Exploration — Using Mongoose to Implement CRUD Operations

Introduction

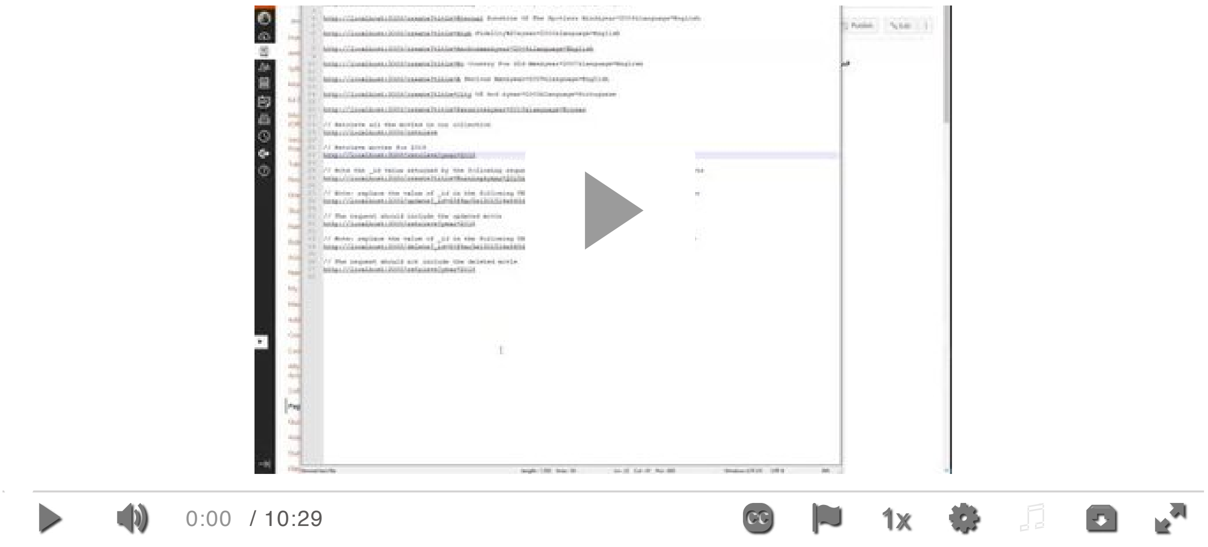
To execute CRUD (create, retrieve, update, delete) operations on a collection in MongoDB, **Mongoose provides a large number of static methods (https://mongoosejs.com/docs/api/model.html)** on the model class generated by Mongoose for a schema. In this exploration, we will study some of these methods and use them to build a web service that provides endpoints to perform the CRUD operations on a movie database stored in MongoDB. We will build this web service by:

- Implementing functions in the **model** layer to perform each of the 4 CRUD operations
- Adding route handlers in the **controller** layer that call the appropriate function in the model layer as an HTTP request is received to perform a particular CRUD operation.

For now, we use a simple approach of adding route handlers with the URLs `create`, `retrieve`, `update` and `delete` with the `GET` HTTP method to perform create, retrieve/read, update and delete operations. The parameters we need to pass to functions in our models layer will be provided as query parameters in the HTTP request to the corresponding route handler.
The **test_requests.txt** file from the **movies-api.zip (https://canvas.oregonstate.edu/courses/1879154/files/94479023?wrap=1)** (https://canvas.oregonstate.edu/courses/1879154/files/94479023/download?download_frd=1) (introduced in the previous Exploration) will allow you to experiment with these CRUD implementations. It provides sample HTTP requests that you can send from your browser to test the Movies app.

NOTE: In an *upcoming* module we will implement a more sophisticated web service that provides CRUD operations via `POST`, `GET`, `PUT` and `DELETE` HTTP methods. These methods, below, are useful if you'll be completing the optional Assignment 6. Regardless, be sure you understand how these two files' variables, functions, filters, and results work, so that Assignment 7 will make sense. **Do not skip the Exercises.**

Create



Once the schema has been defined and we've compiled the model using Mongoose in the model.mjs file, we can then create a document from a new variable, async, and the method **save (https://mongoosejs.com/docs/api.html#document_Document-save)** . Unlike many other CRUD methods which are static, `save` is called on an instance of the model class. The method returns a promise which, if fulfilled, resolves to the document that was saved. By default, a unique ID with a string value is automatically assigned to a newly created document and is available as the property `_id`.

