Learn Git and GitHub without any code!

Using the Hello World guide, you'll start a branch, write comments, and open a pull request.

Read the guide

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

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

main

SEIR-11-08-21 / work / w05 / d5 / 01-02-mongoose-referencing / **mongoose-referencing.md**

cogilvy Add W5D5 — History

0 contributors

Raw | Blame

Executable File 346 lines (230 sloc) 12.3 KB

Mongoose - Referencing Related Data

Learning Objectives

Students Will Be Able To:
Use Referencing to Implement 1:M & M:M Data Relationships
Explain the Difference Between 1:M & M:M Relationships
"Populate" Referenced Documents

Road Map

1. Setup
2. Review the Starter Code
3. The `mongoose-movies` Data Model
4. Referencing *Performers* in the *Movie* Model
5. Associating Movies and Performers
6. *AAU, when viewing a movie's detail page, I want to see a list of the current cast and add a new performer to the list*
7. Essential Questions

Setup

There is starter code to sync with in the mongoose-movies folder:

- `cd ~/code/mongoose-movies`

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

Also, it never hurts to `npm i` just in case.

Lastly, open VS Code and start the server:

- `code .`
- Open a terminal session (`control + backtick`) and `nodemon`
- Browse to `localhost:3000`

Setup - Part Deux

Currently, the `cast` property on the `Movie` model holds an array of strings representing the names of the performers.

During this lesson, we will be updating the array to hold ObjectIds instead.

Mongoose will have a problem if it expects an ObjectId and gets a string instead, so let's clear out any strings that might be in the movie documents.

We will use a Node REPL to update the data and the Perform CRUD Using Mongoose Models in a Node REPL guide can help us.

> Note. All forms/views and controller actions that relied on the cast being an array of strings has been removed and/or adjusted in the starter code accordingly.

Here's the Mongoose statement that will update the `cast` property of all movie documents to an empty array:

```
Movie.updateMany(
  {},  // Query object determines which docs to update
  {cast: []},  // Update object has properties to update
  function(err, result) {console.log(result)}
);
```

Review the Starter Code

The starter code has a few updates from the *Mongoose - Embedding Related Data* lesson's final code:

- The **movies/show.ejs** view shows how you can use EJS to calculate an *average rating* for a movie.
- As you will learn in this lesson, a many-to-many relationship between two data resources such as *movies* and *performers*, requires that those resources already exist. Therefore, the functionality to create *performers* has been implemented to save time. However, rest assured that there is nothing in this code that has not been previously taught - let's checkout the model, router, controller & view.

  > Note: Implementing the code for Performers was the optional exercise previously assigned.

- Be sure to checkout the date "fix" required in the `create` action.
- **Go ahead a create a few performers** - feel free to use these from Star Wars and Caddyshack:

```
Mark Hamill  9/25/1951
Carrie Fisher  10/21/1956
Harrison Ford  7/13/1942
Chevy Chase  10/8/1943
Bill Murray  9/21/1950
```

