Seneca
SCHOOL OF INFORMATION AND COMMUNICATIONS TECHNOLOGY

WEB322

Web Programming Tools and Frameworks

Schedule | Graded Work | Resources | Cyclic Guide | MyApps Instructions | Code examples

WEB322 Week 8 Notes

Week 8 - MongoDB

Introduction to MongoDB

What is MongoDB?

MongoDB is an open source database that stores its data in JSON like format (technically it's stored as BSON data but you will interact with its data in JSON format). MongoDB is classified as a NoSQL database. NoSQL is quickly becoming a popular alternative to traditional Relational Databases (RDBMS). The term NoSQL comes from "Not only SQL" and is intended to mean that it is a type of database system that can store data in non traditional tabular and relational format. This is why MongoDB is considered a NoSQL database. It stores its data using object notation.

How does it relate (no pun intended) to Traditional SQL systems?

When you think of how a traditional relational database system works you have tables and records, primary

and foreign keys, and joins between tables when writing queries.

MongoDB has similar functionality with a few major differences.

Let's look at a comparison table of features so we can become familiar with the terminology of the MongoDB world and NoSQL databases.

RDBMS term	MongoDB term
Table	Collection
Record	Document
Column	Field
Joins	Embed data or link to another collection

This page compares MySQL to MongoDB and explains when it makes sense to use one or the other or both!

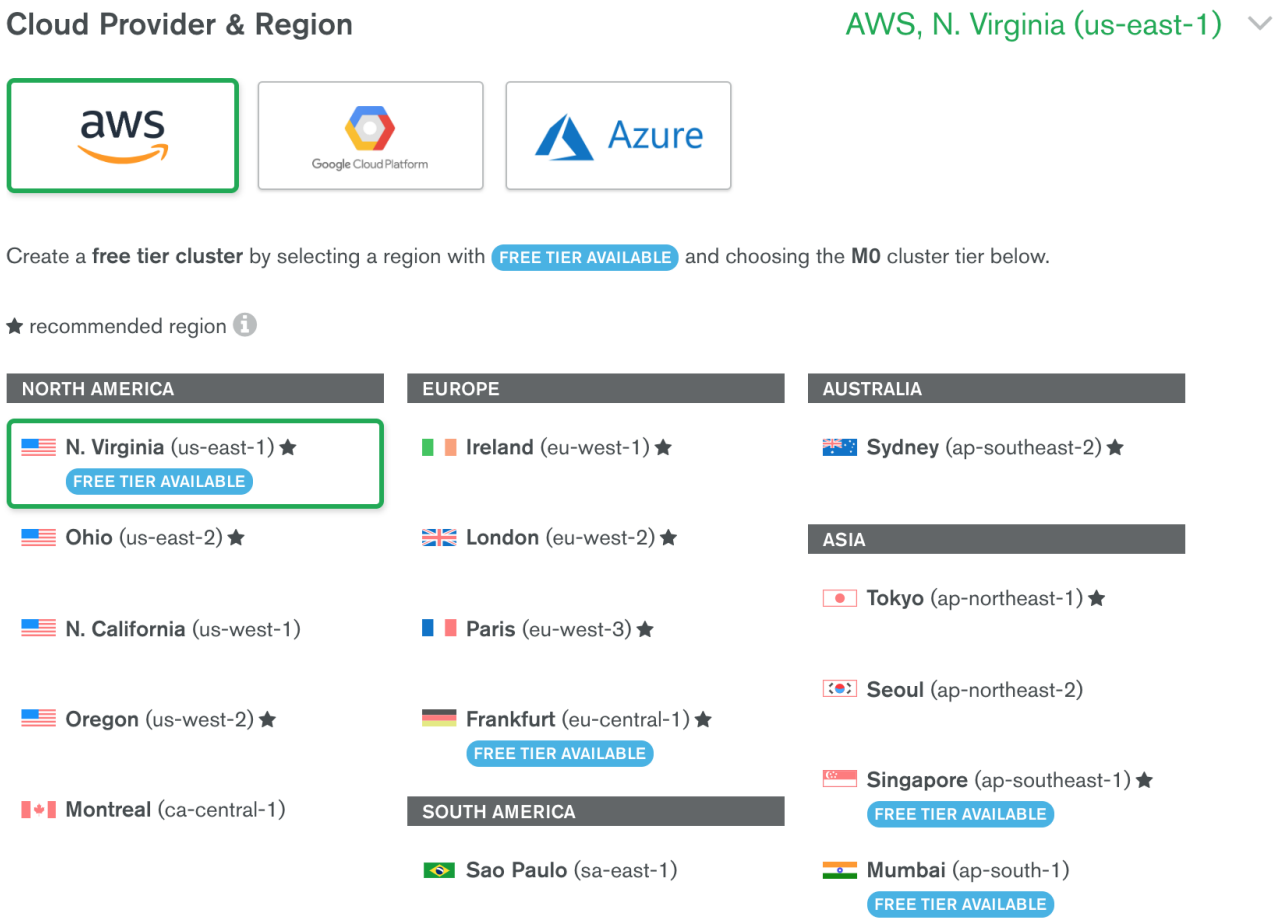
Queries are still queries, and primary/foreign keys can still be called the same but are implemented via schema and indexes. (more on that part later)