Create

The following variable creates a new document in the collection `movies` with the specified value of `title`, `year` and `language` in the model.mjs file.

```
const createMovie = async (title, year, language) => {
    const movie = new Movie({
        title: title,
        year: year,
        language: language
    });
    return movie.save();
}
```

We can then get the new movie document using the asyncHandler within the controller.mjs file:

```
app.get ('/create', asyncHandler(async (req,res) => {
    const movie = await movies.createMovie(
        req.query.title,
        req.query.year,
        req.query.language
        )
    res.send(movie);
}));
```

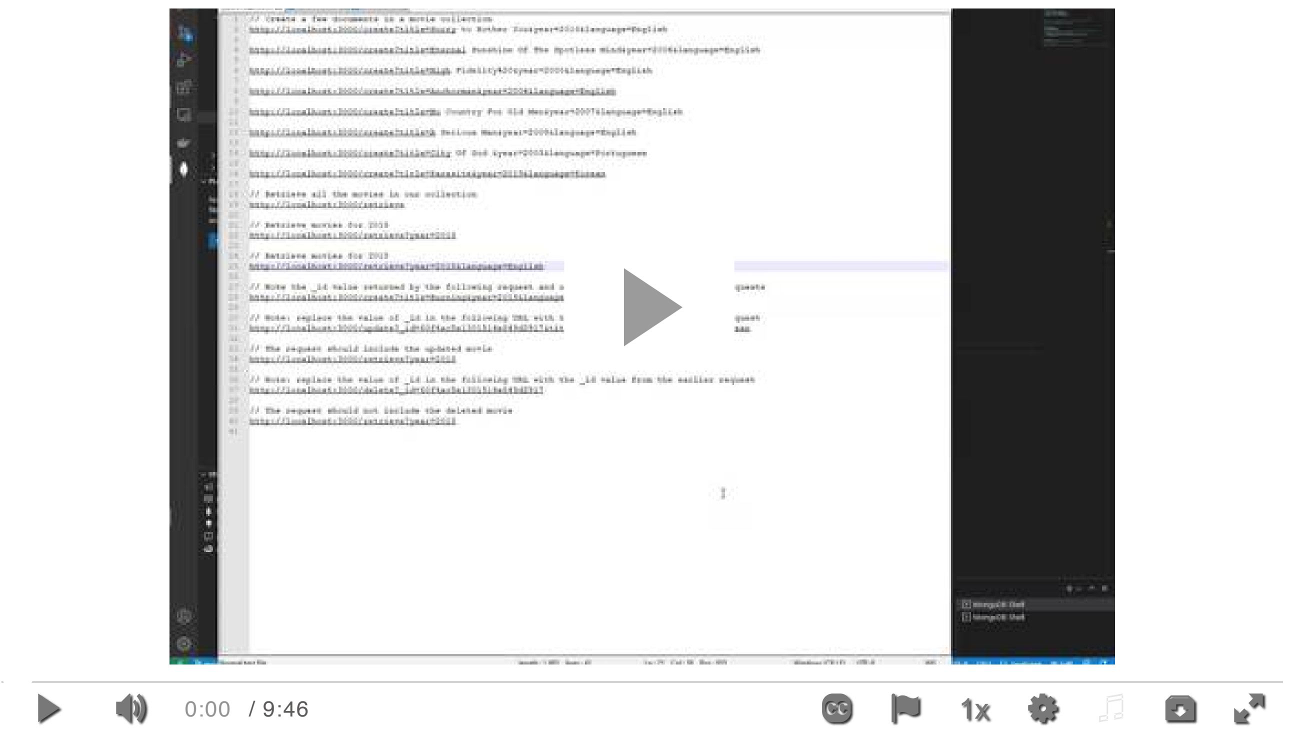
Exercise: create

Use the test-requests.txt file to create 10 documents in the movies collection.

Retrieve

For retrieving documents, Mongoose provides several static methods on the model class it compiles from the schema definition.

