Server-side
Web Development

Unit 15. Node.js i MongoDB

Fidel Oltra, Ricardo Sánchez

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2023-24

Index

1 Node.js and MongoDB. Installing Moongose.

In this unit we are going to learn how to connect and work with a MongoDB Database from Node.

If we are working with Docker we just need to start the container:

```
docker start mongo
```

If you don't have the container, create it with:

```
docker run --name mongo -v mongodata:/data/db
           -d mongodb/mongodb-community-server:latest
```

Now, let's create a folder for our project. For instance, `contacts_Mongo`. Inside this folder, we will install the library `Mongoose`, that makes our work with MongoDB easier. You can find a lot of information about `Moongoose` in the official web.

mongoose official page

To install `Moongose` in our project, we must create the project and the `package.json` file in the project folder (`contactMongo`) using `npm init`.

```
fidel@fidel-DWES:~/projectesNode/contactMongo$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (contactmongo)
version: (1.0.0)
description: Contacts with MongoDB
entry point: (index.js)
test command:
git repository:
keywords:
author: Fidel Oltra
```

```
license: (ISC)
About to write to /home/fidel/projectesNode/contactMongo/package.json:

{
  "name": "contactmongo",
  "version": "1.0.0",
  "description": "Contacts with MongoDB",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Fidel Oltra",
  "license": "ISC"
}

Is this OK? (yes)
npm notice
npm notice New patch version of npm available! 10.2.3 -> 10.2.5
npm notice Changelog: https://github.com/npm/cli/releases/tag/v10.2.5
npm notice Run npm install -g npm@10.2.5 to update!
npm notice
```

Now let's install `mongoose`:

```
fidel@fidel-DWES:~/projectesNode/contactMongo$ npm install mongoose

added 22 packages, and audited 23 packages in 7s

1 package is looking for funding
  run `npm fund` for details

found 0 vulnerabilities
fidel@fidel-DWES:~/projectesNode/contactMongo$
```

Now, we need to import the library `mongoose` in our project. For instance, creating and `index.js` document and adding the line:

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

To connect to the Mongo server (assuming it's already running), we need to call a method called `connect` within the `mongoose` object that we have imported. We will pass the database URL as the first parameter and, optionally, a second parameter with an object containing connection properties. Including this second object is necessary in certain versions of Mongoose to specify some additional options, and in fact, we may see a warning when running the application, but generally, we can connect directly with the connection URL as the first and only parameter.

In the case of accessing a local MongoDB server, we can use a URL like this:

```
mongoose.connect('mongodb://127.0.0.1:27017/contacts');
```

> NOTE: In certain versions, Mongoose also accepts connecting to localhost as the local server name, but in others, it has restricted it to the local IP 127.0.0.1.

2 Schemes and models

To work with our database using `Moongose` we need to define `schemas` and `modules` (associated to collections).

2.1 Defining Schemas

To define a schema, we need to create an instance of the Mongoose Schema class. Therefore, we'll create this object and define the attributes that the corresponding collection will have, along with the data type of each attribute. It is also advisable to separate these definitions into separate files. We can create a subfolder called `models` and store our database schemas and models in it.

For the proposed contacts database in these tests, we can define a schema to store the data for each contact: name, phone number, and age, for example. We would do this in a file called "contact.js" inside the "models" subfolder of our project:

```
// models/contact.js
const mongoose = require('mongoose');

let contactSchema = new mongoose.Schema({
    name: String,
    phone: String,
    age: Number
});
```

The available data types to define the schema are:

- Text (String)
- Numbers (Number)
- Dates (Date)
- Booleans (Boolean)
- Arrays (Array)
- Others (we'll see some more later): Buffer, Mixed, ObjectId

2.2 Applying the Schema to a Model

Once the schema is defined, we need to apply it to a model to associate it with a collection in the database. For this, we have the **`model`** method in Mongoose. As the first parameter, we'll indicate the name of the collection to which to associate the schema. As the second parameter, we'll specify the schema to apply (an object of type `Schema` created previously). We would add these lines **at the end** of our "`models/contact.js`" file:

```
let Contact = mongoose.model('contacts', contactSchema);
module.exports = Contact;
```

> NOTE: If we specify a singular model name, Mongoose will automatically create the collection with the pluralized name. This pluralization may not always be correct, as it simply adds an "s" to the end of the model name if we haven't added it ourselves. For this reason, it is recommended to create models with collection names already in plural.

