Introduction to Computer Science I
COMP 2406 – Fall 2019

More Mongoose

Dave McKenney
david.mckenney@carleton.ca

Learning Outcomes

by the End of this Lecture, Students that have Completed the Reading Assignment and Review Questions should be Able to:

Define Mongoose instance, static, and query methods

Create interlinked documents in Mongoose

Use Mongoose's populate functionality

Use Mongoose inside a web app

Mongoose Instance Methods

Mongoose models/documents in our apps have various pre-defined methods (find, save, etc.)

We can also define our own instance methods

This allows us to give names to operations we perform regularly. It also helps keep our code organized.

Mongoose Instance Methods

For example, in our Products schema, we could define a getVolume( ) method

This method can calculate and return the volume of the calling Product document instance

Mongoose Instance Methods

We could also define a sell(int, callback) method

This method could modify the calling instance, save it to the database and call the callback function

Mongoose Instance Methods

Finally, we could define a findSimilarProducts( ) method

This could find products with similar names, similar prices, etc.

Later, we could factor in past purchasers and reviews of the product to find related products

Mongoose Instance Methods

All of these shorten the amount of code we write in our server that is manipulating products

We could also reduce code duplication if we had to do the same thing multiple places

And we keep all our product-related behaviour in a single file

See 18-ex1-instance-methods.js

Mongoose Static Methods

Mongoose also supports static methods

These are like instance method but are defined at the level of the model in general
(e.g., should not refer to this.someField)

Mongoose Static Methods

For example, searching by name is a common use of our product schema/model

We could create a static method to allow us to Product.findByName(string, callback)

If we decide to change how this functionality should work, we have a single place to do so

Mongoose Static Methods

Alternatively, if your product schema contained a category field, we could define:
Product.findByCategory(string, callback)

Or another common idea:
Product.findOutOfStock(callback)

See 18-ex2-static-methods.js

Mongoose Static Methods

Again, this leads to code that is easier to read, has less duplication, and is well organized

Whether to choose an instance method or static method depends on whether the method deals with a single document or the type of documents in general (i.e., like general OOP)

Mongoose Query Helper Methods

One last type of method we can add to a schema is query helper methods

These can extend the default query methods that Mongoose provides for all models
(e.g., find, where, gt, etc.)

Mongoose Query Helper Methods

Similar to instance methods, this can allow us to assign a name to some specific behaviour

But unlike instance methods, we can incorporate these methods into other queries

Mongoose Query Helper Methods

For example, we could add a byName query helper similar to the findByName static method

So we could then add it into a chaining query like:
Products.find().byName("someString")

See 18-ex3-query-methods.js

Referencing Other Documents

When discussing schemas, it was mentioned that a schema can reference an ObjectID associated with another document

We can also specify the schema (i.e., collection to search in) this ObjectID references

Referencing Other Documents

So we could have the following schemas/models:

```
const personSchema = Schema({
  _id: Schema.Types.ObjectId,
  name: String,
  age: Number,
  stories: [{type: Schema.Types.ObjectId, ref: 'Story'}]
});

const storySchema = Schema({
  author: {type: Schema.Types.ObjectId, ref: 'Person'},
  title: String,
  fans: [{type: Schema.Types.ObjectId, ref: 'Person'}]
});

const Story = mongoose.model('Story', storySchema);
const Person = mongoose.model('Person', personSchema);
```

Referencing Other Documents

We can create an author and story that references the author:

```
const author = new Person({
  name: 'Ian Fleming',
  age: 50
});

author.save(function (err) {
  if (err) throw err;
  const story1 = new Story({
    title: 'Casino Royale',
    author: author._id    // assign the _id from the person
  });

  story1.save(function (err) {
    if (err) throw err;
    //done!
  });
});
```

Referencing Other Documents

The referenced object is stored using its ObjectId

This saves storage space, reduces information transferred for queries, and significantly simplifies updating documents

If you stored the document instead of ID, you would have to find/update all occurrences for every change

See 18-ex5-reference-documents.js

Referencing Other Documents - Populate

If you aren't going to use the referenced document directly, the ID received in the query is sufficient

