
WEB322 – Week 8 – Class 1

WEB322 H

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 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 database system that can store data in non traditional tabular and relational format. This is why MongoDB is considered a No 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, 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. (I

that part later)

Installing MongoDB locally

Lets get MongoDB installed locally. It should take about 5-10 minutes.

OSX

Windows

- Download the latest version of MongoDB for OSX. Get the 64 bit community version with SSL support.
- Extract the .tgz file
- Copy the extracted folder (mongodb-osx-x86_64-x.x.x) into your Applications folder
- Create a .bash_profile file (note: the version of mongoDB maybe different (ie: change x86_64-x.x.x to whatever version using):

```
$ cd ~
$ pwd
/Users/userName
$ touch .bash_profile
$ open -a TextEdit .bash_profile

export PATH="/Applications/mongodb-osx-x86_64-x.x.x/bin:$PATH"

##restart terminal

$ mongo -version
MongoDB shell version: x.x.x
```

MongoDB should now be installed!

Running the MongoDB Server

Now we want to try running the server and issuing commands in the shell.

- Create a new directory for your mongoDB database (called db) - this can be anywhere: why don't we use the Desktop?
- Open a terminal window and enter the command:

```
mongod --dbpath ~/Desktop/db
```

(or if you're on Windows, provide the absolute path (ie: **mongod --dbpath C:\users\username\Desktop\db**)

- This will get the "mongoDB" database engine running

Connecting to the MongoDB shell

Now that we know how to run MongoDB at anytime, we will learn how to connect to the db through the shell and see how we issue commands. We will create a web322 database, add collections for companies and users, and insert one company and or

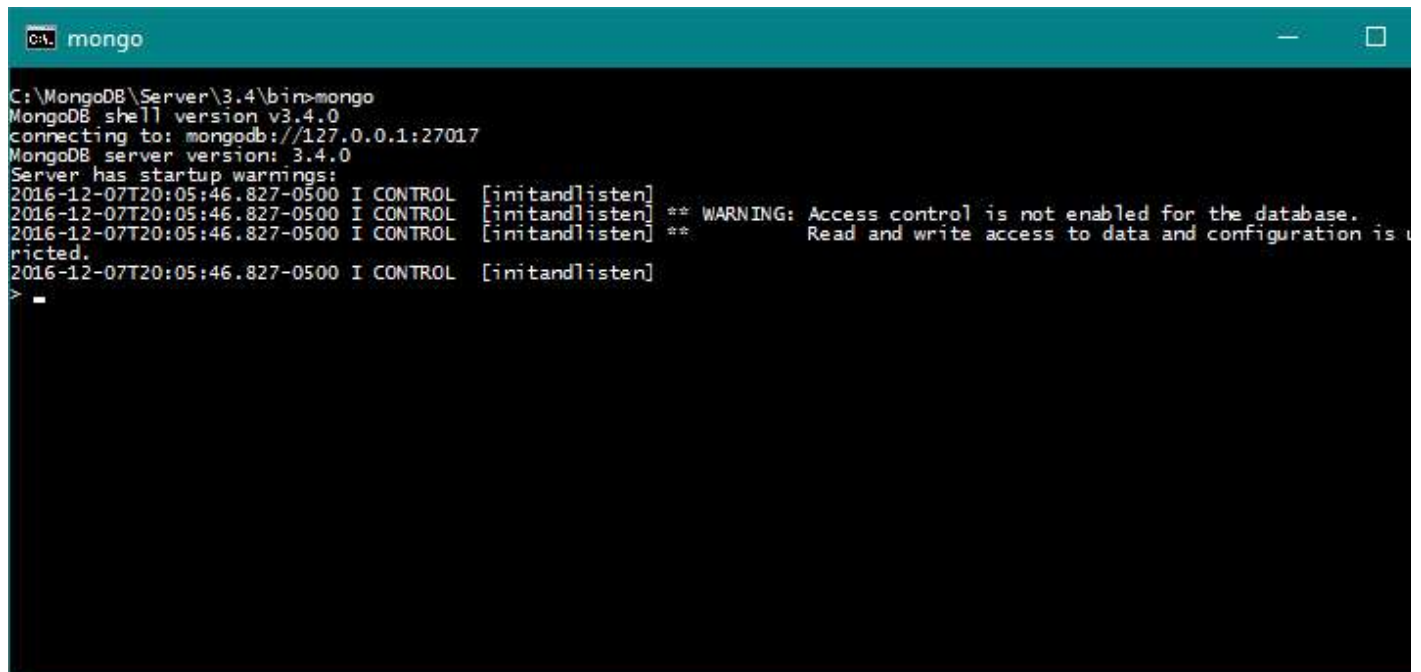
into the collections.

