LAB 20d

NODE AND MONGODB

What You Will Learn

- How to setup MongoDB
- How to import JSON data into Mongo
- How to query data in Mongo
- How to read Mongo data within Node using Mongoose

Approximate Time

The exercises in this lab should take approximately 60 minutes to complete.

Fundamentals of Web Development, 2nd Ed

Randy Connolly and Ricardo Hoar

Textbook by Pearson
http://www.funwebdev.com

Date Last Revised: Feb 23, 2018

Working with MongoDB

In this lab, you will be focusing on the server-side development environment Node.js (or Node for short). Like with PHP, you can work with it locally on your development machine or remotely on a server.

Preparing Directories

1 This lab has additional content that you will need to copy/upload this folder into your eventual working folder/workspace.

2 The data file you will be using has been provided as a zip file. You will need to unzip this. If using a Linux-based environment such as that provided on Cloud9, you will need to use the following command:

```
unzip data.zip
```

Exercise 20d.1 — Setting up MongoDB

1 The mechanisms for installing MongoDB vary based on the operating system.

If you wish to run MongoDB locally on a Windows-based development machine, you will need to download and run the Windows installer from the MongoDB website.

If you want to run MongoDB locally on a Mac, then you will have to download and run the install package.

If you want to run MongoDB on a Linux-based environment, you will likely have to run sudo commands to do so. The Node website provides instructions for most Linux environments.

If you are using a cloud-based development environment (for instance Cloud9), MongoDB might already installed in your workspace. Check by entering the following command in the terminal:

```
mongod
```

If it doesn't work, try the following commands (just for Cloud9):

```
sudo apt-get install -y mongodb-org
mkdir data
echo 'mongod --bind_ip=$IP --dbpath=data --nojournal --rest "$@"' > mongod
chmod a+x mongod
mongod
```

The MongoDB daemon process should now be running. Just like working with MySQL, when working with MongoDB, there is a daemon process that keeps running and which responds to requests. There is also a separate mongo shell that allows you to interactively input commands.

2 You have been provided with a two JSON data files. The first is quite small and is ideal for learning the basics of Mongo syntax. To import this data into MongoDB, you need to use the `mongoimport` program via the following:

```
mongoimport -db funwebdev --collection books --file books.json –jsonArray
```

This instructs the program to create a database named funwebdev, and a collection named books and populate that collection from the file books.json.

3 Examine the file books.json to see the structure of the data you just imported.

4 Import the other file, which is very large, using the following command:

```
mongoimport -db funwebdev --collection stocks --file stocks.json –jsonArray
```

Exercise 20d.2 — Using the Mongo Shell

1 In a separate terminal window, run the Mongo shell via the following command:

```
mongo
```

2 Enter the following command into the Mongo shell:

```
db
```

This will display the current database, which should return test, the default database.

3 Enter the following commands into the Mongo shell:

```
show dbs
use funwebdev
```

The first command displays a list of available databases, while the second makes funwebdev the current database.

4 Enter the following command:

```
show collections
```

A database in MongoDB is composed of collections, which are somewhat analogous to tables in a relational database.

5 Enter the following command:

```
db.books.find()
```

This displays all the data in the books collection. MongoDB uses JavaScript as its query language, not SQL.

6 Enter the following:

```
db.books.find({id: 587})
```

This displays a single record.

7 Enter the following:

```
db.books.find({id: 587}).pretty()
```

This displays a single record formatted more nicely. The equivalent of a SQL WHERE clause is specified via a JavaScript object literal representing the search string.

Notice that unlike a table record in a relational database, a Mongo document is like a JavaScript literal in that it can be hierarchical.

8 Enter the following:

```
db.books.find({"category.secondary" : "Calculus"}).pretty()
```

This finds all books whose secondary category field is equal to Calculus. Notice that you reference objects-within-objects via dot notation and that such a compound name needs to be enclosed in quotes.

9 Enter the following:

```
db.books.find({ "production.pages": {$gte: 950}})
```

This uses a MongoDB comparison operator to retrieve all books whose page count is greater or equal to 950.

10 Enter the following:

```
db.books.find({ "production.pages": {$gte: 950}}).count()
```

This puts the result through the count() function (and should return the number 6).

11 Enter the following:

```
db.books.find({ "production.pages": {$gte: 950}}).sort({title: 1})
```

This sorts the results of the find on the title field.