Retrieve Documents Using find



The following variables in the model.mjs file set-up the ability to retrieve the list of newly-created documents from the `movies` collection, based on a filter or the `_id`. Each document gets the automatic `_id`, along with values for `title`, `year` and `language.`

```
// Retrieve based on a filter and return a promise.
const findMovies = async (filter) => {
    const query = Movie.find(filter);
    return query.exec();
}

// Retrieve basd on the ID and return a promise.
const findById = async (_id) => {
    const query = Movie.findById(_id);
    return query.exec();
}
```

Then, in the controller.mjs file, we devise a filter to look at each parameter of the data and if it is not undefined, then return that filter.

```
  function movieFilter(req) {
    let filter = {};
    if (req.query._id !== undefined) {
        filter._id = req.query._id;
    } if (req.query.title !== undefined) {
         filter.title = req.query.title;
    } if (req.query.year !== undefined) {
         filter.year = req.query.year;
    } if (req.query.language !== undefined) {
        filter.language = req.query.language ;
    }
    return filter;
}
```

We then use that filter in a GET request with the asyncHandler, and send the result to the screen:

```
app.get ('/retrieve', asyncHandler(async (req,res) => {
    const filter = movieFilter(req);
    const result = await movies.findMovies(filter)
    res.send(result);
}));
```

Exercise: retrieve

Use the test-requests.txt file to retrieve the 10 documents, then retrieve a list of movies from 2022.

More detail

The `find` method can be used to retrieve all the documents in a collection or retrieve a subset of documents that match specific criteria. Calling `find` does not execute a retrieval operation. Instead, calling this method returns a `Query` object. Then calling the method `exec` on the `Query` object actually executes the retrieval operation on MongoDB.

find (https://mongoosejs.com/docs/api.html#model_Model.find) takes the following optional parameters:

- `filter`
  - This parameter is an object and is used to match documents.
  - Mongoose creates a Boolean condition for every property of the object and checks this condition on each document.
  - Mongoose converts the filter into a Boolean condition and checks this condition on each document.
  - If any condition evaluates to false for a document, then it will not be included in the result.
  - If every condition in the filter evaluates to true for a document, that document will be included in the result. But even if one condition evaluates to false, then the document will not be included in the result.
  - If this parameter is not provided, it defaults to an empty condition which evaluates to true for every document in the collection.
- `projection`

  - A space-separated list of the properties of the document which we want to be included in the result.
  - If it is not provided, all properties of the document are included.
- `options`
  - This parameter allows further tailoring of what the result should look like.

Instead of passing all the parameters at once to the `find` method, an alternate coding pattern is typically used for easier understanding. In this pattern, we call the `find` method to create a `Query` object and then call various methods on the `Query` object to specify additional filters, projection and options.

Consider the function `findMovies` shown below.

```
findMovies = async (filter, projection, limit) => {
    const query = Movie.find(filter)
        .select(projection)
        .limit(limit);
    return query.exec();
}
```

If we want to retrieve:

- the movies with value of year equal to 2018,
- but only want the property `title` (besides `_id`) of these documents in the result,
- and limit the number of documents in the result to a maximum of 5 documents,

then we can call this function as follows:

```
findMovies({ year: 2018 }, 'title', 5)
```

Here is how this works:

- We call `find` with the value of `filter` parameter set to `{ year: 2018 }`.
  - This returns a `Query` object.
  - Only documents with the value of 2018 for the property `year` will match the query.
  - If we wanted all documents to match the query, we would have passed an empty object, i.e., `{}`.
  - Note that the `Query` object also has other methods that can be used to build up a filter.
  - It is also possible to create complex conditions using Boolean operators such as AND, OR, NOT, etc.
- We call the `select` method on the `Query` object to specify which properties to include in the result.
  - We call the method with the value `title` telling the query to only include the property `title` in the result.
  - Note that the `_id` property is always automatically included in the result.

  - To include multiple properties in the result, we specify these properties separated by a space.
    - For example, to include the properties `title` and `year` in the result, we will call `select` with the value `'title year'`.
  - To include all the properties in the result, we can either not call the `select` method on the query object or call it with an empty string, i.e., `''`.
- We call the `limit` method on the `Query` object to specify the maximum number of documents we want in the result.
  - To include all the documents matching the filtering criteria, we can either not call the `limit` method on the query object or call it with the value `0`.

Other useful methods for retrieval include:

- `findOne`
  - This method creates a query which when executed will return at most one document.
  - We can set filters and projections on the query similar to those for `find`.
- `findById`
  - This method is just a handy way of calling `findOne` when we want to find a document by its ID.

- **Spread syntax (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax)** (ellipses ...) for filtering (in place of *if* statements)

  -
    ```
    const filter = {
            ...(req.query._id && {_id: req.query._id}),<
    ```
    and so on
- **push (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push)**

  -
    ```
    let filter = [];
        if (req.query._id !== undefined) {
            filters.push({_id: req.query._id })
        }
    ```
    and so on.