With mongod running, we will now run "mongo" to connect to the running server process.

Open a new terminal and execute the command:

```
mongo
```

Once connected to the db with the shell, your window will look like this:

A screenshot of a Windows command prompt window titled "mongo". The window shows the execution of the "mongo" command from the directory "C:\MongoDB\Server\3.4\bin". The output displays the MongoDB shell version (v3.4.0) and the connection to the local server (mongodb://127.0.0.1:27017). It also shows the server version (3.4.0) and startup warnings, including a warning that access control is not enabled for the database. The prompt "y>" is visible at the bottom of the terminal output.

```
C:\MongoDB\Server\3.4\bin>mongo
MongoDB shell version v3.4.0
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.0
Server has startup warnings:
2016-12-07T20:05:46.827-0500 I CONTROL [initandlisten]
2016-12-07T20:05:46.827-0500 I CONTROL [initandlisten] ** WARNING: Access control is not enabled for the database.
2016-12-07T20:05:46.827-0500 I CONTROL [initandlisten] ** Read and write access to data and configuration is u
ricted.
2016-12-07T20:05:46.827-0500 I CONTROL [initandlisten]
y>
```

Databases, Collections, and Documents

Now that we are connected to the shell, we can start working with some commands and data. For this example you will see in console that it says:

```
WARNING: Access control is not enabled for the database.
```

This is letting you know you have not setup any user credentials to log in to the db with. This is OK for now, your database server is running locally on your computer. No one will be able to connect to your database outside your computer yet. In a real production development environment, we would enable authentication to the database. This is outside the scope of this course.

Now it's time to try out some commands.

Let's show the databases currently on the server

```
show dbs
```

We can see that there are 2 databases built in and available from a fresh install: admin and local. We won't be using either of these databases in this course.

When we want to create a new database we just have to simply tell the shell we want to use that database and it will automatically create it if it doesn't exist. Try this command out.

```
use web322
```

Now we can run `show dbs` again and you will see it still doesn't exist. This is because `mongodb` will only actually create the db collections required once you start inserting data. So let's insert a company to a `web322_companies` collection!

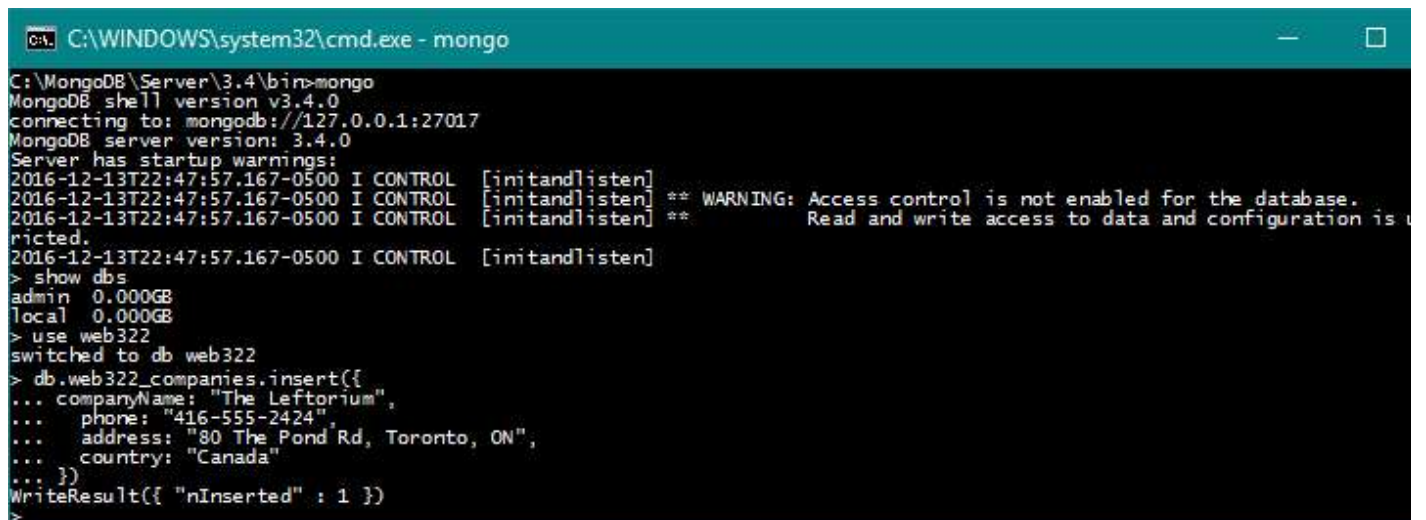
Let's decide on what a company JSON object would look like to decide what fields we want for the document we are about to insert. Some obvious ones we might want are a company name, phone number, address, and country. Let's start with that.

```
{
  companyName: "The Leftorium",
  phone: "416-555-2424",
  address: "80 The Pond Rd, Toronto, ON",
  country: "Canada"
}
```