Setting up a MongoDB Atlas account

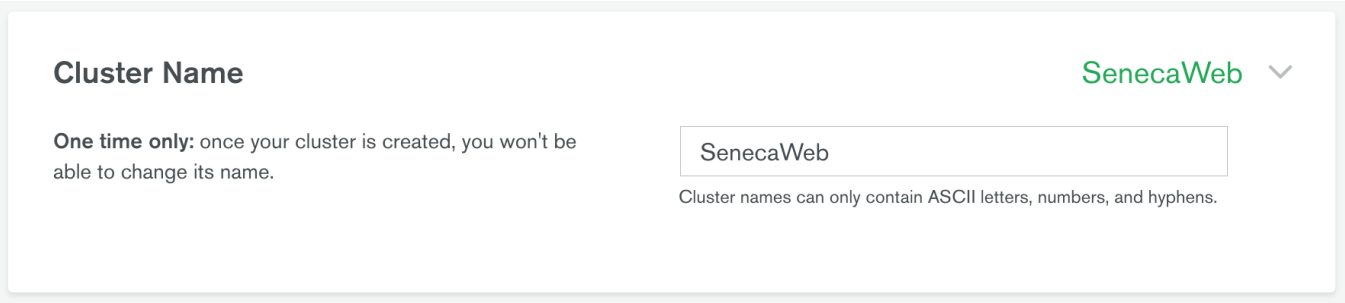
MongoDB Atlas is an online service that hosts MongoDB in the cloud. You can sign up for a free account to use in this course with your Cyclic account.

To get started signup for an account at https://www.mongodb.com/cloud/atlas and click the "Start free" button.

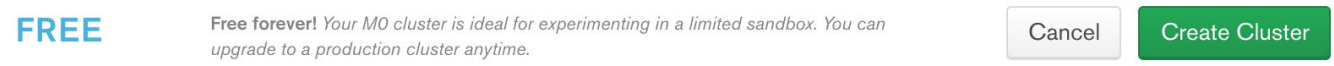
Once your account is setup, you will be taken to the start screen with a modal window suggesting you "Build my first cluster". You can close this modal window, as we will not be creating a cluster just yet. First, we want to ensure that we have selected all the free options, ie:



Next, we should change the cluster name from “Cluster0” to something more recognizable, ie “SenecaWeb”.



