# MASTERING MONGOOSE

The Official Mongoose eBook

By Valeri Karpov

To Pooka,

My loyal writing buddy who sat on my lap for most of this book.

# Table of Contents

# 0: How To Use This Book

Mongoose is an object-document mapping (ODM) framework for Node.js and MongoDB. It is the most popular database framework for Node.js in terms of npm downloads (over 800,000 per month) and usage on GitHub (over 939,000 repos depend on Mongoose).

I'm Valeri Karpov, the maintainer of Mongoose since April 2014. Mongoose was one of the first Node.js projects when it was written in 2010, and by 2014 the early maintainers had moved on. I had been using Mongoose since 2012, so I jumped on the opportunity to work on the project as fast as I could. I've been primarily responsible for Mongoose for most of its 10 years of existence.

This eBook contains the distilled experience of 8 years working with MongoDB and Mongoose, and 6 years working on Mongoose. The goal of this book is to take you from someone who is familiar with Node.js and MongoDB, to someone who can architect a Mongoose-based backend for a large scale business. Here are the guiding principles for this eBook:

- *Focus on principles that will last.* Like Victor Hugo once said, "If a writer wrote merely for his time, I would have to break my pen and throw it away." Mongoose's core concepts have stood the test of time thus far. Most of the ideas and lessons from this book would have been equally applicable in 2015, and I expect they will be equally applicable in 2025.
- *Macro from the micro.* This book focuses on how to use Mongoose in isolation, and only discusses building apps with Express and Mongoose in the final chapter. A lot of tutorials assume there is a way you *should* build an app, independent of business needs, existing code architecture, time constraints, or team composition. This eBook takes the opposite approach: it demonstrates principles for using Mongoose effectively, and then shows how you can apply those principles when building apps.
- *80/20 Rule.* **This book is not meant to be a complete reference.** That's what the documentation is for. This eBook intentionally omits certain functionality that, in the author's opinion, is either not broadly applicable or easy to pick up once you understand the fundamentals. In other words, this eBook is a leaner, more opinionated guide than the Mongoose documentation. Think of this eBook as a guided tour rather than a reference.

Before we get started, here's some technical details.

- All code examples assume Mongoose v5.x unless otherwise specified.
- This book uses the `#` symbol as a convenient shorthand for `.prototype.`. For example, `Document#save()` refers to the `save()` method on instances of the `Document` class.

Mongoose is a powerful tool for building backend applications. Mongoose provides change tracking and schema validation out of the box, letting you work with a schema-less database without worrying about malformed data. Mongoose also provides tools for customizing your database interactions, like middleware and plugins. Mongoose's community also supports a wide variety of plugins for everything from pagination to encryption to custom types.

Are you ready to become a Mongoose master? Let's get started!

# 1: Getting Started

## 1.1: What is Mongoose?

Mongoose is a object-document mapping (ODM) framework for Node.js and MongoDB. This definition is nuanced, so here's an overview of the core benefits of using Mongoose.

### Core Benefits

Mongoose is built on top of the official MongoDB Node.js Driver, which this book will refer to as just "the driver". The driver is responsible for maintaining a connection to MongoDB and serializing commands into MongoDB's wire protocol. The driver is an rapidly growing project and it is entirely possible to build an application without Mongoose. The major benefits of using Mongoose over using the driver directly are:

- **Application-level schema validation:** Mongoose can validate that untrusted data coming in over the network conforms to a declarative schema.
- **Models:** Model-View-Controller (MVC) architecture enjoys widespread adoption. Mongoose models make it easy to write MongoDB code using standard MVC practices.
- **Change tracking:** Mongoose converts vanilla JavaScript assignments into MongoDB atomic operations.
- **Middleware:** Mongoose middleware helps you handle cross cutting concerns. Instead of adding a log statement every time you call `save()`, you can put a single log statement in `pre('save')`.
- **Plugins:** Mongoose has a robust plugin architecture and an active community. As of early 2020, there are over 3000 modules on npm with "mongoose" as a keyword.

In order to clarify what Mongoose is, let's take a look at how Mongoose differs from other modules you may have used.

### ODM vs ORM

Mongoose is an ODM, not an ORM (object-relational mapping). Some common ORMs are ActiveRecord, Hibernate, and Sequelize. Here's the key difference between ORMs and ODMs: ORMs store data in relational databases, so data from one ORM object may end up in multiple rows in multiple tables. In other words, how your data looks in JavaScript may be completely different from how your ORM stores the data.

MongoDB allows storing arbitrary objects, so Mongoose doesn't need to slice your data into different collections. Like an ORM, Mongoose validates your data, provides a middleware framework, and transforms vanilla JavaScript operations into database operations. But unlike an ORM, Mongoose ensures your objects have the same structure in Node.js as they do when they're stored in MongoDB.

Node.js

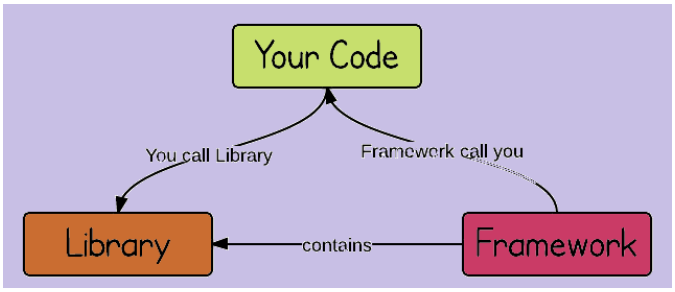
```
const doc = new MyModel({ name: 'foo', tags: ['bar'] });
await doc.save();
```

MongoDB

```
rs:PRIMARY> db.tests.findOne()
{
        "_id" : ObjectId("5c5c52fd785cdd08cedf1fbc"),
        "tags" : [
                "bar"
        ],
        "name" : "foo",
        "__v" : 0
}
rs:PRIMARY>
```

### Framework vs Library

Mongoose is more a framework than a library, although it has elements of both. For the purposes of this book, a *framework* is a module that executes your code for you, whereas a *library* is a module that your code calls. Modules that are either a pure framework or a pure library are rare. But, for example, Lodash is a library, Mocha is a framework, and Mongoose is a bit of both.




A database driver, like the MongoDB Node.js driver, is a library. It exposes several functions that you can call, but it doesn't provide middleware or any other paradigm for structuring code that uses the library. On the other hand, Mongoose provides middleware, custom validators, custom types, and other paradigms for code organization.

The MongoDB driver doesn't prescribe any specific architecture. You can build your project using traditional MVC architecture, aspect-oriented, reactive extensions, or anything else. Mongoose, on the other hand, is designed to fit the "model" portion of MVC or MVVM.

### Summary

Mongoose is an ODM for Node.js and MongoDB. It provides schema validation, change tracking, middleware, and plugins. Mongoose also makes it easy to build apps using MVC patterns. Next, let's take a look at some basic Mongoose patterns that almost all Mongoose apps share.

## 1.2 Connecting to MongoDB

To use Mongoose, you need to open at least one connection to a MongoDB server, replica set, or sharded cluster.

This book will assume you already have a MongoDB instance running. If you don't have MongoDB set up yet, the easiest way to get started is a cloud instance using MongoDB Atlas' free tier. If you

prefer a local MongoDB instance, run-rs is an npm module that automatically installs and runs MongoDB for you.

Here's how you connect Mongoose to a MongoDB instance running on your local machine on the default port.

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

mongoose.connect('mongodb://localhost:27017/mydb');
```

The first parameter to `mongoose.connect()` is known as the *connection string*. The connection string defines which MongoDB server(s) you're connecting to and the name of the database you want to use. More sophisticated connection strings can also include authentication information and configuration options.

The `mydb` section of the connection string tells Mongoose which database to use. MongoDB stores data in the form of *documents*. A document is essentially a Node.js object, and analagous to a row in SQL databases. Every document belongs to a collection, and every collection belongs to a database. For the purposes of this book, you can think of a database as a set of collections.

Although connecting to MongoDB is an asynchronous operation, you don't have to wait for connecting to succeed before using Mongoose. You can, and generally should, `await` on `mongoose.connect()` to ensure Mongoose connects successfully.

```
await mongoose.connect('mongodb://localhost:27017/mydb');
```

However, many Mongoose apps do not wait for Mongoose to connect because it isn't strictly necessary. Don't be surprised if you see `mongoose.connect()` with no `await` or `then()`.

## 1.3 Defining a Schema

To store and query MongoDB documents using Mongoose, you need to define a model. Mongoose models are the primary tool you'll use for creating and loading documents. To define a model, you first need to define a *schema*.

In general, Mongoose schemas are objects that configure models. In particular, schemas are responsible for defining what properties your documents can have and what types those properties must be.

For example, suppose you're creating a model to represent a product. A minimal product should have a string property `name` and a number property `price` as shown below.

```
const productSchema = new mongoose.Schema({
  // A product has two properties: `name` and `price`
  name: String,
  price: Number
});

// The `mongoose.model()` function has 2 required parameters:
// The 1st param is the model's name, a string
// The 2nd param is the schema
const Product = mongoose.model('Product', productSchema);

const product = new Product({
  name: 'iPhone',
  price: '800', // Note that this is a string, not a number
  notInSchema: 'foo'
});

product.name; // 'iPhone'
product.price; // 800, Mongoose converted this to a number
// undefined, Mongoose removes props that aren't in the schema
product.notInSchema;
```

The `mongoose.model()` function takes the model's name and schema as parameters, and returns a class. That class is configured to cast, validate, and track changes on the paths defined in the schema. Schemas have numerous features beyond just type checking. For example, you can make Mongoose lowercase the product's name as shown below.

```
const productSchema = new mongoose.Schema({
  // The below means the `name` property should be a string
  // with an option `lowercase` set to `true`.
  name: { type: String, lowercase: true },
  price: Number
});
const Product = mongoose.model('Product', productSchema);

const product = new Product({ name: 'iPhone', price: 800 });

product.name; // 'iphone', lowercased
```

Internally, when you instantiate a Mongoose model, Mongoose defines native JavaScript getters and setters for every path in your schema on every instance of that model. Here's an example of a

simplified `Product` class implemented using ES6 classes that mimics how a Mongoose model works.

```
class Product {
  constructor(obj) {
    // `_doc` stores the raw data, bypassing getters and setters
    // Otherwise `doc.name = 'foo'` causes infinite recursion
    this._doc = {};
    Object.assign(this, { name: obj.name, price: obj.price });
  }

  get name() { return this._doc.name; }
  set name(v) { this._doc.name = v == null ? v : '' + v; }

  get price() { return this._doc.price; }
  set price(v) { this._doc.price = v == null ? v : +v; }
}

const p = new Product({ name: 'iPhone', price: '800', foo: 'bar' });
p.name; // 'iPhone'
p.price; // 800
p.foo; // undefined
```

### 1.4 Storing and Querying Documents

In Mongoose, a model is a class, and an instance of a model is called a document. This book also defined a "document" as an object stored in a MongoDB collection in section 1.2. These two definitions are equivalent for practical purposes, because a Mongoose document in Node.js maps one-to-one to a document stored on the MongoDB server.

There are two ways to create a Mongoose document: you can instantiate a model to create a new document, or you can execute a query to load an existing document from the MongoDB server.

To create a new document, you can use your model as a constructor. This creates a new document that has **not** been stored in MongoDB yet. Documents have a `save()` function that you use to persist changes to MongoDB. When you create a new document, the `save()` function sends an `insertOne()` operation to the MongoDB server.

```
// `doc` is just in Node.js, you haven't persisted it to MongoDB yet
const doc = new Product({ name: 'iPhone', price: 800 });

// Store `doc` in MongoDB
await doc.save();
```

That's how you create a new document. To load an existing document from MongoDB, you can use the `Model.find()` or `Model.findOne()` functions. There are several other query functions you can use that you'll learn about in Chapter 3.

```
// `Product.findOne()` returns a Mongoose _query_. Queries are
// thenable, which means you can `await` on them.
let product = await Product.findOne();
product.name; // "iPhone"
product.price; // 800

const products = await Product.find();
product = products[0];
product.name; // "iPhone"
product.price; // 800
```

Connecting to MongoDB, defining a schema, creating a model, creating documents, and loading documents are the basic patterns that you'll see in almost every Mongoose app. Now that you've seen the verbs describing what actions you can do with Mongoose, let's define the nouns that are responsible for these actions.

# 2: Core Concepts

So far, you've seen the basic patterns of working with Mongoose: connecting to MongoDB, defining a schema, creating a model, creating a document, and storing a document. In order to do this, Mongoose has 5 core concepts. Let's take a look at these 5 concepts and how they interact.

## 2.1: Models, Schemas, Connections, Documents, Queries

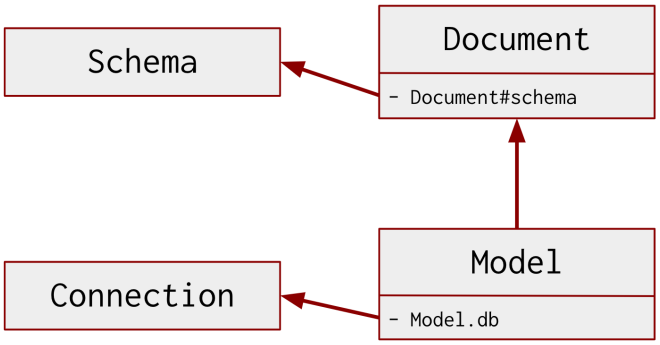
Mongoose has 5 core concepts that every other concept in the framework builds on. The 5 concepts are:

- A *Model* is a class representing an object stored in MongoDB. Models are the primary way you interact with MongoDB using Mongoose.
- A *Schema* is a class representing a model configuration.
- A *Connection* is a class representing a collection of sockets that connect to one or more MongoDB server processes
- A *Document* is an instance of a model. You can `save()` a document to MongoDB.
- A *Query* is class representing an operation that Mongoose will send to MongoDB.

Here's an example of the 5 core concepts in action:

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

// Here's how you create a connection:
const conn = mongoose.createConnection();
conn instanceof mongoose.Connection; // true

// Here's how you create a schema
const schema = new mongoose.Schema({ name: String });

// Calling `conn.model()` creates a new model. In this book
// a "model" is a class that extends from `mongoose.Model`
const MyModel = conn.model('ModelName', schema);
Object.getPrototypeOf(MyModel) === mongoose.Model; // true

// You shouldn't instantiate the `Document` class directly.
// You should use a model instead.
const document = new MyModel();
document instanceof MyModel; // true
document instanceof mongoose.Document; // true

const query = MyModel.find();
query instanceof mongoose.Query; // true
```

Models are the primary concept you will use to interact with MongoDB using Mongoose. In order to create a model you need a connection and a schema. A schema configures your model, and a connection provides your model a low-level database interface. Below is a class diagram representing how the `Document`, `Model`, `Connection`, and `Schema` classes are related.




Under the hood, Mongoose's `Model` class inherits from the `Document` class. When you create a new model using `mongoose.model()`, you create a new class that inherits from `Model`. However, for convenience, Mongoose developers typically use the term "model" to refer to classes that inherit from `Model`, and the term "document" to refer to instances of models.

## 2.2: Documents: Casting, Validation, and Change Tracking

There's two ways to create a model in Mongoose. You can call the top-level `mongoose.model()` function, which takes 2 parameters: a model name and a schema.

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

mongoose.model('MyModel', new mongoose.Schema({}));
```

You can also create a new connection and call the connection's `model()` function:

```
const conn = mongoose.createConnection();

conn.model('MyModel', new mongoose.Schema({}));
```

Remember that a model needs both a connection and a schema. Mongoose has a default connection `mongoose.connection`. That is the connection `mongoose.model()` uses.

```
// `mongoose.model()` uses the default connection
mongoose.model('MyModel', schema);
// So the below function call does the same thing
mongoose.connection.model('MyModel', schema);
```

Note that the return value of `mongoose.model()` is **not** an instance of the `Model` class. Rather, `mongoose.model()` returns a class that extends from `Model`.

```
const MyModel = mongoose.model('MyModel', new mongoose.Schema({}));

// `MyModel` is a class that extends from `mongoose.Model`, _not_
// an instance of `mongoose.Model`.
MyModel instanceof mongoose.Model; // false
Object.getPrototypeOf(MyModel) === mongoose.Model; // true
```

Because `Document` and `Model` are distinct classes, some functions are documented under the `Document` class and some under the `Model` class. For example, `save()` is defined on the `Model` class, not the `Document` class. Keep this in mind when reading Mongoose documentation.

Mongoose documents are responsible for *casting*, *validating*, and *tracking changes* on the paths defined in your schema.

### Casting

*Casting* means converting values to the types defined in your schema. Mongoose converts arbitrary values to the correct type, or tracks a `CastError` if it failed to cast the specified value. Calling `save()` will fail if there was a `CastError`.

```
const schema = Schema({ name: String, age: Number });
const MyModel = mongoose.model('MyModel', schema);

const doc = new MyModel({});
doc.age = 'not a number';
// Throws CastError: Cast to Number failed for value
// "not a number" at path "age"
await doc.save();
```

If casting fails, Mongoose will *not* overwrite the current value of the path. In the below example, `doc.age` will still be 59 after trying to assign 'not a number'.

```
const schema = Schema({ name: String, age: Number });
const Person = mongoose.model('Person', schema);

const doc = new Person({ name: 'Jean-Luc Picard', age: 59 });
doc.age = 'not a number';
doc.age; // 59
```

You can clear the cast error by setting `age` to a value Mongoose can cast to a number. In the below example, Mongoose will successfully save the document, because setting `age` to a number clears the cast error.

```
const doc = new Person({ name: 'Jean-Luc Picard', age: 59 });

doc.age = 'not a number';
doc.age; // 59

doc.age = '12';
doc.age; // 12 as a number

// Saving succeeds, because Mongoose successfully converted the
// string '12' to a number and cleared the previous CastError
await doc.save();
```

By default, Mongoose casting considers `null` and `undefined` (collectively known as *nullish* values) as valid values for **any** type. For example, you can set `name` and `age` to nullish values and save the document.

```
doc.age = null;
doc.name = undefined;
// Succeeds, because Mongoose casting lets `null` and `undefined`.
// `name` and `age` will _both_ be `null` in the database.
await doc.save();
```

**Note:** Even though `name` is set to `undefined`, Mongoose will store it as `null` in the database. MongoDB does support storing `undefined`, but the MongoDB Node.js driver does not support `undefined` because the `undefined` type has been deprecated for a decade, see issue mongodb/js-bson#134 on GitHub.

If you want to disallow `null` values for a path, you need to use validation.

### Validation

Validation is a separate concept, although closely related to casting. Casting converts a value to the correct type. After Mongoose casts the value successfully, validation lets you further restrict what values are valid. However, validation **can't** transform the value being validated, whereas casting may change the value.

When you call `save()`, Mongoose runs validation on successfully casted values, and reports a `ValidatorError` if validation fails.

```
const schema = new mongoose.Schema({
  name: {
    type: String,
    // `enum` adds a validator to `name`. The built-in `enum`
    // validator ensures `name` will be one of the below values.
    enum: ['Jean-Luc Picard', 'William Riker', 'Deanna Troi']
  }
});
const Person = mongoose.model('Person', schema);
const doc = new Person({ name: 'Kathryn Janeway' });
await doc.save().catch(error => {
  // ValidationError: Person validation failed: name:
  // `Kathryn Janeway` is not a valid enum value for path `name`.
  error;
  // ValidatorError: `Kathryn Janeway` is not a valid enum
  // value for path `name`.
  error.errors['name'];
  Object.keys(error.errors); // ['name']
});
```

If `validate()` fails, Mongoose rejects the promise with a `ValidationError`. The `ValidationError` has an `errors` property that contains a hash mapping errored paths to the corresponding `ValidatorError` or `CastError`. In other words, a `ValidationError` can contain multiple `ValidatorError` objects and/or `CastError` objects.

Mongoose documents have a `validate()` function that you can use to manually run validation. The `save()` function calls `validate()` internally, so as a Mongoose user you are not responsible for running validation yourself. However, you may run validation on your own using `validate()`. Just remember: the `validate()` function is asynchronous, so you need to `await` on it.

Mongoose has numerous built-in validators, but built-in validators only apply to certain types. For example, adding `enum` to a path of type `Number` is a no-op.

```
const schema = new mongoose.Schema({
  age: {
    type: Number,
    enum: [59, 60, 61]
  }
});
const Person = mongoose.model('Person', schema);
const doc = new Person({ age: 22 });
// No error, `enum` does nothing for `type: Number`
await doc.validate();
```

Here's a list of all the built-in validators Mongoose supports for different types:

- Strings: `enum`, `match`, `minlength`, `maxlength`
- Numbers: `min`, `max`
- Dates: `min`, `max`

There is one built-in validator that applies to all schema types: the `required` validator. The `required` validator disallows nullish values.

```
const schema = new mongoose.Schema({
  name: {
    type: String,
    // Adds the `required` validator to the `name` path
    required: true
  }
});
const Person = mongoose.model('Person', schema);
const doc = new Person({ name: null });
await doc.validate().catch(error => {
  // ValidationError: Person validation failed: name: Path
  // `name` is required.
  error;
  // ValidatorError: Path `name` is required.
  error.errors['name'];
});
```

### Change Tracking

Mongoose documents have *change tracking*. That means Mongoose tracks vanilla JavaScript assignments, like `doc.name = 'foo'` or `Object.assign(doc, { name: 'foo' })`, and converts them into MongoDB update operators. When you call `save()`, Mongoose sends your changes to MongoDB.

```
const MyModel = mongoose.model('MyModel', mongoose.Schema({
  name: String,
  age: Number
}));

const doc = new MyModel({});
doc.name = 'Jean Valjean';
doc.age = 27;
await doc.save(); // Persist `doc` to MongoDB
```

When you load a document from the database using a query, change tracking means Mongoose can determine the minimal update to send to MongoDB and avoid wasting network bandwidth.

```
// Mongoose loads the document from MongoDB and then _hydrates_ it
// into a full Mongoose document.
const doc = await MyModel.findOne();
doc.name; // "Jean Valjean"
doc.name = 'Monsieur Leblanc';
doc.modifiedPaths(); // ['name']
// `save()` only sends updated paths to MongoDB. Mongoose doesn't
// send `age`.
await doc.save();
```

A Mongoose document is an instance of a Mongoose model. Documents track changes, so you can modify documents using vanilla JavaScript operations and rely on Mongoose to send a minimal update to MongoDB when you `save()`. Mongoose does more with change tracking than just structuring MongoDB updates: it also makes sure your JavaScript assignments conform to a schema. Let's take a closer look at schemas.

## 2.3 Schemas and SchemaTypes

In Mongoose, a *schema* is a configuration object for a model. A schema's primary responsibility is to define the *paths* that the model will cast, validate, and track changes on.

Let's take a closer look at the schema from example 2.2.1. This schema has two explicitly defined paths: `name` and `age`. Mongoose also adds an `_id` path to schemas by default. You can use the `Schema#path()` function to get the SchemaType instance associated with a given path as shown below.

```
const schema = Schema({ name: String, age: Number });
Object.keys(schema.paths); // ['name', 'age', '_id']

schema.path('name'); // SchemaString { path: 'name', ... }
schema.path('name') instanceof mongoose.SchemaType; // true

schema.path('age'); // SchemaNumber { path: 'age', ... }
schema.path('age') instanceof mongoose.SchemaType; // true

// The `SchemaString` class inherits from `SchemaType`.
schema.path('name') instanceof mongoose.Schema.Types.String; // true
```

A Mongoose SchemaType, like the `SchemaString` class, does **not** contain an actual value. There is no string value associated with an instance of `Schema.Types.String`. A `SchemaType` instance

is a configuration object for an individual path: it describes what casting, validation, and other Mongoose features should apply to a given path.

Many Mongoose developers use the terms "schema" and "model" interchangeably. Don't do this! A schema **configures** a model, but a schema instance can't write to the database. Similarly, a document is **not** an instance of a schema. Documents are instances of models, and models have an associated schema. In object-oriented programming terminology, a model **has a** schema, and a model **is a** document.

So far, the only SchemaType features you've seen are casting and validation. In the next section, you'll learn about another SchemaType feature: getters and setters.

## 2.4 Getters and Setters

In Mongoose, getters/setters allow you to execute custom logic when getting or setting a property on a document. Getters are useful for converting data that's stored in MongoDB into a more human-friendly form, and setters are useful for converting user data into a standardized format for storage in MongoDB. First, let's take a look at getters.

### Getters

Mongoose executes your custom getter when you access a property. In the below example, the string path `email` has a custom getter that replaces `'@'` with `' [at] '`.

```
const schema = new mongoose.Schema({ email: String });

// `v` is the underlying value stored in the database
schema.path('email').get(v => v.replace('@', ' [at] '));

const Model = mongoose.model('User', schema);
const doc = new Model({ email: 'test@gmail.com' });

doc.email; // 'test [at] gmail.com'
doc.get('email'); // 'test [at] gmail.com'
```

Keep in mind that getters do **not** impact the underlying data stored in MongoDB. If you save `doc`, the `email` property will be 'test@gmail.com' in the database. Mongoose applies the getter **every** time you access the property. In the above example, Mongoose runs the getter function twice, once for `doc.email` and once for `doc.get('email')`.

Another useful feature of getters is that Mongoose applies getters when converting a document to JSON, including when you call Express' `res.json()` function with a Mongoose document. This makes getters an excellent tool for formatting MongoDB data in RESTful APIs.

```
app.get(function(req, res) {
  return Model.findOne().
    // The `email` getter will run here
    then(doc => res.json(doc)).
    catch(err => res.status(500).json({ message: err.message }));
});
```

To disable running getters when converting a document to JSON, set the `toJSON.getters` option to `false` in your schema as shown below.

```
const userSchema = new Schema({
  email: { type: String, get: obfuscate }
}, { toJSON: { getters: false } });
```

You can also skip running getters on a one-off basis using the `Document#get()` function's `getters` option.

```
// Make sure you don't forget to pass `null` as the 2nd parameter.
doc.get('email', null, { getters: false }); // 'test@gmail.com'
```

#### Setters

Mongoose executes custom setters when you assign a value to a property. Suppose you want to ensure all user emails are lowercased, so users don't have to worry about case when logging in. The below `userSchema` adds a custom setter that lowercases emails.

```
const userSchema = Schema({ email: String });
userSchema.path('email').set(v => v.toLowerCase());
const User = mongoose.model('User', userSchema);

const user = new User({ email: 'TEST@gmail.com' });
user.email; // 'test@gmail.com'
```

Mongoose will call the custom setter regardless of which syntax you use to set the `email` property.

- `doc.email = 'TEST@gmail.com'`
- `doc.set('email', 'TEST@gmail.com')`
- `Object.assign(doc, { email: 'TEST@gmail.com' })`

Mongoose also runs setters on update operations:

- `updateOne()`

- `updateMany()`
- `findOneAndUpdate()`
- `findOneAndReplace()`

For example, Mongoose will run the `toLowerCase()` setter in the below example, because the below example calls `updateOne()`.

```
const { _id } = await User.create({ email: 'test@gmail.com' });

// Mongoose will run the `toLowerCase()` setter on `email`
await User.updateOne({ _id }, { email: 'NEW@gmail.com' });
const doc = await User.findOne({ _id });
doc.email; // 'new@gmail.com'
```

#### Advanced Getters/Setters

Getters and setters are good for more than just manipulating strings. You can also use getters and setters to transform values to and from different types. For example, MongoDB ObjectIds are objects that are commonly represented as strings that look like '5d124083fc741d44eca250fd'. However, you cannot compare ObjectIds using JavaScript's `===` operator, or even the `==` operator.

```
const str = '5d124083fc741d44eca250fd';
const schema = Schema({ objectid: mongoose.ObjectId });
const Model = mongoose.model('ObjectIdTest', schema);
const doc1 = new Model({ objectid: str });
const doc2 = new Model({ objectid: str });

// Mongoose casted the string `str` to an ObjectId
typeof doc1.objectid; // 'object'
doc1.objectid instanceof mongoose.Types.ObjectId; // true

doc1.objectid === doc2.objectid; // false
doc1.objectid == doc2.objectid; // false
```

However, you can use a custom getter that converts the `objectid` property into a string for comparison purposes. This approach also works for deep equality checks like `assert.deepEqual()`.

```
const str = '5d124083fc741d44eca250fd';

const s = Schema({ objectid: mongoose.ObjectId }, { _id: false });
// Add a custom getter that converts ObjectId values to strings
s.path('objectid').get(v => v.toString());

const Model = mongoose.model('ObjectIdTest', s);
const doc1 = new Model({ objectid: str });
const doc2 = new Model({ objectid: str });

// Mongoose now converts `objectid` to a string for you
typeof doc1.objectid; // 'string'
// The raw value stored in MongoDB is still an ObjectId
typeof doc1.get('objectid', null, { getters: false }); // 'object'

doc1.objectid === doc2.objectid; // true
doc1.objectid == doc2.objectid; // true
assert.deepEqual(doc1.toObject(), doc2.toObject()); // passes
```

Mongoose casting converts strings to ObjectIds for you, so in the above example you don't need to write a custom setter. However, if you use a getter to convert the raw value to a different type, you shoud also write a setter that converts back to the correct type.

For example, MongoDB introduced support for decimal floating points in MongoDB 3.4. Mongoose has a `Decimal128` type for storing values as decimal floating points. Below is an example of using `Decimal128` with Mongoose.

```
const accountSchema = Schema({ balance: mongoose.Decimal128 });
const Account = mongoose.model('Account', accountSchema);

await Account.create({ balance: 0.1 });
await Account.updateOne({}, { $inc: { balance: 0.2 } });

const account = await Account.findOne();
account.balance.toString(); // 0.3

// The MongoDB Node driver currently doesn't support
// addition and subtraction with Decimals
account.balance + 0.5;
```

Currently, you can't use add or subtract native JavaScript numbers with MongoDB decimals. Luckily, you can use getters and setters to make MongoDB decimals look like numbers as far as JavaScript is concerned.