2.3 Constraints and Validations

If we define a simple schema like the example of contacts above, we allow any type of value to be added to the fields of the documents. Thus, for example, we could have contacts without names or with negative ages. But with Mongoose, we can provide validation mechanisms that automatically discard documents that do not meet the specifications.

In the official Mongoose documentation, we can find a detailed description of the different validators we can apply. Here, we will only to describe the most important or common ones:

- The `required` validator allows defining that a certain field is mandatory.
- The `default` validator allows specifying a default value for the field if none is specified.
- The `min` and `max` validators are used to define a range of allowed values (minimum and/or maximum) for numeric data (inclusive).
- The `minlength` and `maxlength` validators are used to define a minimum or maximum size of characters for text strings.

- The `unique` validator indicates that the field in question does not allow duplicates (like an alternative key in a relational system). The Mongoose documentation specifies that this is not properly a validator but a helper for indexing, and depending on when it is indexed, it may not work properly.
- The `match` validator is used to specify a regular expression that the field must comply with.

We also can use the validator library which provides a goog number of methods to validate common expressions like emails, urls, etc. We will use it later.

Let's go back to our contacts schema. We will establish that the name and phone are mandatory, and we will only allow ages between 18 and 120 years (inclusive). Also, the name will have a minimum length of 1 character, and the phone will consist of 9 digits, using a regular expression, and it will be a unique key. We can use some more validators, such as `trim`, to clean up leading and trailing whitespace in text data. With all these constraints, the associated schema and model look like this:

```javascript
// models/contact.js
const mongoose = require('mongoose');

let contactSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    phone: {
        type: String,
        required: true,
        unique: true,
        trim: true,
        match: /^\d{9}$/
    },
    age: {
        type: Number,
        min: 18,
        max: 120
    }
});

let Contact = mongoose.model('contacts', contactSchema);
module.exports = Contact;
```

Now we have established the connection to the database and the schema for the data we are going to use. Now, we can add the model to our main file "`index.js`", and we can start performing some basic operations against the database.

```
// index.js
const mongoose = require('mongoose');
const Contact = require(__dirname + "/models/contact");

mongoose.connect('mongodb://127.0.0.1:27017/contacts');

// Now we can perform operations against the database
```

3 Adding Documents

If we want to insert a document into a collection, we must create an object of the corresponding model and call its `save` method. This method returns a promise, so we will use:

- A **then** code block for when the operation has been successful. In this block, we will receive the inserted object as a result, allowing us to examine its data if desired.
- A **catch** code block for when the operation could not be completed. We will receive an object with the error as a parameter, which we can examine for more information about it.

This same `then-catch` pattern will also be used with the rest of the operations later (searches, deletions, or modifications), although the returned result in each case will vary.

So, we would add a new contact to our test collection like this:

```
let contact1 = new Contact({
    name: "Mike",
    phone: "966112233",
    age: 45
});

contact1.save().then(result => {
    console.log("Contact added:", result);
}).catch(error => {
    console.log("ERROR adding contact:", error);
});
```

Add this code to the "`index.js`" file of our project, after connecting to the database. Run the application, and take a look at the result returned when everything works correctly. It will be something like this:

```
{
  name: 'Mike',
  phone: '966112233',
  age: 45,
  _id: new ObjectId("5a12a2e0e6219d68c00c6a00"),
  __v: 0
}
```

Note that we get the same fields that we defined in the schema (name, phone, and age), and two additional fields that we haven't specified:

- **`__v`** refers to the document's version. Initially, all documents start with version 0 when inserted, and this version can be modified when we update the document, as we will see later.
- **`_id`** is an auto-generated code by Mongo for any document in any collection.

Now, let's go to the `mongosh` console and examine the databases in the left panel:

```
$ mongosh

test> use contacts
switched to db contacts
contacts> db.contacts.find()
[
  {
    _id: ObjectId('65a42fa2c08351c80636c8eb'),
    name: 'Mike',
    phone: '966112233',
    age: 45,
    __v: 0
  }
]
```

If we attempt to insert an incorrect contact, we will jump to the catch block. For example, this contact is too old according to the schema definition:

```
let contact2 = new Contact({
    name: "Oldman",
    phone: "965123456",
    age: 200
});

contact2.save().then(result => {
    console.log("Contact added:", result);
}).catch(error => {
    console.log("ERROR adding contact:", error);
});
```

If we take a look at the produced error, we will see a lot of information, but among all that information, there is a `ValidatorError` message with details about the error:

```
ValidatorError: Path `age` (200) is more than the maximum allowed value
↪  (120)
```

This error tells us that the value provided for the “age” field (200) exceeds the maximum allowed value of 120 according to the schema definition.