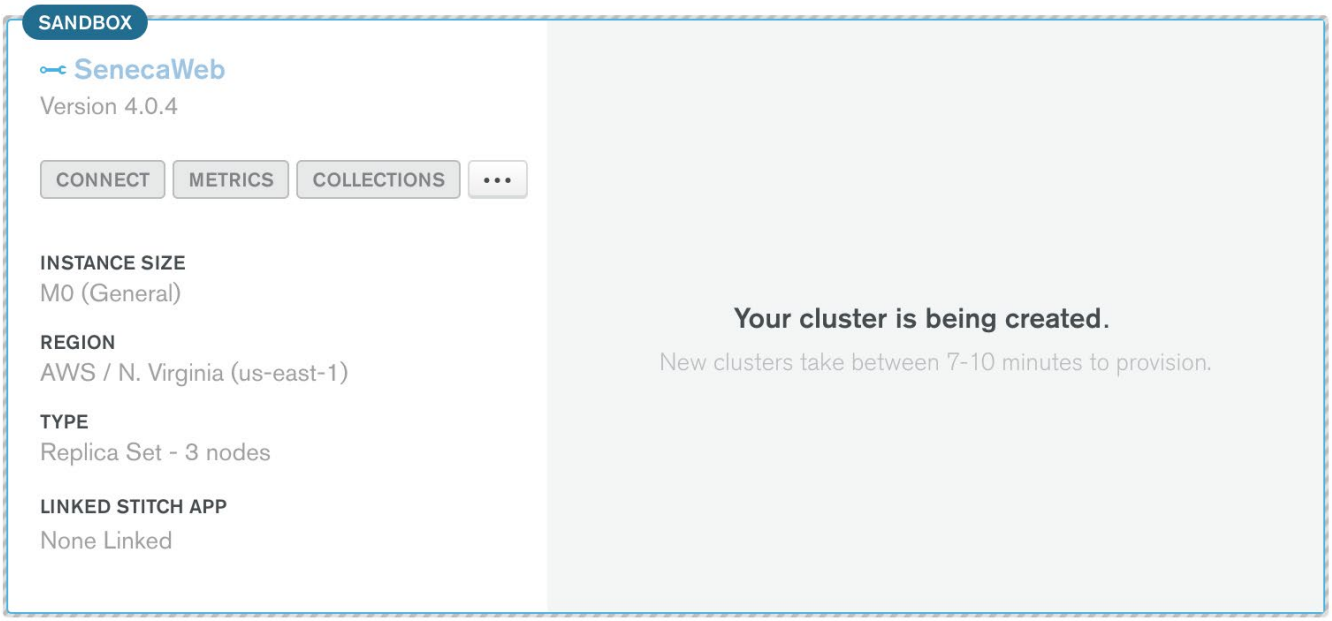
Once this is complete, we can go ahead and “Create Cluster”



Once you pass the Capcha test (proving you’re human), you will be redirected to the main page for managing your clusters in MongoDB Atlas (Note: Feel free to close the “Get Started” checklist on the bottom left corner of the screen)

Here, you should see a “Your cluster is being created.” message. Before we proceed, we must wait for this

to finish (it could be between 7-10 minutes).