```
const accountSchema = Schema({
  balance: {
    type: mongoose.Decimal128,
    // Convert the raw decimal value to a native JS number
    // when accessing the `balance` property
    get: v => parseFloat(v.toString()),
    // When setting the `balance` property, round to 2
    // decimal places and convert to a MongoDB decimal
    set: v => mongoose.Types.Decimal128.fromString(v.toFixed(2))
  }
});
const Account = mongoose.model('Account', accountSchema);

const account = new Account({ balance: 0.1 });
account.balance += 0.2;
account.balance; // 0.3
```

#### Computed Properties using Setters

Many codebases use setters to handle computed properties, properties that are computed from other properties in the document. For example, suppose you want to store a user's email domain (for example, "gmail.com") in addition to the user's email.

```
const userSchema = Schema({ email: String, domain: String });
// Whenever you set `email`, make sure you update `domain`.
userSchema.path('email').set(function(v) {
  // In a setter function, `this` may refer to a document or a query.
  // If `this` is a document, you can modify other properties.
  this.domain = v.slice(v.indexOf('@') + 1);
  return v;
});
const User = mongoose.model('User', userSchema);

const user = new User({ email: 'test@gmail.com' });
user.email; // 'test@gmail.com'
user.domain; // 'gmail.com'
```

Using setters to modify other properties within the document is viable, but you should avoid doing so unless there's no other option. We include it in this book because using setters for computed properties is a common pattern in practice and you may see it when working on an existing codebase. However, we recommend using virtuals or middleware instead.

What can go wrong with computed property setters? First, in `userSchema`, the `domain` property is a normal property that you can set. A malicious user or a bug could cause an incorrect `domain`.

```
// Since `domain` is a real property, you can `set()` it and
// overwrite the computed value.
let user = new User({ email: 'test@gmail.com', domain: 'oops' });
user.domain; // 'oops'

// Setters are order dependent. If `domain` is the first key,
// the setter will actually work.
user = new User({ domain: 'oops', email: 'test@gmail.com' });
user.domain; // 'gmail.com'
```

This means using setters for computed properties is unpredictable. Another common issue is that setters also run on `updateOne()` and `updateMany()` operations. If your setter modifies other properties in the document, you need to be careful to ensure it supports **both** queries and documents. Queries have a `set()` function to make this task easier.

```
const userSchema = Schema({ email: String, domain: String });
userSchema.path('email').set(function setter(v) {
  const domain = v.slice(v.indexOf('@') + 1);
  // Queries and documents both have a `set()` function
  this.set({ domain });
  return v;
});
const User = mongoose.model('User', userSchema);

let doc = await User.create({ email: 'test@gmail.com' });
doc.domain; // 'gmail.com'

// The setter will also run on `updateOne()`
const { _id } = doc;
const $set = { email: 'test@test.com' };
await User.updateOne({ _id }, { $set });
doc = await User.findOne({ _id });
doc.domain; // 'test.com'
```

## 2.5: Virtuals

You can handle computed properties with getters and setters, but virtuals are typically the better choice. In Mongoose, a *virtual* is a property that is **not** stored in MongoDB. Virtuals can have getters and setters just like normal properties. Virtuals have some neat properties that make them ideal for expressing the notion of a property that is a function of other properties.

Consider the previous example of a `domain` property that stores everything after the '@' in the user's email. Instead of a custom setter that updates the `domain` property every time the `email` property is modified, you can create a virtual property `domain`. The virtual property `domain` will have a getter that recomputes the user's email domain every time the property is accessed.

```
const userSchema = Schema({ email: String });
userSchema.virtual('domain').get(function() {
  return this.email.slice(this.email.indexOf('@') + 1);
});
const User = mongoose.model('User', userSchema);

let doc = await User.create({ email: 'test@gmail.com' });
doc.domain; // 'gmail.com'

// Mongoose ignores setting virtuals that don't have a setter
doc.set({ email: 'test@test.com', domain: 'foo' });
doc.domain; // 'test.com
```

Note that, even though the above code tries to `set()` the `domain` property to an incorrect value, `domain` is still correct. Setting a virtual property without a setter is a no-op. This virtual-based `userSchema` can safely accept untrusted data without risk of corrupting `domain`.

The downside of virtuals is that, since they are not stored in MongoDB, you **cannot** query documents by virtual properties. If you try to query by a virtual property like `domain`, you won't get any results. That's because MongoDB doesn't know about `domain`, `domain` is a property that Mongoose computes in Node.js.

```
await User.create({ email: 'test@gmail.com' });

// `doc` will be null, because the document in the database
// does **not** have a `domain` property
const doc = await User.findOne({ domain: 'gmail.com' });
```

### Virtuals and `JSON.stringify()`

By default, Mongoose does not include virtuals when you convert a document to JSON. For example, if you pass a document to the Express web framework's `res.json()` function, virtuals will **not** be included by default.

You need to set the `toJSON` schema option to `{ virtuals: true }` to opt in to including virtuals in your document's JSON representation.

```
// Opt in to Mongoose putting virtuals in `toJSON()` output
// for `JSON.stringify()`, and in `toObject()` output.
const opts = { toJSON: { virtuals: true } };
const userSchema = Schema({ email: String }, opts);
userSchema.virtual('domain').get(function() {
  return this.email.slice(this.email.indexOf('@') + 1);
});
const User = mongoose.model('User', userSchema);

const doc = await User.create({ email: 'test@gmail.com' });

// { _id: ..., email: 'test@gmail.com', domain: 'gmail.com' }
doc.toJSON();
// {"_id":...,"email":"test@gmail.com","domain":"gmail.com"}
JSON.stringify(doc);
```

Under the hood, the `toJSON` schema option configures the `Document#toJSON()` function. JavaScript's native `JSON.stringify()` function looks for `toJSON()` functions, and replaces the object being stringified with the result of `toJSON()`. In the below example, `JSON.stringify()` serializes the value of `prop` as the number `42`, even though `prop` is an object in JavaScript.

```
const myObject = {
  toJSON: function() {
    // Will print once when you call `JSON.stringify()`.
    console.log('Called!');
    return 42;
  }
};

// `{"prop":42}`. That is because `JSON.stringify()` uses the result
// of the `toJSON()` function.
JSON.stringify({ prop: myObject });
```

You can also configure the `toJSON` schema option globally:

```
mongoose.set('toJSON', { virtuals: true });
```

## 2.6: Queries

So far, this book has only used `Model.findOne()` and `Model.updateOne()` as promises using the `await` keyword.

```
const doc = await MyModel.findOne();
```

When you call `Model.findOne()`, Mongoose does **not** return a promise. It returns an instance of the Mongoose `Query` class, which can be used as a promise.

```
const query = Model.findOne();
query instanceof mongoose.Query; // true
// Each model has its own subclass of the `mongoose.Query` class
query instanceof Model.Query; // true

const doc = await query; // Executes the query
```

The reason why Mongoose has a special `Query` class rather than just returning a promise from `Model.findOne()` is for chaining. You can construct a query by chaining *query helper methods* like `where()` and `equals()`.

```
// Equivalent to `await Model.findOne({ name: 'Jean-Luc Picard' })`
const doc = await Model.findOne().
  where('name').equals('Jean-Luc Picard');
```

Each query has a `op` property, which contains the operation Mongoose will send to the MongoDB server. For example, when you call `Model.findOne()`, Mongoose sets `op` to `'findOne'`.

```
const query = Model.findOne();
query.op; // 'findOne'
```

Below are the possible values for `op`. There's a corresponding method on the `Query` class for each of the below values of `op`.

- `find`
- `findOne`
- `findOneAndDelete`
- `findOneAndUpdate`
- `findOneAndReplace`
- `deleteOne`
- `deleteMany`
- `replaceOne`
- `updateOne`
- `updateMany`

You can modify the `op` property before executing the query. Until you execute the query, you can change the `op` by calling any of the above methods. For example, some projects chain

`.find().updateOne()` as shown below, because separating out the query filter and the update operation may make your code more readable.

```
// Equivalent to `Model.updateOne({ name: 'Jean-Luc Picard' },
// { $set: { age: 59 } })`
const query = Model.
  find({ name: 'Jean-Luc Picard' }).
  updateOne({}, { $set: { age: 59 } });
```

Mongoose does **not** automatically execute a query, you need to explicitly execute the query. There's two methods that let you execute a query: `exec()` and `then()`. When you `await` on a query, the JavaScript runtime calls `then()` under the hood.

```
// The `exec()` function executes the query and returns a promise.
const promise1 = Model.findOne().exec();

// The `then()` function calls `exec()` and returns a promise
const promise2 = Model.findOne().then(doc => doc);
// Equivalently, `await` calls `then()` under the hood
const doc = await Model.findOne();
```

## 2.7: The Mongoose Global and Multiple Connections

When you `require('mongoose')`, you get an instance of the Mongoose class.

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

// Mongoose exports an instance of the `Mongoose` class
mongoose instanceof mongoose.Mongoose; // true
```

When someone refers to the "Mongoose global" or "Mongoose singleton", they're referring to the object you get when you `require('mongoose')`. However, you can also instantiate an entirely separate Mongoose instance with its own options, models, and connections.

```
const { Mongoose } = require('mongoose');

const mongoose1 = new Mongoose();
const mongoose2 = new Mongoose();

mongoose1.set('toJSON', { virtuals: true });
mongoose1.get('toJSON'); // { virtuals: true }
mongoose2.get('toJSON'); // null
```

In particular, the Mongoose global has a default `connection` that is an instance of the `mongoose.Connection` class. When you call `mongoose.connect()`, that is equivalent to calling `mongoose.connection.openUri()`.

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

const mongoose1 = new mongoose.Mongoose();
const mongoose2 = new mongoose.Mongoose();

mongoose1.connection instanceof mongoose.Connection; // true
mongoose2.connection instanceof mongoose.Connection; // true

mongoose1.connections.length; // 1
mongoose1.connections[0] === mongoose1.connection; // true

mongoose1.connection.readyState; // 0, 'disconnected'
mongoose1.connect('mongodb://localhost:27017/test',
  { useNewUrlParser: true });
mongoose1.connection.readyState; // 2, 'connecting'
```

When you call `mongoose.createConnection()`, you create a new `connection` object that Mongoose tracks in the `mongoose.connections` property.

```
const mongoose = require('mongoose');
const mongoose1 = new mongoose.Mongoose();

const conn = mongoose1.createConnection('mongodb://localhost:27017/test',
  { useNewUrlParser: true });

mongoose1.connections.length; // 2
mongoose1.connections[1] === conn; // true
```

Why do you need multiple connections? For most apps, you only need one connection. Here's a couple examples when you would need to create multiple connections:

- Your app needs to acess data stored in multiple databases. A Mongoose connection is scoped to exactly one database - if you need to create models on different databases, you need a separate connection.

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

const conn1 = mongoose.createConnection('mongodb://localhost:27017/db1',
  { useNewUrlParser: true });
const conn2 = mongoose.createConnection('mongodb://localhost:27017/db2',
  { useNewUrlParser: true });

// Will store data in the 'db1' database's 'tests' collection
const Model1 = conn1.model('Test', mongoose.Schema({ name: String }));
// Will store data in the 'db2' database's 'tests' collection
const Model2 = conn2.model('Test', mongoose.Schema({ name: String }));
```

- Your app has certain operations that are slow, and you don't want the slow operations to cause performance issues on fast queries. See Section 6.6 for more information on how multiple connections can help speed up your app.

## 2.8: Summary

There are 5 core concepts in Mongoose: models, documents, schemas, connections, and queries. Most Mongoose apps will use all 5 of these concepts.

- A *Model* is a class representing an object stored in MongoDB. To create a model, you need a schema and a connection.
- A *Schema* is class representing a configuration object for a model.
- A *Connection* is a pool of sockets that are connected to one or more MongoDB servers. A connection provides an interface for models to communicate with MongoDB.
- A *Document* is an instance of a model.
- A *Query* is an operation that will be sent to the MongoDB server.

When you create a schema, there are two implicit classes: SchemaTypes and VirtualTypes. You may interact with these directly, but you can build a simple CRUD app without dealing with them.

Finally, when you `require('mongoose')`, you get back an instance of the `Mongoose` class. An instance of the `Mongoose` class tracks a list of connections, and stores some global options. Advanced users may want to create multiple Mongoose instances, but most apps will only use the singleton `Mongoose` instance you get when you `require('mongoose')`.

# 3: Queries

Queries and aggregations are how you fetch data from MongoDB into Mongoose. You can also use queries to update data in MongoDB without needing to fetch the documents.

Mongoose models have several static functions, like `find()` and `updateOne()`, that return Mongoose `Query` objects.

```
const schema = Schema({ name: String, age: Number });
const Model = mongoose.model('Model', schema);

const query = Model.find({ age: { $lte: 30 } });
query instanceof mongoose.Query; // true
```

Mongoose queries provide a chainable API for creating and executing CRUD (create, ead, update, delete) operations. Mongoose queries let you build up a CRUD operation in Node.js, and then send the operation to the MongoDB server.

## 3.1: Query Execution

Mongoose queries are just objects in Node.js memory until you actually execute them. There are 3 ways to execute a query:

1. `await` on the query
2. Call `Query#then()`
3. Call `Query#exec()`

Under the hood, all 3 ways are equivalent. The `await` keyword calls `Query#then()`, and Mongoose's `Query#then()` calls `Query#exec()`, so (1) and (2) are just syntactic sugar for (3).

Here are the 3 ways to execute a query in code form:

```
const query = Model.findOne();

// Execute the query 3 times, in 3 different ways
await query; // 1

query.then(res => {}); // 2

await query.exec(); // 3
```

Note that a query can be executed multiple times. The above example executes 3 separate `findOne()` queries.

The actual operation that Mongoose will execute is stored as a string in the `Query#op` property. Below is a list of all valid query ops, grouped by CRUD verb.

**Read:**

- `count`: return the number of documents that match `filter` (deprecated)
- `countDocuments`: return the number of documents that match `filter`
- `distinct`: return a list of the distinct values of a given field
- `estimatedDocumentCount`: return the number of documents in the collection
- `find`: return a list of documents that match `filter`
- `findOne`: return the first document that matches `filter`, or `null`

**Update:**

- `findOneAndReplace`: same as `replaceOne()` + returns the replaced document
- `findOneAndUpdate`: same as `updateOne()` + returns the replaced document
- `replaceOne`: replace the first document that matches `filter` with `replacement`
- `update`: update the first document that matches `filter` (deprecated)
- `updateMany`: update all documents that match `filter`
- `updateOne`: update the first document that matches `filter`

**Delete:**

- `deleteOne`: delete the first document that matches `filter`
- `deleteMany`: delete all documents that match `filter`
- `findOneAndDelete`: same as `deleteOne()` + returns the deleted document
- `findOneAndRemove`: same as `remove()` + returns the deleted document (deprecated)
- `remove`: delete all documents that match `filter` (deprecated)

You can modify the query `op` before executing. Many developers use `.find().updateOne()` in order to separate out the query filter from the update object for readability.

```
// Chaining makes it easier to visually break up complex updates
await Model.
  find({ name: 'Will Riker' }).
  updateOne({}, { rank: 'Commander' });

// Equivalent, without using chaining syntax
await Model.updateOne({ name: 'Will Riker' }, { rank: 'Commander' });
```

Note that `.find().updateOne()` is **not** the same thing as `.findOneAndUpdate()`. `findOneAndUpdate()` is a distinct operation on the MongoDB server. Chaining `.find().updateOne()` ends up as an `updateOne()`.

## 3.2: Query Operations

There are 17 distinct query operations in Mongoose. 4 are deprecated: `count()`, `update()`, `findOneAndRemove()`, and `remove()`. The remaining 13 can be broken up into 5 classes: reads, updates, deletes, find and modify ops, and other.

### Read

`find()`, `findOne()`, and `countDocuments()` all take in a `filter` parameter and find document(s) that match the `filter`.

- `find()` returns a list of all documents that match `filter`. If none are found, `find()` returns an empty array.
- `findOne()` returns the first document that matches `filter`. If none are found, `findOne()` returns `null`.
- `countDocuments()` returns the number of documents that match `filter`. If none are found, `countDocuments()` returns `0`.

Below is an example of using `find()`, `findOne()`, and `countDocuments()` with the same `filter`.

```
await Model.insertMany([
  { name: 'Jean-Luc Picard', age: 59 },
  { name: 'Will Riker', age: 29 },
  { name: 'Deanna Troi', age: 29 }
]);

const filter = { age: { $lt: 30 } };

let res = await Model.find(filter);
res[0].name; // 'Will Riker'
res[1].name; // 'Deanna Troi'

res = await Model.findOne(filter);
res.name; // 'Will Riker'

res = await Model.countDocuments(filter);
res; // 2
```

### Delete

`deleteOne()`, `deleteMany()` , and `remove()` all take a `filter` parameter and delete document(s) from the collection that match the given filter.

- `deleteOne()` deletes the first document that matches `filter`
- `deleteMany()` deletes all documents that match `filter`
- `remove()` deletes all documents that match `filter`. `remove()` is deprecated, you should use `deleteOne()` or `deleteMany()` instead.

Below is an example of using `deleteOne()`, `deleteMany()`, and `remove()` with the same `filter`.

```
await Model.insertMany([
  { name: 'Jean-Luc Picard', age: 59 },
  { name: 'Will Riker', age: 29 },
  { name: 'Deanna Troi', age: 29 }
]);

const filter = { age: { $lt: 30 } };

let res = await Model.deleteOne(filter);
res.deletedCount; // 1

await Model.create({ name: 'Will Riker', age: 29 });

res = await Model.deleteMany(filter);
res.deletedCount; // 2

await Model.insertMany([
  { name: 'Will Riker', age: 29 },
  { name: 'Deanna Troi', age: 29 }
]);

// `remove()` deletes all docs that match `filter` by default
res = await Model.remove(filter);
res.deletedCount; // 2
```

The return value of `await Model.deleteOne()` is **not** the deleted document or documents. The return value is a result object that has a `deletedCount` property, which tells you the number of documents MongoDB deleted. To get the deleted document, you'll need to use `findOneAndDelete()`.

### Update

`updateOne()`, `updateMany()`, `replaceOne()`, and `update()` all take in a `filter` parameter and an `update` parameter, and modify documents that match `filter` based on `update`.

- `updateOne()` updates the first document that matches `filter`

- `updateMany()` updates all documents that match `filter`
- `replaceOne()` replaces the first document that matches `filter` with the `update` parameter
- `update()` updates the first document that matches `filter`. `update()` is deprecated, use `updateOne()` instead.

Below is an example of using `updateOne()`, `updateMany()`, and `update()` with the same `filter` and `update` parameters:

```
const schema = Schema({ name: String, age: Number, rank: String });
const Model = mongoose.model('Model', schema);
await Model.insertMany([
  { name: 'Jean-Luc Picard', age: 59 },
  { name: 'Will Riker', age: 29 },
  { name: 'Deanna Troi', age: 29 }
]);

const filter = { age: { $lt: 30 } };
const update = { rank: 'Commander' };

// `updateOne()`
let res = await Model.updateOne(filter, update);
res.nModified; // 1

let docs = await Model.find(filter);
docs[0].rank; // 'Commander'
docs[1].rank; // undefined

// `updateMany()`
res = await Model.updateMany(filter, update);
res.nModified; // 2

docs = await Model.find(filter);
docs[0].rank; // 'Commander'
docs[1].rank; // 'Commander'

// `update()` behaves like `updateOne()` by default
res = await Model.update(filter, update);
res.nModified; // 1
```

The `replaceOne()` function is slightly different from `updateOne()` because it *replaces* the matched document, meaning that it deletes all keys that aren't in the update other than `_id`. In the previous example, `updateOne()` added a `'rank'` key to the document, but didn't change the other

keys in the document. On the other hand, `replaceOne()` deletes any keys that aren't in the update.

```
const filter = { age: { $lt: 30 } };
const replacement = { name: 'Will Riker', rank: 'Commander' };

// Sets `rank`, unsets `age`
let res = await Model.replaceOne(filter, replacement);
res.nModified; // 1

let docs = await Model.find({ name: 'Will Riker' });
!!docs[0]._id; // true
docs[0].name; // 'Will Riker'
docs[0].rank; // 'Commander'
docs[0].age; // undefined
```

Like deletes, updates do **not** return the updated document or documents. They return a result object with an `nModified` property that contains the number of documents the MongoDB server updated. To get the updated document, you should use `findOneAndUpdate()`.

### Find and Modify

`findOneAndUpdate()`, `findOneAndDelete()`, `findOneAndReplace()`, and `findOneAndRemove()` are operations that update or remove a single document. Unlike `updateOne()`, `deleteOne()`, and `replaceOne()`, these functions return the updated document rather than simply the number of updated documents.

- `findOneAndUpdate()` works like `updateOne()`, but also returns the document
- `findOneAndDelete()` works like `deleteOne()`, but also returns the document
- `findOneAndReplace()` works like `replaceOne()`, but also returns the document
- `findOneAndRemove()` works like `deleteOne()`. It is deprecated in favor of `findOneAndDelete()`.

Remember that `findOneAndUpdate()` is different than calling `findOne()` followed by `updateOne()`. `findOneAndUpdate()` is a single operation, and it is *atomic*.

For example, if you call `findOne()` followed by an `updateOne()`, another update may come in and change the document between when you called `findOne()` and when you called `updateOne()`. With `findOneAndUpdate()`, that cannot happen.

By default, `findOneAndUpdate()`, `findOneAndDelete()`, and `findOneAndReplace()` return the document as it was **before** the MongoDB server applied the update.

```
const filter = { name: 'Will Riker' };
const update = { rank: 'Commander' };

// MongoDB will return the matched doc as it was **before**
// applying `update`
let doc = await Model.findOneAndUpdate(filter, update);
doc.name; // 'Will Riker'
doc.rank; // undefined

const replacement = { name: 'Will Riker', rank: 'Commander' };
doc = await Model.findOneAndReplace(filter, replacement);
// `doc` still has an `age` key, because `findOneAndReplace()`
// returns the document as it was before it was replaced.
doc.rank; // 'Commander'
doc.age; // 29

// Delete the doc and return the doc as it was before the
// MongoDB server deleted it.
doc = await Model.findOneAndDelete(filter);
doc.name; // 'Will Riker'
doc.rank; // 'Commander'
doc.age; // undefined
```

To return the document as it is **after** the MongoDB server applied the update, use the `new` option.

```
const filter = { name: 'Will Riker' };
const update = { rank: 'Commander' };
const options = { new: true };

// MongoDB will return the matched doc **after** the update
// was applied if you set the `new` option
let doc = await Model.findOneAndUpdate(filter, update, options);
doc.name; // 'Will Riker'
doc.rank; // 'Commander'

const replacement = { name: 'Will Riker', rank: 'Commander' };
doc = await Model.findOneAndReplace(filter, replacement, options);
doc.rank; // 'Commander'
doc.age; // void 0
```

#### Other

The `distinct()` and `estimatedDocumentCount()` operations don't quite fit in any of the other classes. Their semantics are sufficiently different that they belong in their own class.

- `distinct(key, filter)` returns an array of distinct values of `key` among documents that match `filter`.

```
// Return an array containing the distinct values of the `age`
// property. The values in the array can be in any order.
let values = await Model.distinct('age');
values.sort(); // [29, 59]

const filter = { age: { $lte: 29 } };
values = await Model.distinct('name', filter);
values.sort(); // ['Deanna Troi', 'Will Riker']
```

- `estimatedDocumentCount()` returns the number of documents in the collection based on MongoDB's collection metadata. It doesn't take any parameters.

```
// Unlike `countDocuments()`, `estimatedDocumentCount()` does **not**
// take a `filter` parameter. It only returns the number of documents
// in the collection.
const count = await Model.estimatedDocumentCount();
count; // 3
```

In most cases, calling `estimatedDocumentCount()` will give the same result as calling `countDocuments()` with no parameters. `estimatedDocumentCount()` is generally faster than `countDocuments()` because it doesn't actually go through and count the documents in the collection. However, `estimatedDocumentCount()` may be incorrect for one of two reasons:

1. Orphaned documents, which only happen on sharded clusters.
2. There was an unclean shutdown of the database

`estimatedDocumentCount()` is rarely useful when building apps because it doesn't accept a `filter` parameter. You can use `estimatedDocumentCount()` if you want to count the total number of documents in a massive collection, but `countDocuments()` is usually fast enough.

### 3.3: Query Operators

MongoDB supports a wide variety of operators to build up sophisticated `filter` parameters for operations like `find()` and `updateOne()`.

In the previous section, several examples used the `$lte` query operator (also known as a *query selector*) to filter for Star Trek characters whose `age` was less than `30`:

```
// `$lte` is an example of a query selector
const filter = { age: { $lte: 30 } };
const docs = await Model.find(filter);
```

If an object has multiple query selectors, MongoDB treats it as an "and". For example, the below query selector will match documents whose `age` is greater than 50 **and** less than 60.

```
const querySelectors = { $gt: 50, $lt: 60 };
const filter = { age: querySelectors };
```

The most commonly used query selectors fall into one of 4 classes:

### Comparison

- `$eq`: Matches values that strictly equal the specified value
- `$gt`: Matches values that are greater than the specified value
- `$gte`: Matches values that are greather than or equal to the specified value
- `$in`: Matches values that are strictly equal to a value in the specified array
- `$lt`: Matches values that are less than a given value
- `$lte`: Matches values that are less than or equal to the specified value
- `$ne`: Matches values that are not strictly equal to the specified value
- `$nin`: Matches values that are not strictly equal to any of the specified values
- `$regex`: Matches values that are strings which match the specified regular expression

The right hand side of a comparison query selector is the value or values to compare against.

```
// Matches if the `age` property is exactly equal to 42
let filter = { age: { $eq: 42 } };

// Matches if the `age` property is a number between 30 and 40
filter = { age: { $gte: 30, $lt: 40 } };

// Matches if `name` is 'Jean-Luc Picard' or 'Will Riker'
filter = { name: { $in: ['Jean-Luc Picard', 'Will Riker'] } };

// Matches if `name` is a string containing 'picard', ignoring case
filter = { name: { $regex: /picard/i } };
```

### Element/Type

There are two 'element' query selectors. These query selectors help you filter documents based on the type of the value, like whether a value is a string or a number.

- `$exists`: Matches either documents that have a certain property, or don't have the specified property, based on whether the given value is `true` or `false`
- `$type`: Matches documents where the property has the specified type.

Here's how `$exists` works:

```
const schema = Schema({ name: String, age: Number, rank: String });
const Model = mongoose.model('Model', schema);
await Model.create({ name: 'Will Riker', age: 29 });

// Finds the doc, because there's an `age` property
let doc = await Model.findOne({ age: { $exists: true } });

// Does **not** find the doc, no `rank` property
doc = await Model.findOne({ rank: { $exists: true } });

// Finds the doc, because there's no `rank` property
doc = await Model.findOne({ rank: { $exists: false } });

// Finds the doc, because `$exists: true` matches `null` values.
// `$exists` is analagous to a JavaScript `in` or `hasOwnProperty()`
await Model.updateOne({}, { rank: null });
doc = await Model.findOne({ rank: { $exists: true } });
```

`$exists` is useful, but it also matches `null` values. The `$type` query selector is helpful for cases where you want to match documents that are the wrong type.

```
const schema = Schema({ name: String, age: Number, rank: String });
const Model = mongoose.model('Model', schema);
await Model.create({ name: 'Will Riker', age: 29 });

// Finds the doc, because `age` is a number
let doc = await Model.findOne({ age: { $type: 'number' } });

// Does **not** find the doc, because the doc doesn't
// have a `rank` property.
doc = await Model.findOne({ rank: { $type: 'string' } });
```

Below is a list of valid types for the `$type` query selector:

- `'double'`
- `'string'`
- `'object'`
- `'array'`
- `'binData'`
- `'objectId'`
- `'bool'`
- `'null'`
- `'regex'`
- `'int'`
- `'timestamp'`
- `'long'`
- `'decimal'`
- `'number'`: Alias that matches `'double'`, `'int'`, `'long'`, and `'decimal'`

There is a small gotcha when using `$type` with Mongoose numbers: to save space, the underlying MongoDB driver stores JavaScript numbers as 'int' where possible, and 'double' otherwise. So if `age` is an integer, Mongoose will store it as an 'int', otherwise Mongoose will store it as a 'double'.

```
const schema = Schema({ name: String, age: Number, rank: String });
const Model = mongoose.model('Model', schema);
await Model.create({ name: 'Will Riker', age: 29 });

// Finds the doc. Mongoose stores `age` as an int
let doc = await Model.findOne({ age: { $type: 'int' } });

doc.age = 29.5;
await doc.save();

// Does **not** find the doc: `age` is no longer an int
doc = await Model.findOne({ age: { $type: 'int' } });

// Finds the doc, `age` is now a double
doc = await Model.findOne({ age: { $type: 'double' } });

// Finds the doc, 'number' matches both ints and doubles.
doc = await Model.findOne({ age: { $type: 'number' } });
```

You can also invert the `$type` query selector using the `$not` query selector. `$not: { $type: 'string' }` will match any value that is not a string, including documents that do not have the property.