3.1 About the Automatic ID

As seen in the previous insertion tests, each time a document is added to a collection, it is automatically assigned a property called `_id` with an autogenerated code. Unlike other database management systems (such as MariaDB/MySQL, for example), this code is not auto-incrementing but is a string. In fact, it is a 12-byte text that stores important information:

- The document’s creation time (timestamp), allowing us to obtain the exact moment (date and time) of its creation.
- The computer that created the document. This is particularly useful when scaling the application, and we have different Mongo servers accessing the same database. We can identify which of all the servers created the document.
- The specific process of the system that created the document.
- A random counter used to avoid any duplication in case the three previous values coincide in time.

There are specific methods to extract part of this information, specifically the creation moment, but we won’t use them in this course.

Despite having this significant advantage with this autogenerated ID, we can choose to create our own IDs and not use Mongo's (although this is not a good idea):

```
let contactX = new Contact({_id: 2, name: "John", age: 70, phone:
↪   "611885599"});
```

4 Finding Documents

If we want to search for any document or set of documents in a collection, we can use various methods.

4.1 Generic Search with `find`

The most general way to obtain documents is to use the static `find` method associated with the model in question. We can use it without parameters resulting in all documents of the collection as the promise result:

```
Contact.find().then(result => {
    console.log(result);
}).catch(error => {
    console.log("ERROR:", error);
});
```

4.2 Parameterized Search with find

We can also pass a set of search criteria as a parameter to find. For example, to search for contacts with the name "Mary" and age 29, we would do this:

```
Contact.find({name: 'Mary', age: 29}).then(result => {
    console.log(result);
}).catch(error => {
    console.log("ERROR:", error);
});
```

Any call to `find` **will return an array of results**, even if only one or none is found. It is important to keep this in mind to know how to access a specific element of that result. Not getting results will not trigger an error (it won't jump to the catch in that case).

We can also use some comparison operators in case we are not looking for exact data. For example, this query retrieves all contacts with the name “Mary” and ages between 18 and 40:

```
Contact.find({name: 'Mary', age: {$gte: 18, $lte: 40}})
.then(result => {
    console.log('Search result:', result);
})
.catch(error => {
    console.log('ERROR:', error);
});
```

Here you can find a detailed list of operators you can use in searches.

Moreover, parameterized search with find supports other syntax variants, such as using chained methods like where, limit, sort, etc., until you get the desired results in the desired order and quantity. For example, this query shows the first 10 contacts who are of legal age, sorted from oldest to youngest:

```
Contact.find()
  .where('age')
  .gte(18)
  .sort('-age')
  .limit(10)
  .then(...
```

4.3 Other Options: `findOne` or `findById`

There are other alternatives we can use to search for specific documents (and not a set or list of them). These are the `findOne` and `findById` methods. The first one is used similarly to `find`, with the same filtering parameters, but it only returns one (arbitrary) document that matches those criteria (not an array). For example:

```
Contact.findOne({name: 'Mike', age: 39})
.then(result => {
    console.log('Search result:', result);
})
.catch(error => {
    console.log('ERROR:', error);
});
```

The `findById` method is used, as the name suggests, to search for a document given its id. For example:

```
Contact.findById('5ab2dfb06cf5de1d626d5c09')
.then(result => {
    console.log('Search result by ID:', result);
})
.catch(error => {
    console.log('ERROR:', error);
});
```

In these methods, if the query produces no result, we will get `null` as a response, but the `catch` clause will not be triggered.

5 Deleting Documents

To delete documents from a collection, we can use different methods depending on the number of documents we want to delete and the conditions for deletion.

5.1 The `deleteOne` and `deleteMany` Methods

These methods delete documents that meet a certain filtering condition. The first one deletes the first document it finds, and the second one deletes all documents that meet the condition.

```
// Delete all contacts named Mike
Contact.deleteMany({name: 'Mike'}).then(result => {
    console.log(result.deletedCount, "documents deleted");
}).catch (error => {
    console.log("ERROR:", error);
});
```

As seen in the example, both methods return an object with a property called `deletedCount`, which shows how many documents have been deleted.