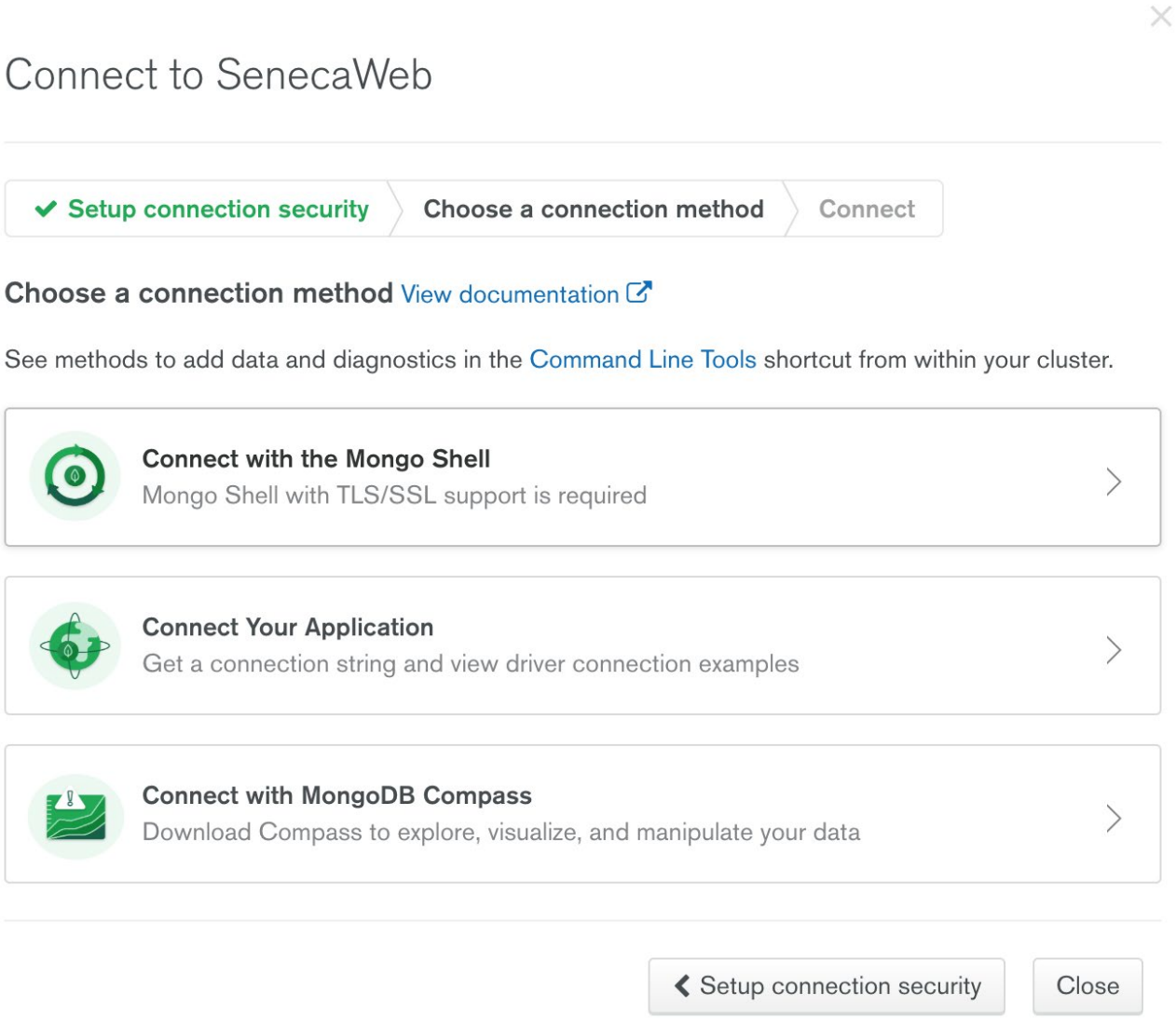


Once your cluster has been created, you can click the “CONNECT” button. This will cause a new modal to appear, which will allow us to Whitelist IP Address to connect to the cluster, as well as to create a “MongoDB User”.

1. For this step, click the “Add a Different IP Address” button and enter “0.0.0.0/0” for the IP Address and “All” for the Description. You can then click the “Add IP Address” button. Essentially, we’re allowing all IP addresses to connect to our cluster. This will make it easier for us to work with the databases in the cluster. If we were creating a cluster for Production, we would ensure that only our deployed app has access.

2. For “Create a MongoDB User”, pick something that you will remember, ie: Username: “dbUser” and Password: (something that you will remember). Once the data is entered, you can click the “Create MongoDB User” button.

Once this is complete, click the “Choose a connection method” button at the bottom of the modal window. This will bring you to a new screen allowing you to choose how you wish to “Connect to SenecaWeb”:



From here, click on the "**Connect Your Application**" button.

Under the first option, make sure that Node.js is selected for "Driver" and "Version" is set to 3.6 or later. Select the connection string and copy it (alternatively hitting the "Copy" button). Next, paste it in a text file for now. You will notice that there's a space for <password> - simply replace this with the actual password that you created for user "dbUser" (above).

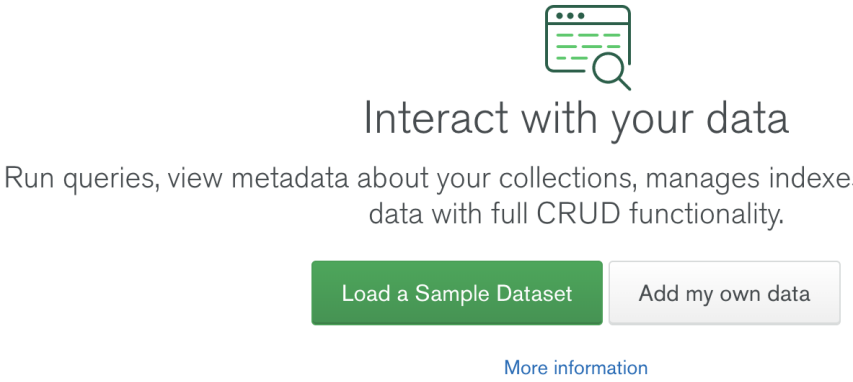
Once you have **copied the connection string** (we will need this for our code below), you can close the modal window using the "Close" button located at the bottom of the modal window.