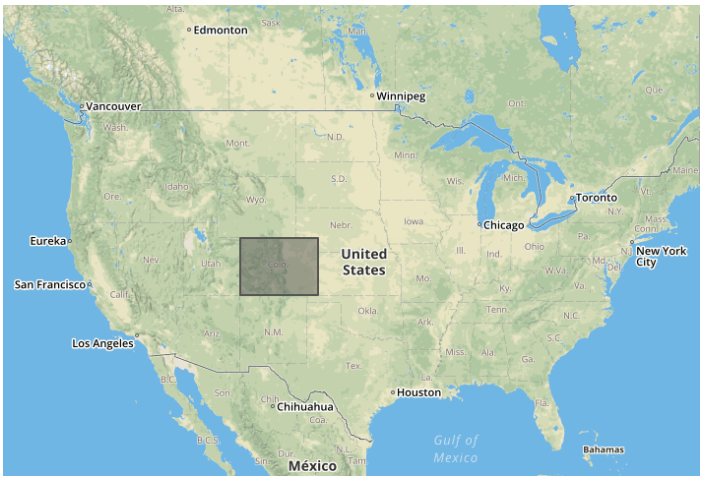
```
const schema = Schema({ name: String, age: Number, rank: String });
const Model = mongoose.model('Model', schema);
await Model.create({ name: 'Will Riker', age: 29 });

// Finds the doc, because Mongoose stores `age` as an int
const querySelector = { $not: { $type: 'string' } };
let doc = await Model.findOne({ age: querySelector });

// Finds the doc, because `doc` doesn't have a `rank` property
doc = await Model.findOne({ rank: querySelector });
```

#### Geospatial

Geospatial operators are like comparison operators for geoJSON data. For example, suppose you have a `State` model that contains all the US states, with the state's boundaries stored as geoJSON in the `location` property.

```
const State = mongoose.model('State', Schema({
  name: String,
  location: {
    type: { type: String },
    coordinates: [[[Number]]]
  }
}));
// The US state of Colorado is roughly a geospherical rectangle
await State.create({
  name: 'Colorado',
  location: {
    type: 'Polygon',
    coordinates: [[
      [-109, 41],
      [-102, 41],
      [-102, 37],
      [-109, 37],
      [-109, 41]
    ]]
  }
});
```

The US state of Colorado is a convenient example because its boundaries are almost a geospatial rectangle, excluding the historical idiosyncracies of Colorado's borders. Here's what the above polygon looks like on a map:




MongoDB has 4 geospatial query selectors that help you query geoJSON data:

- `$geoIntersects`
- `$geoWithin`
- `$near`
- `$nearSphere`

The `$geoIntersects` query selector matches if the document's value intersects with the given value. For example, using `$geoIntersects`, you can take a geoJSON point and see what `State` it is in. A point intersects with a polygon if and only if the point is within the polygon.

```
// Approximate coordinates of the capital of Colorado
const denver = { type: 'Point', coordinates: [-104.9903, 39.7392] };
// Approximate coordinates of San Francisco
const sf = { type: 'Point', coordinates: [-122.5, 37.7] };

let doc = await State.findOne({
  location: {
    // You need to specify `$geometry` if you're using
    // `$geoIntersects` with geoJSON.
    $geoIntersects: { $geometry: denver }
  }
});
doc.name; // 'Colorado'

doc = await State.findOne({
  location: {
    $geoIntersects: { $geometry: sf }
  }
});
doc; // null
```

The `$geoWithin` query selector is slightly different than `$geoIntersects`: it matches if the document's value is entirely contained within the given value. For example, `State.findOne()` will not return a value with `$geoWithin`, because a polygon can't be entirely contained within a point.

However, if you have a `City` model where each city has a `location` that is a geoJSON point, `$geoWithin` lets you find cities that are within the state of Colorado.

If the document value is a geoJSON point, `$geoWithin` and `$geoIntersects` are equivalent. There's no way for a point to partially overlap with another geoJSON feature.

```
const City = mongoose.model('City', Schema({
  name: String,
  location: {
    type: { type: String },
    coordinates: [Number]
  }
}));
let location = { type: 'Point', coordinates: [-104.99, 39.739] };
await City.create({ name: 'Denver', location });

const colorado = await State.findOne();
const $geoWithin = { $geometry: colorado.location };
let doc = await City.findOne({ location: { $geoWithin } });
doc.name; // 'Denver'

// `$geoIntersects` also finds that Denver is in Colorado
const $geoIntersects = { $geometry: colorado.location };
doc = await City.findOne({ location: { $geoIntersects } });
```

Another neat feature of `$geoWithin` is that you can use it to find points that are within a certain number of miles of a given point. For example, Denver is about 1000 miles away from San Francisco. The `$centerSphere` query operator lets you find documents that are within 1000 miles of a given point:

```
const sfCoordinates = [-122.5, 37.7];
// $centerSphere distance is in radians, so convert miles to radians
const distance = 1000 / 3963.2;
const $geoWithin = { $centerSphere: [sfCoordinates, distance] };
let doc = await City.findOne({ location: { $geoWithin } });

// `doc` will be `null`
$geoWithin.$centerSphere[1] = distance / 2;
doc = await City.findOne({ location: { $geoWithin } });
```

The `$geoWithin` operator helps you find points that are within a certain distance of a point, but those points may be in any order. The `$near` and `$nearSphere` operators help you find points that are within a certain distance of a given point and sort them by distance. The `$near` and `$nearSphere` operators require a special '2dsphere' index.

The difference between `$near` and `$nearSphere` is that `$nearSphere` calculates spherical distance (using the Haversine formula), but `$near` calculates Cartesian distance (assuming the Earth is flat). In practice, you should use `$nearSphere` for geospatial data.

```
const createCity = (name, coordinates) => ({
  name,
  location: { type: 'Point', coordinates }
});
await City.create([
  createCity('Denver', [-104.9903, 39.7392]),
  createCity('San Francisco', [-122.5, 37.7]),
  createCity('Miami', [-80.13, 25.76])
]);
// Create a 2dsphere index, otherwise `$nearSphere` will error out
await City.collection.createIndex({ location: '2dsphere' });

// Find cities within 2000 miles of New York, sorted by distance
const $geometry = { type: 'Point', coordinates: [-74.26, 40.7] };
// `$nearSphere` distance is in meters, so convert miles to meters
const $maxDistance = 2000 * 1609.34;
const cities = await City.find({
  location: { $nearSphere: { $geometry, $maxDistance } }
});
cities[0].name; // 'Miami'
cities[1].name; // 'Denver'
```

### Array

Array query selectors help you filter documents based on array properties.

- `$all`: matches arrays that contain all of the values in the given array
- `$size`: matches arrays whose length is equal to the given number
- `$elemMatch`: matches arrays which contain a document that matches the given filter

The need for array query selectors is limited because MongoDB is smart enough to drill into arrays.

For example, suppose you have a `BlogPost` model with an array of embedded `comments`. Each comment has a `user` property. You can find all blog posts that a given user commented on simply

by querying by `comments.user`.

```
let s = Schema({ comments: [{ user: String, text: String }] });
const BlogPost = mongoose.model('BlogPost', s);
await BlogPost.create([
  { comments: [{ user: 'jpicard', text: 'Make it so!' }] },
  { comments: [{ user: 'wriker', text: 'One, or both?' }] },
]);

const docs = await BlogPost.find({ 'comments.user': 'jpicard' });
docs.length; // 1
```

However, what if you want to find all blog posts that both `'wriker'` and `'jpicard'` commented on? That's where the `$all` query selector comes in. The `$all` query selector matches arrays that contain all elements in the given array.

```
await BlogPost.create([
  { comments: [{ user: 'jpicard', text: 'Make it so!' }] },
  { comments: [{ user: 'wriker', text: 'One, or both?' }] },
  { comments: [{ user: 'wriker' }, { user: 'jpicard' }] }
]);

// Find all blog posts that both 'wriker' and 'jpicard' commented on.
const $all = ['wriker', 'jpicard'];
let docs = await BlogPost.find({ 'comments.user': { $all } });
docs.length; // 1

// Find all blog posts that 'jpicard' commented on.
docs = await BlogPost.find({ 'comments.user': 'jpicard' });
docs.length; // 2
```

The `$size` operator lets you filter documents based on array length. For example, here's how you find all blog posts that have 2 comments:

```
// Find all blog posts that have exactly 2 comments
const comments = { $size: 2 };
let docs = await BlogPost.find({ comments });
docs.length; // 1
```

The `$elemMatch` query selector more subtle. For example, suppose you wanted to find all blog posts where the user `'jpicard'` commented `'Make it so!'`. Naively, you might try a query on `{ 'comments.user', 'comments.comment' }` as shown below.

```
await BlogPost.create([
  { comments: [{ user: 'jpicard', text: 'Make it so!' }] },
  { comments: [{ user: 'wriker', text: 'One, or both?' }] },
  {
    comments: [
      { user: 'wriker', text: 'Make it so!' },
      { user: 'jpicard', text: 'That\'s my line!' }
    ]
  }
]);

// Finds 2 documents, because this query finds blog posts where
// 'jpicard' commented, and where someone commented 'Make it so!'.
let docs = await BlogPost.find({
  'comments.user': 'jpicard',
  'comments.text': 'Make it so!'
});
docs.length; // 2
```

Unfortunately, that query doesn't work. The naive approach finds documents where `'comments.user'` is equal to `'jpicard'` and `'comments.comment'` is equal to `'Make it so!'`, but doesn't make sure that the same subdocument has both the correct `user` and the correct `comment`. That's what `$elemMatch` is for:

```
// `$elemMatch` is like a nested filter for array elements.
const $elemMatch = { user: 'jpicard', comment: 'Make it so!' };
let docs = await BlogPost.find({ comments: { $elemMatch } });
docs.length; // 1
```

### 3.4: Update Operators

MongoDB also provides several update operators to help you build up sophisticated `update` parameters to `updateOne()`, `updateMany()`, and `findOneAndUpdate()`. Each update has at least one update operator.

In MongoDB, update operators start with '$'. So far, this book hasn't explicitly used any update operators. That's because, when you provide an `update` parameter that doesn't have any update operators, Mongoose wraps your update in the `$set` update operator.

```
const schema = Schema({ name: String, age: Number, rank: String });
const Character = mongoose.model('Character', schema);
await Character.create({ name: 'Will Riker', age: 29 });

const filter = { name: 'Will Riker' };
let update = { rank: 'Commander' };
const opts = { new: true };
let doc = await Character.findOneAndUpdate(filter, update, opts);
doc.rank; // 'Commander'

// By default, Mongoose wraps your update in `$set`, so the
// below update is equivalent to the previous update.
update = { $set: { rank: 'Captain' } };
doc = await Character.findOneAndUpdate(filter, update, opts);
doc.rank; // 'Captain'
```

The `$set` operator sets the value of the given field to the given value. If you provide multiple fields, `$set` sets them all. The `$set` operator also supports dotted paths within nested objects.

```
const Character = mongoose.model('Character', Schema({
  name: { first: String, last: String },
  age: Number,
  rank: String
}));
const name = { first: 'Will', last: 'Riker' };
await Character.create({ name, age: 29, rank: 'Commander' });

// Update `name.first` without touching `name.last`
const $set = { 'name.first': 'Thomas', rank: 'Lieutenant' };
let doc = await Character.findOneAndUpdate({}, { $set }, { new: true });
doc.name.first; // 'Thomas'
doc.name.last; // 'Riker'
doc.rank; // 'Lieutenant'
```

The `$set` operator is an example of a *field update operator*.

### Field Update Operators

Field update operators operate on fields of any type, as opposed to array update operators or numeric update operators. Below is a list of the most common field update operators:

- `$set`: set the value of the given fields to the given values
- `$unset`: delete the given fields

- `$setOnInsert`: set the value of the given fields to the given values if a new document was inserted in an upsert. Ignored if MongoDB updated an existing document.
- `$min`: set the value of the given fields to the specified values, if the specified value is less than the current value of the field.
- `$max`: set the value of the given fields to the specified values, if the specified value is greater than the current value of the field.

The `$unset` operator deletes all properties in the given object. For example, you an use it to unset a character's age:

```
const schema = Schema({ name: String, age: Number, rank: String });
const Character = mongoose.model('Character', schema);
await Character.create({ name: 'Will Riker', age: 29 });

const filter = { name: 'Will Riker' };
// Delete the `age` property
const update = { $unset: { age: 1 } };
const opts = { new: true };
let doc = await Character.findOneAndUpdate(filter, update, opts);
doc.age; // undefined
```

The `$setOnInsert` operator behaves like a conditional `$set`. It only sets the values if a new document was inserted because of the `upsert` option. If `upsert` is `false`, or the upsert modified an existing document rather than inserting a new one, `$setOnInsert` does nothing.

```
await Character.create({ name: 'Will Riker', age: 29 });

let filter = { name: 'Will Riker' };
// Set `rank` if inserting a new document
const update = { $setOnInsert: { rank: 'Captain' } };
// If `upsert` option isn't set, `$setOnInsert` does nothing
const opts = { new: true, upsert: true };
let doc = await Character.findOneAndUpdate(filter, update, opts);
doc.rank; // undefined

filter = { name: 'Jean-Luc Picard' };
doc = await Character.findOne(filter);
doc; // null, so upsert will insert a new document

doc = await Character.findOneAndUpdate(filter, update, opts);
doc.name; // 'Jean-Luc Picard'
doc.rank; // 'Captain'
```

`$min` and `$max` set values based on the current value of the field. They're so named because if you use `$min`, the updated value will be the minimum of the given value and the current value. Similarly, if you use `$max`, the updated value will be the maximum of the given value and the current value.

```
await Character.create({ name: 'Will Riker', age: 29 });

const filter = { name: 'Will Riker' };
const update = { $min: { age: 30 } };
const opts = { new: true, upsert: true };

let doc = await Character.findOneAndUpdate(filter, update, opts);
doc.age; // 29

update.$min.age = 28;
doc = await Character.findOneAndUpdate(filter, update, opts);
doc.age; // 28
```

#### Numeric Update Operators

Numeric update operators can only be used with numeric values. Both the value in the database and the given value must be numbers.

- `$inc`: increments the given fields by the given values. The given values may be positive (for incrementing) or negative (for decrementing)
- `$mul`: multiplies the given fields by the given values.

```
// Increment `age` by 1 using `$inc`
const filter = { name: 'Will Riker' };
let update = { $inc: { age: 1 } };
const opts = { new: true };
let doc = await Character.findOneAndUpdate(filter, update, opts);
doc.age; // 30

// Decrement `age` by 1
update.$inc.age = -1;
doc = await Character.findOneAndUpdate(filter, update, opts);
doc.age; // 29

// Multiply `age` by 2
update = { $mul: { age: 2 } };
doc = await Character.findOneAndUpdate(filter, update, opts);
doc.age; // 58
```

### Array Update Operators

Array update operators can only be used when the value in the database is an array. They let you add or remove elements from arrays.

- `$push`: add the given value to the end of the array
- `$addToSet`: add the given value to the end of the array, unless there's already an element in the array that is exactly equal to the given value
- `$pull`: remove all elements from the array that match the specified condition
- `$pullAll`: remove all elements that are exactly equal to one of the values in the given array
- `$pop`: remove the first element of the array if the given value is `-1`, or the last element if the given value is `1`

The `$push` operator lets you add an element to the end of the array.

```
const schema = Schema({ title: String, tags: [String] });
const Post = mongoose.model('BlogPost', schema);
const title = 'Introduction to Mongoose';
await Post.create({ title, tags: ['Node.js'] });

// Add 'MongoDB' to the blog post's list of tags
const update = { $push: { tags: 'MongoDB' } };
const opts = { new: true };

let doc = await Post.findOneAndUpdate({ title }, update, opts);
doc.tags; // ['Node.js', 'MongoDB']
```

The `$addToSet` operator is similar to `$push`, except it skips adding duplicates. For example, if `BlogPost.tags` contains `'MongoDB'`, `$addToSet` will skip adding 'MongoDB'.

```
const title = 'Introduction to Mongoose';
await Post.create({ title, tags: ['Node.js'] });

// 'MongoDB' isn't in `tags`, so `$addToSet` behaves like `$push`
const update = { $addToSet: { tags: 'MongoDB' } };
const opts = { new: true };

let doc = await Post.findOneAndUpdate({ title }, update, opts);
doc.tags; // ['Node.js', 'MongoDB']

// Since 'MongoDB' is in `tags`, `$addToSet` will be a no-op.
doc = await Post.findOneAndUpdate({ title }, update, opts);
doc.tags; // ['Node.js', 'MongoDB']
```

The `$addToSet` works with objects, but it will only skip adding if the array contains an object that is *deeply equal* to the given object. In MongoDB, 2 objects are deeply equal if they have the same keys in the same order, and each key's values are deeply equal. For example, here's how `$addToSet` works with an array of `comments`.

```
// Make sure to add `_id: false`, otherwise Mongoose adds a unique
// unique `_id` to every subdoc, and then `$addToSet` will always
// add a new doc.
const commentSchema = Schema({ user: String, comment: String },
  { _id: false });
const schema = Schema({ comments: [commentSchema] });
const Post = mongoose.model('BlogPost', schema);
const comment = { user: 'jpicard', comment: 'Make it so!' };
await Post.create({ comments: [comment] });

const update = { $addToSet: { comments: comment } };
const opts = { new: true };

// Skips adding, the new and old comments are deeply equal
let doc = await Post.findOneAndUpdate({}, update, opts);
doc.comments.length; // 1

// Adds a new comment, because the `comment` property is different
update.$addToSet.comments.comment = 'Engage!';
doc = await Post.findOneAndUpdate({}, update, opts);
doc.comments.length; // 2

// Adds a new comment, the new comment has different keys
delete update.$addToSet.comments.comment;
doc = await Post.findOneAndUpdate({}, update, opts);
doc.comments.length; // 3
```

### 3.5: Sort Order

Operations like `findOne()` and `updateOne()` find the first document in the collection that matches a given filter. But how does MongoDB determine what the first document is?

By default, MongoDB iterates through documents in what MongoDB calls *natural order*. Natural order means whatever order MongoDB decides to store the documents in internally, which isn't guaranteed.

When you execute a `findOne()`, by default MongoDB gives you the first document that matches the given filter in natural order. So if there are multiple documents in the collection that match the given `filter`, there's no guarantee which one MongoDB will return.

```
const schema = new Schema({ name: String, age: Number });
const Character = mongoose.model('Character', schema);
await Character.create({ name: 'Jean-Luc Picard', age: 59 });
await Character.create({ name: 'Will Riker', age: 29 });

// `doc` may be either of the 2 documents inserted above.
// Natural order means MongoDB may return whichever one.
const doc = await Character.findOne();
```

MongoDB has a `sort` option for operations like `findOne()` that let you tell MongoDB what fields to sort on. Mongoose queries also have a `sort()` helper that sets the `sort` option for you.

```
// The 3rd parameter to `findOne()` is an `options` parameter.
// `options.sort` lets you specify what order MongoDB should
// use when checking which documents match `filter`
const options = { sort: { name: 1 } };
// `doc` will always be the doc with name 'Jean-Luc Picard'
let doc = await Character.findOne({}, null, options);

// You can also set the `sort` option using `Query#sort()`.
doc = await Character.findOne({}).sort({ name: 1 });
```

The `Query#sort()` function in Mongoose works with the following operations:

- `find()`
- `updateOne()`
- `replaceOne()`
- `findOneAndUpdate()`
- `findOneAndReplace()`
- `deleteOne()`
- `findOneAndDelete()`

```
// Update the character with the highest `age`. The `sort` option
// can be a string property name, or an object.
await Character.updateOne({}, { rank: 'Captain' }).sort('-age');

// Find all characters, sorted by `age`
const docs = await Character.find().sort({ age: 1 });

// Delete the character whose name comes first alphabetically
// out of all the characters whose age is greater than 25.
await Character.deleteOne({ age: { $gt: 25 } }).sort('name');
```

Sorting is useful for more than just ordering by numeric values like `age`. MongoDB can sort any other type. For example, by default MongoDB sorts strings in *lexicographic order*. Here's how lexicographic ordering compares two strings:

- Compare the first character of both strings based on their ASCII character code, so `'' < 'A' < 'Z' < 'a' < 'z'`.
- If the first two characters are equal, compare the 2nd character of each string.

```
const schema = new Schema({ value: String });
const TestString = mongoose.model('TestString', schema);

await TestString.create([
  { value: 'A' },
  { value: 'a' },
  { value: 'Z' },
  { value: 'z' },
  { value: '' },
  { value: 'aa' }
]);

let docs = await TestString.find().sort({ value: 1 });
docs.map(v => v.value); // ['', 'A', 'Z', 'a', 'aa', 'z']
```

#### Sorting Types Other than Strings and Numbers

MongoDB can also compare values of different types. When sorting values of different types, MongoDB first compares the type, and only compares values if both values are the same type. For example, in MongoDB, a number is always less than a string.

Here's the order of types in MongoDB, from lowest to highest:

- Null
- Numbers
- Strings
- Objects
- Arrays
- BinData: the type Mongoose uses to store Node.js buffers
- ObjectId
- Boolean
- Date
- Timestamp
- Regular Expression

Here's an example of sorting values with different types using Mongoose's `Query#sort()` function.

```
// `value: {}` means Mongoose skips casting the `value` property
const schema = new Schema({ value: {} });
const Test = mongoose.model('Test', schema);
await Test.create([
  { value: 42 },
  { value: 'test string' },
  { value: true },
  { value: null }
]);

const docs = await Test.find().sort({ value: 1 });
docs.map(v => v.value); // [null, 42, 'test string', true]
```

#### With Query Selectors and Update Operators

The update operators `$min` and `$max` rely on comparing two values. `$min` and `$max` depend on the same sort order as the `Query#sort()` function. For example, since `null` is the lowest value in MongoDB's sort order, updating a field with `$min: { field: null }` will always set that field to `null`.

```
// `value: {}` means Mongoose skips casting the `value` property,
// so `value` can be of any type.
const schema = new Schema({ value: {} });
const Test = mongoose.model('Test', schema);
await Test.create([{ value: 42 }]);

const opts = { new: true };
// Does nothing because 'a' > 42 in MongoDB's sort order
let doc = await Test.findOneAndUpdate({}, { $min: { value: 'a' } }, opts)
doc.value; // 42

// Sets `value` to `null` because `null` is smaller than
// any other value in MongoDB's sort order.
doc = await Test.findOneAndUpdate({}, { $min: { value: null } }, opts);
doc.value; // null
```

On the other hand, the comparison query selectors `$gte`, `$lte`, `$gt`, and `$lt` implicitly filter for elements that have the same type as the given value. For example, `field: { $gte: null }` will only match documents whose `field` property is exactly equal to `null`, because `$gte` implicitly filters for values that have the same type as `null`.

```
const schema = new Schema({ value: {} });
const Test = mongoose.model('Test', schema);
await Test.create([{ value: 42 }]);

// Does **not** find the doc. 42 is greater than null in MongoDB
// sort order, but `$gte` only compares values with the same type
let doc = await Test.findOne({ value: { $gte: null } });

// Also doesn't find the doc. `$lte` will only compare docs
// whose `value` is the same type as the given value (string).
doc = await Test.findOne({ value: { $lte: '42' } });
```

### 3.6: Limit, Skip, Project

Sometimes, the result set from `find()` is too big to fit in memory. Mongoose queries have a `limit()` function that sets a limit on the maximum amount of documents that a query can return.

```
const schema = new Schema({ name: String, age: Number });
const Character = mongoose.model('Character', schema);
await Character.create({ name: 'Jean-Luc Picard', age: 59 });
await Character.create({ name: 'Will Riker', age: 29 });
await Character.create({ name: 'Deanna Troi', age: 29 });

// Sort by `age` ascending, and return at most 2 documents.
const docs = await Character.find().sort({ age: 1 }).limit(2);
```

Note that `limit()` takes effect *after* the sort order. Even if you put `.limit()` before `.sort()`, MongoDB will internally sort the documents and then apply the limit.

The `skip()` function is commonly used with `limit()`. `skip(num)` tells MongoDB to skip the first `num` documents.

```
await Character.create({ name: 'Jean-Luc Picard', age: 59 });
await Character.create({ name: 'Beverly Crusher', age: 40 });
await Character.create({ name: 'Will Riker', age: 29 });
await Character.create({ name: 'Deanna Troi', age: 29 });

const docs = await Character.find().
  sort({ age: 1 }). // Sort by `age` ascending
  skip(2). // Skip the 2 documents
  limit(2); // And return at most 2 documents
docs.map(doc => doc.name); // ['Bevery Crusher', 'Jean-Luc Picard']
```

Like `limit()`, MongoDB applies `skip()` *after* applying the sort order. MongoDB applies `skip()` *before* applying `limit()`. So even if you change the order of `skip()`, `limit()`, and `sort()`, MongoDB will still apply them in the following order:

- Sort the documents to match the given sort order
- Skip the first `num` docs
- Collect docs until it reaches the given `limit`

`skip()` and `limit()` are most commonly used for pagination. Imagine the page counter on the bottom of Google's search results:


Here's how you might implement a `getPage()` function that finds the search results for a given page.

```
const resultSchema = Schema({ title: String, order: Number });
const SearchResult = mongoose.model('SearchResult', resultSchema);
for (let i = 1; i <= 25; ++i) {
  await SearchResult.create({ order: i, title: 'test' + i });
}

function getResults(page, resultsPerPage) {
  return SearchResult.find().
    sort({ order: 1 }).
    skip(page * resultsPerPage).
    limit(resultsPerPage);
}

// Returns results 11-20
const docs = await getResults(1, 10);
```

### Projections

Sometimes you want to omit certain fields from a query result. MongoDB supports *projections*, which let you select which fields to include or exclude from the result documents.

Mongoose queries have a `select()` function that let you add a projection to your query. Projections are objects that have the following form:

```
{ field1: <value1>, field2: <value2> }
```

`value1` and `value2` can either be:

- Truthy (`1`, `true`, etc.) to include `field1` and `field2`, and exclude all other fields
- Falsy (`0`, `false`, etc.) to exclude `field1` and `field2`, and include all other fields.