> **Important**: If we use `deleteMany` without specifying any condition, ALL documents in the affected collection will be deleted.

5.2 The `findOneAndDelete` and `findByIdAndDelete` Methods

The `findOneAndDelete` method searches for the document that meets the specified pattern (or the first one it finds that meets it) and deletes it. Additionally, it retrieves the deleted document in the result, so we could undo the operation afterward by adding it again.

```
Contact.findOneAndDelete({name: 'Mike'})
.then(result => {
    console.log("Contact deleted:", result);
}).catch (error => {
    console.log("ERROR:", error);
});
```

In this case, the result parameter is directly the deleted object. The `findByIdAndDelete` method searches for the document with the specified id and deletes it. It also gets the deleted object as a result.

```
Contact.findByIdAndDelete('5a16fed09ed79f03e490a648')
.then(result => {
    console.log("Contact deleted:", result);
}).catch (error => {
    console.log("ERROR:", error);
});
```

In the case of these two methods, if no element is found that meets the filtering criterion, `null` will be returned as the result. In other words, the catch clause will not be triggered for this reason. The catch clause would be triggered, for example, if we specify an id with an invalid format (not having 12 bytes).

> NOTE: In earlier versions of Mongoose, these methods were named `findOneAndRemove` and `findByIdAndRemove`, respectively, and were replaced by these others in more recent versions.

6 Document Modifications or Updates

To make modifications to a document in a collection, we can also use various static methods.

6.1 The `findByIdAndUpdate` Method

The `findByIdAndUpdate` method will search for the document with the specified `id` and replace the fields based on the criteria we provide as the second parameter.

In this link you can check the update operators that we can use in the second parameter of this method.

The most common of all is `$set`, which takes an object with key-value pairs that we want to modify in the original document. For example, here we replace the name and age of a contact with a specific `id`, leaving the phone unchanged:

```
Contact.findByIdAndUpdate('5a0e1991075e9407c4da8b0a',
    {$set: {name:'Mike Smith', age: 40}}, {new:true})
.then(result => {
    console.log("Modified contact:", result);
}).catch (error => {
    console.log("ERROR:", error);
});
```

As an alternative to the above code, we can omit the `$set` keyword, and the code would look like this:

```
Contact.findByIdAndUpdate('5a0e1991075e9407c4da8b0a',
    { name: 'Mike Smith', age: 40 } , { new: true })
.then(result => {
    console.log("Modified contact:", result);
}).catch (error => {
    console.log("ERROR:", error);
});
```

The third parameter that `findByIdAndUpdate` receives is a set of additional options. For example, the new option used in this example indicates whether we want to get the new modified object as a result (true) or the old one before modification (false, useful for undo operations). We can also pass the `runValidators` option as the third parameter, which allows us to validate the fields we modify. By default, this option is disabled, and to validate the fields, we must set it to true.

With `runValidators`, when trying to modify a contact with an incorrect age, it jumps to the catch block. For example, this contact is too young, according to the schema definition:

```
Contact.findByIdAndUpdate('5a0e1991075e9407c4da8b0a',
    { name: 'Mary Ann', age: 10 } , { new: true, runValidators: true })
.then(result => {
    console.log("Modified contact:", result);
}).catch (error => {
    console.log("ERROR:", error);
});
```

6.2 The updateOne and updateMany Methods

These methods can be used to update the data of the first document that matches the search condition or all documents that match that condition, respectively. For example, the following statement sets the age to 20 for the first contact found with the name “Nacho”:

```
Contact.updateOne({name: 'Mike'}, {$set: {age: 20}})
  .then(...
```

`updateMany` updates the data for all contacts with that name:

```
Contact.updateMany({name: 'Mike'}, {$set: {age: 20}})
  .then(...
```

Both methods are useful if we want to search for documents based on fields other than their primary identifier. Otherwise, it is preferable to use `findByIdAndUpdate`.

