SEI-CC / **SEIR-11-08-21**

Code | Issues | Pull requests 1 | Projects | Wiki | Insights

main

SEIR-11-08-21 / work / w05 / d4 / 01-02-mongoose-embedding / **mongoose-embedding.md**

cogilvy-ga Update mongoose-embedding.md

History

1 contributor

Raw | Blame

Executable File 494 lines (338 sloc) 15.2 KB

mongoose

elegant mongodb object modeling for node.js

Mongoose
Embedding Related Data

Learning Objectives

Students Will Be Able To:
Use EJS Partial views
Define schemas for embedding Subdocuments
Embed a Subdocument in its related document

Road Map

1. Setup
2. Review the Starter Code
3. Related Data Resources/Entities - Review
4. Embedding Subdocuments
5. Adding Reviews to a Movie
6. Essential Questions
7. Further Study
    - Retrieve a Subdocument from a Mongoose Array
    - Remove a Subdocument from a Mongoose Array
    - Query for a Document that Contains a Certain Subdocument

Setup

1. Move into the mongoose-movies repo we used in the Intro to Mongoose lesson:
   `cd ~/code/mongoose-movies .`

2. Sync your code with the remote:

   ```
   git fetch --all
   git reset --hard origin/main
   ```

3. It never hurts to ensure that the Node modules are installed:

   ```
   npm install
   ```

4. Open the project's folder in VS Code: `code .`

5. Open an integrated terminal session: `control + backtick`

6. Start the Express app's server: `nodemon`

7. Browse to `localhost:3000`

Review the Starter Code

As you can see, a navigation bar and a bit of styling has been added since the previous lesson.

However, the changes are more than skin deep...

EJS Partial Templates have been implemented! Check out how:

- **movies/index.ejs** & **movies/new.ejs** are using EJS's `include` function to *include* header and footer "partial" templates.
- Check these EJS docs for more info.
- All `res.render()` calls are passing in a `title` property that's being used for the page title and to dynamically add an `active` class to the links in the nav bar.
- Like we did in the express-todos, **show** functionality has been added using a **show** route definition that maps to a `show` controller action for viewing a single movie:

- **views/index.ejs** shows a "DETAILS" link that will send a request to the proper `show` route: `GET /movies/:id`.
- EJS tags write the movie's `_id` into the `href` attribute.
- The `moviesCtrl.show` action is using the `Movie.findById` method and `req.params.id`.

Related Data Resources/Entities - Review

As you may recall, **relationships** exist between types of data (resources/entities).

For example, in the *Mongoose Movies* app, we have the following data relationships:

- ***A Movie has many Reviews; A Review belongs to a Movie***
  `Movie --< Review` (One-To-Many)

- ***A Movie has many Performers; A Performer belongs to many Movies***
  `Movie >--< Performer` (Many-To-Many)

💪 Data Modeling Exercise (2 mins)

Assuming an app for ordering pizza online:

- What would the relationship be between the `Customer` and `Order` data entities?

- What would the relationship be between the `Order` and `MenuItem` data entities?

Assuming an app for booking bands to perform at concerts:

- What would the relationship be between the `Band` and `Concert` data entities?

- What would the relationship be between the `Venue` and `Concert` data entities?

Adding Movie Reviews

A Movie has many Reviews **`Movie --< Review`**

Embedding vs. Referencing

As you'll learn in the next unit, when using a SQL/Relational Database to implement related data, there isn't much flexibility. For example, in addition to the `Movie` model, there would **have** to be a `Review` model that maps to a *reviews* table.

However, modeling data in MongoDB/Mongoose is more flexible, and is left up to the developer to decide between:

1. **Embedding** the related child/many-side data its parent document. For example, in the pizza app, it would probably make sense to copy and embed each **MenuItem** within the **Order** document it belongs to.

2. Using **referencing** where the related data is stored in their own documents and "link" them by storing one document's `ObjectId` in the document it relates to. For example, each **Order** document could have a `customer` property that stores the `_id`.