If you need the content of the referenced document, you could perform additional queries to retrieve it:

```
Story.findOne(function(err, result){
 if(err) throw err;
 People.findById(result.author, function(err, result){
        //do something
 }
}
```

Referencing Other Documents - Populate

Mongoose provides an easy way to do this - populate:

```
Story.findOne({ title: 'Casino Royale' }).
 populate('author').
 exec(function (err, story) {
         if (err) return handleError(err);
         //Do something with story.author
         //which is now a Person document
 });
```

Referencing Other Documents - Populate

This works with arrays of ObjectId too

Populate the array field and each value is replaced by the document it refers too

This has performance implications…

Referencing Other Documents - Populate

You can easily assign/change a reference field:

```
Story.findOne({ title: 'Casino Royale' })
.exec(function(error, story) {
        if (error) throw err;
        story.author = author;
});
```

Mongoose automatically saves just the _id field of the value you are assigning

Referencing Other Documents - Populate

In some cases, the document that is referenced may not exist anymore (e.g., after being removed)

In this case, the populated field will be null

You will have to handle this in your code, if it is something that could happen

Referencing Other Documents - Populate

You can select the fields to populate, so not all fields are returned:

```
Story.findOne({ title: /casino royale/i }).
 populate('author', 'name'). // only return Person name
 exec(function (err, story) {
  if (err) return handleError(err);
  console.log('The author is: ' + story.author.name);
  // prints "The author is Ian Fleming"
  console.log('The authors age is: ' + story.author.age);
  // prints "The authors age is null'

});
```

Referencing Other Documents - Populate

If you have multiple reference fields, you can populate more than one:

```
Story.
  find(...).
  populate('fans').
  populate('author').
  exec();
```

Referencing Other Documents - Populate

If you have multiple reference fields, you can populate more than one:

```
Story.
  find(...).
  populate('fans author'). //Alternate method
  exec();
```

Referencing Other Documents - Populate

One issue: stories have a reference to author, but authors do not have a reference to story

There are several ways to handle these sorts of cases

And there is much debate about which is best…

Referencing Other Documents - Populate

We could push the story reference to the author's stories array any time we assign the author to a story

See 18-ex6-two-way-reference.js

Referencing Other Documents - Populate

Having two-way references is harder to maintain

We need to perform multiple updates when we make a single change

We need to ensure that the references are consistent (e.g., if there is an error saving one, both do not save)

Referencing Other Documents - Populate

Typically, if you have a one-to-many relationship (e.g., authors can have many stories)

Then the one ('child') object references its 'parent' object

Referencing Other Documents - Populate

So assume the stories hold a reference to their author's ID, but the author does not hold a reference to their stories

How can we perform a query like "find this author's stories?"

Referencing Other Documents - Populate

We can find in the stories collection all stories that match the given author's ID:

```
Story.
 find({ author: author._id }).
 exec(function (err, stories) {
  if (err) throw err
  console.log('The stories are an array: ' + stories);
 });
```

Referencing Other Documents - Populate

What about many-to-many relationships?

For example, our store database products had many buyers, and buyers have purchased many products

How could we model this in Mongoose?

Referencing Other Documents - Populate

We could use an array of references

If we expect the array to grow large, this is a bad idea

We may be doing a lot of queries like:
Find products where the large array of buyers contains this user's ID

Referencing Other Documents - Populate

In fact, the Mongo documentation explicitly mentions avoiding arrays that will grow very large

So is there a better way? We will discuss more when we look at the completed store example

If you have many of these relationships to model, a relational database may be a better choice…

Lean Mongoose

Mongoose provides significant utility to us

But this utility requires additional information

So Mongoose documents are significantly more 'heavyweight' than regular Javascript objects (which we get when using Mongo)

Lean Mongoose

To speed up querying and reduce memory footprint, queries can be executed with the lean( ) option

This will then return plain Javascript objects instead of fully-functional Mongoose documents

```
SomeModel.find().lean().exec(...)
```

Completed Store

For an example of Mongoose applied within the context of a web application, see the completed store example

Questions

Questions?