` method to `reviewSchema` to check if a review is recent (e.g., less than a week old). This could be useful for highlighting recent reviews on a product page:

```
reviewSchema.methods.isRecent = function() {
    const oneWeek = 1000 * 60 * 60 * 24 * 7;
    return (Date.now() - this.date.getTime()) < oneWeek;
};
```

How can we use these with product instances? Here is an example:

```
const product = await Product.findById(productId);

console.log("Product volume:", product.getVolume());

product.reviews.forEach(review => {
    if (review.isRecent()) {
        console.log(`${review.user} wrote recent review: "${review.comment}"`);
    }
});
```

We can also define *asynchronous* instance methods that modify documents and interact with the database. For example, we could write a `sell(n)` that sells **n** units of a product. This method reduces the product's stock by **n** if sufficient stock is available, and throws an error otherwise.

Here is how we can implement it:

```
productSchema.methods.sell = async function(amount) {
    if (this.stock >= amount) {
        this.stock -= amount;
        await this.save();
        return `${amount} units sold. Remaining stock: ${this.stock}`;
    } else {
        throw new Error("Not enough stock");
    }
};
```

The logic is straightforward. It checks whether there is enough stock, reduces the stock if possible, saves the change to the database, and returns a confirmation message. If there isn't enough stock, it throws an error. To use this method, we can do the following:

```
const product = await Product.findById(productId);

try {
    const message = await product.sell(3);
    console.log(message);
} catch (err) {
    console.error(err.message);
}
```

Notice the use of `await` when calling `product.sell()`. This ensures that the stock is updated in the database before we attempt to log the result. Without `await`, we might see incorrect or unexpected messages.

For other instance method ideas, we can find products with similar names, prices, or other attributes. Later, we could factor in past purchasers and reviews to find related products. For example, we can define a `findSimilarProducts()` instance method on the `productSchema` that finds products whose category is the same, whose price is within **±20%** of this product's price and whose age range is within ± 1 year:

```
productSchema.methods.findSimilarProducts = async function() {
    // Convert this product's ageRange (e.g., "5+") to a number
    // by taking off the '+' and converting using base 10.
    const thisAge = parseInt(this.ageRange.replace(/\+$/, ""), 10);

    // Find products by category and price
    const candidates = await this.model("Product")
        .find({
            category: this.category,
            price: { $gte: this.price * 0.8, $lte: this.price * 1.2 },
            ageRange: { $exists: true } // ensure ageRange exists
        });

    // Filter by numeric age ±1 year
    return candidates.filter(p => {
        const pAge = parseInt(p.ageRange.replace(/\+$/, ""), 10);
        return (pAge >= thisAge - 1) && (pAge <= thisAge + 1);
    });
};
```

And using it is also quite easy:

```
const product = await Product.findById(productId);

const related = await product.findSimilarProducts();
console.log("Similar products:", related);
```

From these examples, we can see that instance methods operate on document instances, not the model itself. They encapsulate operations we perform often, so we don't have to repeat the same code in multiple places, making our code cleaner and easier to maintain. They work for both top-level documents (e.g., `Product`) and subdocuments (e.g., `Review`).

Query Helper Methods:

In addition to instance methods, **Mongoose** allows us to define ***query helper*** methods. These methods extend the default query functionality (e.g., `find()`, `where()`, `gt()`, etc.) and let us assign a descriptive name to a common query pattern. The key difference from instance methods is that instance methods operate on **a single document**, whereas ***query helper*** methods operate on **query chains**, so they can be combined with other queries for more complex filtering.

For example, consider this code to search for products that contain some text (e.g., toy). We could do this by using regular expressions. This code searches for all products with **"toy"** in the name (the 'i' indicates case-insensitive):

```
const products = await Product.find().where({ name: new RegExp("toy", 'i') });
```

To make it more readable, we could add a `byName` query helper method to `productSchema` that does this for us:

```
productSchema.query.byName = function(name) {
    return this.where({ name: new RegExp(name, 'i') });
};
```

Then to use it, we simply append it in our query chain:

```
// Find all products with names containing "toy"
const products = await Product.find().byName("toy");
console.log(products);
```

This makes our query more readable and reusable.