The following is an example of what an `Order` document might look like:

```
{
  _id: ObjectId('5fbaac9c14c9d08b73bafaa6'),
  orderDate: ISODate('2020-11-22T15:56:12.499Z'),
  customer: ObjectId('5fb81c435dca2113b71c5f21'),
  isPaid: false,
  lineItems: [
    {
      _id: ObjectId('5fb80da98be8990e0186cc37'),
      quantity: 2,
      menuItem: {
        ObjectId('5fb81c435dca2113b71e1c44'),
        description: 'Breadsticks',
        price: 6.99
      }
    },
    {
      _id: ObjectId('5fb80da98be4090e0260cd63'),
      quantity: 1,
      menuItem: {
        ObjectId('5fb81c435dca2113b71e1c44'),
        description: 'One-Topping Pizza - Small',
        price: 12.99
      }
    }
  ]
}
```

> Note that using an array of `lineItems` allows for tracking the quantity of the `menuItem`.

Embed or Reference Reviews?

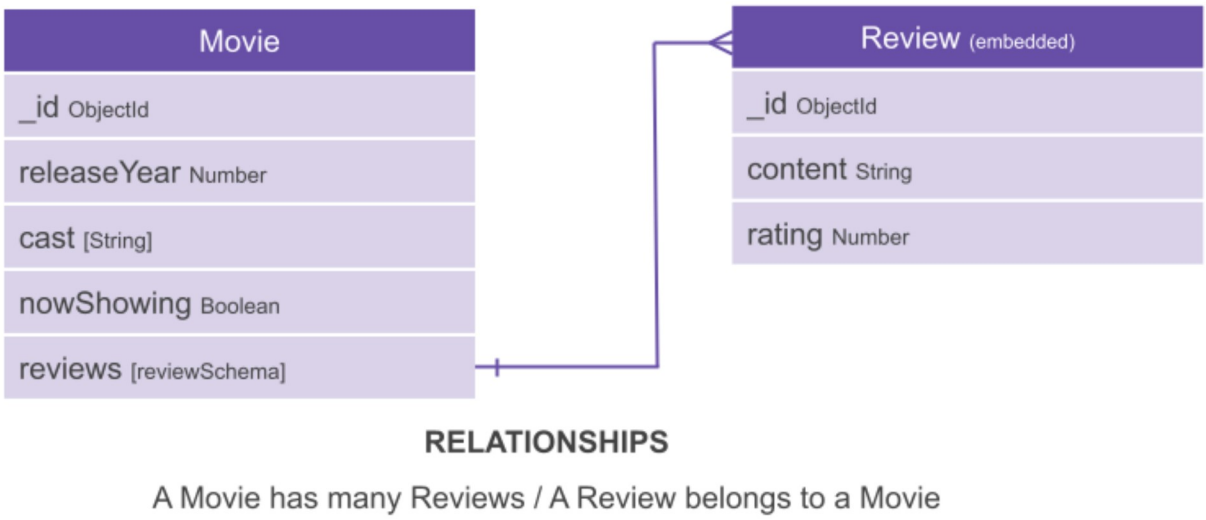
In most apps, a related resource, such as *reviews*, would likely be displayed with its parent being reviewed, in this case a *movie*.

If we stored *reviews* in their own collection by using a `Review` Model, we would have to make separate queries to access the *reviews* for the movies.

Embedding is considerably more efficient.

Yup, when using MongoDB/Mongoose, ***reviews*** are a perfect use case of embedding related data.

Here's what an ERD that models the `Movie --< Review` relationship might look like:



Embedding Subdocuments

When we embed related data, we refer to that embedded data as a **subdocument**.

Subdocuments are very similar to regular documents.

The key difference being that they themselves are not saved directly to the database - they are saved when the parent (top-level) document they are embedded within is saved.

Subdocuments **do** have their own **schema** though.

However, since subdocs are not saved to a collection, we **do not compile a subdoc's schema into a model**.

Creating a Schema for the Review Subdocuments

If a schema is going to be used for embedding subdocs in just **one** Model, then there's no reason to put the schema in its own Node module.