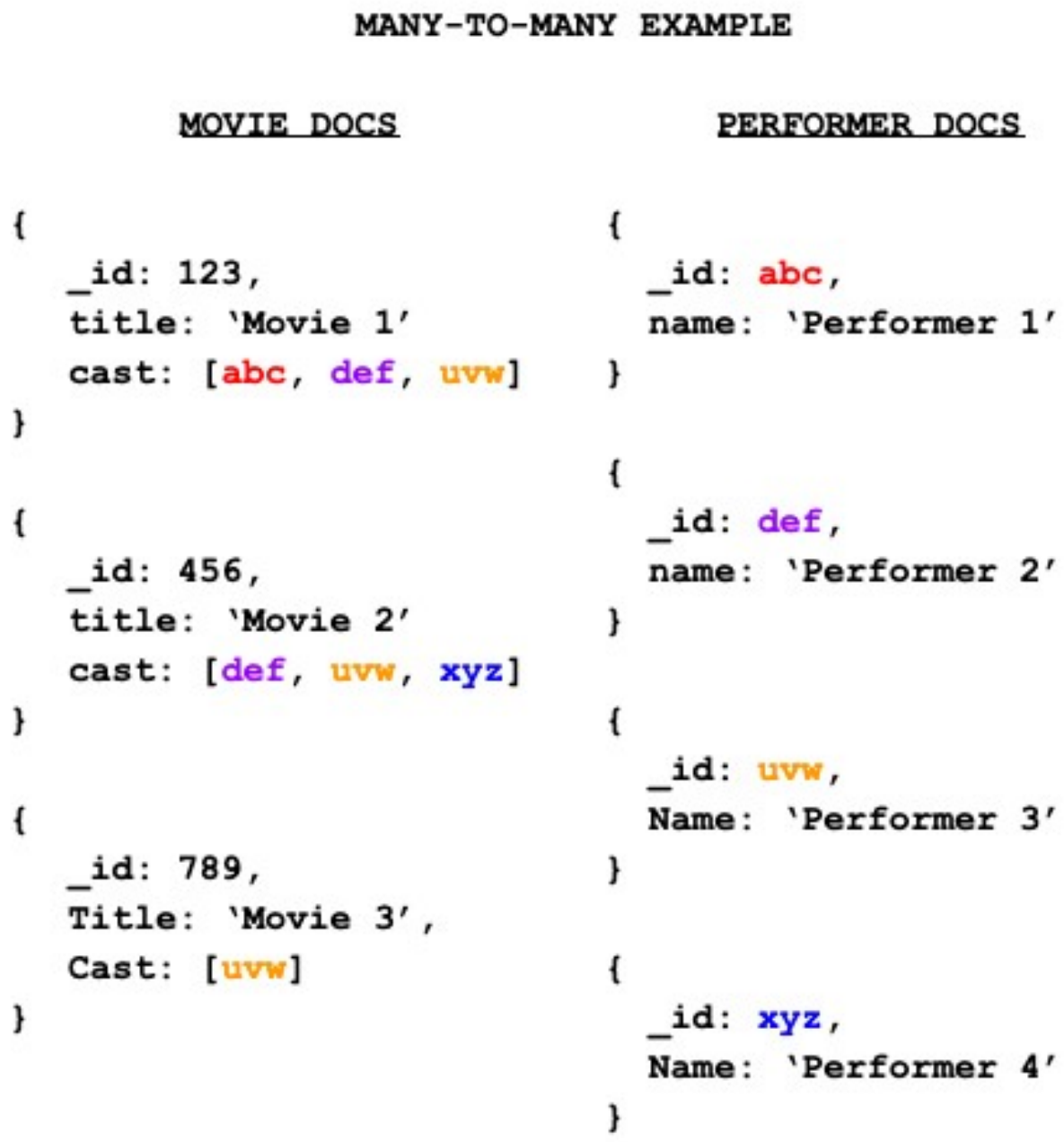
The `mongoose-movies` Data Model

We are going to implement the following data relationship:

- ***A Movie has many Performers; A Performer has many Movies***

`Movie >--< Performer` (Many-To-Many)

