Class Participation Module 10 Day 2

https://web322.ca/notes/week08

Team Name: Ogre Inc.	Your Name: Brock Floyd
Team Member 1 (TL) Brock	5
Team Member 2 Carleigh	5
Team Member 3 Kyle	5
Team Member 4 Jean	5
Team Member 5 Emma	5

Connection String:

The connection string is how you are able to connect your database to your application. Once you have the database online with MongoDB Atlas, you can update the connection string. You will take your original connection string and replace some wording within the connection string with the actual name of the database.

Setting up a Schema:

Before creating a connection and working with MongoDB Atlas, you need to figure out what type of data you want to store. The example used in the reading was about the application needing "company" information. Each "company" within the application has unique properties to them. Example:

```
{
  companyName: "The Kwik-E-Mart",
  address: "Springfield",
  phone: "212-842-4923",
  employeeCount: 3,
  country: "U.S.A"
}
```

To work with the example above in our database, we have to create a "schema." We will continue to use this example to talk about schema.

Company Schema:

A schema can be defined as a map to a MongoDB collection and defines the structure of the documents within the collection it maps to. Schemas also provide over features. To represent the "company" example, we will use this code:

```
var mongoose = require("mongoose");
```

```
var Schema = mongoose.Schema;
var companySchema = new Schema({
  "companyName":  String,
  "address": String,
  "phone": String,
  "employeeCount": {
    "type": Number,
    "default": 0
  },
  "country": String
});

var Company = mongoose.model("web322_companies", companySchema);
```

The above code shows that schemas are like a blueprint for a document that's saved in the database. Additionally, it shows 5 properties and their values for "company". These values can be default or sometimes have an index. The last line in the code above tells mongoose to register the schema as a model and to connect it to the collection. The variables created can also be used to create queries, insert, update, or remove (CRUD) documents from the model .

```
// save the company
kwikEMart.save((err) => {
    if(err) {
      console.log("There was an error saving the Kwik-E-Mart company");
    } else {
        console.log("The Kwik-E-Mart company was saved to the
web322_companies collection");
        Company.findOne({ companyName: "The Kwik-E-Mart" })
        .exec()
        .then((company) => {
            if(!company) {
                console.log("No company could be found");
            } else {
                console.log(company);
            }
            // exit the program after saving and finding
            process.exit();
        })
        .catch((err) => {
            console.log(`There was an error: ${err}`);
        });
    }
});
```

The code above shows “The Kwik-E-Mart” company being added to the database by using Mongoose. Additionally, you will see a findOne() function being called to find the company using Mongoose.

```
/ save the company
kwikEMart.save((err) => {
    if(err) {
      console.log("There was an error saving the Kwik-E-Mart company");
    } else {
        console.log("The Kwik-E-Mart company was saved to the
web322_companies collection");
        Company.findOne({ companyName: "The Kwik-E-Mart" })
        .exec()
        .then((company) => {
            if(!company) {
                console.log("No company could be found");
            } else {
                console.log(company);
            }
            // exit the program after saving and finding
            process.exit();
        })
        .catch((err) => {
            console.log(`There was an error: ${err}`);
        });
    }
});
```

This code above shows the new data added being saved.

.exec():

.Exec() is a call that follows a mongoose query, this is how mongoose returns a promise. .Exec() is required to have a proper promise when using mongoose. It is good practice to follow this format. Complete query => Exec()=>(invoked).then()

Arrays and Recursive Schemas:

A recursive schema is a schema that contains an array of elements with the same schema as the definition. This can be used to store tree structures. In the following example, we are creating a tree of comments from a blog post as an example.

