COURSEWARE

Mongoose

Contents

Overview

Mongoose is a MongoDB object modelling library, or ODM (Object Document Mapper), written in JavaScript.

It provides a simple to use API to work with MongoDB databases.

This allows interaction with the DB directly in JavaScript without having to use a database query language (SQL, etc.).

Mongoose can be used to interact with MongoDB in a structured way.

Installation

Install from NPM:

```
npm install mongoose
```

Import into module:

```
const mongoose = require('mongoose');
```

Connecting to MongoDB

You can connect to MongoDB with:
`mongoose.connect(uri, options)`
This connects using Mongoose's default connection.

```
mongoose.connect('mongodb://localhost:27017/example', { useNewUrlParser: true
});
```

Multiple connections can be made with:

```
const secondConnection = mongoose.createConnection(uri, options)
```

Different connections can use different settings and can be connected to different databases.

```
const conn2 = mongoose.createConnection('mongodb://localhost:27017/example2',{
useNewUrlParser: true });
```

You can check if a connection was successful or not with the callback parameter or with promises.

```
// Callback function
mongoose.connect(uri, opts,
    function (err) {
        if (err) {
            /* handle errors */
        } else {
            /* connection ready */
        }
    });

// Or with promises
mongoose.connect(uri, opts).then(() => { /* connection ready */ }, (err) => { /*
handle errors */ });
```

Schemas

Schemas define the structure of your collections and the shape of the documents within.

A schema is a configuration object for a model.

A `SchemaType` is a configuration object for an individual property.
It says what type a given path should have and what is valid for that path.
Standard types include: String, Date, Boolean etc.

SchemaType Docs

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const productSchema = new Schema({
    name: String,
    price: String,
    location: [{
        aisle: Number,
        shelf: Number
    }],
    dateAdded: {
        type: Date,
        default: Date.now
    },
    onSale: Boolean
});
```

Validation

Validation can be used to control what data is allowed in schema.
There are a number of built-in validators:

- All SchemaTypes have the required validator.
- Numbers have **min** and **max** validators.
- Strings have **enum**, **match**, **minlength**, and **maxlength** validators.

Validation is declared when defining a Schema.

```
const productSchema = new Schema({
    name: {
        type: String,
        required: true,
        minlength: 2
    },
    price: String,
    location: [{
        aisle: {
            type: Number,
            min: [1, 'minimum is 1'],
            max: 20
        },
        shelf: Number
    }], /* ... */
});
```

Models

Models are constructors compiled from Schema definitions.
Models are responsible for creating and reading documents from the MongoDB database.
Call `mongoose.model(name, schema)` to compile the model.
The name argument is the name of the collection the model is for.

Mongoose automatically looks for the plural, lower-case version of this name.

```
const productSchema = new Schema({/*...*/});

const product = mongoose.model('Product', productSchema);
```

Mongoose will look for a collection called 'products'.

Documents

A document represents a one-to-one mapping to a document as stored in MongoDB.

`Model` and `Document` are distinct classes in Mongoose.
The model class is a subclass of the document class.
A document is an instance of it's model.

Subdocuments

A subdocument is a document embedded in another document.

This is represented as nested schemas in Mongoose.

```
const childSchema = new Schema({/*...*/});

const parentSchema = new Schema({
    children: [ childSchema ]
});
```

Creating a child schema is better than using one massive schema as it improves modularity and makes the code much more reusable.

Queries

Mongoose provides a number of static helper functions for simple CRUD operations.
Run your queries against models.
The result can be accessed with a callback function.
Callback arguments: (error, result)
The value of result depends on the function called.
EG: find() returns an array, count() returns a number.

```
Model.deleteMany()
Model.deleteOne()
Model.find()
Model.findById()
Model.findByIdAndDelete()
Model.findByIdAndRemove()
Model.findByIdAndUpdate()
Model.findOne()
Model.findOneAndDelete()
Model.findOneAndRemove()
Model.findOneAndUpdate()
Model.replaceOne()
Model.updateMany()
Model.updateOne()
```

Create

New documents can be created by calling a model constructor.

```
let newdoc = new MyModel({example: 'data'});
```

Saving a document is simple:

```
newdoc.save().then(()=> console.log('done!'));
```

```
const prod = new Product({
    name: 'carrots',
    price: 1.23,
    location: {
        aisle: 13,
        shelf: 3
    },
    onSale: false
});

// Save returns a promise
prod.save().then(() => console.log('complete'));
```

Read

Using `Model.find()` will get an array of documents that matches the query.

```
var Product = mongoose.model('Product', productSchema);

Product.find(                   // find all from product
    { 'onSale': true },         // where onSale = true
    'name price',               // select name, price
    (err, prods) => {           // callback when complete / error
        if (err) {
            console.error('An error occurred:', err);
        } else {
            console.log('Products on sale:');
            prods.forEach((prod) => console.log(prod.name, prod.price));
        }
    }
);
```

Update

Updating a document works similarly.

Mongoose tracks changes you make to documents and generates the update operators automatically.

Calling `save()` on a modified document will update the document in the database.

```
prod.onSale = true;

// await will wait for a promise.
await prod.save();
```

Delete

Documents can be removed in a number of ways:

`Model.deleteOne()` – Delete a single document that matches a query.

`Model.deleteMany()` – Delete multiple documents that match a query.

The returned promise / callback resolves to an object containing:

`ok` – 1 if no error occurred.

`n` – Number of documents deleted.

`deletedCount` – Same as `n`.

Tutorial

There is no tutorial for this module

Exercises

1. Integrate Mongoose into an Express project and connect to a local MongoDB database.
   Log if the connection was successful or not to the console.
2. Create a 'movies' schema. Include fields such as title, description, date released, etc.
   Using subdocuments, include additional nested data such as reviews and actors.