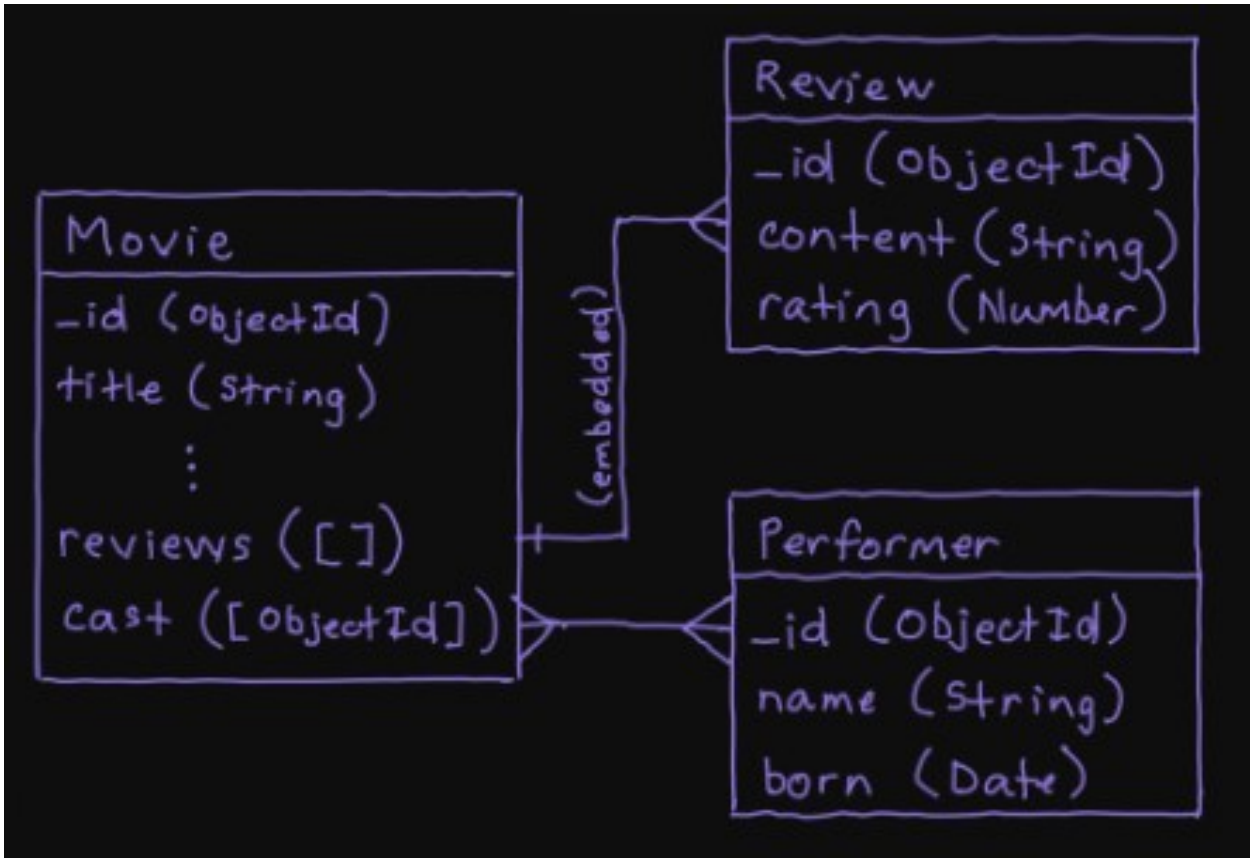
However, unlike we saw with *Reviews* (One-To-Many), multiple Movies can reference the same Performer creating a Many-To-Many relationship. Here's a simplified example:



Entity-Relationship-Diagram (ERD)

As part of the planning for your future projects, you'll need to plan the data model and document it with an Entity-Relationship-Diagram (ERD).

Here's an ERD that documents the data model for mongoose-movies:



Referencing *Performers* in the *Movie* Model

We're going to need to update the `cast` property in the `Movie` model (**models/movie.js**) to hold the `ObjectId` s of performer documents:

```
reviews: [reviewSchema],
cast: [{type: Schema.Types.ObjectId, ref: 'Performer'}]
```

The property type of `ObjectId` (or an array of `ObjectId` s) **is always** used to implement **referencing**.

The `ref: 'Performer'` is optional, but allows us to use the unicorn of Mongoose methods - `populate` .

Contrasting One-to-Many (1:M) and Many-to-Many (M:M) Relationships

The key difference between a `1:M` and a `M:M` relationship:

- In a `1:M` relationship, each of the **many** (child) documents belongs to only **one** (parent) document. Each time we want add a new relationship - **the child**

document must be created.

- In a `M:M` relationship, **existing** documents are referenced and the same document can be referenced over and over. New documents are created only if it's the first of its kind.

Many:Many CRUD

So, before a many-to-many relationship can be created between two documents (often called an **association**), those two documents must first exist.

This requires that the app first provide the functionality to create the two resources independent of each other.

Then, creating the association is a matter of adding the `ObjectId` to an array on the other side of the relationship.

The array property can be on either side (even both, but that's not usually recommended). Usually, the app's functionality reveals which side makes more sense. For example, the viewing of a movie with its performers is slightly easier to code by putting the `cast` array on the `Movie` Model vs. a `movies` array on the `Performer` Model.

> Note: When a relationship exists between the logged in user, for example: `User ---< Post`, it's usually a better practice to add the property that holds the relationship to the other Model, **not** the User Model.

❓ Review Questions

1. **What property type is used in schemas to reference other documents?**
2. **True or False: Assuming `Movie >---< Performer`, when associating a "performer" document with a "movie" document, both documents must already exist in the database.**

Associating Movies and Performers

Now that we've added the `cast` property to the `Movie` model, we're ready to implement the M:M relationship between *movies* and *performers*.