Here is an example of the recursive schema needed to begin the tree.

```
const commentSchema = new Schema({
    comment: String,
    author: String,
    date: Date
});
```

```
commentSchema.add({ comments: [commentSchema] });
```

Here we add a comments field using the recursive schema created previously. This indicates that these comments will consist as an array defined by the recursive schema.

```
var commentChain = new Comment({
    comment: "Star Wars is awesome",
    author: "Author 1",
    date: new Date(),
    comments: [{
        comment: "I agree",
        author: "Author 2",
        date: new Date(),
        comments: [{
            comment: "I agree with Author 2",
            author: "Author 3",
            date: new Date(),
            comments: []
        }]
    }]
});
```

Multiple Connections:

With the use of Mongoose, it is possible to configure multiple connections for an app. If multiple connections are needed, it is possible to make changes so that each database can be connected to. encodeURIcomponent is required if special characters are used within the password.

```
let pass1 = encodeURIComponent("pa$$word1"); // this step is needed if
there are special characters in your password, ie "$"
let db1 =
mongoose.createConnection(`mongodb+srv://dbUser:${pass1}@cluster0.0abc1.mo
ngodb.net/db1?retryWrites=true&w=majority`);

// verify the db1 connection

db1.on('error', (err)=>{
  console.log("db1 error!");
});

db1.once('open', ()=>{
  console.log("db1 success!");
});

// ...

let pass2 = encodeURIComponent("pa$$word2"); // this step is needed if
there are special characters in your password, ie "$"
let db2 =
mongoose.createConnection(`mongodb+srv://dbUser:${pass2}@cluster0.2def3.mo
ngodb.net/db2?retryWrites=true&w=majority`)

// ...

var model1 = db1.model("model1", model1Schema); // predefined
"model1Schema" used to create "model1" on db1

var model2 = db2.model("model2", model2Schema); // predefined
"model2Schema" used to create "model2" on db2
```

createConnection is used in place of connect to save the result as a reference to the connection. Db1 and db2 can be used separately. Connections can be tested with .on() and .once().

Connection Warnings:

Depending on the version of Mongoose that you are using, you may encounter connection warnings. Some examples may be…

```
DeprecationWarning: current URL string parser is deprecated, and will be
removed in a future version. To use the new parser, pass option {
useNewUrlParser: true } to MongoClient.connect.
```

or

```
DeprecationWarning: current Server Discovery and Monitoring engine is
deprecated, and will be removed in a future version. To use the new Server
Discover and Monitoring engine, pass option { useUnifiedTopology: true }
to the MongoClient constructor..
```

To resolve these issues, you can add the suggested options to an object and pass them in as the 2nd parameter to your connect / createConnection method.
Ex.

```
let db1 = mongoose.createConnection("Your connection string goes
here", {useNewUrlParser: true, useUnifiedTopology: true});
```

The rest of CRUD:

save():

Save() represents the first letter of CRUD; “C”. To save, or create, a new document, we must first create the document in code using a reference to the schema object. We can then call a built in method, .save() on the model object in order to save it.
Ex.

```
var kwikEMart = new Company({ ... });

kwikEMart.save((err) => {
  if(err) {
    // there was an error
    console.log(err);
  } else {
    // everything good
    console.log(kwikEMart);
  }
});
```

find():

Find() represents the second letter of CRUD; “R”. To Find or Read, is to search/query the data to find specific information.
Ex.

```
Company.find({ companyName: "The Kwik-E-Mart"})
//.sort({}) //optional "sort" -
https://docs.mongodb.com/manual/reference/operator/aggregation/sort/
.exec()
.then((companies) => {
  // companies will be an array of objects.
  // Each object will represent a document that matched the query

  // Convert the mongoose documents into plain JavaScript objects
  companies = companies.map(value => value.toObject());
```

```
});
```

Selecting Specific Fields

Limit results to use only specific fields by passing the list of fields as a space-separated string in the second parameter to the find() method. Example below:

```
Company.find({ companyName: "The Kwik-E-Mart"}, "address phone")
//.sort({}) //optional "sort" -
https://docs.mongodb.com/manual/reference/operator/aggregation/sort/
.exec()
.then((companies) => {
  // companies will be an array of objects.
  // Each object will represent a document that matched the query

  // Convert the Mongoose documents into plain JavaScript objects
  companies = companies.map(value => value.toObject());
});
```

updateOne() / updateMany()