Creating a Database on MongoDB Atlas

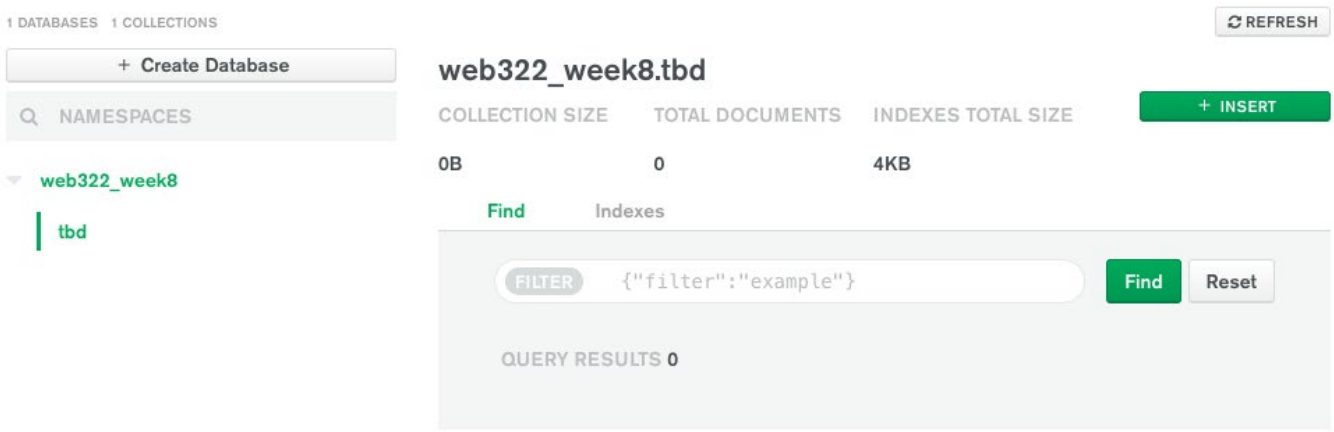
Now that the "SenecaWeb" cluster is all set up, we can add a Database to connect to. We'll be creating a "web322_week8" database for the sake of the example.

To get started, click the blue “SenecaWeb” link from the “Clusters Overview” Section:

This will take you to a detailed view of your “SenecaWeb” cluster. From here, you will notice a “Collections” tab. Click this to open the data on all “Collections” contained in this cluster. Since we have not created any Databases yet, we will be greeted with the following message:



Go ahead and click the “Add my own data” button. This will open a small modal window asking for the “Database Name” and “Collection Name”. For “Database Name” enter “web322_week8” and for “Collection Name” simply enter “tbd” since we don’t know what collections we will have just yet, and we cannot proceed without entering one (Note: ensure that “CAPPED COLLECTION” remains unchecked). With the data entered, click the green “Create” button. Once this is complete, you will be shown the following information under “collections”:



Unfortunately, we cannot have a Database in MongoDB Atlas without a collection, so leave “tbd” there for the time being. We can remove it later.

Updating the connection string to point to our new “web322_week8” database.

Now that we have a new Database online with MongoDB Atlas, we can update our *connection string* to point to it (NOTE: This is what is used in the “mongo.connect” and “mongo.createConnection” methods mentioned below).

To accomplish this, simply take your original *connection string* and look for the last part of the url before the query parameters, ie: **mmongodb.net/**. To connect to a specific database, simply **add** the database name here, ie: **mongodb.net/web322_week8**.