But first, a quick refactor...

AAU, after adding a movie, I want to see its details page

This user story can be accomplished with a quick refactor in the `moviesCtrl.create` action in **controllers/movies/js**:

```
movie.save(function(err) {
  if (err) return res.redirect('/movies/new');
  // res.redirect('/movies');
  res.redirect(`/movies/${movie._id}`);
});
```

Don't forget to replace the single-quotes with back-ticks!

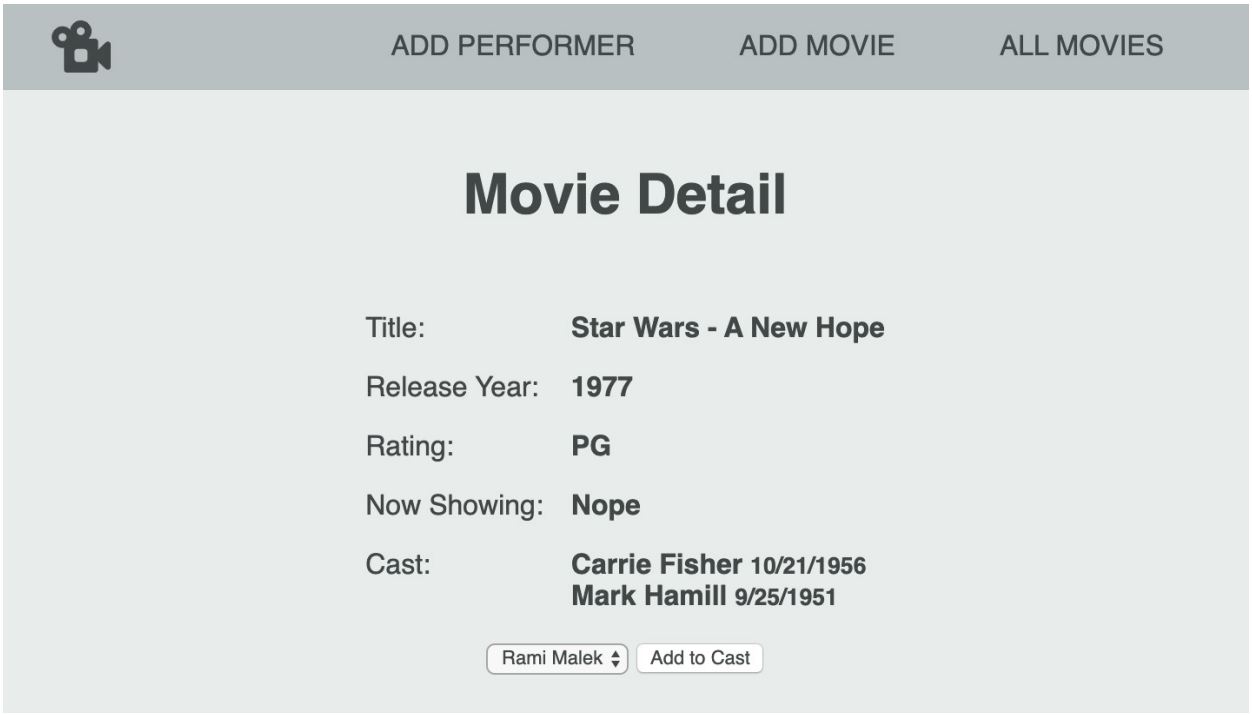
User story done! Now for some fun!

AAU, when viewing a movie's detail page, I want to see a list of the current cast and add a new performer to the list

Let's ponder what it's going to take to implement this user story:

- In **movies/show.ejs**, iterate over the movie's cast and use EJS to render each performer.
- Hold it! Because we are using referencing, there are `ObjectId` s in a movie's `cast` array - not any of the actual performer data! Oh wait, this is what the magical `populate` method is for!
- Using a form with a dropdown, we can send an HTTP request to associate a performer and movie. We will need the list of performers to build the dropdown - however, we only want to include the performers in the dropdown that are not already in the cast!