Therefore, for mongoose-movies, it's fine to write the `reviewSchema` just above the `movieSchema` in **models/movie.js**:

```
const reviewSchema = new Schema({
  content: String,
  rating: {type: Number, min: 1, max: 5, default: 5}
}, {
  timestamps: true
});

const movieSchema = new Schema({
```

With `reviewSchema` defined, we can now use it within the `movieSchema` as follows:

```
const movieSchema = new Schema({
  ...
  nowShowing: { type: Boolean, default: false },
  // reviews is an array of review subdocs!
  reviews: [reviewSchema]
}, {
  timestamps: true
});
```

User Stories

```
AAU, when viewing the detail page for a movie, I want to see a list of the
movie's reviews
```

and

```
AAU, when viewing the detail page for a movie, I want to see a form for adding
a new review
```

Since we will be displaying the form for creating a new review on each movie's detail page (**show.ejs**), we won't need to implement `new` functionality for the reviews resource, thus:

- No route in `routes/reviews.js` for showing a page with a form.

- No `new` controller action in `controllers/reviews.js`
- No `views/reviews/new.ejs` template. In fact, in mongoose-movies, there's no need to even create a `views/reviews` folder.

Cool, so there's no `new` functionality code for reviews, but we certainly need to implement the `create` functionality...

Implementing the Adding Movie Reviews User Story

Let's get coding!

Step 1 - Determine the Proper Route

Routing for a related, also called a nested resource, can be a bit different because we often need to "inform" the server of the nested resource's parent resource.

Let's take a look Routing for Nested Resources section of our Routing Guide.

Using the chart, we find that the proper route for creating a review is:

```
POST /movies/:id/reviews
```

> Note how the path of the route provides to the server the `_id` of the *movie* that the *review* is being created for!

Step 2 - Create the UI that Sends the Request

▸ ❓ What UI did we use to create a To Do?

Open up **movies/show.ejs**...

Here's the form to add under the current `</section>` tag:

```
</section>
<!-- new markup below -->
<br><br><h2>Reviews</h2>
<form id="add-review-form" method="POST"
  action="/movies/<%= movie._id %>/reviews">
  <label>Review:</label>
  <textarea name="content"></textarea>
  <label>Rating:</label>
  <select name="rating">
```

```
    <option value="1">1</option>
    <option value="2">2</option>
    <option value="3">3</option>
    <option value="4">4</option>
    <option value="5" selected>5</option>
  </select>
  <input type="submit" value="Add Review">
</form>
```

> Nothing new, but be sure to review how the value for the `action` is being written.

A touch of styling. **Update** this existing CSS rule on line 69:

```
#new-form *, #add-review-form * {
  font-size: 20px;
  ...
```

and **add** this new CSS to the bottom:

```
#add-review-form {
  display: grid;
  grid-template-columns: auto auto;
  gap: 1rem;
}

#add-review-form input[type="submit"] {
  width: 8rem;
  grid-column: 2 / 3;
  margin-bottom: 2rem;
}
```

Browse to the "details" of a movie.

Yeah, not too pretty but the form's `action` attribute looks pretty sweet!

Step 3 - Define the Route on the Server

As a best practice, let's create a dedicated router module for the reviews resource:

```
$ touch routes/reviews.js
```

and start with the typical router boilerplate:

```
const express = require('express');
const router = express.Router();
// You Do - Require the yet to be created reviews controller


// You Do - Define the Route below


module.exports = router;
```

💪 You Do: Require the reviews controller (yet to be created) and define the route we previously identified for creating a review (1 min)

Now let's require the new router in **server.js**:

```
const moviesRouter = require('./routes/movies');
// new reviews router
const reviewsRouter = require('./routes/reviews');
```

However, before we mount the new router in **server.js**, let's take another look at the paths in Routing for Nested Resources section of our Routing Guide.

Notice how some paths need to start with the parent resource (`posts`) and others with the nested resource `comments` ?

Because of these different "starts with" paths, we won't be able to mount the reviews router in **server.js** to any particular path, instead, to achieve the flexibility we need to mount the router to just `/` :

```
app.use('/movies', moviesRouter);
// Mount the reviews router to root to provide the
// flexibility necessary for nested resources
app.use('/', reviewsRouter);
```

As you know, the server won't be happy until we create and export that `create` action from the controller...