Ok, we've decided on what a company record will look like for now. Let's go ahead and insert this company into the `web322_companies` collection...

```
db.web322_companies.insert({
  companyName: "The Leftorium",
  phone: "416-555-2424",
  address: "80 The Pond Rd, Toronto, ON",
  country: "Canada"
})
```

You should get a result like the following:



```
C:\WINDOWS\system32\cmd.exe - mongo
C:\MongoDB\Server\3.4\bin>mongo
MongoDB shell version v3.4.0
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.0
Server has startup warnings:
2016-12-13T22:47:57.167-0500 I CONTROL [initandlisten]
2016-12-13T22:47:57.167-0500 I CONTROL [initandlisten] ** WARNING: Access control is not enabled for the database.
2016-12-13T22:47:57.167-0500 I CONTROL [initandlisten] ** Read and write access to data and configuration is unrestricted.
2016-12-13T22:47:57.167-0500 I CONTROL [initandlisten]
> show dbs
admin 0.000GB
local 0.000GB
> use web322
switched to db web322
> db.web322_companies.insert({
...   companyName: "The Leftorium",
...   phone: "416-555-2424",
...   address: "80 The Pond Rd, Toronto, ON",
...   country: "Canada"
... })
WriteResult({"nInserted" : 1 })
>
```

You should see the result `writeResult{ nInserted : 1 }`. This means a write succeeded and the number of documents inserted (nInserted) was 1.

We can do a quick query to look at the record from the shell with the following command:

```
db.web322_companies.find()
```

You'll get a result like:

```
{ "_id" : ObjectId("5850c358dbf675a87f27a456"), "companyName" : "The Leftorium",  
  "phone" : "416-555-2424", "address" : "80 The Pond Rd, Toronto, ON", "country" : "Canada" }
```