Let's do this - here's our wireframe:



Replacing *ObjectIds* with the Actual Performer Docs

Let's refactor the `moviesCtrl.show` action so that it will pass the movie with the *performer* documents in its `cast` array instead of `ObjectIds` (in **controllers/movies.js**):

```
function show(req, res) {
  Movie.findById(req.params.id)
    .populate('cast').exec(function(err, movie) {
      res.render('movies/show', { title: 'Movie Detail', movie });
    });
}
```

`populate`, the unicorn method of Mongoose!

We can chain the `populate` method after any query.

When we "build" queries by chaining like above, we need to call the `exec` method to actually run the query (passing in the callback to it).

❓ How does the `populate` method know to replace the `ObjectId`s with `Performer` documents?

Passing the *Performers* for the dropdown

While we're in `moviesCtrl.show`, let's see how we can query for just the *performers* that are not already associated with the *movie*.

First, we're going to need to access the `Performer` model, so require it at the top of **controllers/movies.js**:

```
const Movie = require('../models/movie');
// require the Performer model
const Performer = require('../models/performer');
```

Now we're ready to refactor the `show` action, we'll review as we refactor the code:

```
function show(req, res) {
  Movie.findById(req.params.id)
    .populate('cast').exec(function(err, movie) {
      // Performer.find({}).where('_id').nin(movie.cast) <-- Mongoose query build
      // Native MongoDB approach
      Performer.find(
        {_id: {$nin: movie.cast}},
        function(err, performers) {
          console.log(performers);
          res.render('movies/show', {
            title: 'Movie Detail', movie, performers
          });
        }
      );
    });
}
```

The log will show we are retrieving the *performers* - a good baby step at this point.

Refactor *movies/show.ejs*

There are comments to help us as we refactor **show.ejs** to render:

- The movie's cast of performers, and
- The dropdown and form used to associate a performer with the movie.

It's a bit complex, so we'll review it while we make the changes:

```
<div><%= movie.nowShowing ? 'Yes' : 'Nope' %></div>
<!-- start cast list -->
<div>Cast:</div>
```

```
  <ul>
    <%- movie.cast.map(p =>
      `<li>${p.name} <small>${p.born.toLocaleDateString()}</small></li>`
    ).join('') %>
  </ul>
  <!-- end cast list -->
</section>

<!-- add to cast form below this comment -->
<form id="add-per-to-cast" action="???" method="POST">
  <select name="performerId">
    <%- performers.map(p =>
      `<option value="${p._id}">${p.name}</option>`
    ).join('') %>
  </select>
  <button type="submit">Add to Cast</button>
</form>
```

Now let's add this tidbit of CSS in **public/stylesheets/style.css** to tidy up the cast list:

```
ul {
  margin: 0 0 1rem;
  padding: 0;
  list-style: none;
}

li {
  font-weight: bold;
}
```

Add the Route for the *Add to Cast* Form Post

Let's check out the Routing Guide to find the endpoint (keep looking...).

The route is RESTful, but we'll have to use a non-RESTful name for the controller action because we're creating an association between a movie and a performer, but `create` is already taken...

In **routes/performers.js**

```
router.post('/movies/:id/performers', performersCtrl.addToCast);
```

`addToCast` - not a bad name!

💪 Exercise - Update the Form's `action` Attribute (1 minute)

The form's `action` attribute needs to be set to the "proper" endpoint for adding the association between a movie and a performer.

> Hint: Copy/paste the path of the route we just defined and modify as required - yup, you'll need some squids in there.

Code the *addToCast* Controller Action

Let's write that `addToCast` action in **controllers/performers.js**:

```
const Performer = require('../models/performer');
// add the Movie model
const Movie = require('../models/movie');

module.exports = {
  new: newPerformer,
  create,
  addToCast
};

function addToCast(req, res) {
  Movie.findById(req.params.id, function(err, movie) {
    movie.cast.push(req.body.performerId);
    movie.save(function(err) {
      res.redirect(`/movies/${movie._id}`);
    });
  });
}
```

Reads like a book!

We Did It!

That was fun!

❓ Essential Questions

1. **True or False: The following property in a bookSchema would properly implement a `Book >--< Author` relationship:**

```
const bookSchema = new Schema({
```

```
    authors: [{type: Schema.Types.ObjectId, ref: 'Author'}],
    ...
```

2. **Describe the difference between 1:M & M:M relationships.**
3. **What's the name of the method used to replace an `ObjectId` with the document it references?**

References

- MongooseJS Docs - Populate
- MongooseJS Docs - Queries