For example, here's an example of using `Query#select()` to include or exclude fields.

```
let schema = Schema({ name: String, age: Number, rank: String });
const Character = mongoose.model('Character', schema);
await Character.create([
  { name: 'Will Riker', age: 29, rank: 'Commander' }
]);

// Include `name` and `age`, exclude `rank`
let projection = { name: 1, age: 1 };

let doc = await Character.findOne().select(projection);
doc.name; // 'Will Riker'
doc.rank; // undefined

// Exclude `name` and `age`, include `rank`
projection = { name: false, age: false };

doc = await Character.findOne().select(projection);
doc.name; // undefined
doc.rank; // 'Commander'
```

A projection is either *inclusive* or *exclusive*. That means either all the values in the object must be truthy, or they must all be falsy. If you try to mix and max projections, MongoDB will throw an `MongoError`:

```
const projection = { name: 1, age: 0 };
const err = await Character.findOne().select(projection).
  catch(err => err);
// 'Projection cannot have a mix of inclusion and exclusion.'
err.message;
```

Projections only work with query operations that return a document or an array of documents. For example, you can use projections with `findOneAndUpdate()`:

```
const update = { age: 44, rank: 'Captain' };
const opts = { new: true };
const projection = { name: 1, age: 1 };
// Updates `rank`, but excludes it from the result document
const doc = await Character.findOneAndUpdate({}, update, opts).
  select(projection);

doc.age; // 44
doc.rank; // undefined
```

But projections have no impact on `updateOne()`. For example, you can't exclude `nModified` from the result of `updateOne()` using projections:

```
const update = { age: 44, rank: 'Captain' };
const fields = { nModified: 0 };
const res = await Character.updateOne({}, update).select(fields);

res.nModified; // 1
```

#### Schema-Level Projections

You can define default projections on your schema. For example, you may want to exclude a user's `email` from query results by default. Here's how you can accomplish that using the `select` option for schema types:

```
const User = mongoose.model('User', Schema({
  name: String,
  // Exclude `email` by default
  email: { type: String, select: false }
}));
await User.create({ name: 'John', email: 'john@gmail.com' });
await User.create({ name: 'Bill', email: 'bill@startup.co' });

const docs = await User.find().sort({ name: 1 });
docs[0].name; // 'Bill'
docs[1].name; // 'John'

docs[0].email; // undefined
docs[1].email; // undefined
```

You need to be careful when including paths that are excluded in the schema by default. If you use `.select('email')`, you'll include the `email` field but exclude all other fields.

```
const docs = await User.
  find().
  sort({ name: 1 }).
  select('email');
docs[0].name; // undefined
docs[1].name; // undefined

docs[0].email; // 'bill@startup.co'
docs[1].email; // 'john@gmail.com'
```

If you want to include `email` without excluding all other fields, use `.select('+email')`. When you prefix a field with `+`, that tells Mongoose to skip adding a schema-level projection that excludes `email`, as opposed to adding a projection that excludes everything except `email`.

```
const docs = await User.
  find().
  sort({ name: 1 }).
  select('+email'); // Note the `+` here
docs[0].name; // Bill Johnson
docs[1].name; // John Smith

docs[0].email; // 'bill@startup.co'
docs[1].email; // 'john.smith@gmail.com'
```

### 3.7: Query Casting and Validators

Mongoose casts fields query filters and updates to the path specified in your schema. For example, Mongoose casts strings in the `filter` to ObjectIds if you query by `_id`.

```
const schema = Schema({ name: String });
const Character = mongoose.model('Character', schema);
const ObjectId = mongoose.Types.ObjectId;
const _id = '5dd57639649ce0bd87750caa';
await Character.create([{ _id: ObjectId(_id), name: 'Will Riker' }]);

// Works even though `_id` is a string
const doc = await Character.findOne({ _id });
doc._id instanceof ObjectId; // true
doc._id === _id; // false
```

Mongoose also casts updates to the type specified in your schema. For example, if you call `updateOne()` with `age` as a string as opposed to a number, Mongoose will cast it for you.

```
const schema = Schema({ name: String, age: Number });
const Character = mongoose.model('Character', schema);
await Character.create([{ name: 'Will Riker', age: 29 }]);

const filter = { name: 'Will Riker' };
// Even though `age` is a string, Mongoose will cast `age` to a
// number because that's the type in `schema`.
const update = { age: '30' };
const opts = { new: true };
const doc = await Character.findOneAndUpdate(filter, update, opts);

doc.age; // 30, as a number
```

If Mongoose can't cast a value in `filter` or `update` to the correct type, Mongoose will reject the query with a `CastError`:

```
// This `findOneAndUpdate()` will error out because Mongoose can't
// cast the string 'not a number' to a number.
const filter = { name: 'Will Riker' };
const update = { age: 'not a number' };
const opts = { new: true };
const err = await Character.findOneAndUpdate(filter, update, opts).
  catch(err => err);

// 'Cast to number failed for value "not a number" at path "age"'
err.message;
err.name; // 'CastError'
```

Mongoose runs casting on both filters and updates. So if you try to use a query selector like `$gte` on a number path, and the value isn't something Mongoose can cast to a number, Mongoose will also reject the query with a `CastError`:

```
const filter = { age: { $gte: 'fail' } };
const err = await Character.findOne(filter).catch(err => err);

// 'Cast to number failed for value "fail" at path "age" for
// model "Character"'
err.message;
err.name; // 'CastError'
```

### Query Validation

Mongoose does not validate queries by default, but it can if you enable the `runValidators` option on your query.

```
const schema = Schema({
  name: String,
  rank: { type: String, enum: ['Captain', 'Commander'] }
});
const Character = mongoose.model('Character', schema);
await Character.create([{ name: 'Will Riker', rank: 'Commander' }]);

// By default, Mongoose will let the below update go through, even
// though 'Lollipop' is not in the `enum` of allowed values.
const update = { rank: 'Lollipop' };
let opts = { new: true };
const doc = await Character.findOneAndUpdate({}, update, opts);
doc.rank; // 'Lollipop'

// But if you set `runValidators`, Mongoose will run the `enum`
// validator and reject because of an invalid `rank`.
opts = { runValidators: true };
const err = await Character.findOneAndUpdate({}, update, opts).
  catch(err => err);

// '`Lollipop` is not a valid enum value for path `rank`.'
err.message;
err.name; // 'ValidationError'
```

Mongoose only runs query validation on updates. For example, the below query goes through even though the `filter` option contains an invalid `rank`. That's because the below query operation doesn't write to the database, so there's no risk of storing an invalid `rank`.

```
let opts = { runValidators: true };
// Mongoose executes the below query without error
await Character.findOne({ rank: 'Lollipop' }).setOptions(opts);
```

However, if the query op is `findOneAndUpdate()` or `updateOne()`, and the `upsert` option is set, it is possible that `filter` may end up getting stored in the database. Be careful when using query validators with upserts!

```
// Mongoose will let the below upsert through, which will store
// an invalid `rank` in the database.
const filter = { rank: 'Lollipop' };
const update = { name: 'test' };
const opts = { runValidators: true, upsert: true, new: true };
const doc = await Character.findOneAndUpdate(filter, update, opts);
doc.rank; // 'Lollipop'
```

Update validators only run on paths specified in the `update` parameter. So if `rank` isn't in the update, Mongoose won't run query validators on it. Even if `rank` is invalid in the database, Mongoose will still allow the below update to go through.

```
// Insert an invalid doc in the database
const doc = { _id: new mongoose.Types.ObjectId(), rank: 'Lollipop' };
await Character.collection.insertOne(doc);
// Below update succeeds, even though the document in the database
// has an invalid `rank`
await Character.updateOne({ _id: doc._id }, { name: 'Test' });
```

#### Query Validators with Custom Validators

Query validators are hidden behind a `runValidators` option because query validators have some caveats with custom validators. Conceptually, there are two types of validators:

- Simple validators are validators that only rely on the value being validated
- Complex validators are validators that rely on other values in the document

Most built-in validators, like `enum` and `required`, are simple validators. But, in practice, developers often write complex validators to express constraints that involve multiple properties.

For example, suppose that, in order to have `rank = 'Captain'`, a character's `age` must be at least 30. Here's how you would write a custom validator that handles this for `save()`:

```
const schema = Schema({
  name: String,
  age: Number,
  rank: { type: String, validate: ageRankValidator }
});
function ageRankValidator(v) {
  const message = 'Captains must be at least 30';
  assert(v !== 'Captain' || this.age >= 30, message);
}

const Character = mongoose.model('Character', schema);
const rank = 'Captain';
const doc = new Character({ name: 'Will Riker', age: 29, rank });
const err = await doc.save().catch(err => err);
// 'Character validation failed: rank: Captains must be at least 30'
err.message;
```

But `ageRankValidator` assumes that `this` is the document being validated. Unfortunately, with a query validator, the document being updated might not be in memory. If you're updating the document's `age`, you might not know the document's `rank`.

When Mongoose runs a custom validator as part of query validation, `this` will be the query object. For complex custom validators, you need to check whether `this` is a `Query` or a `Document`, and handle each case appropriately.

```
function ageRankValidator(v) {
  const message = 'Captains must be at least 30';
  if (this instanceof mongoose.Query) {
    const update = this.getUpdate();
    assert(v !== 'Captain' || update.$set.age >= 30, message);
  } else {
    assert(v !== 'Captain' || this.age >= 30, message);
  }
}

const update = { age: 29, rank: 'Captain' };
// The `context` option tells Mongoose to set the value of `this`
// in the validator to the query. Otherwise `this` will be `null`
const opts = { runValidators: true, context: 'query' };
// Throws 'Validation failed: rank: Captains must be at least 30'
await Character.findOneAndUpdate({}, update, opts);
```

## 3.8: Summary

In Mongoose, queries let you load documents from the database as well as execute CRUD operations. Because of limitations with validation, you should generally use `save()` to update Mongoose documents, rather than `updateOne()` or `findOneAndUpdate()`.

However, if you need atomic updates, you should use `findOneAndUpdate()`. And if you need to update multiple documents quickly, `updateMany()` is helpful.

Each Mongoose query has a distinct `op` that determines what the actual operation Mongoose will send to MongoDB is. The `op` can be a create operation like `findOne()`, a replace operation like `replaceOne()`, an update operation like `updateMany()`, a delete operation like `deleteOne()`, or an "other" operation like `distinct()`.

Each query also stores `filter`, `update`, and `options` properties. Depending on the `op`, Mongoose may or may not send these properties to MongoDB. But `filter` defines which documents MongoDB should look for, `update` defines how MongoDB should change those documents, and `options` defines other tuneable parameters.

# 4: Middleware

In Mongoose, middleware lets you attach your own custom logic to built-in Mongoose functions. Middleware functions are sometimes called *hooks*. That's because middleware lets you "hook" into existing functions.

For example, suppose you wanted to log every time you saved a document. You can define a `pre('save')` middleware function that Mongoose will call every time you call `save()`.

```
const schema = Schema({ name: String });

// `middleware` is a function that Mongoose will call for you
// when you call `save()`
schema.pre('save', function middleware() {
  console.log('Saving', this.name);
});

const Model = mongoose.model('Test', schema);
const doc = new Model({ name: 'test' });

// Prints "Saving test"
await doc.save();
```

You can define middleware for many built-in Mongoose functions. `save()` middleware is most common, but you can also define middleware for other document methods, like `validate()` and `remove()`. You can also define middleware for query functions and aggregation framework calls.

Middleware is defined on a schema. To define middleware, you need to use the `Schema#pre()` and `Schema#post()` functions.

## 4.1: `pre()` and `post()`

The `Schema#pre()` and `Schema#post()` functions each take 2 parameters:

- A string `name`.
- A middleware function `fn`.

`name` tells Mongoose which function you want to attach middleware to, and `fn` is the middleware function you want Mongoose to execute. The function you attach middleware to is called the *wrapped function*. `Schema#pre()` tells Mongoose to execute `fn` **before** the wrapped function, and `Schema#post()` tells Mongoose to execute `fn` **after** the wrapped function.

For example, Mongoose calls `pre('save')` middleware before actually sending the updated fields to the database. Then Mongoose calls `post('save')` middleware after it sends the

updated fields to the database and receives a response from MongoDB.

```
const schema = Schema({ name: String });

// Mongoose will execute `preFn()` before saving the document...
schema.pre('save', function preFn() {
  console.log('Saving', this.name);
});
// And `postFn()` after saving the document
schema.post('save', function postFn() {
  console.log('Saved', this.name);
});

const Model = mongoose.model('Test', schema);
const doc = new Model({ name: 'test' });
// Prints "Saving test\nSaved test"
await doc.save();
```

You can call define any number of pre and post hooks for a given function. Mongoose executes middleware functions in series in the order of your `pre('save')` and `post('save')` calls.

```
const schema = Schema({ name: String });
schema.pre('save', () => console.log('pre save 1'));
schema.pre('save', () => console.log('pre save 2'));

const Model = mongoose.model('Test', schema);
const doc = new Model({ name: 'test' });
// Prints "pre save 1" followed by "pre save 2"
await doc.save();
```

Mongoose passes the result `res` of the operation as the first parameter to `post()` middleware. For example, since `await doc.save()` returns a document, Mongoose passes the saved document as the first parameter to `post('save')` middleware functions.

```
schema.post('save', function(doc) {
  console.log('Saved:', doc.name);
});

const Model = mongoose.model('Test', schema);
const doc = new Model({ name: 'test' });
// Prints "Saved: test"
await doc.save();
```

### 4.2: Async Middleware Functions

Middleware functions can be asynchronous. There's two ways to make your hook async:

1) If the middleware function returns a promise (like an `async function`), Mongoose will wait for that promise to settle before executing the next middleware function in the chain.

```
schema.pre('save', async function() {
  console.log('Waiting');
  await new Promise(resolve => setTimeout(resolve, 50));
  console.log('First Done');
});
schema.pre('save', () => console.log('Second'));

const Model = mongoose.model('Test', schema);
const doc = new Model({ name: 'test' });
// Prints "Waiting", "First Done", "Second"
await doc.save();
```

2) Mongoose passes a function `next()` as the 1st parameter to pre hooks and as the 2nd parameter to post hooks. Mongoose waits until you call `next()` to move on to the next hook.

```
schema.pre('save', function(next) {
  console.log('1');
  setTimeout(() => { console.log('2'); next(); }, 50);
});
schema.pre('save', () => console.log('3'));

// For post hooks, `next()` is the 2nd parameter
schema.post('save', function(doc, next) {
  console.log('4');
  setTimeout(() => { console.log('5'); next(); }, 50);
});
schema.post('save', () => console.log('6'));

const Model = mongoose.model('Test', schema);
const doc = new Model({ name: 'test' });
// Prints "1", "2", "3", "4", "5", "6"
await doc.save();
```

### 4.3: Document Middleware

Middleware in Mongoose falls into one of 4 different types:

1. Document middleware
2. Model middleware
3. Aggregation middleware
4. Query middleware

There are 2 primary differences between these different types of middleware.

The first difference is the value of `this` in the middleware functions. Document middleware only applies to methods of the `Model` class (remember that a document is an instance of a model).

That means when you define middleware for `save()`, you're defining document middleware, because `save()` is a method on the Model class. Mongoose calls document middleware functions with `this` set to the the document you're calling the method on.

```
const schema = Schema({ name: String });
schema.pre('save', function() {
  doc === this; // true
});
const Model = mongoose.model('Test', schema);

const doc = new Model({ name: 'test' });
// When you call `doc.save()`, Mongoose calls your pre save middleware
// with `this` equal to `doc`.
await doc.save();
```

The second difference is the value of the `res` parameter to post middleware. For document middleware, `res` is the same as `this`. However, that is not the case for other types of middleware.

```
const schema = Schema({ name: String });
schema.post('save', function(res) {
  res === this; // true
  res === doc; // true
});
const Model = mongoose.model('Test', schema);

const doc = new Model({ name: 'test' });
// When you call `doc.save()`, Mongoose calls your post save middleware
// with `res` equal to `doc`.
await doc.save();
```

There are 5 document functions you can attach middleware to:

- `validate()`
- `save()`
- `remove()`
- `updateOne()`
- `deleteOne()`

Below is an example of applying middleware to multiple document methods:

```
const schema = Schema({ name: String });
schema.post(['save', 'validate', 'remove'], function(res) {
  res === this; // true
  res === doc; // true
});
const Model = mongoose.model('Test', schema);
const doc = new Model({ name: 'test' });

// Triggers post('validate') hook
await doc.validate();

// Triggers post('save') **and** post('validate') hook
await doc.save();

// Triggers post('remove') hook
await doc.remove();
```

### `save()` Triggers `validate()` Middleware

Under the hood, Mongoose attaches a `pre('save')` middleware to all models that calls `validate()`. That means `save()` triggers `validate()` middleware.

```
const schema = Schema({ name: String });
schema.pre('validate', () => console.log('pre validate'));
const Model = mongoose.model('Test', schema);

const doc = new Model({ name: 'test' });

// Prints "pre validate"
await doc.save();
```

Keep in mind that Mongoose calls `validate()` in a pre hook when you call `save()`. That means Mongoose runs `post('validate')` middleware **before** `pre('save')` middleware. Here's the

order of execution for `save()` and `validate()` middleware when you call `save()`:

1. `pre('validate')`
2. `post('validate')`
3. `pre('save')`
4. `post('save')`

Here's an example of what happens when you have `pre('validate')`, `post('validate')`, `pre('save')`, and `post('save')` middleware on the same schema:

```
const schema = Schema({ name: String });
schema.pre('validate', () => console.log('pre validate'));
schema.post('validate', () => console.log('post validate'));
schema.pre('save', () => console.log('pre save'));
schema.post('save', () => console.log('post save'));

const Model = mongoose.model('Test', schema);
const doc = new Model({ name: 'test' });

// Prints "pre validate", "post validate", "pre save", "post save"
await doc.save();
```

### 4.4: Model Middleware

Mongoose has middleware for the `insertMany()` function. The `insertMany()` function takes an array of objects, validates them, and sends them to the MongoDB server in bulk. Because `insertMany()` makes only 1 request to the database regardless of the number of documents, it can be faster than calling `save()` on each document individually.

The `insertMany()` function is a static function on the model class, not a method like `save()`. In `insertMany()` hooks, `this` is the model, and the `res` parameter is the inserted documents.

```
const schema = Schema({ name: String });
schema.post('insertMany', function(res) {
  this === Model; // true
  res[0] instanceof Model; // true
  res[0].name; // 'test'
});

const Model = mongoose.model('Test', schema);

// Triggers `post('insertMany')` hooks
await Model.insertMany([{ name: 'test' }]);
```

The reason why model middleware is inherently different from document middleware is that the wrapped function `insertMany()` is a static function on the model class, whereas `save()` and `validate()` are methods on the model class. `Schema#pre()` and `Schema#post()` let you attach middleware to different categories of functions from one place, based on the function's name.

### 4.5: Aggregation Middleware

Mongoose models have an `aggregate()` function that helps you use the MongoDB aggregation framework. The aggregation framework gives you the flexibility to filter and transform documents in much more sophisticated ways than a simple query.

For now, it is sufficient to know that the `Model.aggregate()` function's first parameter is an array of objects called `pipeline`. A pipeline is an array of stages: each stage defines a transformation on a set of documents. For example, the `$match` stage filters a set of documents, so a pipeline that contains a single `$match` stage is equivalent to a `find()`.

```
const schema = Schema({ name: String, age: Number });
const Model = mongoose.model('Character', schema);

await Model.create({ name: 'Jean-Luc Picard', age: 59 });
await Model.create({ name: 'Will Riker', age: 29 });
await Model.create({ name: 'Deanna Troi', age: 29 });

// The below is equivalent to `Model.find({ age: { $gte: 30 } })`
const docs = await Model.aggregate([
  {
    // To add a `$match` stage, you add an object to the array
    // with a `$match` property. The value of the `$match` property
    // is a filter. The aggregation framework only lets documents
    // that match the filter pass the `$match` stage.
    $match: { age: { $gte: 30 } }
  }
]);
docs.length; // 1
docs[0].name; // 'Jean-Luc Picard'
```

Although `aggregate()` is a static function on the `Model` class, aggregation middleware is slightly different from model middleware. That's because when you call `Model.aggregate()`, Mongoose returns an `Aggregate` object, **not** a promise.

The `Aggregate` class is a tool for building aggregation pipelines via chaining syntax. Aggregation objects are thenables, so `await` executes the aggregation and returns the result.

```
// Note that there's no `await` here.
const aggregate = Model.aggregate([
  { $match: { age: { $gte: 30 } } }
]);
// Mongoose's `Aggregate` class is a tool for building
// aggregation pipelines using function call chaining.
aggregate instanceof mongoose.Aggregate; // true
```

In aggregation middleware functions, `this` is an `Aggregate` object, and `res` is the result of the aggregation call. `res` is always an array.

```
const schema = Schema({ name: String, age: Number });

schema.pre('aggregate', function() {
  this instanceof mongoose.Aggregate; // true
  const pipeline = this.pipeline();
  pipeline[0]; // { $match: { age: { $gte: 30 } } }
});
schema.post('aggregate', function(res) {
  this instanceof mongoose.Aggregate; // true
  res.length; // 1
  res[0].name; // 'Jean-Luc Picard'
});

const Model = mongoose.model('Character', schema);

// Triggers `pre('aggregate')` and `post('aggregate')`
await Model.aggregate([{ $match: { age: { $gte: 30 } } }]);
```

#### When Does Middleware Execute?

Mongoose doesn't implicity execute aggregation objects. You need to explicitly execute an aggregation in one of 3 ways:

1. `await` on the aggregation object
2. Call `Aggregate#then()`
3. Call `Aggregate#exec()`

Under the hood, methods (1) and (2) call `Aggregate#exec()`, so these 3 ways are purely syntactic sugar. The key takeaway here is that Mongoose does not run aggregation middleware until you `exec()` the aggregation.

```
const schema = Schema({ name: String, age: Number });
schema.pre('aggregate', function() {
  console.log('Called');
});

const Model = mongoose.model('Character', schema);

// Does **not** trigger aggregation middleware
const agg = Model.aggregate([{ $match: { age: { $gte: 30 } } }]);

// Mongoose only runs aggregation middleware when you `exec()`
await agg.exec();
```

#### Modifying the Pipeline

`pre('aggregate')` and `post('aggregate')` middleware serve different purposes:

- `pre('aggregate')` is for modifying the aggregation before it executes.
- `post('aggregate')` is for checking and modifying the result of the aggregation after it executes.

The most common use case for aggregation middleware is using `pre('aggregate')` to add stages to all `aggregate()` calls. For example, suppose you have a model `User` with a property `isDeleted`. You want to make sure aggregation pipelines never touch users that have `isDeleted` set. This pattern is known as a *soft delete*.

Here's how you can add a pipeline stage in `pre('aggregate')` that ensures all aggregation pipelines ignore soft deleted users.

```
const s = Schema({ name: String, age: Number, isDeleted: Boolean });

s.pre('aggregate', function() {
  // Prepend a `$match` stage to every aggregation pipeline that
  // filters out documents whose `isDeleted` property is true
  this.pipeline().unshift({
    $match: {
      isDeleted: { $ne: true }
    }
  })
});
```

Given this `pre('aggregate')` middleware, all `aggregate()` calls will filter out soft deleted documents. For example:

```
await User.create({ name: 'Jean-Luc Picard', age: 59 });
await User.create({ name: 'Will Riker', age: 29 });
await User.create({ name: 'Tasha Yar', isDeleted: true });

// Will **not** return the Tasha Yar doc, because
// that user is soft deleted.
const $match = { age: { $lte: 30 } };
const docs = await User.aggregate([{ $match }]);
docs.length; // 1
docs[0].name; // Will Riker
```

### 4.6: Query Middleware

Query middleware is conceptually similar to aggregation middleware. Mongoose models have several static functions, like `find()` and `updateOne()`, that return Mongoose `Query` objects. Mongoose doesn't execute a query until you either:

1. `await` on the query
2. Call `Query#then()`
3. Call `Query#exec()`

Like aggregation middleware, Mongoose only runs query middleware when you execute the query.

```
const schema = Schema({ name: String, age: Number });
// Attach pre('find') middleware
schema.pre('find', function() {
  console.log('Find', this.getQuery());
});
const Model = mongoose.model('Model', schema);

// Doesn't print anything
const query = Model.find({ age: { $lte: 30 } });

// Prints "Find { age: { $lte: 30 } }" from the `find()` middleware
await query;
```

Here's a list of all query functions that you can attach middleware to:

- `Query#count()`
- `Query#countDocuments()`
- `Query#deleteOne()`
- `Query#deleteMany()`
- `Query#distinct()`

- `Query#estimatedDocumentCount()`
- `Query#find()`
- `Query#findOne()`
- `Query#findOneAndDelete()`
- `Query#findOneAndRemove()`
- `Query#findOneAndReplace()`
- `Query#findOneAndUpdate()`
- `Query#remove()`
- `Query#replaceOne()`
- `Query#update()`
- `Query#updateMany()`
- `Query#updateOne()`

That means, for example, you can define `pre('findOneAndUpdate')` or `post('findOne')` middleware that will only execute when you call `findOneAndUpdate()` or `findOne()`.

```
schema.pre('findOneAndUpdate', () => console.log('update'));
schema.post('findOne', () => console.log('findOne'));
const Model = mongoose.model('Model', schema);

// Doesn't trigger any middleware
const query = Model.findOneAndUpdate({}, { name: 'test' });

// Prints "update"
await query.exec();

// Prints "findOne"
await Model.findOne({});
```

Remember that you can change the query's `op` before you execute it. Just because you see a `Model.find()` doesn't necessarily mean Mongoose will execute `find` middleware - Mongoose only executes the middleware for the query's `op`.

```
schema.pre('find', () => console.log('find'));
schema.pre('updateOne', () => console.log('updateOne'));
const Model = mongoose.model('Model', schema);

const query = Model.find({ name: 'Jean-Luc Picard' });
query.updateOne({}, { age: 70 });
query.op; // 'updateOne'

// Triggers `updateOne` middleware, **not** `find` middleware.
await query.exec();
```

You may also have noticed that there are several query methods that don't have corresponding middleware, like the `Query#findById()` method. The `findById()` method is just a thin syntactic wrapper around `findOne()`, and it triggers `findOne()` middleware. Calling `schema.pre('findById')` doesn't do anything.

```
schema.pre('findOne', () => console.log('findOne'));
const Model = mongoose.model('Model', schema);

// Prints "findOne". There is no middleware for `findById()`.
await Model.findById(new mongoose.Types.ObjectId());
```

## Name Conflicts

So far query middleware is pretty similar to aggregation middleware. The reason why this book explains query middleware last is because some query middleware functions have the same name as document middleware functions:

- `updateOne`
- `deleteOne`
- `remove`

What happens when you call `schema.pre('updateOne')`? By default, Mongoose treats `updateOne` and `deleteOne` middleware as just query middleware. In other words, `schema.pre('updateOne')` registers a pre hook for just `Query#updateOne()`, **not** `Document#updateOne()`.

Because `Document#updateOne()` calls `Query#updateOne()` under the hood, calling `doc.updateOne()` triggers `updateOne` query middleware as shown below.

```
schema.pre('updateOne', function() {
  console.log('updateOne', this.constructor.name);
});
const Model = mongoose.model('UserModel', schema);
const doc = await Model.create({ name: 'Jean-Luc Picard', age: 59 });

// Prints "updateOne model" followed by "updateOne Query".
// `Document#updateOne()` triggers both document and query hooks
await doc.updateOne({ age: 60 });

// Prints "updateOne Query"
await Model.updateOne({ _id: doc._id }, { age: 61 });
```

To switch between document middleware and query middleware, `Schema#pre()` and `Schema#post()` support an optional parameter `options`. Set `options.document` to `true` to tell

Mongoose to trigger document `updateOne` middleware.

Keep in mind that, since `Document#updateOne()` calls `Query#updateOne()`, calling `Document#updateOne()` will trigger your middleware function **twice**: once as query middleware and once as document middleware.

```
// The `options.document` parameter tells Mongoose to call
// 'updateOne' hooks as both document and query middleware
schema.pre('updateOne', { document: true }, function() {
  console.log('updateOne', this.constructor.name);
});
const Model = mongoose.model('UserModel', schema);
const doc = await Model.create({ name: 'Jean-Luc Picard', age: 59 });

// Prints "updateOne model" followed by "updateOne Query"
await doc.updateOne({ age: 60 });
```

To avoid Mongoose double calling your middleware function, you can set `options.query` to `false`. That tells Mongoose to not call this particular middleware function as query middleware for `updateOne()`.

```
// Only call this middleware function as document middleware for
// updateOne, not as query middleware.
schema.post('updateOne', { document: true, query: false }, function() {
  console.log('updateOne', this.constructor.name);
});
const Model = mongoose.model('UserModel', schema);
const doc = await Model.create({ name: 'Jean-Luc Picard', age: 59 });

// Prints "updateOne model"
await doc.updateOne({ age: 60 });
```

For `updateOne()` and `deleteOne()`, the `document` and `query` options let you switch between document and query middleware. The `query` option is `true` by default, the `document` option is `false` by default.

Unfortunately, `remove()` middleware is slightly different. In Mongoose 5, for `remove()` middleware, `options.document` is `true` by default and `options.query` is `false` by default. That is because `Document#remove()` middleware is older than the concept of query middleware, so for backwards compatibility Mongoose has maintained that Mongoose only calls `schema.pre('remove')` hooks as document middleware by default.

```
// By default, `schema.pre('remove')` only registers document
// middleware, **not** query middleware.
schema.pre('remove', function() {
  console.log('remove', this.constructor.name);
});
const Model = mongoose.model('UserModel', schema);
const doc = await Model.create({ name: 'Jean-Luc Picard', age: 59 });

// Prints "remove model"
await doc.remove({ age: 60 });
```

You can make Mongoose trigger query middleware for `remove()` by using `schema.pre('remove', { query: true }, fn)`. However, `remove()` is deprecated, so you should use `deleteOne()` where possible instead.

### 4.7: Error Handling Middleware

Error handling middleware is a special type of `post()` middleware that Mongoose calls when an error occurs. To register error handling middleware, you need to register a post hook with 3 parameters: `function(err, res, next) {}`.

```
// This is normal post middleware.
schema.post('save', () => console.log('this wont print'));
// If a post middleware function takes exactly 3 parameters, Mongoose
// will treat it as error handling middleware.
schema.post('save', function errorHandler(err, doc, next) {
  console.log('Error:', err.message);
  next(err);
});
const Model = mongoose.model('UserModel', schema);

try {
  // Prints "Error: UserModel validation failed..." because
  // of the `errorHandler()` function
  await Model.create({ age: 'not a number' });
} catch (error) {
  error.message; // "UserModel validation failed..."
}
```

Mongoose calls error handling middleware if a `pre()` hook errored out, if the wrapped function errored out, or if a preceeding `post()` hook errored out.

```
// Errors in pre hooks will trigger error handling middleware
schema.pre('save', () => { throw new Error('Oops!)' });

// Wrapped function errors trigger error handling middleware
await Model.create({ age: 'not a number' });

// Errors in post hooks also trigger error handling, but only if the
// error handler is defined after the hook that errors out.
schema.post('save', () => { throw new Error('Oops!'); });
```

An error handling middleware must take exactly 3 parameters. That also means your error handling middleware **must** call `next()`, otherwise Mongoose will hang.

A common use case for error handling middleware is converting error messages to a more user friendly format. For example:

```
schema.post('save', function errorHandler(err, doc, next) {
  // By default, duplicate `_id` errors look like this:
  // "E11000 duplicate key error collection: test.usermodels"
  // Error handling middleware can make the error more readable
  if (err.code === 11000) return next(Error('Duplicate _id'));
  next(err);
});
const Model = mongoose.model('UserModel', schema);
const doc = await Model.create({ name: 'test' });
await Model.create({ _id: doc._id }); // Throws "Duplicate _id"
```

## 4.8: Summary

In Mongoose, middleware is broken up into 4 types based on the value of `this` and `res`:

- Document middleware
- Model middleware
- Aggregation middleware
- Query middleware

You register middleware using the `Schema#pre()` and `Schema#post()` functions. Mongoose executes `pre()` middleware before the wrapped function, and `post()` middleware after the wrapped function.

Error handling middleware is a special type of `post()` middleware that only executes if an error occurs. Error handling middleware is useful for logging errors and making error messages more human readable.

## 5: Populate

Mongoose lets documents reference other documents, and `populate()` is how you load referenced documents. For example, suppose you have two models: `Person` and `Group`. Each person belongs to at most one group. In other words, there is a *one-to-many* relationship between groups and people.

```
const Group = mongoose.model('Group', Schema({ name: String }));
const Person = mongoose.model('Person', Schema({
  name: String,
  group: {
    type: mongoose.ObjectId,
    // `ref` means `group` references the 'Group' model
    ref: 'Group'
  }
}));

const jedi = await Group.create({ name: 'Jedi Order' });
await Person.create({ name: 'Luke Skywalker', group: jedi._id });

const doc = await Person.findOne().populate('group');
doc.group.name; // 'Jedi Order'
```

In the above example, Luke Skywalker belongs to the Jedi Order group, by virtue of the fact that Luke Skywalker's `group` property has the same value as the Jedi Order's `_id` property.