Update

Mongoose provides many different methods for updating documents. These vary along different dimensions. One important dimension to consider is whether a method completely replaces the document with the values that are provided or just updates a subset of the properties.

In the model.mjs file, we provide a variable for updating with async and use updateOne to filter through the data, then return a result that tells us how many were modified:

```
const updateMovie = async (filter, update) => {
   const result = await Movie.updateOne(filter, update);
   return result.modifiedCount;
};
```

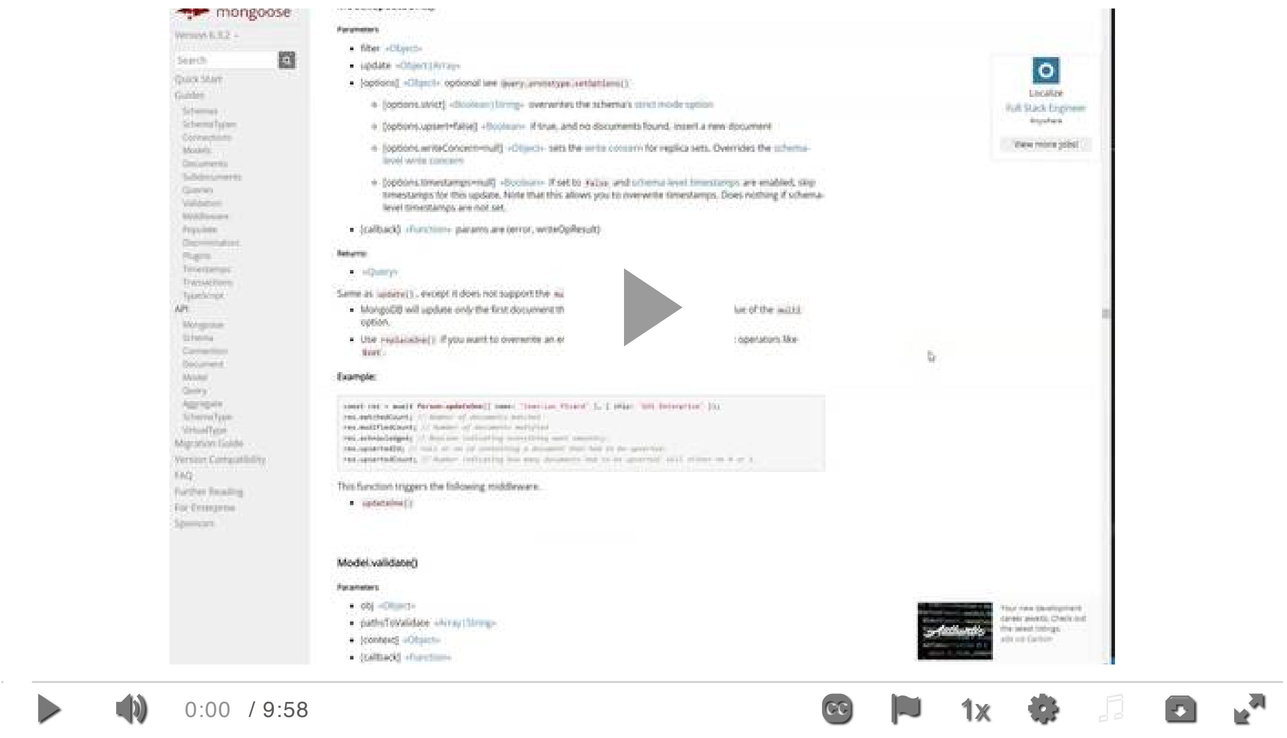
And, in the controller.mjs file, we first find the movie by `_id` and then filter through the data, checking to see if it exists. If it exists, then update the movie and print the number of updated documents. But if not, then send an error message.

```
app.get('/update', (req, res) => {
    // Find the movie via the _id and if found, filter,
    // make the update, and print the number of updated documents.
    movies.findById(req.query._id)
        .then(movie => {
            if (movie !== null) {
                const update = {};
                if (req.query.title !== undefined) {
                    update.name = req.query.name;
                }
                if (req.query.year !== undefined) {
                    update.year = req.query.year;
                }
                if (req.query.language !== undefined) {
                    update.language = req.query.language;
                }
                movies.updateMovie({ _id: req.query._id }, update)
                    .then(updateCount => {
                        res.send({ updateCount: updateCount });
                    })
                    .catch(error => {
                        console.error(error);
                        res.send({ Error: 'The document was not updated.'});
                    });
            } else {
                res.send({ Error: 'The document was not found.' });
            }
        })
        .catch(error => {
            console.error(error);
            res.json({ Error: error });
        });

});
```

Exercise: update

Use the test-requests.txt file to create, retrieve, and update a newly-created document in the movies collection.