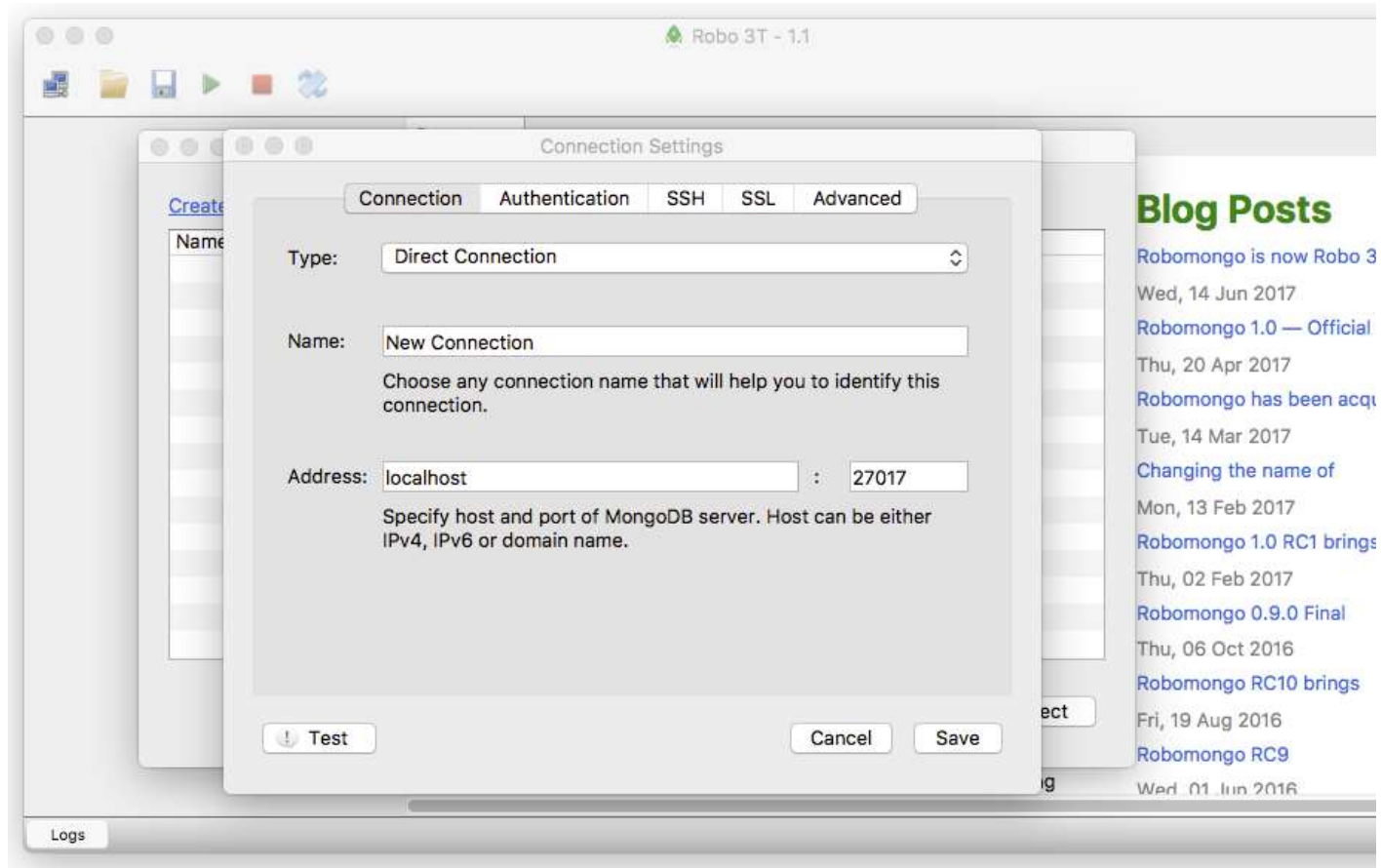
You'll notice it's the same as the document you inserted with the added field of "_id". Every document in MongoDB gets an _id property by default. These are **globally unique** to the entire database and probably even the world! So no need to worry about conflicts anytime you do a find by the _id!

Now let's take a look at the data in the database using a GUI tool.