Populate is most commonly used with ObjectIds, but you can `populate()` with any built-in Mongoose type. For example, `populate()` still works if the `Group` model has a numeric `_id`.

```
const Group = mongoose.model('Group', Schema({
  _id: Number,
  name: String
}));
const Person = mongoose.model('Person', Schema({
  name: String,
  group: { type: Number, ref: 'Group' }
}));
await Group.create({ _id: 66, name: 'Jedi Order' });
await Person.create({ name: 'Luke Skywalker', group: 66 });

const doc = await Person.findOne().populate('group');
doc.group.name; // 'Jedi Order'
```

The `Query#populate()` function tells Mongoose to execute an additional query to load documents from the model referenced by the `ref` property.

### 5.1: The `ref` Property

Mongoose lets you specify a `ref` property on schema paths, like the `group` path from the previous example. The `ref` property can be one of 3 different types:

1. String: the name of the model that Mongoose should use to load referenced documents.
2. Model: the model that Mongoose should use to load referenced documents.
3. Function: a function that returns either a model name as a string, or a model.

The `ref` property is usually a string, but you can also set it to a model instance:

```
const Group = mongoose.model('Group', Schema({
  _id: Number,
  name: String
}));
const Person = mongoose.model('Person', Schema({
  name: String,
  // `ref` can also be a Mongoose model as opposed to a string
  group: { type: Number, ref: Group }
}));
await Group.create({ _id: 66, name: 'Jedi Order' });
await Person.create({ name: 'Luke Skywalker', group: 66 });

const doc = await Person.findOne().populate('group');
doc.group.name; // 'Jedi Order'
```

You can also make the `ref` property conditional based on the document by setting `ref` to be a function. If `ref` is a function, Mongoose will call the function with the document being populated as the first argument and as `this`.

```
const Person = mongoose.model('Person', Schema({
  name: String,
  groupKind: String,
  // `ref` can also be a function that takes the document being
  // populated as a parameter. That means you can make `ref`
  // conditional based on the document's properties.
  group: { type: Number, ref: doc => doc.groupKind }
}));
```

The `ref` function should return either a Mongoose model or a model name as a string.

What makes `ref` functions so useful is that `populate()` also works with multiple documents. Using `ref` functions, you can make Mongoose populate the same path in multiple documents using different models with a single `populate()` call:

```
const Person = mongoose.model('Person', Schema({
  name: String,
  groupKind: String,
  group: { type: Number, ref: doc => doc.groupKind }
}));
const companySchema = Schema({ _id: Number, name: String });
const Company = mongoose.model('Company', companySchema);

await Group.create({ _id: 66, name: 'Jedi Order' });
await Company.create({ _id: 5, name: 'Cloud City Mining' });
await Person.create([
  { name: 'Luke Skywalker', groupKind: 'Group', group: 66 },
  { name: 'Lando Calrissian', groupKind: 'Company', group: 5 }
]);

const docs = await Person.find().sort({ name: 1 }).populate('group');

// The `group` property now contains multiple unrelated models.
docs[0].group instanceof Company; // true
docs[1].group instanceof Group; // true
docs[0].group.name; // 'Cloud City Mining'
docs[1].group.name; // 'Jedi Order'
```

### `ref` in Arrays

Mongoose also supports `populate()` within arrays. For example, instead of storing a `group` property with a `Person` document, you can store an array of `members` on a `Group` document.

There is still a one-to-many relationship between `Group` documents and `Person` documents, just the way the data is stored in MongoDB is slightly different. In this case, the 'Jedi Order' group has an array of `members`. Without `populate()`, `members` is an array of ObjectIds.

When you `populate()`, Mongoose replaces the first ObjectId with the 'Luke Skywalker' `Person`, because the first element of `members` is an ObjectId that is equal to Luke Skywalker's ObjectId.

```
const Group = mongoose.model('Group', Schema({
  name: String,
  // `members` is an array of ObjectIds with `ref = 'Person'`
  members: [{ type: mongoose.ObjectId, ref: 'Person' }]
}));
let Person = mongoose.model('Person', Schema({ name: String }));

const luke = await Person.create({ name: 'Luke Skywalker' });
await Group.create({ name: 'Jedi Order', members: [luke._id] });

const jedi = await Group.findOne().populate('members');

jedi.members[0] instanceof Person; // true
jedi.members[0].name; // 'Luke Skywalker'
```

Note that `members` is an array of ObjectIds. It is **not** a document array. However, you can `populate()` paths underneath document arrays. For example, suppose that you also want to store each member's `rank`.

```
const Group = mongoose.model('Group', Schema({
  name: String,
  members: [{
    person: { type: mongoose.ObjectId, ref: 'Person' },
    rank: String
  }]
}));

const luke = await Person.create({ name: 'Luke Skywalker' });
const members = [{ person: luke._id, rank: 'Jedi Knight' }];
await Group.create({ name: 'Jedi Order', members });

// `populate()` can "drill down" into document arrays. It loops
// though each element in `members` and populates the `person` path.
const jedi = await Group.findOne().populate('members.person');

jedi.members[0].rank; // 'Jedi Knight'
jedi.members[0].person.name; // 'Luke Skywalker'
```

### 5.2: `populate()` Syntax

The `Query#populate()` function usually takes a single string parameter containing the `path` to populate. `Query#populate()` also accepts two other types of arguments:

1. An `options` object that contains a `path` property.
2. An array of `path` strings or `options` objects.

### `options` Syntax

You can use the `options` syntax to configure tuneable parameters for `populate()`. To specify the path to populate, you need to use the `path` option:

```
await Group.create({ _id: 66, name: 'Jedi Order' });
await Person.create({ name: 'Luke Skywalker', group: 66 });

const doc = await Person.findOne().populate({ path: 'group' });
doc.group.name; // 'Jedi Order'
```

Mongoose supports numerous populate options. For example, there's the `model` option, which lets you override the `ref` property. You can even `populate()` paths that don't have a `ref` property, as long as you specify the `model` option.

```
const groupSchema = Schema({ _id: Number, name: String });
const Group = mongoose.model('Group', groupSchema);
const Person = mongoose.model('Person', Schema({
  name: String,
  // Note that 'ref' points to a model that isn't 'Group'
  group: { type: Number, ref: 'OtherModel' }
}));

await Group.create({ _id: 66, name: 'Jedi Order' });
await Person.create({ name: 'Luke Skywalker', group: 66 });

// The `model` option overrides the model Mongoose populates
const doc = await Person.findOne().populate({
  path: 'group',
  model: Group
});
doc.group.name; // 'Jedi Order'
```

You can think of `ref` as the default model to populate from, and the `model` option as an override.

You can also specify `sort`, `skip`, `limit`, and `match` parameters to further filter the documents that `populate()` finds. The `match` parameter adds additional properties to the `filter` that populate uses to find the referenced documents.

```
const Group = mongoose.model('Group', Schema({
  name: String,
  members: [{ type: mongoose.ObjectId, ref: 'Person' }]
}));
let personSchema = Schema({ name: String, isDeleted: Boolean });
const Person = mongoose.model('Person', personSchema);

const members = await Person.create([
  { name: 'Luke Skywalker' },
  { name: 'Anakin Skywalker', isDeleted: true }
]);
const name = 'Jedi Order';
await Group.create({ name, members });

// Mongoose doesn't know to filter out documents with `isDeleted`
let doc = await Group.findOne({ name }).populate('members');
doc.members.length; // 2
doc.members[1].name; // 'Anakin Skywalker'

// You can use `match` to filter out docs with `isDeleted` set
const match = { isDeleted: { $ne: true } };
const path = 'members';
doc = await Group.findOne({ name }).populate({ path, match });
doc.members.length; // 1
doc.members[0].name; // 'Luke Skywalker'
```

With `sort`, `limit`, and `skip`, you can sort the results of populate, and limit extremely large arrays.

```
const people = await Person.create([
  { name: 'Mace Windu' },
  { name: 'Obi-Wan Kenobi' },
  { name: 'Yoda' },
  { name: 'Anakin Skywalker' }
]);
await Group.create({ name: 'Jedi Order', members: people });

const path = 'members';
const opts = { path, sort: { name: 1 }, skip: 1, limit: 2 };
const doc = await Group.findOne().populate(opts);
// ['Mace Windu', 'Obi-Wan Kenobi']
doc.members.map(doc => doc.name);
```

### Array Syntax

You can call `populate()` with an array of strings to populate multiple paths:

```
const Group = mongoose.model('Group', Schema({
  name: String,
  leader: { type: mongoose.ObjectId, ref: 'Person' },
  people: [{ type: mongoose.ObjectId, ref: 'Person' }]
}));

const mace = await Person.create({ name: 'Mace Windu' });
const adi = await Person.create({ name: 'Adi Gallia' });
const people = [mace, adi];
await Group.create({ name: 'Jedi Order', leader: mace, people });

let doc = await Group.findOne().populate(['leader', 'people']);
doc.leader.name; // 'Mace Windu'
doc.people.map(doc => doc.name); // ['Mace Windu', 'Adi Gallia']
```

If you call `populate()` with an array, Mongoose will treat each element as a separate set of populate options. This means that, with one `populate()` call, you can populate one path using a simple `path` string and another path an `options` object.

```
const Group = mongoose.model('Group', Schema({
  name: String,
  leader: { type: mongoose.ObjectId, ref: 'Person' },
  people: [{ type: mongoose.ObjectId, ref: 'Person' }]
}));
const schema = Schema({ name: String, age: Number });
const Person = mongoose.model('Person', schema);

let mace = await Person.create({ name: 'Mace Windu', age: 53 });
let yoda = await Person.create({ name: 'Yoda', age: 900 });
let anakin = await Person.create({ name: 'Anakin Skywalker' });
const people = [mace, yoda, anakin];
await Group.create({ name: 'Jedi Order', leader: mace, people });

const match = { age: { $gte: 100 } };
const opts = ['leader', { path: 'people', match }];
const doc = await Group.findOne().populate(opts);
doc.leader.name; // 'Mace Windu'
doc.people.map(doc => doc.name); // ['Yoda']
```

### 5.3: Virtual Populate

So far, this chapter has only used *conventional populate*. Conventional populate is used to `populate()` real paths in the schema. Virtual populate is a slightly different form of `populate()` that instead operates on virtuals.

Virtual populate has several advantages over conventional populate:

- With conventional populate, `members` can be an array of ObjectIds or documents depending on whether it was populated. Virtual populate doesn't change the structure of the document.
- Checking whether a property is populated is easy: check if the virtual field is not nullish.
- Virtual populate can populate by different `foreignField` properties, whereas conventional populate always uses `_id` as the `foreignField`.

Here's how you can model the `Person` and `Group` models using virtual populate:

```
const Group = mongoose.model('Group', Schema({ name: String }));

const personSchema = Schema({
  name: String,
  groupId: mongoose.ObjectId
});
personSchema.virtual('group', {
  ref: 'Group',
  localField: 'groupId',
  foreignField: '_id',
  justOne: true
});
const Person = mongoose.model('Person', personSchema);
const groupId = await Group.create({ name: 'Jedi Order' });
await Person.create({ name: 'Luke Skywalker', groupId });

const person = await Person.findOne().populate('group');
person.group.name; // 'Jedi Order'
```

Using `foreignField`, you can also populate a `Group` document's `people` without changing the non-virtual fields:

```
const groupSchema = Schema({ name: String });
groupSchema.virtual('people', {
  ref: 'Person',
  localField: '_id',
  foreignField: 'groupId',
  justOne: false
});
const Group = mongoose.model('Group', groupSchema);
const schema = Schema({ name: String, groupId: 'ObjectId' });
const Person = mongoose.model('Person', schema);

const groupId = await Group.create({ name: 'Jedi Order' });
await Person.create({ name: 'Luke Skywalker', groupId });

const jedi = await Group.findOne().populate('people');
jedi.people.map(doc => doc.name); // ['Luke Skywalker']
```

Note that, with virtual populate, you call `populate()` on a virtual. That means that `Group._id` and `Person.groupId` remain ObjectIds when you call `populate('people')`. On the other hand, with conventional `populate()`, `Person.group` can be either an ObjectId or a Mongoose document, depending on whether you populated `group`.

### The `justOne` Option

The `justOne` option for virtuals tells Mongoose whether the populated virtual should be an array, or a single (potentially `null`) document.

```
const groupSchema = Schema({ name: String });

const ref = 'Person';
const localField = '_id';
const foreignField = 'groupId';
// If `justOne` is true, the populated virtual will be either
// a document, or `null` if no document was found.
const opts = { ref, localField, foreignField, justOne: true };
groupSchema.virtual('people', opts);

// If `justOne` is false, the populated virtual will be an
// array containing zero or more documents.
opts.justOne = false;
groupSchema.virtual('people', opts);
```

With conventional populate, Mongoose can infer `justOne` based on the type of the schema path:

```
let groupSchema = Schema({
  name: String,
  // Behaves like `justOne: false`
  people: [{ type: mongoose.ObjectId, ref: 'Person' }]
});

groupSchema = Schema({
  name: String,
  // Behaves like `justOne: true`
  people: { type: mongoose.ObjectId, ref: 'Person' }
});
```

If you set `justOne` to `true` and there are multiple linked documents, Mongoose will pick the first one according to MongoDB's natural sort order. To override the sort order, set the `options.sort`:

```
const groupSchema = Schema({ name: String });
groupSchema.virtual('person', {
  ref: 'Person',
  localField: '_id',
  foreignField: 'groupId',
  justOne: true
});
const Group = mongoose.model('Group', groupSchema);

const Person = mongoose.model('Person',
  Schema({ name: String, groupId: mongoose.ObjectId }));
const groupId = await Group.create({ name: 'Jedi Order' });
await Person.create({ name: 'Luke Skywalker', groupId });
await Person.create({ name: 'Obi-Wan Kenobi', groupId });

let jedi = await Group.findOne().populate({
  path: 'person',
  options: { sort: { name: 1 } }
});
jedi.person.name; // 'Luke Skywalker'
```

### The `count` Option

Usually `populate()` loads the referenced documents, but virtual populate supports an alternative. If you set the `count` option, the populated virtual will contain the number of referenced documents, rather than the documents themselves.

The names of virtual populate with `count` options are usually prefixed with `num`, like `numCities` below.

```
const countrySchema = Schema({ name: String });
countrySchema.virtual('numCities', {
  ref: 'City',
  localField: '_id',
  foreignField: 'countryId',
  count: true // `numCities` will be a number, not an array
});
const Country = mongoose.model('Country', countrySchema);
let citySchema = Schema({ name: String, countryId: 'ObjectId' });
const City = mongoose.model('City', citySchema);

let country = await Country.create({ name: 'Switzerland' });
const docs = [{ name: 'Bern' }, { name: 'Zurich' }].
  map(doc => Object.assign(doc, { countryId: country._id }));
await City.create(docs);

country = await Country.findOne().populate('numCities');
country.numCities; // 2
```

Note that `count` **only** works with virtual populate. There is no way to get just the `count` using conventional populate.

The `count` option makes several other populate options meaningless. If you set `count` to true, Mongoose will ignore the `justOne`, `select`, `sort`, and `limit` options. The only populate option that is useful with `count` is `match`: you can count referenced documents that match a given filter.

The `count` option is primarily useful for performance reasons: if there are a lot of referenced documents, sending all those documents over the wire from the MongoDB server to your app can take a long time. `count` can help you avoid sending all the referenced documents over the wire and increase performance by reducing bandwidth usage.

## 5.4: One-To-One, One-To-Many, Many-To-Many

There are 3 types of relationships between related models `A` and `B`:

#### One-to-One

Each document in `A` refers to at most one document in `B`, and each document in `B` refers to at most one document in `A`. For example, there is a one-to-one relationship between a `Country` and its capital `City`.

```
const Country = mongoose.model('Country', Schema({
  name: String,
  capital: { type: 'ObjectId', ref: 'City' }
}));
const City = mongoose.model('City', Schema({ name: String }));

const dc = await City.create({ name: 'Washington, D.C.' });
const manila = await City.create({ name: 'Manila' });
const [{ name }] = await Country.create([
  { name: 'United States', capital: dc },
  { name: 'Phillipines', capital: manila }
]);

const usa = await Country.findOne({ name }).populate('capital');
usa.capital.name; // 'Washington, D.C.'
```

In Mongoose, a one-to-one relationship is typically represented as a property on one side of the relationship. In the above example, the relationship between `Country` and `City` is stored as an ObjectId property `capital` on `Country` documents. The reason why the property is on `Country` versus on `City` is readability: `Country.capital` is easier to read than `City.capitalOf`.

When using virtual populate, you should suffix the relationship property with `Id`. For example, you can store the relationship in a `capitalId` property, and then define a virtual `capital`.

```
const schema = Schema({ name: String, capitalId: ObjectId });
schema.virtual('capital', {
  ref: 'City',
  localField: 'capitalId',
  foreignField: '_id',
  justOne: true
});
const Country = mongoose.model('Country', schema);
const City = mongoose.model('City', Schema({ name: String }));

const oslo = await City.create({ name: 'Oslo' });
const bern = await City.create({ name: 'Bern' });
await Country.create({ name: 'Norway', capitalId: oslo });
await Country.create({ name: 'Switzerland', capitalId: bern });

let v = await Country.findOne({ name: 'Norway' }).
  populate('capital');
v.capital.name; // 'Oslo'
```

#### One-to-Many

Each document in `A` refers to at most one document in `B`, but each document in `B` may refer to multiple documents in `A`. For example, there is a one-to-many relationship between a `Country` documents and the `City` documents that are part of the `Country`.

```
const countrySchema = Schema({ name: String });
const Country = mongoose.model('Country', countrySchema);
const City = mongoose.model('City', Schema({
  name: String,
  country: { type: mongoose.ObjectId, ref: 'Country' }
}));

const country = await Country.create({ name: 'United States' });
const { name } = await City.create({ name: 'NYC', country });
await City.create({ name: 'Miami', country });

const nyc = await City.findOne({ name }).populate('country');
nyc.country.name; // 'United States'
```

What if, instead of wanting to populate a city's `country`, you want to populate all the `cities` in a given country? You need to use virtual populate:

```
const countrySchema = Schema({ name: String });
countrySchema.virtual('cities', {
  ref: 'City',
  localField: '_id',
  foreignField: 'country',
  justOne: false
});
const Country = mongoose.model('Country', countrySchema);
const schema = Schema({ name: String, country: 'ObjectId' });
const City = mongoose.model('City', schema);

let usa = await Country.create({ name: 'United States' });
let canada = await Country.create({ name: 'Canada' });
await City.create({ name: 'New York', country: usa });
await City.create({ name: 'Miami', country: usa });
await City.create({ name: 'Vancouver', country: canada });

usa = await Country.findById(usa._id).populate('cities');
usa.cities.map(city => city.name); // ['New York', 'Miami']
```

The `Country` model is called the "one" side of the relationship, and the `City` model is called the "many" side of the relationship.For example, you can also represent the relationship between `Country` and `City` using conventional populate by storing a list of `cities` by id:

```
const countrySchema = Schema({
  name: String,
  // You can also represent one-to-many as an array of ObjectIds
  cities: [{ type: ObjectId, ref: 'City' }]
});
const Country = mongoose.model('Country', countrySchema);
const City = mongoose.model('City', Schema({ name: String }));
```

You can get away with representing a one-to-many relationship by storing an array of ids on the "one" side of the relationship. But, for performance reasons, you are usually better off representing a one-to-many relationship by storing a single id on the "many" side of the relationship. You will learn more about why in the section on the Principle of Least Cardinality.

#### Many-to-Many

Each document in `A` may refer to multiple documents in `B`, and each document in `B` may refer to multiple documents in `A`. For example, there is a many-to-many relationship between `Person` documents and `Group` documents: one person may be part of many groups, and a group may have many `members`.

```
const Group = mongoose.model('Group', Schema({
  name: String,
  members: [{ type: mongoose.ObjectId, ref: 'Person' }]
}));
const Person = mongoose.model('Person', Schema({ name: String }));

const luke = await Person.create({ name: 'Luke Skywalker' });
const han = await Person.create({ name: 'Han Solo' });
// Each `Group` has multiple members, and the Luke Skywalker
// document is part of multiple groups.
await Group.create({ name: 'Jedi Order', members: [luke] });
const name = 'Rebel Alliance';
await Group.create({ name, members: [luke, han] });

const group = await Group.findOne({ name }).populate('members');
group.members[0].name; // 'Luke Skywalker'
group.members[1].name; // 'Han Solo'
```

To populate the other side of a many-to-many relationship, you should use a populate virtual. For example, if you want to populate a `Person` document's `groups`, you can use a virtual as shown below.

```
const Group = mongoose.model('Group', Schema({
  name: String,
  members: [{ type: mongoose.ObjectId, ref: 'Person' }]
}));
const personSchema = Schema({ name: String });
personSchema.virtual('groups', {
  ref: 'Group',
  localField: '_id',
  // `populate()` is smart enough to drill into `foreignField` if
  // `foreignField` is an array
  foreignField: 'members'
})
const Person = mongoose.model('Person', personSchema);

const doc = await Person.findOne({ name: 'Luke Skywalker' }).
  populate({ path: 'groups', options: { sort: { name: 1 } } });
doc.groups[0].name; // 'Jedi Order'
doc.groups[1].name; // 'Rebel Alliance'
```

You can represent most many-to-many relationships using an array on one side of the many-to-many. However, MongoDB limits documents to 16 MB in size, and documents that are anywhere near 16 MB are cumbersome. For example, it takes over 10 seconds to download 16 MB on a 3G connection, so documents that big aren't suitable for web app backends.

For example, suppose you're looking to implement a Twitter-like social network, where each user can follow tens of thousands of other users, and popular users can have millions of followers.

```
const userSchema = Schema({
  // Won't work well if a user has millions of followers - might
  // run into the 16 MB document size limit.
  followers: [{
    type: mongoose.ObjectId,
    ref: 'User'
  }]
});
```

The solution to this problem is to use a mapping collection, analagous to an SQL mapping table. Mongoose supports populating recursively (also known as *deep populate*), so you can populate a user's followers:

```
// `Follow` represents `follower` following `followee`.
// No risk of `Follow` documents growing to 16MB.
const Follow = mongoose.model('Follow', Schema({
  follower: { type: ObjectId, ref: 'User' },
  followee: { type: ObjectId, ref: 'User' }
}));
const userSchema = Schema({ name: String });
userSchema.virtual('followers', {
  ref: 'Follow',
  localField: '_id',
  foreignField: 'followee'
});
const User = mongoose.model('User', userSchema);

const user1 = await User.create({ name: 'Mark Hamill' });
const user2 = await User.create({ name: 'John Smith' });
const user3 = await User.create({ name: 'Mike Jackson' });
await Follow.create({ follower: user2, followee: user1 });
await Follow.create({ follower: user3, followee: user1 });

// Find all of Mark Hamill's followers by populating
// `followers` and then `followers.follower`
const populate = { path: 'follower' };
const opts = { path: 'followers', populate };
let doc = await User.findOne({ name: /Hamill/ }).populate(opts);

doc.followers[0].follower.name; // 'John Smith'
doc.followers[1].follower.name; // 'Mike Jackson'
```

### 5.5 Deep Populate

Deep populate is how you can populate documents that are related via an intermediate relationship. For example, suppose you have 3 models: `Product`, `Category`, and `Order`. Each order contains one or more products, and each product belongs to one or more categories. In other words, `Order` is related to `Category` via `Product`.

```
// One-to-many relationships: Order - Product, Product - Category
const Order = mongoose.model('Order', Schema({
  products: [{ type: mongoose.ObjectId, ref: 'Product' }]
}));
const Product = mongoose.model('Product', Schema({
  name: String,
  categories: [{ type: mongoose.ObjectId, ref: 'Category' }]
}));
const categorySchema = Schema({ name: String });
const Category = mongoose.model('Category', categorySchema);

// Create sample documents
const phones = await Category.create({ name: 'Phones' });
const books = await Category.create({ name: 'Books' });
const [iphone, book] = await Product.create([
  { name: 'iPhone', categories: [phones] },
  { name: 'Snow Crash', categories: [books] }
]);
await Order.create({ products: [iphone, book] });
await Order.create({ products: [book] });

// Deep populate `products` and `products.categories`
const orders = await Order.find().sort({ name: -1 }).populate({
  path: 'products',
  // The `populate` option populates each product's categories
  populate: { path: 'categories' }
});
orders[0].products[0].categories[0].names; // 'Phones'
```

Deep populate also works when populating the same model. For example, suppose you have a `User` model and each user has an array of `friends` that point to other `User` documents. Deep populate lets you populate friends of friends. Below is the `User` model:

```
const User = mongoose.model('User', Schema({
  name: String,
  friends: [{ type: mongoose.ObjectId, ref: 'User' }]
}));
```

Here's how you can populate a user's friends of friends.

```
const luke = await User.create({ name: 'Luke Skywalker' });
const yoda = await User.create({ name: 'Yoda', friends: [luke] });
await User.create({ name: 'Mace Windu', friends: [yoda] });

// Populate Mace Windu's friends of friends
const doc = await User.findOne({ name: 'Mace Windu' }).populate({
  path: 'friends',
  populate: { path: 'friends' }
});
doc.friends[0].friends[0].name; // 'Luke Skywalker'
```

There's no limit to how many levels you can populate. For example, you can also populate friends of friends of friends by nesting `populate` twice:

```
const han = await User.create({ name: 'Han Solo' });
const luke = await User.create({ name: 'Luke', friends: [han] });
const yoda = await User.create({ name: 'Yoda', friends: [luke] });
await User.create({ name: 'Mace Windu', friends: [yoda] });

// Populate Mace Windu's friends of friends of friends
const path = 'friends';
const doc = await User.findOne({ name: 'Mace Windu' }).populate({
  path,
  populate: { path, populate: { path } }
});
doc.friends[0].friends[0].friends[0].name; // 'Han Solo'
```

## 5.6: Manual Population

When you call `populate()`, Mongoose executes a query under the hood to fetch the referenced documents. However, if you already have the referenced document, you can *manually populate* the property by setting it to the referenced document. For example, suppose you have 2 models: `Country` and `City`.

```
const countrySchema = Schema({
  name: String,
  capital: { type: 'ObjectId', ref: 'City' }
});
const Country = mongoose.model('Country', countrySchema);
const City = mongoose.model('City', Schema({ name: String }));
```

You can set a `Country` document's capital to `City` document, and Mongoose will treat the `capital` as a populated path.

```
const country = await Country.create({ name: 'Switzerland' });
country.capital = new City('Bern');

country.capital.name; // 'Bern'
!!country.populated('capital'); // true
```

The `Document#populated()` function is how you check if a path is populated. If calling `populated(path)` function returns a truthy value, that means `path` is populated. In the above example, setting the city's `capital` to a `City` document marks the `capital` path as populated.

You can also set an array of refs to an array of documents, so long as *all* the documents are instances of the referenced model.

```
const Country = mongoose.model('Country', Schema({
  name: String,
  cities: [{ type: mongoose.ObjectId, ref: 'City' }]
}));
const City = mongoose.model('City', Schema({ name: String }));

const country = await Country.create({ name: 'Switzerland' });
country.cities = [new City('Bern'), new City('Basel')];
country.cities[0].name; // 'Bern'
!!country.populated('cities'); // true
```

However, if you set an array of refs and one document isn't a `City`, Mongoose will depopulate the entire array.

```
const country = await Country.create({ name: 'Switzerland' });
const city = await City.create({ name: 'Bern' });
// The 2nd doc isn't a city, so Mongoose depopulates the array.
country.cities = [city, country];

country.cities[0].name; // undefined
!!country.populated('cities'); // false
```

## 5.7: Populating Across Databases

Many people ask why they should use `populate()` as opposed to MongoDB's `$lookup` aggregation operator, which also can load referenced documents. One reason is that `populate()`

has fewer restrictions: `$lookup` doesn't support loading referenced documents from different databases, or from sharded collections.

What does it mean to populate a document from a different database? Remember that each Mongoose model is associated with exactly one Mongoose connection via the `Model#db` property. Each Mongoose connection points to exactly one MongoDB database - in MongoDB, a *database* is a container for collections.

In the below example, `Model1` is in database `db1`, and `Model2` is in database `db2`. But `populate()` is able to pull a document referenced in `Model2` from `db1`. The key detail is that `ref` is a model, not a string.

```
const host = 'mongodb://localhost:27017';
const db1 = await mongoose.createConnection(`${host}/db1`);
const db2 = await mongoose.createConnection(`${host}/db2`);

const M1 = db1.model('Test', Schema({ name: String }));
// Note that `ref` below is a **Model**, not a string!
const M2 = db2.model('Test', Schema({
  name: String,
  doc: { type: mongoose.ObjectId, ref: Model1 }
}));

const doc1 = await M1.create({ name: 'model 1' });
await M2.create({ name: 'model 2', doc: doc1._id });

const doc2 = await M2.findOne().populate('doc');
doc2.doc.name; // model 1
```

If you set `ref` to a model, Mongoose will use that model as the foreign model. That model could be associated with an entirely different Mongoose connection. In the above example the models are connected to different databases, but they could also be connected to different MongoDB servers.

## 5.8: Summary

Mongoose queries and documents have a `populate()` function that you can use to load referenced documents from different models. You can either `populate()` a real property in your schema (conventional populate), or you can `populate()` a virtual property in your schema (virtual populate). Virtual populate is more flexible and avoids mutating the document, but conventional populate is simpler.

