Exploration — Advanced Operations Using Mongoose

This reading is optional. You will need it for the assignment.

Introduction

We can now perform basic CRUD operations on a collection using Mongoose. In this exploration, we look at how to create more advanced filters that we can use in querying collections, as well as the details of some Mongoose methods for updating data.

Boolean Operators for Complex Conditions

When querying databases, many times an app is interested in documents that meet very specific criteria. Like other DBMSs, MongoDB supports specifying conditions using many different Boolean operators, such as AND, OR, NOR, etc.

Filtering Documents Using AND

We can require that the result of query contains only those documents that match **all of the specified conditions** by calling **the and method on a query (https://mongoosejs.com/docs/api.html#query_Query-and)** with an array containing all the filters we want to apply on the documents. Note that if the array is empty then all the documents will match the query.

Example: and

Consider we want to find all movies that were released in 2004 and the language was English. The filters would be

```
let filters = [{ year: 2004}, {language: 'English'}]
```

We can call use these filters to return the matching documents as follows:

```
const findMoviesUsingAnd = async (filters) => {
    const query = Movie.find();
    if(filters.length > 0){
      query.and(filters);
    }
    return query.exec();
}
```

In the previous exploration, to retrieve documents we had called `Movie.find()` with **an object** to get back a `Query` object.

- When we passed an empty object to `Movie.find()`, the query returns all the documents in the collection.
- When we passed a non-empty object, e.g., `{year: 2018}`, to the query then the documents matching that **one filter** were returned.

Contrast this with the above example where we call `Movie.find()` without an argument to get a `Query` object.

- If the argument `filters` is a non-empty array, then we pass this non-empty **array of filter objects** to the method `query.and()` and then execute the query to get those documents that match **all the filters** in that array AND-ed together.
- If the argument `filters` is an empty array, we don't call `query.and()` and simply execute the query which will return all the documents in the collection.

Filtering Documents Using OR

We can require that the result of query contains only those documents that match **any of the specified conditions** by calling **the or method on a query (https://mongoosejs.com/docs/api.html#query_Query-or)** with an array containing all the filters we want to apply on the documents. Note that if the array is empty then all the documents will match the query.

Example: or

Consider we want to find all movies that were released in 2004 or for which the language was English. The filters would be

```
let filters = [{ year: 2004}, {language: 'English'}]
```

We can call use these filters to return the matching documents as follows:

```
const findMoviesUsingOr = async (filters) => {
    const query = Movie.find();
    if(filters.length > 0){
      query.or(filters);
    }
    return query.exec();
}
```

Updating Existing Documents

Mongoose provides a few methods for updating existing documents. In the previous exploration, we gave an example using `replaceOne`. That example required that we already have access to the document, which we can do by querying to find the document first. If we don't have access to the full

document or don't want to run a separate find query, then we can make use of other mongoose methods such as `updateOne` or `findOneAndUpdate` to partially update a document. We describe how to use `findOneAndUpdate.`

findOneAndUpdate

Looking at the documentation for the method **findOneAndUpdate (https://mongoosejs.com/docs/api.html#model_Model.findOneAndUpdate)** , we can see that it takes three parameters: conditions (object), update (object) and options (object).

- `conditions`
  - This parameter is of type object and is used to match documents.
  - If multiple documents match the condition, then one matching document is picked at random.
  - To match a document with multiple conditions AND-ed together, specify all the conditions as properties of this object.
  - Examples:
    - To match a document using the value of `_id` we can use the following condition
      - `let condition = {"_id": document_id};`
    - To match a document whose property `year` has the value 2008 and the property `language` has the value English, we can use the following condition
      - `let conditions = {"year": 2008, "language": "English"};`
- `update`
  - This parameter is of type object and contains the updates that we would like to make to the document.
  - To update a property of the document, specify it as a property of this object.
  - To update multiple properties of the document, specify each of these as properties of this object.
  - Any property that is not specified in this object is left unchanged.
  - Examples:
    - To update only the property `year` of a document to the value 2019, while leaving all the other properties of the document unchanged, we specify the following value for the `update` parameter
      - `let update = {"year": 2019}`;
    - To update the property `year` of a document to the value 2019 and the property `language` to the value Punjabi, while leaving all the other properties of the document unchanged, we specify the following value for the `update` parameter
      - `let update = {"year": 2019, "language": "Punjabi"};`
- `options`
  - Lastly the `options` parameter allows us to specify special conditions.
  - An example case of this would be if we wanted to return the updated document to the user after the query. For this we could specify the `new` option in our query as follows:
  - `let options = {new: true};`

This function returns a promise.

- If no document was found matching the `conditions` argument, then the value of the promise resolves to null .
- If a document was found matching the `conditions` argument, then the promise resolves to that document.

Summary

MongoDB supports a wide variety of operations that are exposed by various Mongoose methods. As you build more complex API that use Mongoose, review the capabilities of MongoDB and Mongoose to find out the API that best matches the functionality you want to implement.