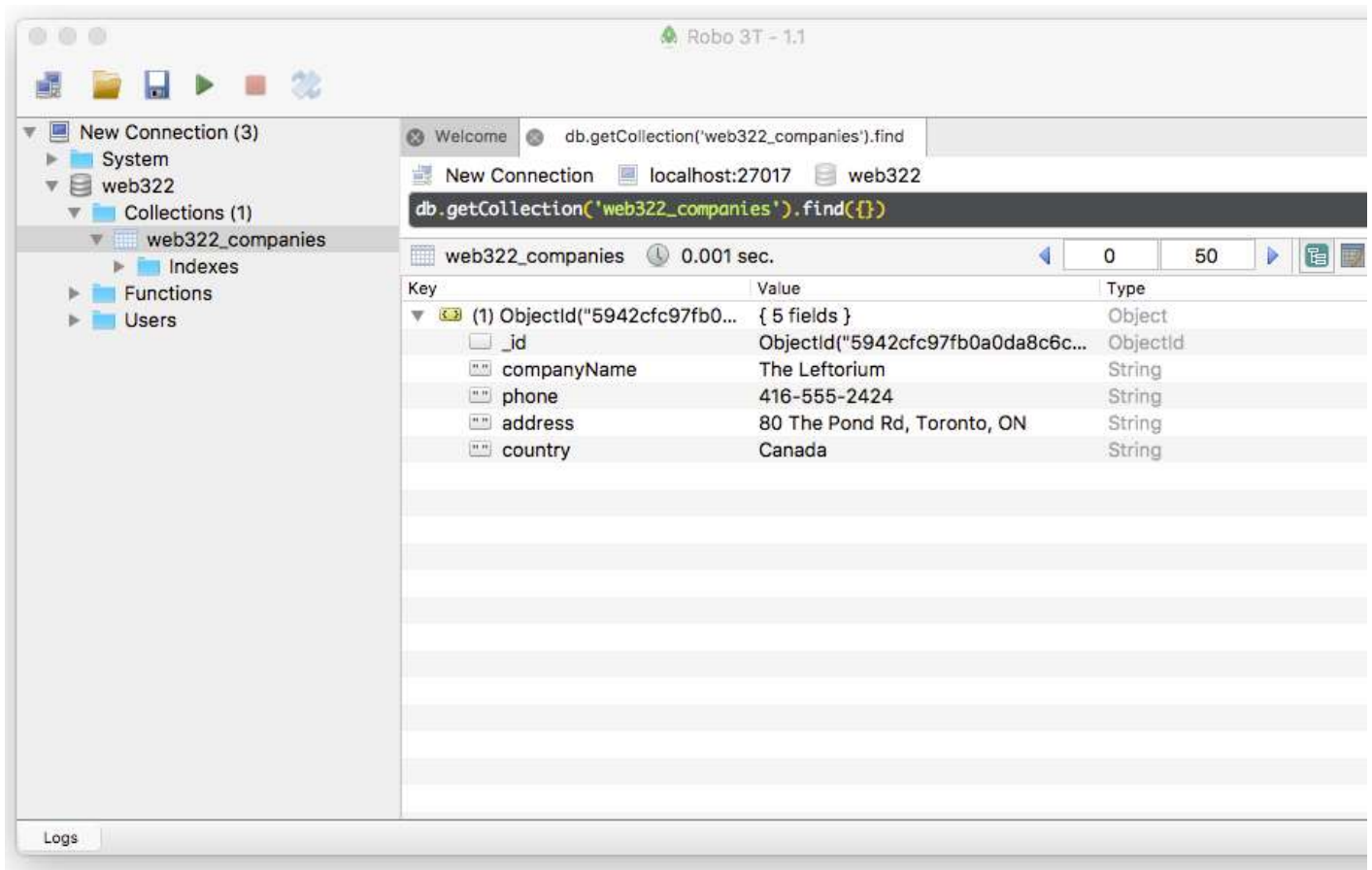
Tools

RoboMongo (Robo 3T) is an open source project that is free to use and [available to download](#) for Windows, Mac, and Linux.

install Robo 3T and connect to the DB with host: localhost, and port: 27017 to inspect our inserted document.



web322_companies. to view the documents inside this collection just double click on it and a window will open in table view w contents. You should see your document as a row in the table with all it's properties.



Now let's setup an online account with MLAB and create our own mongodb in the cloud for use with Heroku.

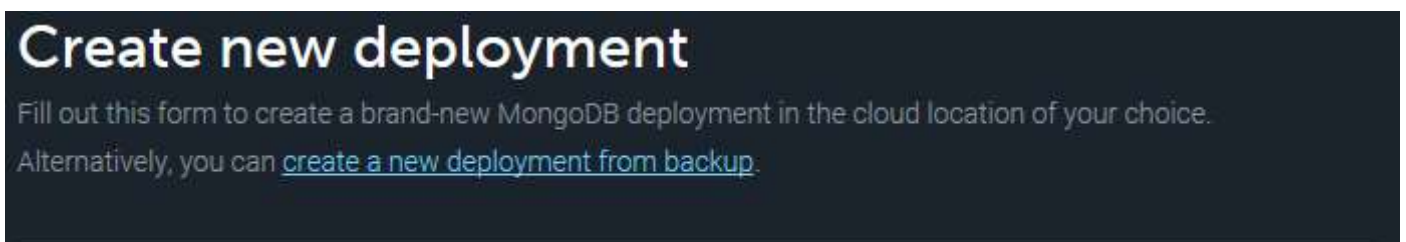
Setting up an MLAB account






Mlab 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 GitHub account.