When a property is populated, it is either `null` or contains one or more Mongoose documents. With conventional populate, Mongoose will populate a single, potentially `null`, document or an

array of documents depending on whether the path is a primitive or an array in your schema:

```
const schema = Schema({
  // If populated, `user` will either be a document or `null`
  user: { type: mongoose.ObjectId, ref: 'User' }
  // If `populated`, `friends` will be an array of documents
  friends: [{ type: mongoose.ObjectId, ref: 'User' }]
});
```

With virtual populate, Mongoose will populate a single, potentially `null`, document or an array of documents depending on the value of the `justOne` option:

```
const groupSchema = Schema({
  leaderId: mongoose.ObjectId,
  memberIds: [mongoose.ObjectId]
});

const ref = 'Person';
const foreignField = '_id';
// If populated, `leader` will be a document, or `null` if no
// document was found, because `justOne` is true.
groupSchema.virtual('leader',
  { ref, localField: 'leaderId', foreignField, justOne: true });

// If populated, `members` will be an array of zero or more
// documents, because `justOne` is false.
groupSchema.virtual('members',
  { ref, localField: 'memberIds', foreignField, justOne: false });
```

When you `save()` a document with populated paths, Mongoose first depopulates the document before saving. So even if you use conventional populate, you don't have to worry about `populate()` changing the data stored in the database.

`populate()` is a powerful tool, but it comes with performance implications. If you are used to using SQL, it is tempting to build your MongoDB documents using third normal form and use `populate()` to dereference all of those relationships. However, abusing `populate()` can lead to lackluster performance and confusing queries.

With Mongoose, you can use denormalization as a way to simplify and speed up your queries. You'll learn more about when to denormalize versus when to `populate()` in the next chapter.

# 6: Schema Design and Performance

The ability to `populate()` referenced documents introduces a key issue in schema design: what should you store in a separate collection versus what should you store in the same collection?

Before MongoDB and similar NoSQL databases became popular, schema design came down to third normal form: "every non-key must provide a fact about the key, the whole key, and nothing but the key." 3rd normal form told you how to structure your database schemas, but not how to make your schema performant or easy to query.

Schema design in Mongoose is more nuanced. In this chapter, you'll learn several schema design principles that will help you avoid performance bottlenecks. These principles are solid guidelines to follow for getting consistent performance from Mongoose. Keep in mind that these principles are *guidelines*, not iron clad laws.

First up is the Principle of Least Cardinality.

## 6.1 Principle of Least Cardinality

The Principle of Last Cardinality is a rule of thumb for how to store one-to-many and many-to-many relationships. For example, suppose you have 2 models: `Person` and `Rank`. Each person has exactly one rank, but multiple people can have the same rank.

There are several ways to represent the relationship between `Person` and `Rank`. One way is to store a `rankId` property on the `Person` model:

```
const schema = Schema({ name: String, rankId: ObjectId });
schema.virtual('rank', {
  ref: 'Rank',
  localField: 'rankId',
  foreignField: '_id',
  justOne: true
});
const Person = mongoose.model('Person', schema);
const Rank = mongoose.model('Rank', Schema({ name: String }));

const rank1 = await Rank.create({ name: 'Captain' });
const rank2 = await Rank.create({ name: 'Lieutenant' });

await Person.create({ name: 'James Kirk', rankId: rank1 });
await Person.create({ name: 'Hikaru Sulu', rankId: rank2 });

let doc = await Person.findOne({ name: /Kirk/ }).populate('rank');
doc.rank.name; // 'Captain'
```

Another way is to represent the relationship as an array `peopleIds` on the `Rank` model. Virtual populate makes it easy to express the relationship between a person and their `rank` using either schema structure:

```
const schema = Schema({ name: String });
schema.virtual('rank', {
  ref: 'Rank',
  localField: '_id',
  foreignField: 'peopleIds',
  justOne: true
});
const Person = mongoose.model('Person', schema);

const rankSchema = Schema({ name: String, peopleIds: [ObjectId] });
const Rank = mongoose.model('Rank', rankSchema);

let names = ['James T. Kirk', 'Uhura', 'Hikaru Sulu'];
let p = await Person.create(names.map(name => ({ name })));
await Rank.create({ name: 'Captain', peopleIds: [p[0]] });
await Rank.create({ name: 'Lieutenant', peopleIds: [p[1], p[2]] });

let doc = await Person.findOne({ name: /Kirk/ }).populate('rank');
doc.rank.name; // 'Captain'
```

Which approach should you use? The Principle of Least Cardinality says: *store relationships in a way that minimizes the size of individual documents*. Small documents are easier to work with: they take up less memory and are sending them over a network is faster.

If you store the `rank` relationship using an array `peopleIds` on the `Rank` document, an individual `Rank` document can grow massive. With tens of thousands of captains, you'll see performance degradation when you load an individual `Rank` document.

The situation is even worse if you try to `populate()` several `Person` documents' `rank` property at once. **Every** person's `rank` will contain the `_id` of every person with that rank.

```
const docs = await Person.find().sort({ name: 1 }).populate('rank');

// Each `Person` has a `rank` document, and each `rank` document
// stores `peopleIds`. If you have thousands of captains, this data
// would be too bulky to send to a client.
docs[0].rank.peopleIds.length; // 1
docs[1].rank.peopleIds.length; // 2
docs[2].rank.peopleIds.length; // 2
```

Therefore, the correct way to store the `rank` relationship is by storing a `rankId` property on the `Person` document. In general, the Principle of Least Cardinality implies that, for a one-to-many relationship, you should **always** represent the one-to-many relationship with a single `id` property on the "many side" of the relationship.

Given two models `A` and `B`, where one `B` document has many `A` documents but one `A` document has at most one `B` document, you should store the relationship like this:

```
const schema = Schema({ name: String, bId: ObjectId });
schema.virtual('b', {
  ref: 'B',
  localField: 'bId',
  foreignField: '_id',
  justOne: true
});
const A = mongoose.model('A', schema);

const B = mongoose.model('B', Schema({ name: String }));
```

The Principle of Least Cardinality is partially because of the fact that MongoDB limits documents to 16MB in size. If you have an array that rapidly grows without bound, you may end up with a 16MB document. But primarily, the Principle of Least Cardinality is about conserving bandwidth when loading documents.

Network bandwidth between your app server and your MongoDB server is a scarce resource, and you should conserve it judiciously. Projections are helpful, but it is also very easy to forget projections when writing new code. It is better to structure your schemas to keep your documents small, so a missing projection doesn't make bandwidth a bottleneck for your app.

#### Many-To-Many Relationships

The Principle of Least Cardinality also applies to many-to-many relationships. To represent a many-to-many relationship, you should store the relationship on the side that has the smaller "many".

For example, suppose you have two models: `Character` and `Show`. Each character can appear in multiple shows, and each show has multiple characters. This is a many-to-many relationship.

Should you store a list of `showIds` on the `Character` model or a list of `characterIds` on the `Show` model? It depends on the structure of the data.

In the case of television shows, the same character rarely appears on multiple shows, and it is unheard of for one character to appear on thousands of shows. On the other hand, a show may have hundreds of characters. Given these assumptions, the Principle of Least Cardinality says to

store a list of `showIds` on the `Character` model, because that minimizes the likelihood of huge arrays.

```
const schema = Schema({ name: String, showIds: [ObjectId] });
schema.virtual('shows', {
  ref: 'Show',
  localField: 'showIds',
  foreignField: '_id',
  justOne: false
});
const Character = mongoose.model('Character', schema);
const Show = mongoose.model('Show', Schema({ name: String }));

const shows = await Show.create([
  { name: 'Star Trek' },
  { name: 'Star Trek: The Next Generation' }
]);

await Character.create([
  { name: 'James T. Kirk', showIds: [shows[0]] },
  { name: 'Leonard McCoy', showIds: [shows[0], shows[1]] }
]);

let v = await Character.findOne({ name: /McCoy/ }).populate('shows');
v.shows[0].name; // 'Star Trek'
v.shows[1].name; // 'Star Trek: The Next Generation'
```

### Mapping Collections

Sometimes it is impossible to represent a many-to-many relationship via an array on one side of the relationship, because both sides can grow without bound. One example is Twitter followers - a user can have millions of followers, and a user can follow hundreds of thousands of people.

Another example is users and events. Say you're building a clone of Meetup.com. Thousands of users may attend a given `Event`, and a single `User` may attend thousands of events. Furthermore, as time goes on, the number of events a user has attended may grow without bound.

To work around cases where both sides of the many-to-many may grow without bound, you can use a *mapping collection*. A mapping collection stores one document for each mapping between a `User` and an `Event`. Mapping collections are analagous to mapping tables in SQL. To dereference a mapping collection, you need to use deep populate:

```
const userSchema = Schema({ name: String });
userSchema.virtual('attended', {
  ref: 'Attendee',
  localField: '_id',
  foreignField: 'user',
  justOne: false,
  // Recursively populate the `Attendee` model's 'event'
  options: { populate: 'event' }
});
const User = mongoose.model('User', userSchema);
const Event = mongoose.model('Event', Schema({ name: String }));
// A mapping collection: 1 doc per mapping from `Person` to `Event`
const Attendee = mongoose.model('Attendee', Schema({
  user: { type: ObjectId, ref: 'User' },
  event: { type: ObjectId, ref: 'Event' }
}));

const e1 = await Event.create({ name: 'Khitomer Conference' });
const e2 = await Event.create({ name: 'Enterprise-B Maiden Voyage' });
const users = await User.create([{ name: 'Kirk' }, { name: 'Spock' }]);

await Attendee.create({ event: e1, user: users[0] });
await Attendee.create({ event: e1, user: users[1] });
await Attendee.create({ event: e2, user: users[0] });

let doc = await User.findOne({ name: 'Kirk' }).populate('attended');
doc.attended[0].event.name; // 'Khitomer Conference'
doc.attended[1].event.name; // 'Enterprise-B Maiden Voyage'
```

You should avoid using mapping collections if possible. Executing an additional query to dereference the mapping collection before you can dereference the actual relationship is slower than storing an array on one of the related models. Using a mapping collection is generally better than having an array that grows without bound.

## 6.2: Principle of Denormalization

The Principle of Least Cardinality deals with how you should structure relationships between documents so you can dereference relationships using `populate()` without letting your documents grow too big. The Principle of Denormalization helps determine whether you need to use `populate()` at all.

Consider the example of the relationship between `Character` and `Rank`. In SQL, third normal form dictates that `Character` and `Rank` should be separate models. If you store that a character's

`rank` is the string 'Captain', what happens if you want to update the 'Captain' rank to instead be the string 'Director'? If you store `rank` as a string, you would have to update every character document.

```
// If you want to change the rank 'Captain' to be 'Director',
// you would need to update every `Character` document.
const schema = Schema({ name: String, rank: String });
const Character = mongoose.model('Character', schema);

await Character.updateMany({ rank: 'Captain' }, { rank: 'Director' });
```

But the question is, how often do you want to change a rank's name? Suppose you're building an app similar to IMDb. The original Star Trek series character Captain Kirk will always be Captain Kirk, so the likelihood that you will want to update every character's rank at once is small.

On the other hand, it would be very valuable to allow searching for characters based on their `rank`. And, unfortunately, if `Rank` is a separate model, Mongoose `populate()` does **not** let you filter by `rank.name`.

```
const User = mongoose.model('User', Schema({
  name: String,
  rank: { type: ObjectId, ref: 'Rank' }
}));
const Rank = mongoose.model('Rank', Schema({ name: String }));

const captain = await Rank.create({ name: 'Captain' });
const commander = await Rank.create({ name: 'Commander' });
await User.create({ name: 'James T. Kirk', rank: captain });
await User.create({ name: 'Spock', rank: commander })

// The `match` option only affects the populated documents.
// This query says to find all users, sort them by name,
// and then populate their `rank` only if `name = 'Captain'`
const docs = await User.find().sort({ name: 1 }).populate({
  path: 'rank',
  match: { name: 'Captain' }
});

docs.length; // 2
docs[1].name; // Spock
docs[1].rank; // null
```

The Principle of Denormalization says that *a document should store all the properties you want to query by*. An alternative phrasing is simply "store what you query for." *Denormalization* means breaking 3rd Normal Form and storing the same property on multiple models.

The Principle of Denormalization exists because querying by a property that's embedded in the document is relatively fast, and you can optimize these queries via indexes. Also, saving an extra `populate()` makes your queries faster. In the case of `Character` and `Rank`, the Principle of Denormalization says to embed the `rank` as a string in the `Character` model.

The Principle of Denormalization encourages you to embed data into a document where possible, even if you need to duplicate the data. For example, suppose you have a `User` model and a `Car` model, with a one-to-many relationship from users to cars.

Each car has a `licensePlate` property. Suppose you want to search for users by their car's license plate in addition to searching by user properties like `name`.

```
const User = mongoose.model('User', Schema({ name: String }));
const Car = mongoose.model('Car', Schema({
  description: String,
  licensePlate: String,
  owner: { type: ObjectId, ref: 'User' }
}));

// Find users whose name contains 'Crockett'. What if you also
// want to filter by users that have a car whose license plate
// starts with 'ZAQ'?
await User.find({ name: /Crockett/ });
```

The Principle of Denormalization implies that, if your app requires users to quickly search users by `licensePlate`, you should store a `licensePlates` on the `User` model. You can rely on Mongoose middleware to keep the `licensePlates` property in sync.

```
const userSchema = Schema({ name: String, licensePlates: [String] });
const carSchema = Schema({
  licensePlate: String,
  owner: { type: ObjectId, ref: 'User' }
});
// Use middleware to update the corresponding user's
// license plates whenever we update a vehicle.
carSchema.post('save', async function() {
  const allCars = await Car.find({ owner: this.owner });
  const licensePlates = allCars.map(car => car.licensePlate);
  await User.updateOne({ _id: this.owner }, { licensePlates });
});
const User = mongoose.model('User', userSchema);
const Car = mongoose.model('Car', carSchema);

const owner = await User.create({ name: 'Sonny Crockett' });
await User.create({ name: 'Davy Crockett' });
await Car.create({ licensePlate: 'ZAQ178', owner });

let doc = await User.findOne({ licensePlates: /ZAQ/ });
doc.name; // 'Sonny Crockett'
```

The key idea behind denormalization is that you can make your most common operations (like queries by `licensePlate`) fast at the cost of making rare operations (updating your `licensePlate`) slower. If querying by `licensePlate` is not a common task, you shouldn't denormalize.

One key philosophical difference between the Principle of Denormalization and 3rd Normal Form is that 3rd Normal Form prescribes how you should store your data regardless of how you use your data. The Principle of Denormalization flips the script and encourages you to structure your data based on how you use it.

## 6.3: Principle of Data Locality

The Principle of Denormalization tells you that each document should store the properties you want to query by in order to make queries faster. However, that's not the only reason why you might want to embed some or all of a document's properties in another document.

The idea of *data locality* means keeping data that is commonly accessed together in the same document. If you find yourself always needing to `populate()` a certain property, that property is a good candidate for denormalization. The Principle of Data Locality says that *a document should contain all data necessary to display a web page for that document*.

One example is a standard blog. I think it is no coincidence that NoSQL databases became popular shortly after the rise of blogging. In order to display a `BlogPost`, you also need to load the blog post's `author`, `tags`, and `comments`. In SQL, rendering a single blog post requires way too many joins. The result is unreadable queries and rapidly decaying performance.

In MongoDB, the right way is to store all the data related to a blog post in one document.

```
const blogPostSchema = Schema({
  title: String,
  content: String,
  authorId: String,
  // Denormalize all the details to render the blog post,
  // avoiding extra `populate()` calls.
  author: userSchema.pick(['name', 'email']),
  tags: [String]
});
```

The `Schema#pick()` function is designed to make denormalization easier. Calling `userSchema.pick(['name', 'email'])` creates a new schema with the same `name` and `email` properties as `userSchema`: same types, same validators, etc. This lets you embed a few properties, and ignore the remaining irrelevant properties.

In Mongoose, you are responsible for keeping denormalized properties up to date. However, Mongoose middleware makes it fairly easy to keep embedded properties consistent. For example, the below code demonstrates how to update all blog posts when you `save()` a user.

```
const schema = Schema({ name: String, email: String, age: Number });
// When updating a user, update all blog posts and comments.
schema.post('save', async function() {
  const update = { $set: { author: this } };
  await BlogPost.updateMany({ authorId: this._id }, update);
});
const User = mongoose.model('User', schema);

// `BlogPost.author` embeds information from 'User'
const BlogPost = mongoose.model('BlogPost', Schema({
  title: String,
  content: String,
  authorId: mongoose.ObjectId,
  author: schema.pick(['name', 'email']),
  tags: [String]
}));
```

Sometimes, you don't want to update embedded data. For example, it is a reasonable design decision for a comment to store the user's `name` and `email` as they were when the user posted it. That way, if a blog admin is moderating comments, they don't have to worry about a user changing their `name` to something offensive after the comment was approved. If you choose to go that route, you don't even need to update blog posts when a `User` is updated.

Intentionally keeping embedded data when the standalone document changes is a more useful design pattern than many 3rd normal form advocates think. Ensuring that malicious users can't update comments after moderation is one use case. Another is storing a snapshot of data as it was when the user performed an action.

Suppose you're building a parking app. Each time a user parks, you embed the vehicle's license plate so law enforcement can verify the owner of the vehicle has paid.

```
const vehicleSchema = Schema({
  make: String,
  model: String,
  plate: String,
  ownerId: mongoose.ObjectId
});
const parkingSchema = Schema({
  endsAt: Date,
  vehicleId: mongoose.ObjectId,
  vehicle: vehicleSchema.pick(['plate'])
});
```

Now suppose your app has been around for several years, and your users start updating their vehicles. If a user changes their license plate today, should it affect their `Parking` documents from two years ago? No, because the vehicle that parked at that time had a particular license plate. Changing the `plate` two years later may be "correct" from a theoretical 3NF perspective, but in practice it is a data integrity issue.

## 6.4: Indexes

A database *index* is a way to make certain queries faster at the expense of making updates somewhat slower. For example, suppose you have a `User` model that has 10,000 documents. Without an index, MongoDB needs to scan all 10,000 documents every time you want to find all users whose `firstName` is `'John'`.

The term *collection scan* means a query where MongoDB needs to iterate through the entire collection. Whether a collection scan is a problem depends on the size of the collection and your use case. For example, if you create 120,000 user documents, a collection scan can take over 50 milliseconds depending on your hardware specs and how much load there is on the database.

```
const docs = [];
for (let i = 0; i < 120000; ++i) {
  docs.push({ firstName: 'Agent', lastName: 'Smith' });
}
docs.push({ firstName: 'John', lastName: 'Smith' });

const userSchema = Schema({ firstName: String, lastName: String });
const User = mongoose.model('User', userSchema);
await User.insertMany(docs);

const start = Date.now();
const res = await User.find({ firstName: 'John' });
const elapsed = Date.now() - start;

elapsed; // Approximately 104 on my laptop
```

Depending on your app, this may not be a problem. Maybe your app doesn't allow searching for users by name. Or maybe that is an admin-only feature that doesn't need to be fast. But if searching by name is a core feature for your app, you need to avoid collection scans.

**Rule of Thumb:** If your end users expect to be able to access the results of a query nearly instantaneously (50ms), your query should avoid performing a collection scan on models with more than 100,000 documents. If you have less than 10,000 documents, collection scans should not be a problem.

So how do you avoid a collection scan? The answer is to *build an index* on the `firstName` property.

```

// Mongoose builds indexes automatically. However, if you
// need to wait for indexes to finish building, you should
// wait for `User.init()` to finish.
await User.init();
await User.insertMany(docs);

const start = Date.now();
const res = await User.find({ firstName: 'John' });
const elapsed = Date.now() - start;

elapsed; // Approximately 14 on my laptop
```

## Defining Indexes

Most apps define indexes in their Mongoose schemas. You can define indexes in your schema definition using `index: true`:

```
// `schema` has 2 indexes: one on `name`, and one on `email`.
const schema = new Schema({
  name: { type: String, index: true },
  email: { type: String, index: true }
});
```

You can also define an index by using the `Schema#index()` function:

```
const schema = new Schema({ name: String, email: String });
// Add 2 separate indexes to `schema`
schema.index({ name: 1 });
schema.index({ email: 1 });
```

## Building Indexes

You can define indexes in your schema, but indexes live on the MongoDB server. In order to actually use an index, you need to *build the index* on the MongoDB server.

Mongoose automatically builds all indexes defined in your schema when you create a model:

```
const schema = new Schema({ name: String, email: String });
schema.index({ name: 1 });
schema.index({ email: 1 });
// Mongoose tries to build 2 indexes on the 'tests' collection. If
// the indexes already exist, Mongoose doesn't do anything.
const Model = mongoose.model('Test', schema);
```

When you create a new model, Mongoose automatically calls that model's `createIndexes()` function. You can disable automatic index builds using the `autoIndex` schema option:

```
const opts = { autoIndex: false }; // Disable auto index build
const schema = Schema({ name: { type: String, index: true } }, opts);
const Model = mongoose.model('Test', schema);
await Model.init();
// Does **not** have the index on `name`
const indexes = await Model.listIndexes();
```

You can also call `createIndexes()` yourself to build all the schema's indexes.

```
const opts = { autoIndex: false };
const schema = Schema({ name: { type: String, index: true } }, opts);
const Model = mongoose.model('Test', schema);

await Model.init();
let indexes = await Model.listIndexes();
indexes.length; // 1

await Model.createIndexes();
indexes = await Model.listIndexes();
indexes.length; // 2
```

Once an index is built, it remains on the MongoDB server forever, unless someone explicitly drops the index. Dropping a database or dropping a collection also drops all indexes. That means once your indexes are built, you usually don't have to worry about them again.

Although Mongoose automatically builds all indexes defined in your schema, Mongoose does **not** drop any existing indexes that aren't in your schema. You can use the `syncIndexes()` function to ensure that the indexes in MongoDB line up with the indexes in your schema:

```
let schema = Schema({ name: { type: String, index: true } });
let Model = mongoose.model('Test', schema);
await Model.init();

// Now suppose you change the property `name` to `fullName`.
// By default, Mongoose won't drop the `name` index
schema = Schema({ fullName: { type: String, index: true } });
mongoose.deleteModel('Test');
Model = mongoose.model('Test', schema);
await Model.init();

// There are now 3 indexes in the database: one on `_id`, one
// on `name`, and one on `fullName`.
let indexes = await Model.listIndexes();
indexes.length; // 3

// Calling `Model.syncIndexes()` builds all indexes in your schema
// and drops all indexes that are not in your schema.
await Model.syncIndexes();
// 2 indexes, one on `_id` and one on `fullName`
indexes = await Model.listIndexes();
indexes.length; // 2
```

## Compound Indexes

A *compound index* is an index on multiple properties. There are two ways to define indexes on multiple propertes. First, using `Schema#index()`:

```
let schema = Schema({ firstName: String, lastName: String });

// Define a compound index on { firstName, lastName }
schema.index({ firstName: 1, lastName: 1 });

const Model = mongoose.model('Test', schema);
await Model.init();

const indexes = await Model.listIndexes();
indexes.length; // 2
indexes[1].key; // { firstName: 1, lastName: 1 }
```

The alternative is to define 2 `SchemaType` paths in your schema with the same index `name`. Mongoose groups indexes with the same name into a single compound index.

```
let schema = Schema({
  firstName: {
    type: String,
    index: { name: 'firstNameLastName' }
  },
  lastName: {
    type: String,
    index: { name: 'firstNameLastName' }
  }
});

const indexes = schema.indexes();
indexes.length; // 1
indexes[0][0]; // { firstName: 1, lastName: 1 }
```

Why are compound indexes useful? Indexes aren't useful unless they're *specific* enough. You'll learn more about index specificity in the next section, but the intuition is that you want to minimize the number of documents that MongoDB needs to look through to answer your query.

The below example shows that indexes don't just magically make your queries fast. In degenerate cases, like below where every document has the same `firstName`, having a bad index can be worse than having no index at all.

```
let schema = Schema({ firstName: String, lastName: String });
// Querying by { firstName, lastName } will be slow, because
// there's only an index on `firstName` and every document
// has the same `firstName`.
schema.index({ firstName: 1 });

const User = mongoose.model('User', schema);
const docs = [];
for (let i = 0; i < 120000; ++i) {
  docs.push({ firstName: 'Agent', lastName: 'Smith' });
}
docs.push({ firstName: 'Agent', lastName: 'Brown' });
await User.insertMany(docs);

const start = Date.now();
let res = await User.find({ firstName: 'Agent', lastName: 'Brown' });
const elapsed = Date.now() - start;

// Approximately 315 on my laptop, 3x slower than if no index!
elapsed;
```

On the other hand, if you build a compound index on `{ firstName, lastName }`, MongoDB can find the 'Agent Brown' document almost instantaneously.

```
let schema = Schema({ firstName: String, lastName: String });
schema.index({ firstName: 1, lastName: 1 });

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

const start = Date.now();
let res = await User.find({ firstName: 'Agent', lastName: 'Brown' });
const elapsed = Date.now() - start;

elapsed; // Approximately 10 on my laptop
```

## Unique Indexes

A `unique` index means that MongoDB will throw an error if there are multiple documents with the same value for the indexed property. One benefit of unique indexes is that bad index specificity is impossible.

There are several ways to declare a `unique` index. First, you can set `unique: true` on a property in your schema definition:

```
let schema = Schema({
  email: {
    type: String,
    unique: true
  }
});
const User = mongoose.model('User', schema);
await User.init();

// Unique index means MongoDB throws an 'E11000 duplicate key
// error' if there are two documents with the same `email`.
const err = await User.create([
  { email: 'agent.smith@source.com' },
  { email: 'agent.smith@source.com' }
]).catch(err => err);

err.message; // 'E11000 duplicate key error...'
```

You can also set the `unique` option to `true` when calling `Schema.index()`. This lets you define a compound unique index:

```
let schema = Schema({ firstName: String, lastName: String });
// A compound unique index on { firstName, lastName }
schema.index({ firstName: 1, lastName: 1 }, { unique: true });

const indexes = schema.indexes();
indexes.length; // 1
indexes[0][0]; // { firstName: 1, lastName: 1 }
indexes[0][1].unique; // true
```

An important note about compound unique indexes: in the above example, there may be duplicate `firstName` and `lastName`, but the combination of `firstName` and `lastName` must be unique.

For example, there can only be at most one document that matches `{ firstName: 'Agent', lastName: 'Smith' }`. But there can be many documents with `firstName` 'Agent' **or** `lastName` 'Smith'.

## The `_id` Index

MongoDB automatically creates an index on `_id` whenever it creates a new collection. This index is `unique` under the hood. Unfortunately, the `listIndexes()` function doesn't report the `_id` index as unique: this is a known quirk with MongoDB.

```
const User = mongoose.model('User', Schema({ name: String }));
await User.createCollection();

// MongoDB always creates an index on `_id`. Even though
// `listIndexes()` doesn't say that the `_id` index is unique,
// the `_id` index **is** a unique index.
const indexes = await User.listIndexes();
indexes.length; // 1
indexes[0].key; // { _id: 1 }
indexes[0].unique; // undefined

// Try to create 2 users with the exact same `_id`
const _id = new mongoose.Types.ObjectId();
const users = [{ _id }, { _id }];
const err = await User.create(users).catch(err => err);

err.message; // 'E11000 duplicate key error...'
```

The `_id` index ensures that `Model.findOne({ _id })` and `Model.findById(id)` are almost always fast queries. If those are slow, that is usually indicative of a different problem that you'll learn about in the slow trains section.

You **cannot** drop the `_id` index or change it in any way.

## Indexes and the Principle of Denormalization

The Principle of Denormalization is doubly important when it comes to indexes. You can only build indexes on a property in that model! For example, suppose you have 2 models: `User` and `Group`.

```
const User = mongoose.model('User', Schema({
  name: String,
  group: { type: mongoose.ObjectId, ref: 'Group' }
}));
const Group = mongoose.model('Group', Schema({ name: String }));
```

You can build an index on `User.group` to enable fast queries by `group`. But building an index on `Group.name` won't help you enable fast queries for users by group name. Remember that when you call `populate('group')`, Mongoose executes a separate query for groups **after** first querying for users.

```
// Finds all users, and then populates the user's `group` if the
// group's name is 'Jedi'. Even if you have an index on `Group.name`,
// this query will run a collection scan on `User`.
User.find().populate({ path: 'group', match: { name: 'Jedi' } });
```

If you want to enable fast queries for users by their group name, you should embed the group name in the `User` model and build an index on `User.groupName`.

```
const User = mongoose.model('User', Schema({
  name: String,
  groupName: { type: String, index: true },
  group: { type: mongoose.ObjectId, ref: 'Group' }
}));

const Group = mongoose.model('Group', Schema({ name: String }));
```

## 6.5: Index Specificity and `explain()`

The previous section explained how to define an index, but glossed over how to check what index MongoDB is using and how to evaluate the effectiveness of an index. This section will cover the notion of index specificity and how to read the output of MongoDB's `explain()` helper, which tells you what index MongoDB used to search the collection.

### The `explain()` Helper

Mongoose queries have a `Query#explain()` helper that tells MongoDB to return stats and information about how it executed the query, rather than the results of the query.