Parameters:

- filter «Object»
- update «Object|Array»
- [options] «Object» optional see Query.prototype.setOptions()
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option
  - [options.upsert=false] «Boolean» if true, and no documents found, insert a new document
  - [options.writeConcern=null] «Object» sets the write concern for replica sets. Overrides the schema-level write concern
  - [options.timestamps=null] «Boolean» If set to false and schema-level timestamps are enabled, skip timestamps for this update. Note that this allows you to overwrite timestamps. Does nothing if schema-level timestamps are not set.
- [callback] «Function» params are (error, writeOpResult)

Returns:

- «Query»

Same as update(), except it does not support the [illegible]

- MongoDB will update only the first document th [illegible] ue of the multi option.
- Use replaceOne() if you want to overwrite an e [illegible] operation like $set.

Example:

```
const res = await Person.updateOne({ name: 'Jean-Luc Picard' }, { ship: 'USS Enterprise' });
res.matchedCount; // [illegible]
res.modifiedCount; // [illegible]
res.acknowledged; // [illegible]
res.upsertedId; // [illegible]
res.upsertedCount; // [illegible]
```

This function triggers the following middleware.

- updateOne()

Model.validate()

Parameters:

- obj «Object»
- pathsToValidate «Array|String»
- [context] «Object»
- [callback] «Function»

0:00 / 9:58

More detail

- If we want to update some properties of a document, but want to leave the other properties unchanged we can use the method **updateOne (https://mongoosejs.com/docs/api.html#model_Model.updateOne)** .
- On the other hand, if we want to completely replace the existing document, we can use the method **replaceOne (https://mongoosejs.com/docs/api/model.html#model_Model.replaceOne)** .

The Mongoose method `updateOne` is a static method created by Mongoose on the model class. It takes 2 required arguments:

- `filter`
  - This parameter is of type object and is used to match documents to be updated.
  - 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 filter = {"_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 filter = {"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"};`

Delete

Again different methods are provided to delete documents. Typically, we know the unique ID of the document we want to delete. In that case, we can use the Mongoose method **deleteOne (https://mongoosejs.com/docs/api/model.html#model_Model.deleteOne)** with the ID of the document we want to delete. This function returns an object with the property `deletedCount` that specifies how many documents were deleted.

In the model.mjs file, we provide two variables; one for deleting by _id and one by deleting using the movieFilter (we can use it retrieval and for deleting).

```
// Delete based on the ID.
const deleteById = async (_id) => {
    const result = await Movie.deleteOne({_id: _id});
    return result.deletedCount;
};

// Delete based on filter.
const deleteByProperty = async (filter) => {
    const result = await Movie.deleteMany(filter);
    return result.deletedCount;
}
```

Then, in the controller.mjs file, we use those variables in functions; one for deleting by _id and one for deleting by a property.

```
function deleteById(req, res) {
    movies.deleteById(req.query._id)
        .then(deletedCount => {
            res.send({ deletedCount: deletedCount });
        })
        .catch(error => {
            console.error(error);
            res.send({ error: 'Request failed' });
        });
}

// Delete based on the filter
function deleteByProperty(req, res) {
    const filters = movieFilter(req);
    movies.deleteByProperty(filters)
        .then(deletedCount => {
            res.send({ deletedCount: deletedCount });
        })
        .catch(error => {
            console.error(error);
            res.send({ error: 'Request failed' });
        });
}
```

Then, we use a GET request to delete bby either of those options after checking to see if it exists.

```
app.get('/delete', (req, res) => {
    if (req.query._id !== undefined) {
        deleteById(req, res);
    } else {
        deleteByProperty(req, res);
    }
});
```

Exercise: delete

Use the test-requests.txt file to delete a document from the movies collection and print the number of deletions. Confirm the deletion by retrieving.

Summary

In this exploration, we created a web app for managing a database with information about movies. Our app stores its data in MongoDB. We studied the Mongoose API to identify methods we can use to perform CRUD operation. In our program, we used this API Mongoose to interact with MongoDB.

Additional Resources

Here are some references to learn more about the topics we discussed in this exploration.

- MDN has a good **tutorial on using Mongoose (https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose)** as does **Freecodecamp (https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57/)**.