Mongoose.js

When we work with MongoDB in node, we won’t work directly with the MongoDB driver. Instead we will use a popular open source module that wraps up the Mongo driver and provides extra functionality, such as: “providing a straight-forward, schema-based solution to model your application data as well as including built-in type casting, validation, query building, business logic hooks and more, out of the box”.

Installing mongoose is easy when you have node installed. Just use npm install to grab it.

```
npm install mongoose
```

This will save it to your package.json file and allow you to require it and start building model files and working with documents.

Setting up a schema

Before we look at how to establish a connection to our MongoDB Atlas DB and work with the data using Mongoose, let’s first determine the type of data that we wish to store. For example, let’s say that our application requires “company” information to be persisted. Each “company” used by our system can be represented using the following properties (ie, its “shape”), as illustrated below for “The Kwik-E-Mart”:

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

```
}
```

To begin working with “companies” like this in our database using Mongoose, the first step is to create a “schema”.

Company schema

From the documentation: “Everything in Mongoose starts with a Schema. Each schema maps to a MongoDB collection and defines the shape of the documents within that collection”. So, for us to work with a specific collection in our MongoDB database, we must first define a “schema”, which defines the structure of the documents to be added to the collection (as well as to provide other features such as “validators”, etc.).

To represent the above company data as a Mongoose Schema, we can use the following 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);
```

Essentially, a schema is like a blueprint for a document that will be saved in the DB. Here, we define the fields that can exist on a document for this collection, and setting their expected types, default values, and sometimes if they are required, or have an index on them.

In the above code, we have defined a Company schema with 5 properties as discussed, and set their types appropriately. The employee count is not just a simple number, we also want to include a default value of 0 of the count field is not supplied. Using defaults where it makes sense to have them is good practice.

The last line of code tells mongoose to register this schema (companySchema) as a model and connect it to the web322_companies collection (Note: the “web322_companies” collection will be automatically created if it doesn’t exist yet). We can then use the Company variable to make queries against this collection and insert, update, or remove documents from the Company model.

With this in mind, let’s go ahead and add “The Kwik-E-Mart” to the database using Mongoose.

```
// 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":  String,
  "address": String,
  "phone": String,
  "employeeCount": {
    "type": Number,
    "default": 0
  },
  "country": String
});
// register the Company model using the companySchema
// use the web322_companies collection in the db to store documents
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().then(()=>{
      console.log("The Kwik-E-Mart company was saved to the
web322_companies collection");
}).catch(err=>{
      console.log("There was an error saving the Kwik-E-Mart company");
});
```

Now we can add a findOne() call to find this company using Mongoose.

Modify the save call in the code above to look like the following:

```
// save the company
kwikEMart.save().then(()=>{
  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}`);
  });
}).catch(err=>{
  console.log("There was an error saving the Kwik-E-Mart company");
});
```

NOTE If you examine the output, you will notice that the data returned includes two extra fields, added by default to our document:

- _id: A unique ObjectID
- _v: The versionKey

.exec()

The .exec() call is added after a mongoose query to tell mongoose to return a promise. If you leave out the .exec(), mongoose will still work with .then() calls but the object returned will not be a proper promise. It is good practice to always use .exec() after your query has been setup and before the .then() method is invoked.

Arrays and Recursive Schemas

Before we move on to "The rest of the CRUD" below, let's take a quick look at how we can define a "recursive" schema. Essentially, this is a schema that contains an array of elements with the same schema as the definition. We can use this to store tree structures such as file / folder hierarchies or comment trees for a blog post. For example: say we wish to store a tree of comments, where each comment can have one or more comments, which can have one or more comments, and so on. We can specify our recursive "commentSchema" using the following code:

```
const commentSchema = new Schema({
```

```
    comment: String,
    author: String,
    date: Date
});

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

Here, we add a “comments” field with a type of “[commentSchema]” to the original “commentSchema”. Using this syntax, we indicate that all “comments” will consist of an Array defined by “commentSchema”. Now, we can easily create documents that appear in this format, ie:

```
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: []
        }]
    }]
});
```

Quick Note: Multiple Connections

Using Mongoose, it is also possible to have multiple connections configured for your application. If this is the case, we just have to make a few small changes on how we **connect** to each DB, and how we define our models (**NOTE**: the use of the “encodeURIComponent” is necessary if your password contains special characters, ie “$”) :

```
// ...

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.mongod
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.mongod
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

// ...
```

Instead of using **“connect”**, we instead use **“createConnection”** and save the result as a reference to the connection (ie: **“db1”** and **“db2”** from above). We can then use **db1** or **db2** to create models on each database separately. Additionally, if we want to *test* the connection, we can use the **.on()** and **.once()** methods of each connection.