When you execute a query with `.explain()`, you get back a complex, deeply nested object that describes how MongoDB executed the query. The `explain` object may look baffling at first, but you only need to look at a few key properties to figure out what index MongoDB is using.

```
let schema = Schema({ firstName: String, lastName: String });
schema.index({ firstName: 1, lastName: 1 });
const User = mongoose.model('User', schema);

const firstName = 'Agent';
const lastName = 'Smith';
const res = await User.findOne({ firstName, lastName }).explain();

// Object with properties like `queryPlanner` & `executionStats`
res;
```

Below is the full value of the `res.queryPlanner` object from the above example.

```
queryPlanner:
   { plannerVersion: 1,
     namespace: 'test.users',
     indexFilterSet: false,
     parsedQuery:
      { '$and':
         [ { firstName: { '$eq': 'Agent' } },
           { lastName: { '$eq': 'Smith' } } ] },
     winningPlan:
      { stage: 'LIMIT',
        limitAmount: 1,
        inputStage:
         { stage: 'FETCH',
           inputStage:
            { stage: 'IXSCAN',
              keyPattern: { firstName: 1, lastName: 1 },
              indexName: 'firstName_1_lastName_1',
              isMultiKey: false,
              multiKeyPaths: { firstName: [], lastName: [] },
              isUnique: false,
              isSparse: false,
              isPartial: false,
              indexVersion: 2,
              direction: 'forward',
              indexBounds:
               { firstName: [ '["Agent", "Agent"]' ],
                 lastName: [ '["Smith", "Smith"]' ] } } } }
```

There's a lot of data in `res.queryPlanner`. However, most of it doesn't really matter for our case. The way to read this output to check what index MongoDB used is as follows:

1. The `winningPlan` property describes the *query plan* that MongoDB decided to use to execute the query.
2. Query plans are described recursively via stages. The above query plan has 3 stages: 'LIMIT', 'FETCH', and 'IXSCAN'. The stages in the plan object are in reverse order. MongoDB actually executed them in the following order: 'IXSCAN', 'FETCH', 'LIMIT'.
3. The 'IXSCAN' stage is a shorthand for "index scan." When you see 'IXSCAN', that means MongoDB used an index. The `keyPattern` property shows what index MongoDB actually used. In this case, the index was `{ firstName: 1, lastName: 1 }`.

Using the above process, you end up finding that MongoDB used the `{ firstName: 1, lastName: 1 }` index. Conversely, if you drop the index on `{ firstName, lastName }`, the

`explain()` output has the below `queryPlanner` property:

```
{ queryPlanner:
   { plannerVersion: 1,
     namespace: 'test.users',
     indexFilterSet: false,
     parsedQuery:
      { '$and':
         [ { firstName: { '$eq': 'Agent' } },
           { lastName: { '$eq': 'Smith' } } ] },
     winningPlan:
      { stage: 'LIMIT',
        limitAmount: 1,
        inputStage:
         { stage: 'COLLSCAN',
           filter:
            { '$and':
               [ { firstName: { '$eq': 'Agent' } },
                 { lastName: { '$eq': 'Smith' } } ] },
           direction: 'forward' } },
     rejectedPlans: [] }
```

The above `queryPlanner` only has 2 stages: 'COLLSCAN' and 'LIMIT'. The 'COLLSCAN' stage means MongoDB did a collection scan. In other words, MongoDB had to search through every document in the collection to answer your query.

## What is Index Specificity?

There is another property in the `explain()` output called `executionStats`. This property can tell you how many documents MongoDB had to search to answer your query.

```
let schema = Schema({
  firstName: String,
  lastName: String,
  test: String
});
schema.index({ firstName: 1, lastName: 1 });
const User = mongoose.model('User', schema);
const firstName = 'Agent';
const lastName = 'Smith';

// Insert 120k + 1 documents. 2 documents are different.
const docs = [];
for (let i = 0; i < 120000 - 1; ++i) {
  docs.push({ firstName, lastName });
}
docs.push({ firstName, lastName, test: 'test' });
docs.push({ firstName, lastName: 'Brown' });
await User.insertMany(docs);

let res = await User.
  find({ firstName, lastName: 'Brown' }).explain();
// { stage: 'IXSCAN', nReturned: 1, ... }
res[0].executionStats.executionStages.inputStage;

// Only one result, but has to scan 120000 documents to get it!
const filter = { firstName, lastName, test: 'test' };
res = await User.find(filter).explain();
// { stage: 'IXSCAN', nReturned: 120000, ... }
res[0].executionStats.executionStages.inputStage;
```

The `nReturned` property from an 'IXSCAN' stage is how you quantify index specificity. Smaller `nReturned` means that the index is more *specific*, which usually means faster queries.

In the above example, the `{ firstName, lastName }` index has bad index specificity, because there are 120000 documents with the same `firstName` and `lastName`. If you happen to search for the wrong `firstName` and `lastName` pair, you end up with a slow query even though you have an index.

What is a good index specificity? Similar to the rule of thumb for collection scans: if you need the query to be near-instantaneous, avoid index scans of more than 100k documents. Below 10k documents should be near instantaneous.

How do you ensure index specificity? One way is to mark an index as `unique`. If an index is unique, you're guaranteed an index specificity of at most 1.

Another way is to be careful about building indexes on properties that can take on a limited number of values. Building an index on a single boolean property is generally a bad idea. Similarly, be careful about building an index on a string property with an `enum`. If there are only 4 allowed values for a given property, an index on that property will suffer from bad index specificity unless you're querying by a value that is extremely rare.

## Indexes for Sorting

Indexes aren't just for finding documents quickly. They can also help MongoDB handle expensive sorting operations. Building an index on `lastName` means that you can get all users sorted by `lastName` faster than getting all users sorted by `firstName`.

```
let schema = Schema({ firstName: String, lastName: String });
// Build an index on `lastName` in reverse order
schema.index({ lastName: -1 });

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

const docs = [];
for (let i = 0; i < 200000; ++i) {
  docs.push({ firstName: 'Agent', lastName: 'Smith' });
  docs.push({ firstName: 'Agent', lastName: 'Brown' });
  docs.push({ firstName: 'Agent', lastName: 'Thompson' });
}
await User.deleteMany({});
await User.insertMany(docs);

let start = Date.now();
await User.find().sort({ firstName: 1 });
let elapsed = Date.now() - start;
elapsed; // Approximately 751

start = Date.now();
await User.find().sort({ lastName: -1 });
elapsed = Date.now() - start;
elapsed; // Approximately 641, about 15% faster
```

If you look use `explain()` with the `User.find().sort({ lastName: -1 })` query above, the `queryPlanner.winningPlan` property will look like this:

```
{ stage: 'FETCH',
  inputStage:
   { stage: 'IXSCAN',
     keyPattern: { lastName: -1 },
     indexName: 'lastName_-1',
     ...
     direction: 'forward',
     indexBounds: { lastName: [Array] } } }
```

On the other hand, `User.find().sort({ firstName: 1 })` has a slightly different `explain()` output. Instead of an `IXSCAN` followed by a `FETCH`, MongoDB executed a `COLLSCAN` followed by a `SORT`.

```
{ stage: 'SORT',
  sortPattern: { firstName: 1 },
  inputStage:
   { stage: 'SORT_KEY_GENERATOR',
     inputStage: { stage: 'COLLSCAN', direction: 'forward' } } }
```

The `lastName: -1` query used an index scan and then fetched individual documents in order, whereas the `firstName: 1` query scanned the entire collection and then sorted afterwards.

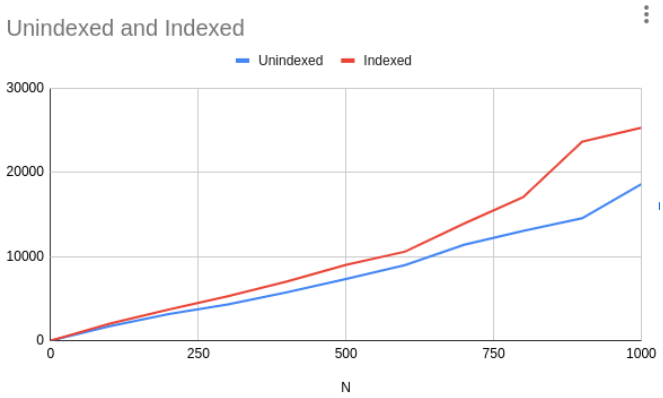
Using an index for sorting is important because MongoDB imposes a 32MB limit on the total size of documents in a sort operation that doesn't use an index. For example, if you want to sort 600k documents with `{ firstName, lastName }` without an index, you'll see the below error message:

```
MongoError: Executor error during find command :: caused by ::
Sort operation used more than the maximum 33554432 bytes of RAM.
Add an index, or specify a smaller limit.
```

### Why Not Index Everything?

If indexes are so key for performance, why not just build an index on every property? There are a couple reasons. First of all, indexes necessarily make writing to the database slower. That's because MongoDB needs to update the index every time you create a new document, delete a document, or update an existing document in a way that affects the index.

The performance impact on updates is generally neglibile until you get up to hundreds of indexes. Below is a graph representing the relative performance of updating `N` indexed versus `N` unindexed keys.




The key takeaway from this graph is that adding 1 or 2 extra indexes usually doesn't make a big difference in performance. Keep in mind that these are indexes on *individual fields*. If you have an index on an array field with thousands of elements, that will have a similar impact to having thousands of indexes.

## 6.6: Slow Trains and Aggregations

Sometimes you may run into a query that should be fast, but instead is painfully slow. For example, the 2nd query in the below example takes 1 second.

```
const conn = await mongoose.createConnection(uri, { poolSize: 1 });
const Model = conn.model('Test', Schema({ name: String }));

await Model.create({ name: 'test' });

// First, run a slow query that will take about 1 sec, but don't
// `await` on it, so you can execute a 2nd query in parallel.
const p = Model.find({ $where: 'sleep(1000) || true' }).exec();

// Run a 2nd query that _should_ be fast, but isn't.
const startTime = Date.now();
await Model.findOne();

Date.now() - startTime; // Slightly more than 1000
```

There's a reason why this 2nd query is slow, and it has to do with the `poolSize` option in the above example. A MongoDB server can only execute a single operation on a given socket at a time. In other words, the number of concurrent operations your Mongoose connection can handle is limited by the `poolSize` option.

In the above example, MongoDB can't execute the 2nd query until the 1st query is done. You can think of sockets as lanes in a tunnel. A fast race car can get through the tunnel quickly. But if the tunnel only has one lane and there's a slow bus in the tunnel, the race car can only go as fast as the bus.

What can you do to avoid slow operations clogging up your connection pool? Here's a few options:

### 1) Increase `poolSize`

Increasing `poolSize` is a one liner to minimize the impact of slow queries. Using `poolSize = 1` is not recommended, in practice you should use at least the default `poolSize = 5`. Below is an example of setting `poolSize` to 10.

```
mongoose.connect(uri, { poolSize: 10 });
```

However, too many connections can cause performance issues and can cause you to hit OS-level resource limits. So increasing `poolSize` to 10k is most likely a bad idea. In production, I generally use `poolSize = 10`.

### 2) Separate connections for known slow operations

Slow queries only slow down operations on the same connection. So you can just create another connection for slow operations! You can think of this as having a separate tunnel just for slow buses and trucks, so faster traffic can go unimpeded through the main tunnel.

```
const conn1 = await mongoose.createConnection(uri, { poolSize: 1 });
const conn2 = await mongoose.createConnection(uri, { poolSize: 1 });
const Model1 = conn1.model('Test', Schema({ name: String }));
const Model2 = conn2.model('Test', Schema({ name: String }));

await Model1.create({ name: 'test' });

// Because this operation is on a separate connection, it won't
// slow down operations on `Model2`
const p = Model1.find({ $where: 'sleep(1000) || true' }).exec();

const startTime = Date.now();
const doc = await Model2.findOne();

doc.name; // 'test'
Date.now() - startTime; // Much less than 1000
```

### 3) Break one slow operation into many fast operations

Another approach for avoiding slow operations is to avoid paradigms that allow you to combine multiple operations into one operation, like aggregations or `bulkWrite()` calls.

The `$lookup` aggregation operator can be especially bad for performance: `$lookup` executes a separate query for every document that comes in to the `$lookup` stage. So an index miss on a

`$lookup` stage causes 1 collection scan for every document coming into the stage!

```
const parentSchema = Schema({ _id: Number });
const Parent = mongoose.model('Parent', parentSchema);
const childSchema = Schema({ name: String, parentId: Number });
const Child = mongoose.model('Child', childSchema);

const parents = [];
for (let i = 0; i < 10000; ++i) parents.push({ _id: i });
await Parent.insertMany(parents);

const cs = [];
for (let i = 0; i < 10000; ++i) cs.push({ name: i, parentId: i });
await Child.insertMany(cs);

const startTime = Date.now();
await Parent.aggregate([{
  $lookup: {
    from: 'Bar',
    localField: '_id',
    foreignField: 'fooId',
    as: 'bars'
  }
}]);
// Takes about 200ms on my laptop even though there's only 10k
// documents. Performance degrades as O(N^2) because of $lookup.
Date.now() - startTime;
```

### 4) Set the `maxTimeMS` option

The `maxTimeMS` option for queries and aggregations sets the maximum amount of time MongoDB will execute a query before throwing an error. Because MongoDB will error out after `maxTimeMS`, it will unblock any other query that's waiting for the slow query.

In the below example, MongoDB throws an 'operation exceeded time limit' error after 10ms because the aggregation is too slow.

```
const parentSchema = Schema({ _id: Number });
const Parent = mongoose.model('Parent', parentSchema);
const childSchema = Schema({ name: String, parentId: Number });
const Child = mongoose.model('Child', childSchema);

const parents = [];
for (let i = 0; i < 10000; ++i) parents.push({ _id: i });
await Parent.insertMany(parents);

const cs = [];
for (let i = 0; i < 10000; ++i) cs.push({ name: i, parentId: i });
await Child.insertMany(cs);

const startTime = Date.now();
const err = await Parent.aggregate([{
  $lookup: {
    from: 'Bar',
    localField: '_id',
    foreignField: 'fooId',
    as: 'bars'
  }
}]).option({ maxTimeMS: 10 }).catch(err => err);

err.message; // 'operation exceeded time limit'
Date.now() - startTime; // About 10
```

Note that `maxTimeMS` does **not** count the amount of time spent waiting behind a slow query. If you send a 50ms query with `maxTimeMS = 100`, and that query is blocked behind a slow query for 500ms, the query will succeed after 550ms. This means that `maxTimeMS` is great for finding queries that are actually slow, as opposed to queries that are just blocked behind an actual slow query.

## 6.7: Manual Sharding

Not to be confused with MongoDB's built-in sharding, *manual sharding* is a fancy way of saying that you can create a second MongoDB server or replica set, and move collections that are causing performance degradation to the second cluster.

That way, if you have one collection that's straining your hardware and bogging down the other collections, you can move that one collection to separate hardware. For example, if one collection is heavily indexed and taking up too much of available RAM, you can move that collection to a separate cluster to free up some RAM.

With an SQL database, you normally wouldn't do manual sharding because you're dependant on JOINs. But Mongoose `populate()` is built with multi-cluster setups in mind. Populating across databases (see Chapter 5) means you can `populate()` data from a completely different MongoDB server in the same way you `populate()` data from the same database.

```
// 2 different MongoDB servers
const host1 = 'mongodb://mongodb1:27017';
const host2 = 'mongodb://mongodb2:27017';
const db1 = await mongoose.createConnection(`${host1}/db`);
const db2 = await mongoose.createConnection(`${host2}/db`);

const M1 = db1.model('Test', Schema({ name: String  }));
// Note that `ref` below is a **Model**, not a string!
const M2 = db2.model('Test', Schema({
  name: String,
  doc: { type: mongoose.ObjectId, ref: Model1 }
}));

const name = 'model 2';
const doc1 = await M1.create({ name: 'model 1' });
await Model2.create({ name, doc: doc1._id });

const doc2 = await M2.findOne({ name }).populate('doc');
doc2.doc.name; // model 1
```

Manual sharding comes with some limitations. Most notably, you can't use `$lookup` to pull data from a different cluster in aggregations. You would also need a separate change stream for each database if you wanted to watch for changes on all your data.

Don't manually shard unless you need to! Applying the design principles from this chapter is enough for most use cases. Manual sharding is a last resort when you have one or two collections that are taking up too many resources and there's no way to reduce the impact of those collections.

## 6.8: Summary

Getting consistent performance from MongoDB comes down to a few simple design principles:

1. **Principle of Least Cardinality:** store one-to-many and many-to-many relationships in a way that minimizes the size of individual documents. No need for a mapping collection if one side of the many-to-many is small.
2. **Principle of Denormalization:** store what you query for. If you need to query model `A` by a property of model `B`, embed that property in model `A`.

3. **Principle of Data Locality:** a document should contain all data necessary to display a web page for that document. If you find yourself always `populate()`-ing a property, you should consider embedding that property.
4. **Indexes:** build indexes on common queries once your collection starts getting up to about 10k documents.
5. **Slow Trains:** many fast operations are better than one slow operation. Avoid `$lookup` and `bulkWrite()` in frequently executed code unless absolutely necessary. Use separate connections for known slow queries.
6. **Manual Sharding:** if all else fails, isolate collections with heavy resource demands on a separate replica set on separate hardware.

Some of these principles may seem counterintuitive. Denormalization is generally frowned upon in SQL because of update anomalies, but since SQL databases usually run behind a cache, there is still a risk of update anomalies, it is just pushed to the caching layer. Caching is generally unnecessary with MongoDB.

Breaking up a `bulkWrite()` into multiple operations can also seem counterintuitive, because, in isolation, one `bulkWrite()` is faster than hundreds of `updateOne()` operations. But one slow `bulkWrite()` can cause a slow train that delays other queries. This principle is similar to threading in operating systems: any individual program would run faster if the operating system didn't switch between threads under the hood, but thread switching prevents any one program from delaying other programs.

# 7: App Structure

The previous chapters have explained the fundamental building blocks of Mongoose: Mongoose's core concepts, and how to get consistent performance from MongoDB and Mongoose.

This chapter will describe how the core concepts come together for building real apps. In particular, this chapter will describe common mistakes people make when building Mongoose apps, and how to avoid them.

## 7.1: Exporting Schemas vs Models

So far, every code sample in this book has been a single file. That is because one clean script with no outside dependencies is short, easy to understand, and easy to copy/paste. But odds are you won't build a Mongoose app as a single file. Not everyone can be Pieter Levels.


So how do you handle multiple files with Mongoose? In practice, most developers end up exporting one model per file. A common pattern is to have a `user.js` file that exports a user model.

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

const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  // ... other properties
});

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

The above pattern is called the *export model pattern*. The export model pattern has some neat properties that make it appealing for simple apps:

1. No need for dependency injection or any other additional frameworks. If you want to use the `User` model, you can do `const User = require('./path/to/user')`.
2. Fits neatly into the model-view-controller paradigm. Many Mongoose apps have 3 directories: `models`, `views`, and `controllers`, with every file in the `models` directory exporting a single model.
3. Easy to test models in isolation. Just `require('./path/to/model')` in your favorite test framework and start testing.

But the export model pattern comes with tradeoffs. A Mongoose model is associated with exactly 1 Mongoose connection. Section 2.7 explains that, when you call `mongoose.model()`, you create a model on the default Mongoose connection. The default Mongoose connection is the one that Mongoose initializes when you call `mongoose.connect()`.

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

// Equivalent:
mongoose.connection.model('User', userSchema);
```

As a consequence, when you use the export model pattern, all your models are limited to sharing the same connection. So slow queries from one model can slow down faster queries from another model. Using the same connection means you're limited in how you can use a separate connection for slower queries as recommended in Section 6.6.

If your app has several known slow queries, the conventional export model pattern can limit your options. One potential workaround is to `require()` in the right connection in your model file. This means each model is responsible for deciding which connection it should use.

```
// `user.js` queries should be on the "fast connection"
const fastConn = require('../connections/fast.js');
const { Schema } = require('mongoose');

const schema = Schema({ name: String, email: String, /*...*/ });
module.exports = fastConn.model('User', schema);

// On the other hand, `pageView.js` queries are for slow
// analytics queries, and should be on the "slow connection"
// to avoid slowing down fast queries.
const slowConn = require('../connections/slow.js');
const { Schema } = require('mongoose');

const schema = Schema({ url: String, time: Date, /*...*/ });
module.exports = slowConn.model('PageView', schema);
```

This alternative export model pattern works because the value returned by `require()` is a *singleton*. Each file that calls `require('../connections/slow.js')` gets the exact same copy of the slow connection instance.

The downside of this alternative export model pattern is that one model may not always be fast or slow. For example, suppose you have a `User` model. Operations like finding a user by their email address need to be fast. But what if you have an analytics dashboard for admins that uses a complex aggregation to calculate user cohort retention?

Splitting up fast operations and slow operations isn't always possible by model. One workaround for the case of the `User` model needing to support both fast and slow operations is to have one app for the fast operations, and a separate app for the slow operations.

### The Export Schema Pattern

Another alternative is to use the *export schema pattern*. With this pattern, instead of exporting a model, your files export mongoose schemas. Below is a simplified `userSchema.js` file.

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

const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  // ... other properties
});

module.exports = userSchema;
```

However, schemas aren't useful unless they're compiled into models. The export schema pattern typically has a `models/index.js` file that exports a function is responsible for decorating a Mongoose connection with models.

```
module.exports = db => {
  const schemas = ['User', /*...*/];
  for (const schema of schemas) {
    db.model(schema, require(`${schema.toLowerCase()}Schema.js`));
  }

  return db;
}
```

You can then instantiate multiple Mongoose connections and decorate them with all your models. You can then have a separate `connections` folder, with several files that export connections with associated models.

```
// connections/fast.js
const models = require('../models');
const mongoose = require('mongoose');

const conn = mongoose.createConnection(process.env.MONGODB_URI);
module.exports = models(conn);

// connections/slow.js
const models = require('../models');
const mongoose = require('mongoose');

const conn = mongoose.createConnection(process.env.MONGODB_URI);
module.exports = models(conn);
```

Another alternative is to register your connections with a dependency injection framework or some other inversion-of-control pattern.

The export schema pattern has several advantages over the export model pattern:

1. Allows you more flexibility in registering the same model on multiple connections.
2. Easier schema reuse. For example, suppose you want to denormalize `User` data in a `Request` model. You can `require('./userSchema')` to get access to the user schema in your `requestSchema.js` file.
3. Cleaner integration with dependency injection.

The downside is that the export schema pattern is more complex and harder to work with. If your app is simple and doesn't need multiple connections or dependency injection, the export model pattern is likely the better choice.

But if you expect your app to rapidly grow in complexity, the export schema pattern is better. In particular, if you expect to build a user-facing app with admin features on top of a monolithic API, you should use the export schema pattern.

## 7.2: Directory Structures

Directory structures are a common debate topic, and it is very easy for directory structure debates to devolve into bikeshedding. This section will describe a few directory structures for Mongoose schemas, models, and connections, and common pitfalls to avoid when laying out a Mongoose project. This section will only discuss structuring the Mongoose part of your code - directory structure for your API is a subject for later.

### Directory Structure for the Export Models Pattern

The directory structure for the export models pattern is simple: you create a `models` directory that contains all of your models.

```
models/
  - User.js
  - Photo.js
  - Like.js
    more...
```

The goal of the export models pattern is to enable using the model with nothing but a `require()` call. There is a catch though: suppose your `User.js` file looks like what you see below.

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

module.exports = mongoose.model('User', mongoose.Schema({
  name: String,
  email: String
}));
```

When you `require('path/to/User')`, you would still need to call `mongoose.connect()`, otherwise `User.find()` and other database operations will hang. How can you make it so that you have a single entry point for calling `mongoose.connect()` without sacrificing the convenience of the export model pattern?

Remember that `require()` returns a *singleton*. If you `require()` the same file multiple times from different files, you get the exact same object reference. So the trick is to put a `connect.js` file in your `models` directory that all models `require()` in.

```
models/
  - User.js
  - Photo.js
  - Like.js
    More...
  - connect.js
```

For example, below is what `User.js` should look like.

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

require('./connect');

module.exports = mongoose.model('User', mongoose.Schema({
  name: String,
  email: String
}));
```

The `connect.js` is responsible for calling `mongoose.connect()` with the correct parameters. The below example uses an environment variable, but this approach also works with config files or any other synchronous mechanism for pulling configs.

```
const opts = {
  useNewUrlParser: true,
  useUnifiedTopology: true
};

require('mongoose').connect(process.env.MONGODB_URI, opts).
  catch(err => {
    // If there's an error connecting to MongoDB, throw an uncaught error
    // that kills the process.
    process.nextTick(() => { throw err; });
  });
```

The `connect.js` file doesn't export anything. The first time you `require()` in a model file, the `connect.js` file will attempt to connect to MongoDB. Subsequent `require('./connect')` calls don't do anything.

This directory structure has some limitations. It works well for the basic case of a simple API or backend service that can't do anything if it can't connect to MongoDB. But if you want your service to keep running if initial connection fails, you may need a different directory structure.

#### Export Schema Pattern: Export Connection

A more flexible alternative to the export model pattern that still retains a lot of the good properties of the export model pattern is exporting a connection. Instead of calling `require()` on individual model files, you call `require()` on the whole models directory, which gives you a connection object. Below is the directory structure.

```
models/
  - User.schema.js
  - Photo.schema.js
  - Like.schema.js
    More...
  - index.js
```

The `*.schema.js` files export the appropriate schema:

```
const { Schema } = require('mongoose');

module.exports = Schema({
  name: String,
  email: String
});
```

The `index.js` file is responsible for compiling models for these schemas on a Mongoose connection. You can use the default connection:

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

mongoose.model('User', require('./User.schema.js'));
mongoose.model('Photo', require('./Photo.schema.js'));

let opts = { useUnifiedTopology: true, useNewUrlParser: true };
mongoose.connect(process.env.MONGODB_URI, opts);

module.exports = mongoose.connection;
```

Now, to use your models, instead of using `require()` to pull in individual models, you would instead `require()` in the connection.

```
const db = require('path/to/models');

// Get the `User` model and use it
const User = db.model('User');
User.findOne().then(user => console.log(user));
```

This approach has two benefits over the export model pattern's directory structure. First, since you export the *connection*, the code that calls `require()` can handle connection errors. So, if your app needs to keep running even if Mongoose fails to connect, this directory structure is helpful. Unfortunately, exporting the connection makes it a bit more difficult to access the models - you need to use the `.model()` function.

However, this approach is still dependent on singletons. You can handle multiple connections with this approach: just have a separate file that exports a different connection. But you would need a separate file for each connection.

#### Export Schema Pattern: Export Function

The `index.js` file doesn't have to export a connection. Another pattern is to export a function that creates a new connection.

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

module.exports = async function createConnection(uri, opts) {
  const db = mongoose.createConnection(uri, opts);

  db.model('User', require('./User.schema.js'));
  db.model('Photo', require('./Photo.schema.js'));

  return db;
};
```

This approach loses the key benefit of being able to use Mongoose immediately on calling `require()`. But it also gives you extra flexibility to create connections as you see fit. For example, to avoid slow trains, you may create separate connections for different Express sub-apps or different websocket events. You should use this approach if you're using dependency injection.

Another alternative is to return an object containing all models, as opposed to the connection itself.

```
module.exports = async function createConnection(uri, opts) {
  // Will throw an error if initial connection fails.
  const db = await mongoose.createConnection(uri, opts);

  const User = db.model('User', require('./User.schema.js'));
  const Photo = db.model('Photo', require('./Photo.schema.js'));

  return { User, Photo };
};
```

This alternative encapsulates the connection and makes it easier to access the models. However, exporting a function that returns a hash of models is not a common pattern because then you need to list out every model in your export.

#### Takeaways

There are numerous directory structures and patterns you can use to organize Mongoose schemas, models, and connections. You can choose to have individual files export schemas or models: exporting models is more convenient, but also more limiting.

You can have a top-level `index.js` file in `models` that exports models or a connection. Exporting models is more convenient for syntax and hides the connection object, which is better if you want to encapsulate error handling. On the other hand, exporting connections is better if you want to defer error handling to the calling code.

Which pattern is better is up to the unique needs of your app. Since the export model pattern is easiest to work with, you should prefer to use the basic export model pattern unless you have a good reason to use one of the more sophisticated patterns.

### 7.3: Custom Validators vs Middleware

Another common task when building Mongoose apps is handling validation that depends on multiple fields. Suppose you have a `Profile` model that has a `photos` property. If the profile's `status` property is `'PUBLISHED'`, the profile needs at least 2 `photos`.

Below is a sample `profileSchema` that demonstrates the issue. How do you make it so that `photos` must have length 2 if `status` is `PUBLISHED`?

```
const profileSchema = Schema({
  photos: [String],
  status: { type: String, enum: ['PENDING', 'PUBLISHED'] }
});
```

There are two ways. First, you can define a custom validator on `photos`:

```
const profileSchema = Schema({
  photos: {
    type: [String],
    validate: function(v) {
      return this.status !== 'PUBLISHED' || (v && v.length > 1);
    }
  },
  status: { type: String, enum: ['PENDING', 'PUBLISHED'] }
});
```

Another way is to define a `pre('save')` hook that checks `photos` if `status` is 'PUBLISHED'.

```
const profileSchema = Schema({
  photos: [String],
  status: {
    type: String,
    enum: ['PENDING', 'PUBLISHED']
  }
});

profileSchema.pre('save', function() {
  if (this.status === 'PUBLISHED' && this.photos.length < 2) {
    throw Error('Published profile must have at least 2 photos');
  }
});
```

Which approach is better? Remember that Mongoose only runs validation on modified paths, so the first approach suffers from a severe limitation. If you modify `status` without modifying `completedAt`, the first approach won't throw an error.

```
const Profile = mongoose.model('Profile', Schema({
  photos: {
    type: [String],
    validate: function(v) {
      if (this.status !== 'PUBLISHED') return true;
      return v != null && v.length >= 2;
    }
  },
  status: {
    type: String,
    enum: ['PENDING', 'PUBLISHED']
  }
}));
const doc = await Profile.create({ status: 'PENDING' });

doc.status = 'PUBLISHED';
// Doesn't throw because `photos` is not modified.
await doc.save();
```

When you call `save()`, Mongoose only validates modified paths. There is one exception: the `required` validator. For example, suppose your `profileSchema` also has a `publishedAt` property that should be required if the profile is published.

```
const Profile = mongoose.model('Profile', Schema({
  publishedAt: {
    type: Date,
    required: function() { return this.status === 'PUBLISHED' }
  },
  status: {
    type: String,
    enum: ['PENDING', 'PUBLISHED']
  }
}));
const doc = await Profile.create({ status: 'PENDING' });

doc.status = 'PUBLISHED';
const err = await doc.save().catch(err => err);
err.message; // 'publishedAt: Path `publishedAt` is required.'
```

That's because Mongoose does some extra work under the hood to make sure the `required` validator runs when the paths the `required` validator depends on changes. But Mongoose does **not** run custom validators unless the path the validator is registered to is modified.