Step 4 - Create and Code the Controller Action

Let's create the controller module:

```
$ touch controllers/reviews.js
```

Open the new **controllers/reviews.js** and let the coding commence:

```
const Movie = require('../models/movie');

module.exports = {
  create
};
```

Note that although we don't have a `Review` model thanks to using embedding, we will certainly need to require the `Movie` model because we will need it to read and update the movie document that we're adding a review to.

Let's write the `create` function:

```
function create(req, res) {
  Movie.findById(req.params.id, function(err, movie) {
    // We can push subdocs into Mongoose arrays
    movie.reviews.push(req.body);
    // Save any changes made to the movie doc
    movie.save(function(err) {
      // Step 5:  Respond to the Request (redirect if data has been changed)
      res.redirect(`/movies/${movie._id}`);
    });
  });
}
```

As you can see, we simply push in an object that's compatible with the embedded document's schema, call `save` on the parent doc, and redirect to wherever makes sense for the app.

Before we start adding reviews, let's update the **show.ejs** template to render a movie's reviews...

Displaying a Movie's Reviews

All that's left is to update **movies/show.ejs** to render the movie's reviews. Time permitting, let's type it in, otherwise we can copy/paste then review.

```
</form>
```

```
<!-- New markup below -->

<% if (movie.reviews.length) { %>
  <table>
    <thead>
      <tr>
        <th>Date</th>
        <th>Review</th>
        <th>Rating</th>
      </tr>
    </thead>
    <tbody>
      <% movie.reviews.forEach(function(r) { %>
        <tr>
          <td><%= r.createdAt.toLocaleDateString() %></td>
          <td><%= r.content %></td>
          <td><%= r.rating %></td>
        </tr>
      <% }); %>
    </tbody>
  </table>
<% } else { %>
  <h5>No Reviews Yet</h5>
<% } %>
```

Try Adding Reviews

Assuming no typos, you should be able to add reviews - congrats!

Let's wrap up with some essential questions before you start on the lab to practice this stuff!

Don't forget to check out the
Further Study section which shows you how to:

- Retrieve a subdocument embedded in a Mongoose array
- Remove a subdocument from a Mongoose array, and
- Query for a document that contains a certain subdocument!

❓ Essential Questions

1. **True or False: All schemas must be compiled into a Model.**

2. **Is it more efficient to embed or reference related data?**
3. **True or False: An embedded subdocument must have its `save` method called to be persisted to the database.**

Further Study

Retrieve a Subdocument from a Mongoose Array

Mongoose arrays have an `id()` method used to find a subdocument based on the subdoc's `_id`:

```
const reviewDoc = movieDoc.reviews.id('5c5ce1be03563ad5540e93e2');
```

> Note that the string argument represents the `_id` of the *review* subdoc, not the *movie* doc.

Remove a Subdocument from a Mongoose Array

Mongoose arrays have a `remove()` method that can be used to remove subdocuments by their `_id`:

```
movieDoc.reviews.remove('5c5ce1be03563ad5540e93e2');
```

Subdocuments themselves also have a `remove()` method that can be used to remove them from the array:

```
// remove the first review subdoc
movieDoc.reviews[0].remove();
```

Query for a Document that Contains a Certain Subdocument

There's an amazing syntax that you can use to query documents based upon the properties of subdocs.

```
// Find the movie that contains a certain review
Movie.findOne({'reviews._id': req.params.reviewId}, function(err, movie) {
  // Wow, movie will be the doc that contains the review with an _id
  // that equals that of the reviewId route parameter!
});
```

> Note that the **dot** property syntax must be enclosed in quotes.

For another example, let's say you wanted to find all movies with at least one review with a 5 rating:

```
Movie.find({'reviews.rating': 5}, function(err, movies) {
  console.log(movies);  // wow!
});
```

`reviews.rating` represents the array and a property on the subdocs within that array!

References

- MongooseJS Docs - Subdocuments