Introduction to Computer Science I
COMP 2406 – Fall 2019

Introduction to 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:

Identify advantages of using Mongoose

Create schemas to represent document types

Perform document validation using Mongoose

Perform document querying using Mongoose

Introduction to Mongoose

Last lecture we incorporated MongoDB into an Express-based Node.js app

Today, we will look at a related tool: Mongoose

Introduction to Mongoose

Mongoose is an Object Document Mapper (ODM)

This provides a mapping from documents in the database to objects in our program

Provides many useful features to simplify our programming and interaction with Mongo

Introduction to Mongoose

Mongoose allows us to define a 'schema' for different document types

Schemas allow us to define things like: the fields in a document type, the types of those fields, validation rules for those fields

Mongoose provides automatic handling of type conversion and validation when saving documents

Introduction to Mongoose

Mongoose provides further utility methods for performing CRUD operations

Additionally, it provides automatic reconnection attempts and automated keep-alive actions

Introduction to Mongoose

Mongoose is an external module, so:

npm install mongoose

Note: if you require mongoose, you do not have to explicitly require mongo

Getting Started – Connecting to a Database

The first step to using Mongoose is to establish a connection

The Mongoose module exports a connection property

You can use this property to access the primary Mongoose connection to Mongo

See 17-ex1-basic-connection.js

Getting Started – Creating a Schema

Now that we have connected to a database, we can create our first basic schema and build a model

The basic user schema will have two properties: a first name and a last name

See ex2-simple-schema.js

Getting Started – Creating/Saving Documents

Once we have a model reference, we can create documents based on that model

Mongoose provides model methods .save and .create for saving/adding documents to the database

See 17-ex3-creating-documents.js

Getting Started – Finding Documents

Mongoose has model methods .find and .findOne that search for documents of that type in the database

We will see more advanced usage later, but you can perform simple searches like:
UserModel.find({firstName: "someName"}, ...};

See 17-ex4-simple-finding.js

Creating More Complex Schemas

The previous examples used a very basic schema

Mongoose supports significantly more functionality than we have seen so far

First, there are a number of schema types…

Mongoose Schema Types

Mongoose Schema Types:

String, Number, Boolean, Date

Array – which can define the type it contains

Buffer – binary data (images, PDFs, other files)

ObjectId – reference to another document

Mixed – anything

Build a More Complex Schema

Consider the product documents we worked with…

```
let productSchema = Schema({
  name: String,
  price: Number,
  stock: Number,
  dimensions: {
     x: Number,
     y: Number,
     z: Number,
  },
  reviews: [Schema.Types.ObjectId],
  buyers: [Schema.Types.ObjectId],
});
```

See 17-ex5-basic-product-schema.js

Schema Validation

One of Mongoose's biggest advantages is automatic validation of fields

We can specify which fields are required, the valid values for each field, and default values for each field

Validation is automatically executed before saving a document into the database

Schema Validation

To start specifying more field requirements in the schema, we can use an object as the value

For example, in our previous example we had:
name: String

We could change this to:
name: {type: String, required: true}

Build a More Complex Schema

```
let productSchema = Schema({
  name: {type: String, required: true},
  price: {type: Number, required: true},
  stock: {type: Number, required: true},
  dimensions: {
    x: Number,
    y: Number,
    z: Number,
  },
  reviews: [Schema.Types.ObjectId],
  buyers: [Schema.Types.ObjectId],
});
```

Validation Errors

Now, when we try to save a product, there MUST be a name, price, and stock value specified

If not, an error will be thrown

This can then be handled in the callback function...

Validation Errors

The error object will have a key 'errors', with the value being an array of ValidatorError objects

Useful fields in ValidatorError include:
kind: what kind of validator was invalid
message: error message
path: the field name
value: the value of the field

Additionally, the error object has a 'message' property that summarizes

Validation Errors

So you can extract information from the error thrown…

See 17-ex6-required-fields.js

Further 'required' Validation

The value of 'required' is true/false

But this value can be dynamically generated (e.g., generated by a function)

This allows you to define custom rules for whether a field is required or not

Further 'required' Validation

For example, we could require a 'price' only if the 'stock' value is greater than 0:

```
price: {type: Number, required: function(){ return
this.stock > 0; } }
```