To get started sign up for an account at www.mlab.com

Once your account is setup and you have verified it you can login and create a new deployment. Choose any option that is free. Amazon Web Services has a single node deployment in US EAST region that is free with 500mb of space to use. This is plenty for the course. Name your database web322_week8.



Cloud Provider:

Region: Amazon's US East (Virginia) Region (us-east-1) ▼

Plan ([view plan details](#)):

Single-node
Replica set cluster

These plan(s) are perfect for development/testing/staging environments as well as for utility instances that do not require high availability.

Standard Line

The most economical plans for applications running on AWS.

☐ **Sandbox** (shared, 0.5 GB) FREE

Once your deployment is setup you will need to add a user to that database

⚠ A database user is required to connect to this database. To create one now, visit the 'Users' tab and click the 'Add database user' button.

Collections
Users
Stats
Backups
Tools

Database Users + Add database user

[None at this time]

THE MONGODB CONNECTION STRING FORMAT

Create a user, choose a password, and then after your user is created look just above the users table on the mlab website for connection string for your db. It will have the following format:

```
mongodb://<dbuser>:<dbpassword>@<subdomain>.mlab.com:<port>/web322_week8
```

The <dbuser> is your db user name, the <dbpassword> is your users password, the <subdomain> will be given to you as well, <port> and it will end with /web322_week8 which is the database name.

This is the standard notation for a connection URI to connect to a MongoDB database. For a local install it has no user or password and simply follows this format:

```
mongodb://localhost:27017/web322_week8
```

The default port for a local install is 27017.

You can connect to your mlab database server using the Robo 3T tool to test the connection. Enter the following details for a connection

- Connection Tab - Address: dsxxxxxx.mlab.com (where xxxxxx will be specific to your own mLab Database)
- Connection Tab - Port: 27892 (or whatever was provided by mLab)
- Authentication Tab - Database: Your mLab database name
- Authentication Tab - User Name: Your user for your new mLab database
- Authentication Tab - Password: Your password for your new mLab database user
- Authentication Tab - Auth Mechanism: (leave this at the default - SCRAM-SHA-1)

Once you are able to connect to your Mlab DB from Robo 3T you can save this connection info for later to use with the week 8 example to connect your Heroku build of week 8 to your Mlab db and test out the photo uploader album online!

For now, keep track of your database username and password for later and remember your database name as well (web322_1 you followed the instructions above). If you save the connection details in to Robo 3T you can look at them later if you forget a good idea to do this now.

Now it's time to work with MongoDB in a node.js app.

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 for declaring schemas/models, validating documents having virtual properties on a model, and instance and static methods on a model object.

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

```
npm install mongoose --save
```

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

MongoDB cheat sheet

There is a very detailed MongoDB cheat sheet available for download as a PDF that you can use as a reference for looking up commands and features for the MongoDB shell. Each command links directly to the official documentation for more information. [Cheat sheet is available here.](#)

Querying MongoDB with Mongoose in Node.js

Now that we have mongoose installed we can start looking at how the basic CRUD operations work (note: the examples in this are based on Mongoose version 4).

Let's start with an example app that will make use of at least one route for each of the main CRUD operations: Create, Read, Update, Delete. In the Mongo shell and in Mongoose we use the functions: insert(), find(), update(), and remove().

Let's define a simple schema for the two types of objects mentioned earlier in the lecture: companies and users.

Setting up a schema

COMPANY SCHEMA

For our company we will want at a minimum a company name, address, phone number, employee count, and country, to match the structure of the company object we have already inserted. MongoDB will automatically include the `_id` field, so we don't need a schema (all documents inserted get an `_id` field by default). The previous document did not have an employee count so we will ensure the default for the count is 0 if this field does not exist on a document. The next time a document is saved which has missing fields that the schema supports, any defaults will be applied.

```
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);
```

A schema is like a blueprint for a document that will be saved in the DB. It's somewhat like a class or struct definition in other languages you are used to. We are defining the fields that can exist on a document for this collection, and setting their expected default values, and sometimes if they are required, or have an index on them.