12 Enter the following:

```
db.books.find({ "production.pages": {$gte: 950},
                "category.main": "Mathematics" })
```

This adds another criteria (equivalent to an SQL AND).

13 Enter the following:

```
db.books.find({title: /Basic/})
```

This uses a regular expression to do a query equivalent to SQL LIKE.

14 Enter the following:

```
db.books.find({title: /Basic/}, { title:1, isbn10:1})
```

Here you have added a second parameter to find(), in which you specify the projection (i.e., the fields you wish to return/display).

15 Enter the following:

```
exit
```

Exercise 20d.3 — Adding Data

1 If not already running, run the Mongo shell via the following commands:

```
mongo
```

2 Ensure you are using the correct database via:

```
use funwebdev
```

3 Enter the following:

```
db.sales.insert({ "id": "1",  "amount": 20.00 })
db.sales.insert({ "id": "2",  "amount": 30.00 })
db.sales.insert({ "id": "3",  "amount": 40.00 })
```

Notice that you can create a new collection simply by adding new objects.

4 Enter the following:

```
db.sales.find()
```

5 Enter the following:

```
db.sales.update( {"id": "3"}, { "id": "3", "amount": 50.00 })
```

6 Enter the following:

```
db.sales.find()
```

The above exercises illustrated just the simplest queries in MongoDB. More complicated queries using aggregation functions add a significant layer of complexity. The next exercise illustrates a few examples, but a more complete exploration is beyond the scope of this lab.

Exercise 20d.4 — Using Aggregate Functions

1 If not already running, run the Mongo shell via the following commands:

```
mongo
```

2 Ensure you are using the correct database via:

```
use funwebdev
```

3 Enter the following:

```
db.sales.aggregate([ { $group: { _id: null, total: {$sum: "$amount"}} } ])
```

The aggregate function allows you to perform aggregations similar to GROUP BY in SQL. It is typically provided with a $group specifier. The `_id` is used to indicate which fields to group the aggregator on. In this case, we are not grouping but summing a value across the entire collection so it is set to null. The `$sum` indicates we want a sum on the field named `amount`.

3 Enter the following:

```
db.books.find({title: /Algebra/}).pretty()
```

Returns all books with Algebra in the title.

4 Enter the following:

```
db.books.aggregate([ { $match: {title: {$regex: /Algebra/}} } ])
```

This does the same thing.

3 Enter the following:

```
db.books.aggregate( [

    { $match: { title: /Algebra/ } },
    { $group: { _id: "$category.secondary", count : { $sum : 1 } } }

])
```

Here the results of the $match are "piped into" the $group specifier. In this case, we are separating the books with the word Algebra in the title by their secondary subcategory and then counting how many there are in each subcategory.

4 Enter the following:

```
db.books.aggregate( [

    { $match: { title: /Algebra/ } },
    { $group: { _id: "$category.secondary",
      amount : { $sum : "$production.pages" } } }

])
```

Now we are displaying the sum of the pages.

5 Change the function from $sum to $avg and test.

6 Enter the following:

```
db.books.aggregate( [
    { $group: { _id: "$category.main",
      avgPages : { $avg : "$production.pages" } } },
    { $sort: { "category.main" : 1 } }

])
```

Now the results from the $group are piped into the $sort specifier.

7 Enter the following:

```
db.books.aggregate( [
    { $group: { _id: "$category.main",
      avgPages : { $avg : "$production.pages" } } },
    { $sort: { "avgPages" : 1 } }

])
```

Notice that you can reference the field created by the $group in the $sort.

USING MONGODB IN NODE

There are several API libraries for accessing MongoDB data in Node. In this lab, you will be making use of the Mongoose package, which uses a lightweight ORM (Object-Relational Mapping) approach. Since Mongoose is an ORM, you will interact with the database by defining a schema rather than run MongoDB queries.

Exercise 20d.5 — TESTING MONGOOSE

1 In the terminal, type the following command:

```
npm init
```

It doesn't really matter how you answer these questions.

2 Enter the following commands:

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

3 Now create a new file with the following content and save it as book-server.js. You may need to modify the connection string based on your MongoDB installation.

