Lab 4: Getting Started with MongoDB

Please note: No demos or hand-ins are required for this lab.

Learning Objectives:

- Introducing MongoDB
- Simple Mongoose Schemas with in a MEAN Stack
- Writing a REST API for MongoDB
- MongoDB for UnitTesting
- The Official MongoDB Rust Driver

Introducing MongoDB:

- MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.
- In MongoDB each entry in a database is called a *document*.
- In MongoDB a collection of documents is called a *collection*.
- In Mongoose the definition of a document is called a *schema*.
- Each individual data entity defined in a schema is called a *path*.
- MongoDB stores documents as BSON, which is binary JSON (JavaScript Serialized Object Notation).

Relational Vs. Document Databases

- In a relational database, a column defines the name and data type and each row would be a different entry as:

firstName	middleName	lastName	maidenName	nickname
Simon	David	Holmes		Si
Sally	June	Panayiotou		
Rebecca		Norman	Holmes	Bec

- In MongoDB the concept of rows still exists but columns are removed from the picture. Rather than a column defining what should be in the row, each row is a document, and this document both defines and holds the data itself. For example:

```
firstName:          middleName:         lastName:           nickname:
"Simon"             "David"             "Holmes"            "Si"

lastName:           middleName:         firstName:
"Panayiotou"        "June"              "Sally"

maidenName:         firstName:          lastName:           nickname:
"Holmes"            "Rebecca"           "Norman"            "Bec"
```

- MongoDB can be better than RDBMS as:

- MongoDB is a document database in which one collection holds different documents. Number of fields, content and size of the document can differ from one document to another.
- Structure of a single object is clear.
- No complex joins.
- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
- Tuning.
- Ease of scale-out − MongoDB is easy to scale.
- Conversion/mapping of application objects to database objects not needed.
- Uses internal memory for storing the (windowed) working set, enabling faster access of data.

Installing MongoDB:

- MongoDB is also available for Windows, Mac OS X, and Linux. Detailed instructions about all of the following options are available in the MongoDB online documentation.
  - Install MongoDB on Windows
  - Install MongoDB on Linux
  - Install MongoDB on macOS

What is Mongoose and how does it work?

- Mongoose was built specifically as a MongoDB Object-Document Modeler (ODM) for Node applications.
- The key principles is that you can manage your data model from within your application.
- No need for any direct interactions with databases or external frameworks or relational mappers;
- A *model* is the compiled version of a schema. All data interactions using Mongoose go through the model.

Simple Mongoose Schemas in a MEAN Stack:

- The schema bears a very strong resemblance to the data itself. The schema defines the name for each data path, and the data type it will contain. For example:

```
{
  "firstname" : "Simon",
  "surname" : "Holmes",
  _id : ObjectId("52279effc62ca8b0c1000007")
}
```

Example MongoDB document

```
{
  firstname : String,
  surname : String
}
```

Corresponding Mongoose schema

- In a MEAN stack, a MongoDB schema is defined schema in the model folder alongside db.js
- Assuming the Mongoose Schema will hold information on the "locations" of some objects, inside the models folder within the app_server add the following line in the locations.js file:

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

- Next, this file is brought into the application by requiring it in db.js as:

```
require('./locations');
```

- Mongoose gives you a constructor function for defining new schemas, which you typically assign to a variable so that you can access it later.

```
var locationSchema = new mongoose.Schema({ });
```

- For example: for the location data as follows:

```
locations: [{
name: 'Starcups',
address: '125 High Street, Reading, RG6 1PS',
rating: 3,
facilities: ['Hot drinks', 'Food', 'Premium wifi'],
distance: '100m'
}]
```

- A MongoDB schema can be defined as:

```
var locationSchema = new mongoose.Schema({
name: String,
address: String,
rating: Number,
facilities: [String]
});
```

- We can add some basic validation in the code as:

```
var locationSchema = new mongoose.Schema({
name: {type: String, required: true},
address: String,
rating: {type: Number, "default": 0, min: 0, max: 5},
facilities: [String]
});
```

Creating more complex schemas with subdocuments

- MongoDB can handle fairly complex schemas with nested documents. An example of such a schema is as follows:

```
location: {
  name: 'Starcups',
  address: '125 High Street, Reading, RG6 1PS',
  rating: 3,
```

```
  facilities: ['Hot drinks', 'Food', 'Premium wifi'],
  coords: {lat: 51.455041, lng: -0.9690884},
  openingTimes: [{
    days: 'Monday - Friday',
    opening: '7:00am',
    closing: '7:00pm',
    closed: false
  },{
    days: 'Saturday',
    opening: '8:00am',
    closing: '5:00pm',
    closed: false
  },{
    days: 'Sunday',
    closed: true
  }],
  reviews: [{
    author: 'Simon Holmes',
    rating: 5,
    timestamp: '16 July 2013',
    reviewText: 'What a great place. I can\'t say enough good things about
it.'
  },{
    author: 'Charlie Chaplin',
    rating: 3,
    timestamp: '16 June 2013',
    reviewText: 'It was okay. Coffee wasn\'t great, but the wifi was fast.'
  }]
}
```

