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 / **03-04-mongoose-flights-lab-part-3.md**

cogilvy Add W5D5 **History**

0 contributors

Raw Blame

```
Executable File   117 lines (72 sloc)   6.22 KB
```

Mongoose "Flights" Lab - Part 3

Intro

Today in the *Mongoose - Referencing Related Data* lesson you:

- Created a `Performer` Model.
- Created a many-to-many relationship, `movie >--< performer` by adding a `cast` property in the `Movie` Model that references *performer* documents.
- Created routes and a controller for the *performers* data resource.
- Implemented functionality for creating *performers*.
- Populated the `cast` property with *performer* docs and displayed them with the movie on the movie's show view.

- Implemented functionality for adding *performers* to a movie's `cast` (if the don't already exist in the cast).

Similar to what we did in the lesson, in this lab you'll be adding functionality to the `mongoose-flights` project you created in *part 1* and have continued to work on in *part 2* of the lab.

The final version of `mongoose-flights`, as a result of completing parts 1 - 3 of this lab, is a DELIVERABLE.

Goal

The goal of this lab is to practice referencing related data.

You will add the ability to create *tickets* for a given *flight* in the `mongoose-flight` project.

The relationship between the data entities is:
`Flight --< Ticket`
A flight has many tickets / A ticket belongs to a flight

Styling is secondary, spend time on it only after the functionality has been implemented.

Exercises

1. Create a `ticketSchema` that will be compiled into a `Ticket` Model with the following properties:

Property	Type	Validations	Default Value
`seat`	`String`	Must be 'A1' thru 'F99' (see hints)	n/a
`price`	`Number`	Minimum of `0`	n/a
`flight`	`ObjectId`	Include `ref: 'Flight'` to enable population	n/a

Hints

Notice how we don't *have* to use an array to implement the 1:M relationship between `Flight` and `Ticket`. Instead, referencing the `ObjectId` of the *flight* in the `flight` property of a *ticket* enables the relationship. FYI, to implement this 1:M relationship, we *could* have put a `tickets` array on the `Flight` model instead. Yup, unlike M:M relationships, 1:M doesn't require the use of an array property - just an ObjectId on the "belongs to" side (child side) of the relationship.

Define the `seat` property as follows:
`seat: {type: String, match: /[A-F][1-9]\d?/}` - that's what we call a regular expression that's being assigned to the `match` validator. Now for the best part, which just might blow your mind! You ready? Here it is... HTML `<input>` tags have a `pattern` attribute that accept a regex pattern; and if what's typed in the `<input>` doesn't match the pattern, the form can't be submitted! Here's what your `<input>` should look like for entering the seat:

```
<input name="seat" required pattern="[A-F][1-9]\d?">
```

That regex pattern will match the following characters:

- An `A` thru `F` character, followed by
- a `1` thru `9` character, followed by
- zero or one `0` thru `9` character.

We'll cover more about regular expressions soon enough in SEI, but this opportunity to preview them was too hard to pass up! Combined with the HTML `pattern` attribute, they provide an excellent way to perform *client-side* validation of inputs.

2. Implement the following user story:
AAU, when viewing the detail page for a flight, I want to view a list of the tickets that have been created for that flight

Hints

To show a list of *tickets* that belong to a *flight* in the `flights/show.ejs`, the flight controller's `show` action is going to have pass that array of flights to be rendered. This is going to require the `show` action to make a separate query (inside of the callback of the `Flight.findById` call) to retrieve the flights as follows:

```
Flight.findById(req.params.id, function(err, flight) {
    Ticket.find({flight: flight._id}, function(err, tickets) {
      // Now you can pass both the flight and tickets in the res.render call
        ...
    });
  });
```

Note that there's no reason to `populate` the `flight` property because in this case, you already have obtained the *flight* using `Flight.findById`.

For future reference though, here's how to populate a *ticket's* `flight` property:

```
  Ticket.findById(req.params.id)
    .populate('flight')
    .exec(function(err, ticket) {...
```

3. Also on the flight's `show` view, display a **New Ticket** link (perhaps styled to look like a button) that when clicked, shows the ticket's `new` view used to create a *ticket* for the *flight*. When the form is submitted, create the *ticket* on the server and redirect back to the *flight's* `show` view.

   Hints

   To display the view with the form for adding a ticket, the path of the `href` for the **New Ticket** link will need to include the flight's `_id`. The path should match this route defined on the server: `/flights/:id/tickets/new`. The `req.params.id` can now be passed to the **tickets/new.ejs** and used for the ticket form's `action` attribute...

   If you use the "proper" route for the ticket form's `action` attribute, the `ticketsCtrl.create` action will have access to the `_id` of the *flight* the *ticket* is being created for - you got this!

   In the controller action, there **will not** be a `flight` property on the `req.body` object. You must add that property yourself before using `req.body` to create the *ticket*. Failure to do so will result in the *ticket* being created without a `flight` property that references the *flight* it belongs to - so if newly added tickets are not showing up with the flight, this is probably the cause.

More Hints

- Learn it, know it, live it... When adding functionality to the app:

i. Identify the "proper" Route (Verb + Path)
ii. Create the UI that issues a request that matches that route.
iii. Define the route on the server and map it to a controller action.
iv. Code and export the controller action.
v. `res.render` a view in the case of a GET request, or `res.redirect` if data was changed.

Bonuses

1. Style the app.

2. Add a feature to delete a flight's *ticket*.

Deliverable?

The final version of `mongoose-flights`, as a result of completing parts 1 - 3 of this lab, is a DELIVERABLE.