Connection Warnings

Depending on the version of mongoose that you’re using, you may encounter warnings such as:

```
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, simply add the suggested options to an object and pass them in as the 2nd parameter to your connect / createConnection method, ie:

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

The rest of the CRUD

Now that we've discussed find, let's talk about the other functions used with mongoose to do the rest of the CRUD functionality. Here are some examples of other find methods, inserting, updating, and removing documents.

save()

To "save" (create) a new document, we must first create the document in code using a reference to the schema object we want. Then we can call a built in method, .save() on the model object to save it.

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

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

find()

```
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

If we wish to limit the results to include only specific fields, we can pass the list of fields as a space-separated string in the second parameter to the find() method, ie:

```
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());
});
```

For complex queries (ie: "greater than", "in", "or", etc, etc.) see the Mongoose Query Guide and the MongoDB documentation under Query and Projection Operators

Note: You will notice that in our "then" callback function(s), we have line:

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

This is to ensure that our returned "companies" Mongoose documents are converted to plain JavaScript objects.

updateOne() / updateMany()

We use the schema object to update vs an instance of a model like we did with save(). update() takes 3

arguments: the query to select which documents to update, the fields to set for the documents that match the query (see update operators, ie: $set, $push and $addToSet), and an option for if you want to update multiple matching documents or only the first match.

```
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()

```
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);
});
```

Indexes

Indexes are a large enough topic on their own for an entire week of lecture. For the scope of this course we will talk about the most common type of index used in MongoDB that you will need for every collection.

Unique indexes

A unique index is applied at the database level and can be attached to one or more fields of a document. The first example of a unique index would be on the _id field of all documents that MongoDB adds automatically. As mentioned above every document gets an _id field added to it by default and that _id value is globally unique across the DB. MongoDB automatically adds the _id field to your documents but it also adds a unique index to the _id field for the schema. Mongoose schemas can be customized to add your own additional unique index constraints to other fields as needed. Mongoose will add the indexes, if they don't exist, to the collections in your db when your app starts up and initializes.

The most common use for this is when you want to enforce a unique value across all documents in a collection on a certain field. A perfect use case for this would be on the companyName for our company schema above. It wouldn't make sense to have multiple companies with the same name in the system. To add a unique index in to the companyName field, we just have to add unique:true to the schema declaration from before.

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

Remember: Your indexes are stored **in MongoDB** and will be enforced by the database.

Let's look at the finalized code and what happens when we try to insert a company with a companyName that already exists when the companyName has a unique index.

```
// 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().then(() => {
  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);
      }
    })
    .catch((err) => {
      console.log(`There was an error: ${err}`);
    });
}).catch(err => {
    console.log(`There was an error saving the Kwik-E-Mart company:
${err}`);
});
```

Running it a second time:

```
$ node week8
There was an error saving the Kwik-E-Mart company: MongoServerError: E11000
duplicate key error collection: web322.web322_companies index:
companyName_1 dup key: { companyName: "The Kwik-E-Mart" }
```

As you can see MongoDB threw back an error (E11000 duplicate key error). This is the most common form of error you'll encounter on saving a document to the database. You can handle it and act according to what your application should do.

Week 8 example

The week 8 example explores connecting to a mongoDB database and saving records for metadata about a photo upload. It allows the owner to upload a photo, add a name, email, and caption to the photo, and save it. The photo itself will be written to the file system and the supporting data about the photo will be saved in a document in the web322_week8_photos collection. Try it out with a local install of MongoDB and then try creating a MongoDB Atlas account and connecting your MongoDB Atlas db to your week8 example running on Cyclic.

Sources

- MongoDB official documentation
- Mongoose documentation
- MongoDB Cheat Sheet