Referencing Objects:

Recall earlier, that we used an `ObjectId` along with a `ref: 'User'` to refer to a specific **User** object in the database:

```
const reviewSchema = new mongoose.Schema({
    user: { type: mongoose.Schema.Types.ObjectId, required: true, ref: 'User' },
    rating: { type: Number, required: true, min: 1, max: 5 },
    comment: { type: String, maxlength: 1000 },
    date: { type: Date, default: Date.now }
});
```

Once we have that ID, we can access everything about the **User** that created this product review. In fact, for each user, we may also want to keep track of the products that they reviewed. Again, we can use the `ref:` key for this. Here is an example of what our `userSchema` may look like:

```
const userSchema = new mongoose.Schema({
    username: { type: String, required: true, unique: true, trim: true },
    email: { type: String, required: true, unique: true, lowercase: true },
    password: { type: String, required: true },
    location: { type: String },
    joinedAt: { type: Date, default: Date.now },

    // References to all products this user has reviewed
    reviewedProducts: [ { type: mongoose.Schema.Types.ObjectId, ref: "Product" } ]
});
```

An alternative to storing the ID would be to store some of the information from the user right into the `reviewSchema`:

```
const reviewSchema = new mongoose.Schema({
    user: { username: { type: String, required: true }, location: { type: String } },
    rating: { type: Number, required: true, min: 1, max: 5 },
    comment: { type: String, maxlength: 1000 },
    date: { type: Date, default: Date.now }
});
```

In this embedded approach, each review stores a small snapshot of the user's public information, such as their **username** and **location**. This data is duplicated in the review document so it can be displayed directly without querying the **User** collection. Only these selected fields are embedded … any other private or sensitive information, like email or password, is not included and remains in the **User** collection.

There are advantages to storing the ID instead of storing embedded information:

- ✓ saves storage space (i.e., no duplicated data)
- ✓ reduces the amount of data transferred in queries
- ✓ makes updates much simpler.

There are also disadvantages of storing the ID instead of the embedded information:

- ✘ requires populating the reference to retrieve full user info (e.g., username and location).
- ✘ slightly more complex queries and potentially slower retrieval compared to embedded data.

Now, when we have the reference to the `user` from the review, or we have the product reference from the user's `reviewedProducts`, how do we get that full user or full product document? To retrieve the full document, **Mongoose** provides the `.populate()` method, which fetches the referenced documents and replaces the **ObjectIds** with the actual document data.

So, if for example, we wanted to extract the username and location for each review of a product, we could do this:

```
// Get a product with all its reviews and populate the user info for each review
const product = await Product.findById(productId)
    .populate("reviews.user", "username location"); // only bring back username & location

console.log(product.reviews);
```

In this example, we first find a product by its `productId`. Each review in `product.reviews` contains a `user` field that stores only the user's `ObjectId`. By calling `.populate()` as shown above, **Mongoose** fetches the full **User** documents for those `ObjectIds` and replaces the `user` field in each review with an object containing only the `username` and `location`. This way, we can directly access `review.user.username` and `review.user.location` without needing a separate query.

The console output would look something like this:

```
[
  {
    user: { _id: "64f8c9a2b1a3e5a7d2f12345", username: "Yuki", location: "Tokyo, Japan" },
    rating: 5,
    comment: "Amazing quality, my nephew loved it!",
    date: 2025-09-08T18:34:12.000Z
  },
  {
    user: { _id: "64f8c9a2b1a3e5a7d2f12346", username: "Carlos", location: "Madrid, Spain" },
    rating: 3,
    comment: "It's okay, but shipping took too long.",
    date: 2025-09-06T14:20:00.000Z
  },
  {
    user: { _id: "64f8c9a2b1a3e5a7d2f12347", username: "Amina", location: "Cairo, Egypt" },
    rating: 4,
    comment: "Good value for the price, sturdy enough.",
    date: 2025-09-05T11:45:00.000Z
  }
]
```

We could also access the user information directly:

```
product.reviews.forEach(review => {
    console.log(`${review.user.username} from ${review.user.location} rated ${review.rating}`);
});
```

Which would display …

```
Yuki from Tokyo, Japan rated 5
Carlos from Madrid, Spain rated 3
Amina from Cairo, Egypt rated 4
```

From this chapter, we saw that **Mongoose** provides a powerful framework for working with **MongoDB**. It allows us to …

- ✓ define schemas and models to structure and validate data,
- ✓ manage relationships between documents to connect related data efficiently,
- ✓ create instance methods for document-specific logic, and
- ✓ define query helpers for reusable, chainable queries.

Together, these features make it easier to build applications that are organized, maintainable, and scalable, while implementing real-world business logic efficiently.