6.3 Update Document Version

We have seen that, among the attributes of a document, in addition to the id autogenerated by Mongo, a version number is created in an attribute `__v`. This version number refers to the version of the document itself so that, if it is later modified (for example, with a call to `findByIdAndUpdate`), it can also be indicated with a change in version that this document has changed since its original version. If we wanted to do that with the previous example, it would be enough to add the `$inc` operator to indicate that it increases the version number, for example, by one:

```
Contact.findByIdAndUpdate('5a0e1991075e9407c4da8b0a',
    {$set: {name:'Mike Smith', age: 40},
    $inc: {__v: 1}}, {new:true})
.then(...
```

7 Mongoose and Promises

Previously, we mentioned that operations like `find`, `save`, and the other methods we've used with Mongoose return a promise, but that's not entirely accurate. What these methods return is a `thenable`, an object that can be treated with the corresponding `then` method. However, there are alternative ways to call these methods, and we can use one or the other as needed.

7.1 Calls as Simple Asynchronous Functions

The methods provided by Mongoose are simply asynchronous tasks, this is to say we can call them and define a response callback that will be executed when the task is completed. To do this, we add an additional parameter to the method, the callback, with two parameters: the error that will occur if the method does not run successfully, and the returned result if the method runs without issues. These two parameters must be indicated in this same order (first the error, and then the correct result). If, for example, we want to search for a contact by its ID, we can do something like this:

```
Contact.findById('35893ad987af7e87aa5b113c', (error, contact) => {
    if (error)
        console.log("Error:", error);
    else
        console.log(contact);
});
```

Now, let's do something more complex: we search for the contact by its ID, once completed, we increase its age by one year, and save the changes. In this case, the code might look like this:

```
Contact.findById('35893ad987af7e87aa5b113c', (error, contact) => {
    if (error)
        console.log("Error:", error);
    else {
        contact.age++;
        contact.save((error2, contact2) => {
            if (error2)
                console.log("Error:", error2);
            else
                console.log(contact2);
        });
    }
});
```

As we can see, when linking an asynchronous call (`findById`) with another (`save`), what happens is a **nesting of callbacks**, with their corresponding if..else structures. This phenomenon is known as ***callback hell*** or ***pyramid of doom*** because it produces a rotated pyramid on the left side of the code (with its peak pointing to the right), which will be larger as we add more calls. In other words, we will be indenting the code more and more by nesting calls within calls, and this management can become difficult to handle.

7.2 Calls as Promises

Let's go back to what we know how to do. How would we link the two previous operations using promises? Remember: find a contact by its ID and increase its age by one year.

We could also create a callback hell by nesting then clauses, with something like this:

```
Contact.findById('35893ad987af7e87aa5b113c')
.then(contact => {
    contact.age++;
    contact.save()
    .then(contact2 => {
        console.log(contact2);
    }).catch(error2 => {
        console.log("Error:", error2);
    });
}).catch (error => {
    console.log("Error:", error);
});
```

However, promises allow chaining then clauses without nesting them, leaving a single catch block at the end to catch any errors that may occur in any of them. To do this, it is enough that within a then, the result of the next promise is returned. The previous code could be rewritten like this:

```
Contact.findById('35893ad987af7e87aa5b113c')
.then(contact => {
    contact.age++;
    return contact.save();
}).then(contact => {
    console.log(contact);
}).catch (error => {
    console.log("Error:", error);
});
```

This form is cleaner and clearer when we want to perform complex operations. However, it can be further simplified using `async/await`.

7.3 Calls with Async/Await

The **`async/await`** specification allows calling a series of asynchronous methods synchronously and waiting for them to finish before moving on to the next task. The only requirement to do this is that these calls must be made from within an asynchronous function, declared with the `async` keyword.

To perform the previous example, we need to declare an asynchronous function with any name we want (for example, `updateAge`), and inside it, call each asynchronous function preceded by the `await` keyword. If the call is going to return a result (in this case, the result of the promise), it can be assigned to a constant or variable. With this, the code can be rewritten like this, and we simply call the updateAge function when we want to execute it:

```
async function updateAge() {
    let contact = await Contact.findById('35893...');
    contact.age++;
    let savedContact = await contact.save();
    console.log(savedContact);
}

updateAge();
```

We would still need to handle the error section: in the two previous cases, there was a `catch` clause or an error parameter to check and display the corresponding error message. How do we manage this with async/await? By using `await`, we are converting asynchronous code into synchronous code, and therefore, error handling is a simple exception handling with try..catch:

```
async function updateAge() {
    try {
        let contact = await Contact.findById('35893...');
        contact.age++;
        let savedContact = await contact.save();
        console.log(savedContact);
    } catch (error) {
        console.log("Error:", error);
    }
}
```

```
updateAge();
```

7.4 Which One to Choose?

Any of these three options can be used in any situation, as needed. In this course, we will use the promise approach for simple tasks, as it allows for a clear separation of correct and incorrect execution code, and we will use the `async/await` specification for complex or linked tasks, where one method depends on the result of the previous one to start.