Use schema object updateOne()/updateMany() to update documents in a collection. updateOne() will only update most top document with matching filter, updateMany() will update all documents. Be sure to use arguments for the query to select which documents to update and the fields to set for the documents that match the query.

```
Company.updateOne( // can also use updateMany to update multiple documents
at once
  { ... query ... },
  { $set: { ... fields to set ... } }
).exec();
```

Example:

```
Company.updateOne(
  { companyName: "The Kwik-E-Mart"},
  { $set: { employeeCount: 3 } }
).exec();
```

deleteOne() / deleteMany()

The deleteMany() returns a document containing the deleteCount field that stores the number of deleted documents.

To delete a single document from a collection, you can use the deleteOne() method.

```
Company.deleteOne({ ... query ... }) // can also use deleteMany to delete
multiple documents at once
.exec()
.then();

Example:
Company.deleteOne({ companyName: "The Kwik-E-Mart" })
.exec()
.then(() => {
  // removed company
  console.log("removed company");
})
.catch((err) => {
  console.log(err);
});
```

Unique Indexes:

Unique Indexes are identifiers that can be assigned to all fields within a database. A unique index may be assigned to more than one field. MongoDB automatically adds a unique index to the id field of each document. Unique indexes are able to be called from across the database. Unique index parameters can be added to fields using Mongoose. If indexes are non-existent they will be automatically applied on startup. Indexes are commonly applied to fields that have unique values applied across many documents. The example given is assigning a unique index to for a company name field. “Unique true” needs to be added to this field in order to enforce this. The example below shows this:

```
/ define the company schema
var companySchema = new Schema({
  "companyName":  {
    "type": String,
    "unique": true
  },
  "address": String,
  "phone": String,
  "employeeCount": {
    "type": Number,
    "default": 0
  },
  "country": String
});

// require mongoose and setup the Schema
var mongoose = require("mongoose");
var Schema = mongoose.Schema;
```

```
// connect to Your MongoDB Atlas Database
mongoose.connect("Your connection string here");

// define the company schema
var companySchema = new Schema({
  "companyName": {
    type: String,
    unique: true
  },
  "address": String,
  "phone": String,
  "employeeCount": {
    "type": Number,
    "default": 0
  },
  "country": String
});
var Company = mongoose.model("web322_companies", companySchema);

// create a new company
var kwikEMart = new Company({
  companyName: "The Kwik-E-Mart",
  address: "Springfield",
  phone: "212-842-4923",
  employeeCount: 3,
  country: "U.S.A"
});

// save the company
kwikEMart.save((err) => {
  if(err) {
    console.log(`There was an error saving the Kwik-E-Mart company:
${err}`);
  } else {
    console.log("The Kwik-E-Mart company was saved to the web322_companies
collection");
  }
  Company.find({ companyName: "The Kwik-E-Mart" })
  .exec()
  .then((company) => {
    if(!company) {
      console.log("No company could be found");
    } else {
      console.log(company);
    }
    // exit the program after saving
```

```
    process.exit();
  })
  .catch((err) => {
    console.log(`There was an error: ${err}`);
  });
});
```

NOTE
Indexes are kept in MongoDB and applied by the database. If the field has a unique index and you try to give it a different value, it will return an error. This is shown in the example below:

```
$ node week8
There was an error saving the Kwik-E-Mart company:
WriteError({"code":11000,"index":0,
"errmsg":"E11000 duplicate key error collection: web322.web322_companies
index:
companyName_1 dup key: { : \"The Kwik-E-Mart\" }","op":{"companyName":"The
Kwik-E-
Mart","address":"Springfield","phone":"212-842-4923","country":"U.S.A","_i
d":"5864148f8e6a802830973e76","employeeCount
":3,"__v":0}})
[ { _id: 586412c78b50ee22cc9691d6,
    companyName: 'The Kwik-E-Mart',
    address: 'Springfield',
    phone: '212-842-4923',
    country: 'U.S.A',
    __v: 0,
    employeeCount: 3 } ]
```