What does this mean for building apps? In general, you should avoid custom validators that rely on multiple paths. Usually, if you can do something with custom validators, it is also easy to do with a `pre` hook. If you need complex validation that depends on multiple fields, better to use middleware than custom validators.

### 7.4: Pagination

Think about the page counter on the bottom of Google's search results:


Suppose you were implementing an app that let you go through pages of git commits, like GitHub. Below is a simplified `Commit` model:

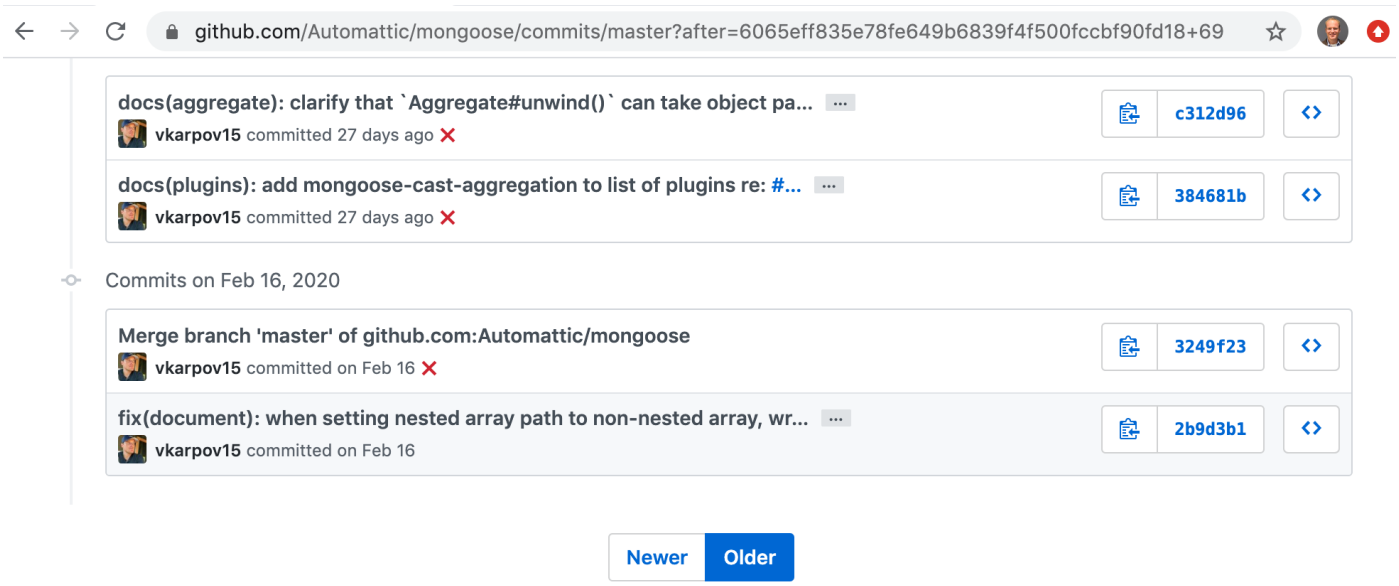
```
const Commit = mongoose.model('Commit', Schema({
  repo: String,
  message: String,
  hash: String,
  diff: String,
  order: Number
}));
```

For this simple example, suppose that the `order` field stores the order of commits, so the first commit to the repo has `order = 0` and the 2nd has `order = 1`. That isn't how git commits are actually ordered, but this model is a simplifying assumption to make this example more accessible.

There are a couple ways you can model pagination. The first way is using the `Query#limit()` and `Query#skip()` query helpers. Assuming you display 5 commits per page, getting the `N`th page means getting 5 commits after skipping the first `N * 5` commits.

```
function getPage(repo, pageNum, perPage) {
  return Commit.find({ repo }).
    sort({ order: -1 }).
    skip(pageNum * perPage).
    limit(perPage);
}

const Commit = mongoose.model('Commit', Schema({
  repo: String,
  message: String,
  hash: String,
  diff: String,
  order: Number
}));

const repo = 'mongoosejs/test';
for (let i = 1; i <= 100; ++i) {
  await Commit.create({ repo, message: `# ${i}`, order: i });
}

const page1 = await getPage('mongoosejs/test', 0, 5);
page1.map(doc => doc.order); // [100, 99, 98, 97, 96]
const page2 = await getPage('mongoosejs/test', 1, 5);
page2.map(doc => doc.order); // [95, 94, 93, 92, 91]
```

However, if you look closely at GitHub, you'll see that isn't how they handle pagination over commits. Below is a screenshot of GitHub's list of commits to the Mongoose repo's `master` branch.



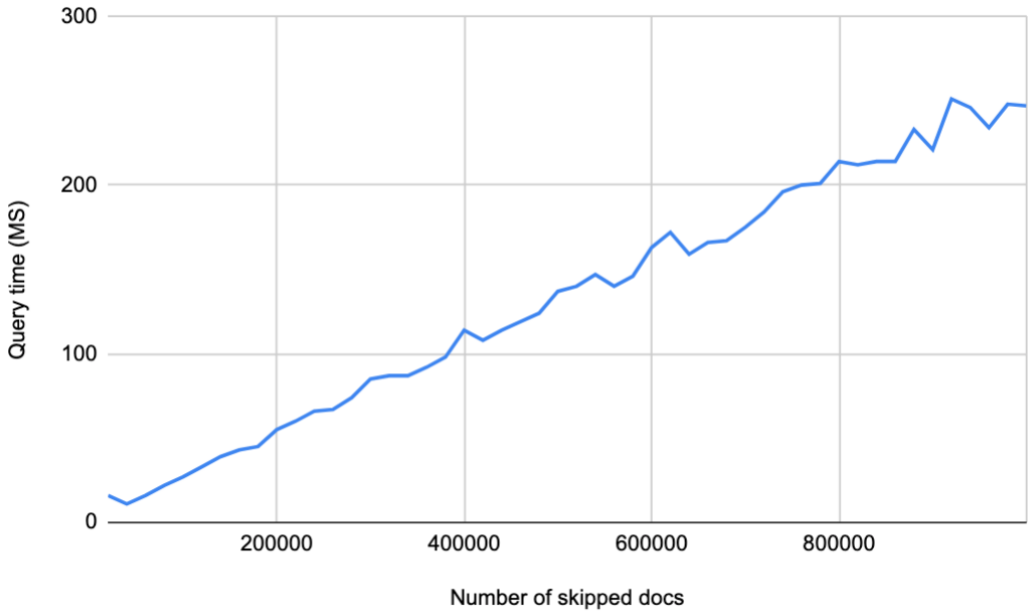

Note that there is no button to go to page 3, there's only "previous" and "next" buttons. And instead of a page number, there's a `?after` in the URL that points to a git commit hash.

Here's how you can implement GitHub-style pagination using Mongoose:

```
const firstPage = (repo, perPage) => Commit.find({ repo }).
  sort({ order: -1 }).limit(perPage);

const nextPage = (repo, perPage, $lt) => Commit.
  find({ repo, order: { $lt } }).
  sort({ order: -1 }).limit(perPage);
const prevPage = (repo, perPage, $gt) => Commit.
  find({ repo, order: { $gt } }).
  sort({ order: -1 }).limit(perPage);

const page1 = await firstPage('mongoosejs/test', 5);
page1.map(doc => doc.order); // [100, 99, 98, 97, 96]

const page2 = await nextPage('mongoosejs/test', 5,
  page1[page1.length - 1].order);
page2.map(doc => doc.order); // [95, 94, 93, 92, 91]
```

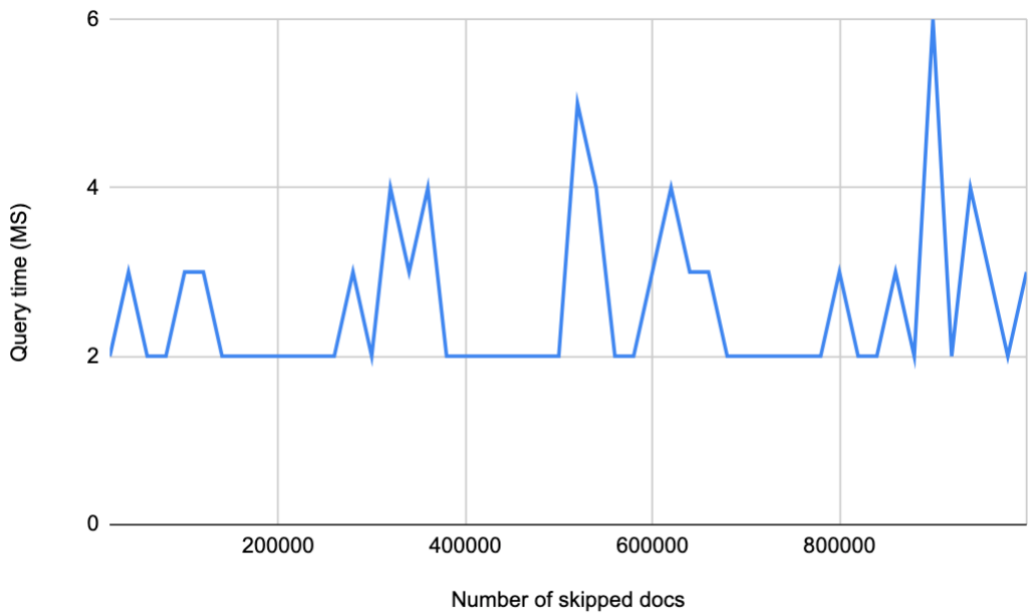
As recently as 2014, GitHub had standard pagination with page numbers for commits, but then they changed it. Why did GitHub change it? One possible reason is performance.

For example, suppose you had a git repo that had hundreds of thousands of commits. The more you `skip()`, the slower your query gets. Below is a graph of `skip()` number vs query time in MS.




The problem is that indexes can't help you with `skip()`. MongoDB still needs to loop through every document you skip. Which means performance degrades linearly as `skip()` size grows.

On the other hand, GitHub style pagination lets you leverage indexes to help with pagination. The key idea is that `Commit.find({ order: { $gt: 1000 } })` can leverage an index on `order`, which means you can get consistent performance as the page count grows.




There are several tradeoffs between these two pagination patterns. The `skip` approach is much easier to implement: there are several plugins, like `mongoose-paginate-v2`, that handle `skip()`-based pagination. The `skip()` based approach is also more flexible: you can sort documents in any order and still use `skip()`. With the `$gt`-based approach, you can only sort by `order`.

The `$gt`-based pagination pattern is better for performance. If you find yourself telling MongoDB to `skip()` hundreds of thousands of documents, your pagination queries will be slow and you'll end up with slow trains. However, for most applications, the `skip()`-based approach is fast enough. Unless you find your users are regularly causing skips of more than 10k documents, using `skip()` for pagination should be fine.

## 7.5: Storing Passwords and OAuth Tokens

Most Mongoose apps have to implement some sort of authentication, using passwords or OAuth or both. A common mistake Mongoose beginners make is storing password hashes directly on their `User` model.

```
const User = mongoose.model('User', Schema({
  email: { type: String, required: true },
  password: { type: String, required: true }
}));

// Later
const bcrypt = require('bcryptjs');
const promisify = require('util').promisify;

// Hash password before storing
let pw = await promisify(bcrypt.hash).call(bcrypt, 'taco', 3);
await User.create({
  email: 'taco@theleague.com',
  password: pw
});
```

What is wrong with this approach? Too many opportunities for mistakes. Leaking a user's password hash is a serious security issue, so you only want to access a user's password hash when absolutely necessary: when logging in. The common workaround is to use Mongoose's *schema-level projections*.

```
const User = mongoose.model('User', Schema({
  email: { type: String, required: true },
  password: {
    type: String,
    required: true,
    // `password` will be excluded when you do `User.find()`
    // unless you explicitly project it in.
    select: false
  }
}));

let user = await User.findOne();
user.password; // undefined
user = await User.findOne().select('+password');
user.password; // String containing password hash
```

Mongoose schema level projections work fine. As long as you don't accidentally return a user that you pulled from a `User.find().select('+password')` query, you won't leak password hashes. However, it is far too easy to make a mistake and leak a user with a password hash. As a technical architect, your job is to make it as hard as possible to make a major mistake.

Instead of storing password hashes and OAuth tokens on the user documents, you should create a separate `AuthenticationMethod` model.

```
const schema = Schema({
  type: {
    type: String,
    enum: ['PASSWORD', 'FACEBOOK_OAUTH']
  },
  // `secret` stores the password hash for password auth,
  // or the access token for Facebook OAuth.
  secret: String,
  user: { type: ObjectId, ref: 'User' }
});
const AuthenticationMethod =
  mongoose.model('AuthenticationMethod', schema);
```

This way, instead of relying on Mongoose to always exclude the password hash, you separate the password hash from the user. If there's no `authenticationMethod` property on the `User` model, there's no way you'll end up with a user that has an embedded password hash.

This is another application of the Principle of Data Locality: if there's a property that you usually want to *exclude* from your document, it is better to store it in a separate collection. Passwords are

a good example: why store password hashes with the `User` document when you almost always want to hide them?

## 7.6: Integrating With Express

Most Mongoose apps also use other frameworks, because Mongoose can only communicate with MongoDB, not with HTTP or websockets. Express is the most popular web server framework for Node.js, and also the most common framework used with Mongoose. In this section, you'll see how to set up a basic Express API using the export model pattern. Note that this app is an educational example, not a production-ready app.

The API will be for a user directory app. As a user, you will be able to log in, modify your profile, and search for other users by name. Not the most complex app, but enough to demonstrate how the app design principles from this chapter come together. You can find the raw code for this app under the `headless-user-directory` folder in the `mongoose-sample-apps` GitHub repo that you received with this ebook.

First, here's the app's directory structure.

```
scripts/
  - seed.js
src/
  - api/
      - checkAuth.js
      - findUsers.js
      - index.js
        more...
  - models/
      - AccessToken.js
      - AuthenticationMethod.js
      - User.js
      - connect.js
test/
  - api.test.js
index.js
package.json
```

This app uses the export models pattern. The `src/models` directory doesn't have an `index.js` file, but it does have a `connect.js`. So to use Mongoose, all you need to do is `require()` in a model.

The only new model here is the `AccessToken` model. This model is responsible for answering the question "is a given user logged in?" Once a user logs in using their `AuthenticationMethod`, your app should create a new `AccessToken` and send that token to the client.

In order for the client to prove that they're logged in, they send the `AccessToken` id to the server. For HTTP-based apps, the client typically sends the token in the request's `Authorization` header. Here's what the `AccessToken` model looks like.

```
const crypto = require('crypto');
const mongoose = require('mongoose');
const { Schema } = mongoose;
require('./connect');

let Model = mongoose.model('AccessToken', Schema({
  _id: {
    type: String,
    required: true,
    default: () => crypto.randomBytes(50).toString('base64')
  },
  user: { type: 'ObjectId', ref: 'User', required: true }
}));

module.exports = Model;
```

The `AccessToken` model has two properties: a cryptographically secure random string `_id` and an associated user. The `crypto` module is a built-in Node.js module that handles some common cryptography tasks.

To see the `AccessToken` model in action, open up the `src/api/checkAuth.js` file. This file exports an Express middleware which checks the request's `Authorization` header. If the `Authorization` header contains a valid access token id, it sets the `req.user` property to the logged in user. If not, the `checkAuth` middleware throws an error.

```
const AccessToken = require('../models/AccessToken');
const User = require('../models/User');

module.exports = function checkAuth(req, res, next) {
  AccessToken.findOne({ _id: req.headers.authorization }).
    orFail().
    then(({ user }) => User.findOne({ _id: user }).orFail()).
    then(user => {
      req.user = user;
      next();
    }).
    catch(next);
};
```

Note that Express 4 doesn't have good support for async/await. So this app makes heavy use of promise chaining and Mongoose's `Query#orFail()` helper. The `orFail()` helper tells Mongoose to throw an error if `findOne()` returns no results.

Next, let's take a look at the `src/api/index.js` file. This is the entry point for the Express app. The order of Express `get()`, `post()`, and `use()` calls are important: putting `app.use(checkAuth)` means everything *after* that `use()` call requires authentication.

```
const app = require('express')();
// Ensure that `req.query` values are always strings or nullish
app.set('query parser', 'simple');
app.use(express.json());

app.post('/login', require('./login'));
app.post('/register', require('./register'));

// The rest of the functionality requires being logged in.
app.use(require('./checkAuth'));

app.put('/user', require('./updateUser'));
app.get('/users', require('./findUsers'));

// Error handling middleware
app.use(function(err, req, res, next) {
  res.status(500).json({ message: err.message });
});

module.exports = app;
```

Similar to the export models pattern, this Express app is a singleton, because the `index.js` file exports a single Express app. This pattern's benefits and limitations are similar to those of the export models pattern: easy and convenient, lacking flexibility, but good enough for a simple app.

The app has 4 endpoints:

- `POST /login` checks username and password, and returns an access token if login succeeded.
- `POST /register` creates a new user and authentication method.
- `PUT /user` updates an existing user.
- `GET /user` searches for users.

The `login` and `register` endpoints are the most transferrable. Most apps that store authentication methods in MongoDB have similar endpoints. Below is the `src/api/login.js` file.

```
const AccessToken = require('../models/AccessToken');
const AuthenticationMethod =
  require('../models/AuthenticationMethod');
const User = require('../models/User');
const bcrypt = require('bcryptjs');

module.exports = function handleLogin(req, res, next) {
  const { email, password } = req.body;
  let user;

  User.findOne({ email }).orFail().
    then(({ _id }) => {
      user = _id;
      return AuthenticationMethod.findOne({ user }).orFail();
    }).
    then(({ secret }) => bcrypt.compare(password, secret)).
    then(success => {
      if (!success) return next(Error('Incorrect Password'));
      return AccessToken.create({ user });
    }).
    then(({ _id }) => res.json({ token: _id })).
    catch(next);
};
```

Step by step, here's what the `login.js` file does:

1. Given an email and password, find the user by their email address.
2. Find the user's authentication method.
3. Verify the password against the password hash using `bcrypt.compare()`.

4. Create an access token and return the token's `_id`.

Given an access token, the client can now authenticate as a user and access the authenticated functionality.

Next, below is the `src/api/register.js` file. The `POST /register` endpoint needs to create a new user, hash their password, and save a new `AuthenticationMethod`.

```
const AuthenticationMethod =
  require('../models/AuthenticationMethod');
const User = require('../models/User');
const bcrypt = require('bcryptjs');

module.exports = function register(req, res, next) {
  User.create(req.body).
    then(user => {
      return bcrypt.hash(req.body.password, 4).
        then(secret => AuthenticationMethod.create({ user, secret })).
        then(() => user);
    }).
    then(user => res.json(user)).
    catch(next);
};
```

Now that you can create a user and log in, let's see how to implement the actual functionality of the app. Below is the `POST /user` route, which lets a user update their profile.

```
module.exports = function updateUser(req, res, next) {
  req.user.set(req.body);
  req.user.save().
    then(() => res.json(req.user)).
    catch(next);
};
```

The `POST /user` route is very simple. That's because it relies on Mongoose models to do the heavy lifting of validating the input data. If a malicious user tries to set their `firstName` property to an object, for example, Mongoose's `save()` function will error out.

*You should rely on Mongoose to do basic validation in your Express apps.* There isn't any need to check whether `email` is unique or check that `firstName` is a string, Mongoose will do that all for you.

So given this API, how do you interact with it? Check out the `test/api.test.js` file. Below is a sample test from that file, which registers as a user and then makes sure you can log in as the

user, using the Axios HTTP library.

```
const urlRoot = 'http://localhost:3000';
const user = {
  firstName: 'Taco',
  lastName: 'MacArthur',
  email: 'taco@theleague.com',
  password: 'taco'
};

let res = await axios.post(`${urlRoot}/register`, user);
assert.equal(res.data.firstName, 'Taco');
assert.strictEqual(res.data.password, undefined);

const count = await mongoose.model('AuthenticationMethod').
  countDocuments({});
assert.equal(count, 1);

res = await axios.post(`${urlRoot}/login`, user);
assert.ok(res.data.token);
```

That's it for building a basic RESTful API on top of Mongoose using Express. The key takeaways are:

1. Make use of promise chaining and `orFail()`, because Express 4 doesn't support async functions.
2. Leverage Mongoose's casting and validation instead of validating API data by hand.
3. Be careful about the order in which you create new models. For example, make sure you call `User.create()` before `AuthenticationMethod.create()`, so you can validate user data first and get a user id to associate with the `AuthenticationMethod`.

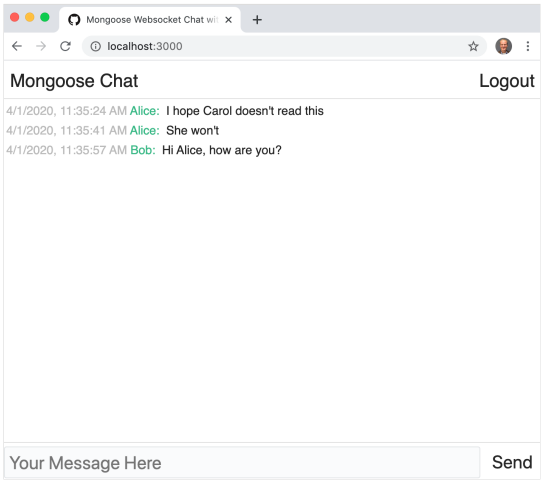
## 7.7: Integrating with Websockets

Express and HTTP are the most common frameworks used with Mongoose. Let's take a look at a slightly different framework to see what assumptions are transferrable from the Express app.

This section's app is a basic realtime chat using Express, Mongoose, and the `ws` package for websockets. The frontend is a minimal vanilla JavaScript app that assumes evergreen browsers.

Below is what the websocket chat frontend looks like. The app's seed script creates 2 users: Alice and Bob. Their emails are `alice@mit.edu` and `bob@mit.edu`, and their passwords are 'alice' and 'bob', respectively. If Alice sends a message, the backend pushes that message out over a websocket so both she and Bob can see that message in realtime.

You can find the raw code for this app under the `vanilla-websocket-chat` folder in the `mongoose-sample-apps` GitHub repo that you received with this eBook.




Below is the app's directory structure.

```
public/
  - js/main.js
  - index.html
  - style.css
src/
  - api/
      - websocket/index.js
      - checkAuth.js
      - index.js
        more...
  - client/index.js
  - models/
      - AuthenticationMethod.js
      - Message.js
      - User.js
      - connect.js
index.js
package.json
```

### `index.js` Entrypoint

Much of this app is the same as the Express-based user directory from Section 7.6. Below is the `api/index.js` file. The key difference between this API and the user directory API is that `api/index.js` exports a function that returns an Express server, as opposed to exporting an Express app. This difference is to support websockets: the `on('upgrade')` event is only available Express servers, which is what `app.listen()` returns.

```
const express = require('express');
const handleUpgrade = require('./websocket');

module.exports = (app, port) => {
  app.set('query parser', 'simple');
  app.use(express.json());
  app.post('/login', require('./login'));
  app.post('/register', require('./register'));

  // The rest of the functionality requires being logged in.
  app.use(require('./checkAuth'));
  app.put('/user', require('./updateUser'));
  app.get('/messages', require('./messages'));
  // Error handling middleware
  app.use(function(err, req, res, next) {
    res.status(500).json({ message: err.message });
  });

  const server = app.listen(port);
  server.on('upgrade', handleUpgrade);
  return server;
};
```

### `handleUpgrade()` and JWTs

The `handleUpgrade()` function is the entry point to the websocket functionality. The websocket functionality will be responsible for creating `Message` documents, and sending the newly created `Message` document to all connected clients.

The trick is that the websocket code needs to be responsible for authentication as well as sending/receiving messages. Instead of using the `AccessToken` pattern for authentication, this app uses JSON Web Tokens (JWTs), a common alternative. A JWT stores user data encrypted with a secret key: if the server can decrypt the JWT and the JWT is properly formatted, it is considered valid.

Below is the `handleUpgrade()` function in `websocket/index.js`. Note that it tries to decrypt the JWT, and, if decryption is successful, it checks to see that the JWT contains a valid `User` id. If it successfully finds a user, `handleUpgrade()` then emits a `'connection'` event. If there is no such user, `handleUpgrade()` closes the socket.

```
module.exports = function handleUpgrade(request, socket, head) {
  const header = request.headers['sec-websocket-protocol'];
  const token = header == null ?
    null : header.slice('access_token, '.length);

  Promise.resolve().
    then(() => jsonwebtoken.verify(token, secretToken)).
    then(data => User.findOne({ _id: data.userId }).orFail()).
    then(user => {
      request.user = user;
      server.handleUpgrade(request, socket, head,
        socket => server.emit('connection', socket, request));
    }).
    catch(() => socket.destroy());
};
```

#### Sending a Message

The `websocket/index.js` file is also responsible for handling the `'connection'` event. Once that event is emitted, the client can send messages on the socket. On the server side, websockets are event emitters that emit a `'message'` event when they receive a message.

```
server.on('connection', (socket, req) => {
  const { user } = req;
  const userName = user.name;
  socket.on('message', msg => {
    Promise.resolve().
      then(() => JSON.parse(msg)).
      then(msg => Message.create({ ...msg, user, userName })).
      then(doc => server.clients.forEach(c => sendSuccess(c, doc))).
      catch(err => sendError(socket, err.message));
  });
  function sendError(socket, message) {
    socket.send(JSON.stringify({ error: true, message }));
  }
  function sendSuccess(socket, res) {
    socket.send(JSON.stringify({ error: false, res }));
  }
});
```

JavaScript passes the `msg` as a string to your 'message' event handler. This server assumes the `msg` is a JSON string representing an object that looks like `{ body: 'this is my message' }`.

If `msg` is valid JSON, the server relies on the `Message` model to verify that the message has the correct properties.

The order of the document keys in `Message.create()` is very important. Make sure you put `user` and `userName` **after** `...msg`. Otherwise a malicious user could spoof a message from another user by putting a `user` property in their websocket message.

Like with the Express HTTP-only user directory, you can rely on Mongoose to validate data you're inserting into your database. Since the `Message` model indicates that `body`, `user`, and `userName` are all required, Mongoose will make sure those properties are set. However, you're responsible for making sure that the user is correct and a malicious user can't spoof the `user` property.

```
const mongoose = require('mongoose');
require('./connect');

const schema = mongoose.Schema({
  user: { type: 'ObjectId', required: true, ref: 'User' },
  userName: { type: String, required: true, trim: true },
  body: { type: String, required: true }
}, { timestamps: true });

schema.index({ createdAt: -1 });

module.exports = mongoose.model('Message', schema);
```

Note that the `Message` model embeds the `User.name` property as `Message.userName`. This is an application of the principle of data locality. To render a message, the client needs the user's name. Embedding `userName` means you don't need an extra `populate()` every time the client loads messages, which reduces complexity and helps performance.

#### Loading Messages

There's one more important detail to note about the `Message` model: the index on `createdAt`. When the client loads, it should load the most recent messages, and there can be a massive amount of messages. Below is the query to load messages from `src/api/message.js`:

```
module.exports = function messages(req, res, next) {
  Message.find().sort({ createdAt: -1 }).limit(100).
    then(messages => res.json({ messages })).
    catch(next);
};
```

To make sure this query is consistently fast even if there's millions of messages, you need an index on `createdAt: -1`. The `timestamps` option automatically adds a `createdAt` property to

the schema.

This app doesn't implement pagination, but you may want to implement loading past messages when the user scrolls up. The `Message` is a good candidate for GitHub-style pagination using `createdAt`: there is already an index on `createdAt`, and there is no need to sort on any other property. Plus, there can easily be millions of messages to scroll through, so using `skip()` could cause performance issues.

That's it for the websocket API. The takeaways for the websocket API are similar to those for the HTTP API: lean on Mongoose to take care of repetitive validation, and use `orFail()` to make promise chaining cleaner for frameworks that don't support async functions. The chat app also shows an example of where to use indexes: the `Message` collection can grow huge, so make sure common queries are fast.

## 7.8: Summary

Mongoose makes building apps easy by taking care of repetitive validation tasks and providing a framework for organizing business logic that is independent of data input. For example, the last two sections built out a vanilla HTTP server and a websocket server using essentially the same models, and with minimal framework-based logic.

Both apps in this chapter used the export model pattern and directory structure. This pattern does have limitations in terms of flexibility because of singletons, and in terms of performance because of slow trains. But the convenience of the export model pattern, namely the ability to `require()` a model and use it immediately, makes it a compelling option for smaller apps. If you want to build an admin dashboard or compose multiple apps in one Node.js process, you shouldn't use the export model pattern.

Both apps in this chapter have `seed` scripts that you can run using `npm run seed`. The purpose of a seed script is to "seed" the local database with some minimal data so a user can tinker with your app. Seed scripts are a handy tool to help people get started with your app, and they can also double as test fixtures.

Finally, both apps make heavy use of schema design to minimize the amount of work that goes into casting and validating user input. You'll know you're using Mongoose correctly when your business logic looks extremely flat, without a lot of `if` statements or `assert` calls. Your schemas should bake in what it means to `create()` a new user or `find()` a list of messages using types, validators, and middleware, so you can easily plug this functionality into apps that speak HTTP, websockets, pub/sub, or any other I/O format.