'this' refers to the document being validated

See 17-ex7-required-function.js

Further 'required' Validation

Additionally, you can specify default values for fields

If a value is not given, then the default value is used:

```
dimensions: {
        x: {type: Number, default: 1},
        y: {type: Number, default: 1},
        z: {type: Number, default: 1}
}
```

Built-In Validators

'required' is one built-in validator that works for all schema types

Number types have two more built-in validators: min and max

String types have: enum (match an item in an array), match (regular expression), minlength, maxlength

In addition, you can specify error messages…

Built-In Validators

```
price: {
    type: Number,
    required: [true, "You need a price..."],
    min: [0, "You can't pay people to buy it..."]
 }
```

Built-In Validators

```
name: {
        type: String,
        required: true,
        minlength: 3,
        maxlength: 50
}
```

See 17-ex-8-built-in-validation.js

Custom Validators

You can also create your own custom validation function - add a 'validate' key to the specification of the field with the value being an object

Add to this object: a validator key with the value being a function returning true/false, and a message key with a string value

So, for example, we could limit the volume of our products to a certain amount...
See 17-ex9-custom-validators.js

The 'Product' Schema

Now that we have defined a product schema, we can create our dataset as we did in the Mongo examples

See 17-ex10-product-inserter.js

Summary of Validation in Mongoose

Validation in Mongoose is VERY useful

We no longer have to put so much effort into validating user data

If typecasting fails or the value does not pass validation, we can just catch/handle the error

Cleaning Up Our Code

We should be interested in keeping our code clean

Multiple schema definitions in a file gets messy

A good practice is to define a single scheme in a single module – that module exports the model creation function

See ex11-requiring-model.js and ProductModel.js

Using Models to Query

Mongoose also provides added utility when querying using models

We can avoid some of the complexities involved when making Mongo queries

This cleans up our code, makes it easier to read/understand, and less likely to have mistakes

Using Models to Query

We previously saw an example of a basic find:

Products.find(function(err, results){ … });

An interesting note about Mongoose queries is that they can be created without being executed

Using Models to Query

If a callback function is specified, the query is executed immediately

Otherwise, the query can be saved to a variable:

```
let findAll = Products.find();
```

And executed at a later time:

```
findAll.exec(function(err, result){ ... });
```

See 17-ex12-saving-query.js

Using Models to Query

Furthermore, we can add additional Mongoose query methods to build complex queries

The first method we will look at is .where

This allows us to specify constraints on a field that must match

Using Models to Query

The query Product.find({name: "Plastic Fork"})

Can be expressed as:

Product.find().where("name").equals("Plastic Fork")

But we can chain many conditions together...

Using Models to Query

So the query Product.find ({price: {$gte: 100, $lte: 300}})

Can be expressed as:

Product.find( ).where("price").gte(100).lte(300)

Using Models to Query

Mongoose supports the same types of operators as Mongo:

.gt(Number), .gte(Number), .lt(Number), .lte(Number)

.equals(value)

.in(Array) .nin(Array)

.regex(*RegularExpress*)

.size(Number) – matches array size

Using Models to Query

Each of the previous operators is applied to the field that was last specified in by .where(...)

See 17-ex13-advanced-finding.js for some examples

Using Models to Query

Another useful query method is .select

This allows you to specify which fields to have returned (i.e., the 'projection' from Mongo)

List the fields in a string separated by spaces…

See 17-ex14-selecting-fields.js

Using Models to Query

A few more useful methods:

.skip(integer)

.limit(integer)

.count() – returns number matched

See 17-ex15-more-query-methods.js

Using Models to Query

Finally, you can specify the sort order of documents that are returned using .sort

Specify a space-separated list of field names

To sort in decreasing order, prefix path with –

See 17-ex16-sorting-results.js

Mongoose Queries

Other than find/findOne, Mongoose model's provide many other query methods…

Mongoose Model Query Methods

deleteOne/deleteMany

findByIdAndDelete(id) – note: id can be a string

findByIdAndUpdate(id, {…updates…})

findOneAndDelete, findOneAndUpdate

replaceOne

updateOne/updateMany

Mongoose Model Query Methods

A very important note: update( ), updateMany( ), findOneAndUpdate( ), etc. do NOT execute the validation steps used when calling save( )

The best practice for updating a document involves:

1. **Finding the document**
2. **Making changes to the fields**
3. **Calling the save( ) method on that document**

Questions?

Questions?