7.5 Collection Relationships

Our contact database is very simple, with only a single collection called `contacts`, and its documents have three fields: `name`, `phone`, and `age`. We are going to add more information, and for this, we'll continue working on the `ContactsMongo` project.

7.6 Define a Simple Relationship

If we want to add, for each contact, their favorite restaurant, so multiple contacts can have the same favorite restaurant. For this, we can define this schema and model (in a file called `models/restaurant.js`):

```
// models/restaurant.js
let restaurantSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    address: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    phone: {
        type: String,
        required: true,
        unique: true,
        trim: true,
        match: /^\d{9}$/
    }
});

let Restaurant = mongoose.model('restaurants', restaurantSchema);
module.export = Restaurant;
```

And we associate it with the contact schema with a new field:

```
// models/contact.js
let contactSchema = new mongoose.Schema({
    name: {
        ...
    },
    phone: {
        ...
    },
    age: {
        ...
    },
    favouriteRestaurant: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'restaurants'
    }
});


let Contact = mongoose.model('contacts', contactSchema);
module.exports = Contact;
```

Note that the data type of this new field is `ObjectId`, indicating that it references an `id` from a document in this or another collection. Specifically, through the `ref` property, we indicate which model or collection this id refers to (the restaurants model, which translates to the restaurants collection in MongoDB).

7.7 Defining a Multiple Relationship

Let's take one step further and define a relationship that allows associating multiple elements from one collection with an element from another collection (or the same collection). For example, let's allow each contact to have a set of pets. We define a new schema for pets, storing their name and type (dog, cat, etc.), in the file `models/pet.js`.

```
// models/pet.js
let petSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    type: {
        type: String,
        required: true,
        enum: ['dog', 'cat', 'others']
    }
});

let Pet = mongoose.model('pets', petSchema);
module.exports = Pet;
```

Observe how the enum validator in a schema can be used to force a certain field to only accept certain values.

To allow a contact to have multiple pets, we add a new field in the contact schema that will be an array of ids, associated with the previously defined pets model:

```
// models/contact.js
let contactSchema = new mongoose.Schema({
    name: {
        ...
    },
    phone: {
        ...
    },
    age: {
        ...
    },
    favouriteRestaurant: {
        ...
    },
    pets: [{
        type: mongoose.Schema.Types.ObjectId,
        ref: 'pets'
    }]
});

let Contact = mongoose.model('contacts', contactSchema);
module.exports = Contact;
```

In this case, notice how the way to define the reference to the pets collection is the same (setting the data type as `ObjectId`, referencing the pets model), but additionally, the data type of this `pets` field is an array (specified by the brackets when defining it).

7.8 Inserting Related Elements

If we want to insert a new contact and, at the same time, specify their favorite restaurant and/or pets, we should do it in parts, as would happen in a relational system:

1. First, we would add the favorite restaurant to the restaurant collection, and/or the pets to the pet collection (unless they already exist, in which case, we would get their id):

```
let restaurant1 = new Restaurant({
    name: "La Tagliatella",
    address: "C.C. San Vicente s/n",
    phone: "965678912"
});
restaurant1.save().then(...

let pet1 = new Pet({
    name: "Otto",
    type: "dog"
});
pet1.save().then(...
```

2. Then, we would add the new contact specifying the id of their favorite restaurant, added previously, and/or the ids of their pets in an array:

```
let contact1 = new Contact({
    name: "Mike",
    phone: 677889900,
    age: 40,
    favouriteRestaurant: '5acd3c051d694d04fa26dd8b',
    pets: ['5acd3c051d694d04fa26dd90',
           '5acd3c051d694d04fa26dd91']
});
contact1.save().then(...
```

Certainly, in a "real" operation, we won't have to manually add the ids of related documents. It would be enough to choose them from some dropdown to get their id.

7.9 On Referential Integrity

Referential integrity is a concept related to relational databases, ensuring that the values of a foreign key will always exist in the table it refers to. Applied to a Mongo database, we might think that the ids of a field linked to another collection should exist in that collection, but it doesn't have to be the case.