In the above, 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 if the count field is not supplied. Using defaults makes sense to have them as 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.

Now let's add a second company to the database using mongoose in node.js instead of the mongo shell.

Here is a simple app you can run with node.js and it will insert another company to the db.

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

// connect to the localhost mongo running on default port 27017
mongoose.connect("mongodb://localhost/web322");
```

```
// 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((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");
  }
  // exit the program after saving
  process.exit();
});
```

Now we can add a `findOne()` call to find this company using mongoose. Modify the save call in the code above to look like this:

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

```

    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}');
});
});

```

A couple of new things are used above. Let's go over them briefly.

.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 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.

.THEN() AND .CATCH()

The .then() function is chained after the exec() and it will receive the results of the query that was run. For example, if the query is finding a document, you will receive the document as an argument to the then() function. .then() can take 2 functions, the first is a callback function that receives the results of the query, the second function is an error handling function. You can also use a .catch() after a .then() instead to catch an error as shown above.

```

.then((result) => {
  // use result
})
.catch((error) => {
  // use the error
})

OR

.then((result) => {
  // use the result
}, (error) => {
  // use the error
})

```

Notice the first method is .then(function).catch(function) and the second method is .then(function1, function2)

Most people agree the second method is harder to read but there are some cases where you might need to use this format. Typically it is better to use the .then().catch() format because if an error occurs either before the .then() (in the query for example), or in the .then() function the .catch() can handle it. When you use .then(onSuccess, onError) the problem is that if onSuccess generates an error, onError won't catch it. onError will only handle errors from the promise call before the .then(). Using .catch() can handle errors in all places. This is why it is the generally preferred pattern.

.CATCH() IS JUST A .THEN()

Behind the scenes, `.catch()` is just a wrapper around `.then()` which only handles error functions. It is identical to using `.then(null, function)`. You can consider the following code identical:

```
myPromise
  .then(onSuccess)
  .then(null, onError)
```

```
myPromise
  .then(onSuccess)
  .catch(onError)
```

In both cases an error from the initial `myPromise` or an error from `onSuccess` will call the error handling function `onError`

Quick Note: Multiple Connections

Using Mongoose, it is also possible to have multiple connections configured for your application, ie:

"`mongodb://user:pass@ds000001.mlab.com:54321/db1`" and "`mongodb://user:pass@ds000002.mlab.com:65432/db2`". If the case, we just have to make a few small changes on how we **connect** to each DB, and how we define our models:

```
// ...

let db1 = mongoose.createConnection("mongodb://user:pass@ds000001.mlab.com:54321/db1");

// verify the db1 connection

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

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

// ...

let db2 = mongoose.createConnection("mongodb://user:pass@ds000002.mlab.com:65432/db2");

// ...

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

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

```
// ...
```

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

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.

FIND()

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

INSERTING A NEW DOCUMENT

First we 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 leftorium = new Company({ ... });

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

UPDATE()

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 find 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.update(  
  { ... query ... },  
  { $set { ... fields to set ... } },  
  { multi: true|false })  
  .exec();
```

Example:

```
Company.update({ companyName: "The Leftorium"},  
  { $set: { employeeCount: 4 } },  
  { multi: false })  
  .exec();
```

REMOVE()

```
Company.remove({ ... query ... })  
  .exec()  
  .then();
```

Example:

```
Company.remove({ companyName: "The Leftorium" })  
  .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 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 index would be on the `_id` field of all documents that MongoDB adds automatically. As mentioned above every document gets `_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 additional unique index constraints to other fields as needed. Mongoose will add the indexes, if they don't exist, to the collection 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 to the schema declaration from before.

```
// define the company schema
```