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

mongoose.connect('mongodb://localhost:27017/funwebdev');
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function callback () {
   console.log("connected to mongo");
});
```

4 Test this (by running `node book-server` in the terminal) to make sure you can connect to the database. Remember than Mongo server has to be running in its own terminal window.

If it works, you should see the "connect to mongo" message.

Exercise 20d.6 — Implementing Mongoose-Based API

1 Remind yourself of the structure of each book item by examining single-book.json.

2 Add the following require statements to the top of book-server.js.

```
var express = require('express');
var parser = require('body-parser');
```

3 Add the following at the end of book-server.js.

```
// define a schema that maps to the structure of the data in MongoDB
var bookSchema = new mongoose.Schema({
  id: Number,
  isbn10: String,
  isbn13: String,
  title: String,
  year: Number,
  publisher: String,
  production: {
      status: String,
      binding: String,
      size: String,
      pages: Number,
      instock: Date
  },
  category: {
    main: String,
    secondary: String
  }
});
```

4 Now add the following:

```
// “compile” the schema into a model
var Book = mongoose.model('Book',bookSchema);
```

5 Add on the necessary “wiring” for express as follows:

```
// create an express app
var app = express();
// tell node to use json and HTTP header features in body-parser
app.use(parser.json());
app.use(parser.urlencoded({extended: true}));
```

6 Add in the following route:

```
//  handle GET requests for [domain]/api/books  e.g. return all books
app.route('/api/books')
    .get(function (req,resp) {
        // use mongoose to retrieve all books from Mongo
        Book.find({}, function(err, data) {
            if (err) {
                resp.json({ message: 'Unable to connect to books' });
             } else {
                 // return JSON retrieved by Mongo as response
                 resp.json(data);
            }
        });
    });
```

7 Add the final wiring code as follows then save.

```
// Use express to listen to port
let port = 8080;
app.listen(port, function () {
    console.log("Server running at port= " + port);
});
```

8 Execute `node book-server` and then in your browser, test by making the following request (note: may or may not be https):

https://*your-domain-here*/api/books

This should display all the books in JSON format.

9 Add the following route and test.

```
// handle requests for specific book: e.g., [domain]/api/books/0321886518
app.route('/api/books/:isbn')
    .get(function (req,resp) {
        Book.find({isbn10: req.params.isbn}, function(err, data) {
            if (err) {
                resp.json({ message: 'Book not found' });
            } else {
                resp.json(data);
            }
        });
    });
```

10 Execute `node book-server` and then in your browser, test by making the following request (note: may or may not be https):

https://*your-domain-here*/api/books/0321886518

This should display all the data for just a single book.

11 Add the following route and test.

```
// handle requests for books with specific page ranges:
// e.g., [domain]/api/books/pages/500/600
app.route('/api/books/pages/:min/:max')
    .get(function (req,resp) {
        Book.find().where('production.pages')
                   .gt(req.params.min)
                   .lt(req.params.max)
                   .sort({ title: 1})
                   .select('title isbn10')
                   .exec( function(err, data) {
                             if (err) {
                                resp.json({ message: 'Books not found' });
                             } else {
                                resp.json(data);
                             }
                          });
    });
```

This shows how Mongoose can be used to construct a more complex MongoDB query.

12 Execute `node book-server` and then in your browser, test by making the following request (note: may or may not be https):

https://*your-domain-here*/api/books/pages/500/600

This should display all the books whose page count is between 500 and 600 pages.

Exercise 20d.7 — Your Turn

1 Create a new Node web service called stock-service.js.

2 Examine the file single-stock.json and use it to define the Mongoose schema. NOTE: your schema definition doesn't have to define/match every field in the MongoDB data. For instance, there is over 70 fields in the financials object: you only need to define a few of them.

3 Define the following web services:

https://*your-domain-here*/api/stocks/AMZN

Returns the complete stock information for the specified symbol (AMZN in this example)

https://*your-domain-here*/api/stocks/sector/Industrials

Returns just the Symbol and Company_Name fields for each stock with a matching Sector

https://*your-domain-here*/api/stocks/prices/2015-11-16/AMZN

Returns just the Price objects for specified symbol for the specified date. Because the price data is an array element, it is a bit tricky to return just a single sub-element from a sub-array. You will need to construct your query using the Mongoose find() method similar to the following:

```
.find( { 'Prices.Name': req.params.symb,
         'Prices.Date': new RegExp(req.params.date)},
       {'Prices.$': 1 },
       function(err, data) { ... });
```

The emphasized text tells MongoDB to return just the specified sub-element.