Continuing with the previous example, if we try to insert a contact with a restaurant id that doesn't exist in the restaurant collection, it will let us do it, as long as that id is valid (i.e., has a length of 12 bytes). Therefore, it's up to the programmer to ensure that the ids used in insertions that involve a reference to another collection actually exist. To facilitate the task, there are some libraries on the NPM repository that we can use, such as this one, although its use goes beyond the contents of this course, and we won't see it here.

In the case of deletion, we might encounter a similar situation: if, following the example of contacts, we want to delete a restaurant, we should be careful with the contacts that have it assigned as their favorite restaurant, as the id will cease to exist in the restaurant collection. Thus, it would be advisable to choose between these two options, although both require manual handling by the programmer:

- Deny the operation if there are contacts with the selected restaurant.
- Reassign (or set to null) the favorite restaurant of those contacts before deleting the selected restaurant.

8 Subdocuments

Mongoose also provides the possibility of defining **subdocuments**. Let's see a specific example of this, and for that, we are going to create an alternative version of our contacts example. Copy the project folder that we have been working on so far and name the new copy ContactsMongo_v2.

In this new project, in our `index.js` file, let's connect to a new database, which we'll call `contacts_subdocuments`, to avoid interfering with the previous database:

```
mongoose.connect('mongodb://127.0.0.1:27017/contacts_subdocuments');
```

And let's regroup the three schemas we have created so far (restaurants, pets, and contacts), and merge them into the `contacts` schema. We will leave only one file in the models folder, which will be `contact.js`, with this content (we omit some of the code that is the same as the previous example):

```
// Restaurants
let restaurantSchema = new mongoose.Schema({
    ... // Code for the restaurant schema
});

// Pets
let petSchema = new mongoose.Schema({
    ... // Code for the pet schema
});

// Contacts
let contactSchema = new mongoose.Schema({
    name: {
        ...
    },
    phone: {
        ...
    },
    age: {
        ...
    },
    favouriteRestaurant: restaurantSchema,
    pets: [petSchema]
});

let Contact = mongoose.model('contacts', contactSchema);
module.exports = Contact;
```

Notice the lines referring to the `favouriteRestaurant` and `pets` properties. This is how you associate an entire schema as the data type of a field in another schema. Thus, we turn the schema into a part of the other, creating **subdocuments** within the main document. Also, note that we haven't defined models for restaurants or pets since they will not have their own collection now.

At first glance, a subdocument may seem somewhat equivalent to defining a relationship between collections. However, the main difference between a subdocument and a relationship between documents from different collections is that the subdocument is embedded within the main document and is different from any other object that may exist in another document, even if their fields are the same. On the contrary, in the simple relationship seen earlier between restaurants and contacts, a favorite restaurant could be shared by multiple contacts simply by linking to the same restaurant id. But this way, we create the restaurant for each contact, differentiating it from other restaurants, even if they are the same. The same would happen with the array of pets: the pets would be different for each contact, even if we wanted them to be the same or shareable.

8.1 Inserting documents with subdocuments

If we want to create and save a contact that contains the favorite restaurant and pets as subdocuments, we can create the entire object and do a single save:

```
let contact1 = new Contact({
    name: 'Mike',
    phone: 966112233,
    age: 39,
    favouriteRestaurant: {
        name: 'La Tagliatella',
        address: 'C.C. San Vicente s/n',
        phone: 961234567
    }
});
contact1.pets.push({name:'Otto', type:'dog'});
contact1.pets.push({name:'Piolín', type:'other'});
contact1.save().then(...
```

In this example, two possible ways of filling in the subdocuments of the main document are shown: on-the-fly when creating the document (case of the restaurant) or afterwards by accessing the fields and assigning them values (case of the pets).

In the database, we will see that there is only one collection, contacts, and by examining the inserted elements, we will see that they contain the defined subdocuments embedded.

8.2 When to define relationships and when to use subdocuments?

The answer to this question may seem complex or evident, depending on how we have understood the concepts seen so far, but let's try to give some basic rules to distinguish when to use each concept:

- Use relationships between collections when you want to share the same document from one collection among multiple documents from another. Thus, in the case of favorite restaurants, it seems logical to use a relationship between collections based on the restaurant id, allowing multiple contacts to share the same instance of a favorite restaurant.
- Use subdocuments when information sharing does not matter, or when simplicity of defining an object outweighs associativity between collections. In other words, applied to the example of pets, if you want to easily access the pets of a contact, regardless of whether another contact has the same pets, you will use subdocuments.