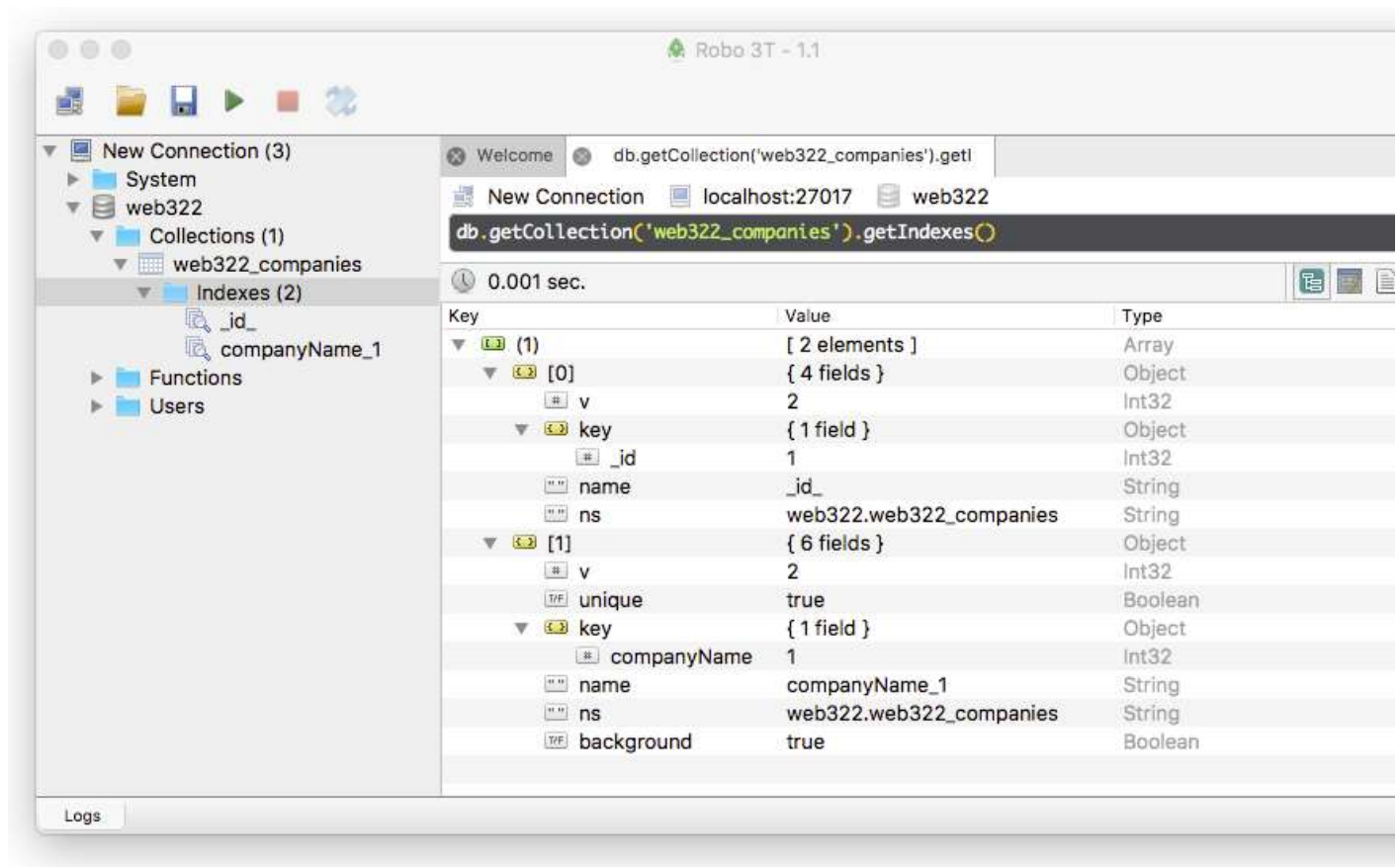


```

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. You can view your indexes that exist w 3T.



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

```

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

```

```
// connect to the localhost mongo running on default port 27017
mongoose.connect("mongodb://localhost/web322");

// 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}');
    });
});
```

```
});
```

Running it a second time:

```
$ 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","_id":"5864148f8e6a8028
":3,"__v":0}}})
[ { _id: 586412c78b50ee22cc9691d6,
  companyName: 'The Kwik-E-Mart',
  address: 'Springfield',
  phone: '212-842-4923',
  country: 'U.S.A',
  __v: 0,
  employeeCount: 3 } ]
```

As you can see MongoDB threw back an error (E11000 duplicate key error). This is the most common form of error you'll encounter 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 a user 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, inspect the data with Robo 3T, and then try creating a Mlab account and connecting your Mlab db to your example running on Heroku.