In the case of subdocuments, there is, therefore, a pending subject: the potential duplication of information. If two people have the same pet, we must create two identical objects for both people, thus duplicating the data of the pet. However, this data duplication will make it easier to access the pets of a person without resorting to other tools that we will see next.

9 Advanced Queries

Now that we know how to define different types of linked collections, let's see how to define queries that use these relationships to extract the information we need. We will continue working with the ContactsMongo_v1 project in this case.

9.1 Populations (`populate`)

The act of relating documents from one collection to documents from another through their corresponding ids allows us to obtain, in a single listing, information from both collections, though an intermediate step is needed. For example, if we want to retrieve all information about our contacts, related to the restaurant and pet collections, we can do something like this:

```
Contact.find().then(result => {
    console.log(result);
});
```

However, this command is limited to displaying the ids of the favorite restaurants and pets, not their complete data. To achieve this, we have to use a very useful method provided by Mongoose called `populate`. This method allows us to incorporate the information associated with the specified model. For instance, if we want to include all information about the favorite restaurant of each contact in the previous list, we do something like this:

```
Contact.find().populate('favouriteRestaurant').then(result => {
    console.log(result);
});
```

If we had more related fields, we could link multiple populate statements, one after another, to populate them all. For example, this is how we would populate both the restaurant and the pets:

```
Contact.find()
.populate('favouriteRestaurant')
.populate('pets')
.then(result => {
    console.log(result);
});
```

There are other options for populating fields. For example, we might want to populate only part of the information, such as just the name of the restaurant. In that case, we use additional parameters in the populate method:

```
Contact.find()
.populate('favouriteRestaurant', 'name')
...
```

9.2 Queries that involve multiple collections

Establishing a general query on a collection is simple, as we have seen in previous sessions. We can use the find method to retrieve documents that meet certain criteria, or alternatives like `findOne` or `findById` to get the document that satisfies the filtering.

NoSQL databases, such as MongoDB, are not designed to query information from multiple collections, which, in part, encourages the use of independent collections based on subdocuments to add additional information.

Suppose we want to obtain the data of favorite restaurants for contacts who are over 30 years old. If we had an SQL database, we could solve this with a query like the following:

```
SELECT * FROM restaurants
WHERE id IN
(SELECT favouriteRestaurant FROM contacts
WHERE age > 30)
```

However, this is not possible in MongoDB, or at least not so straightforward. It would be necessary to split this query into two parts: first, get the ids of the restaurants for people over 30 years old, and then, with another query, get the data of those restaurants. It could look something like this:

```
Contact.find({age: {$gt: 30}}).then(resultContacts => {
    let restaurantIds = resultContacts.map(contact =>
    ↪   contact.favouriteRestaurant);
    Restaurant.find({_id: {$in: restaurantIds}})
    .then(finalResult => {
        console.log(finalResult);
    });
});
```

Notice that the first query retrieves all contacts over 30 years old. Once obtained, we map to get only the ids of the favorite restaurants, and we use that list of ids in the second query to get the restaurants whose id is in that list.

9.3 Other Options in Queries

When we use the `find` method or similar ones, there are additional options that allow specifying which fields to retrieve, sorting criteria, maximum result limits, etc. We have seen some examples about that. Let's now explore in more detail some of these options:

- If we want to specify exactly **which fields to retrieve** in the listing, we can provide a string of field names in the find parameters. Alternatively, we can use the `select` method, which is chained to the normal find call, specifying the fields to retrieve. The following examples achieve the same result: displaying the name and age of contacts over 30 years old:

```
Contact.find({age: {$gt: 30}}, 'name age').then(...
Contact.find({age: {$gt: 30}}).select('name age').then(...
```

- To **sort** the listing by a specific criterion, we can chain the find call with the `sort` method. We specify the field to sort by and the order (1 for ascending order, -1 for descending order). The following listing shows contacts sorted from oldest to youngest. An equivalent alternative is to prepend the minus sign - to the field to indicate descending order:

```
Contact.find().sort({age: -1}).then(...
Contact.find().sort('-age').then(...
```

- To **limit** the number of results obtained, we use the `limit` method, specifying how many documents to retrieve. The following example shows, from oldest to youngest, the first 5 contacts:

```
Contact.find().sort('-age').limit(5).then(...
```