- In a relational database, you'd create these as separate tables, and join them together in a query when you need the information.
- In a document database, anything that belongs specifically to a parent document should be contained within that document.
- MongoDB offers the concept of subdocuments to store repeating, nested data. Subdocuments are very much like documents in that they have their own schema and each is given a unique _id by MongoDB when created. But subdocuments are nested inside a document and they can only be accessed as a path of that parent document.
- **Subdocuments** are defined in Mongoose by using nested schemas. So that's one schema is nested inside another. An example can be:

```
var openingTimeSchema = new mongoose.Schema({
days: {type: String, required: true},
opening: String,
closing: String,
closed: {type: Boolean, required: true}
});

var locationSchema = new mongoose.Schema({
```

```
name: {type: String, required: true},
address: String,
rating: {type: Number, "default": 0, min: 0, max: 5},
facilities: [String],
coords: {type: [Number], index: '2dsphere'},
openingTimes: [openingTimeSchema]
});
```

- With this in place, we could now add multiple opening time subdocuments to a given location, and they would be stored within that location document

Compiling Mongoose schemas into models

- In Mongoose, a model is a compiled version of the schema. Once compiled, a single instance of the model maps directly to a single document in your database.
- The following command would compile the locationSchema that we have created above:

```
mongoose.model('Location', locationSchema, 'Locations');
```

Using the MongoDB shell to create a MongoDB database and add data

- The MongoDB shell is a command-line utility that gets installed with MongoDB, and allows you to interact with any MongoDB databases on your system.
- To start the Mongo shell, open terminal or command prompt and type:

```
$ mongo
```

- This should respond in the terminal with a couple of lines (like these two), confirming the shell version and that it's connecting to a test database, for example:

```
MongoDB shell version: 2.4.6
connecting to: test
```

- Next enter the command that will show you a list of all of the local MongoDB databases:

```
> show dbs
```

- This will return a list of the local MongoDB database names and their sizes, for example:

```
local 0.078125GB
test (empty)
```

- Then, to use a specific dataset type:

```
> use local
```

- Once you're using a particular database, it's really easy to output a list of the collections using the following command:

```
> show collections
```

- To see the contents of a collection, type:

```
db.collectionName.find(queryObject)
```

- To save new content into a collection, type:

```
db.collectionName.save(queryObject)
```

- For updating an old document or adding a sub-document, MongoDB has an update command that accepts two arguments, the first being a query so that it knows which document to update, and the second contains instructions on what to do when it has found the document:
- Here is an example of the update command on the location schema:

```
> db.locations.update({
  name: 'Starcups'
}, {
  $push: {
    reviews: {
      author: 'Simon Holmes',
      id: ObjectId(),
      rating: 5,
      timestamp: new Date("Jul 16, 2013"),
      reviewText: "What a great place."
    }
  }
})
>
```

Writing a REST API for MongoDB:

- REST stands for REpresentational State Transfer, which is an architectural style rather than a strict protocol. REST is stateless—it has no idea of any current user state or history.
- In basic terms, a REST API takes an incoming HTTP request, does some processing, and always sends back an HTTP response.
- In a MEAN stack, there is normally a set of API URLs for each collection in MongoDB.
- Four request methods used in REST API as:

Request method	Use	Response
POST	Create new data in the database	New data object as seen in the database
GET	Read data from the database	Data object answering the request
PUT	Update a document in the database	Updated data object as seen in the database
DELETE	Delete an object from the database	Null

- Our API will return one of three things for each request:
  - A JSON object containing data answering the request query
  - A JSON object containing error data
  - A null response

Creating the routes and starting the application

- we'll have an index.js file in the app_api/routes folder that will hold all of the routes we'll use in the API.
- In a MEAN stack, we can reference this file in the main application file app.js as:

```
var routes = require('./app_server/routes/index');
var routesApi = require('./app_api/routes/index');
```

- To tell the application when to use the routes, type:

```
app.use('/', routes);
app.use('/api', routesApi);
```

- To specify the request methods in the routes, simply type:

```
router.get('/location', ctrlLocations.locationInfo);
router.post('/locations', ctrlLocations.locationsCreate);
```

- Using the other methods of PUT, and DELETE is as simple as switching out the get / post with the respective keywords of post, put, and delete.
- To enable the application to start we can create placeholder functions for the controllers. We need to declare them inside the app_api/routes folder.
- Controller placeholders can be created with JSON return objects as:

```
module.exports.locationsCreate = function (req, res) {
  res.status(200);
  res.json({"status" : "success"});
};
```

- To make the API talk to the database using Mongoose, we first need to require Mongoose into the controller files, and then bring in the model as:

```
var mongoose = require('mongoose');
var Loc = mongoose.model('Location');
```

- The first line gives the controllers access to the database connection, and the second brings in the Location model so that we can interact with the Locations collection.
- We can find a single document in MongoDB using Mongoose using the findById method and run any query on the identified document using the exec method as:

```
Loc
  .findById(locationid)
  .exec(function(err, location) {
    console.log("findById complete");
});
```

- However, to get this working in the context of the controller we need to do two things:
    - Get the location id parameter from the URL and pass it to the findById method.
    - Provide an output function to the exec method.

Create, Update or Delete a new document in MongoDB

- The construct to create a new document in MongoDB can be written as follows:

```
Loc.create(dataToSave, callback);
```

- Constraints and default values can be added to MongoDB schemas. An example of a set of constraints for the location schema can be as follows:

```
var locationSchema = new mongoose.Schema({
  name: {type: String, required: true},
  address: String,
  rating: {type: Number, "default": 0, min: 0, max: 5},
  coords: {type: [Number], index: '2dsphere', required: true},
  reviews: [reviewSchema]
});
```

- If either of these fields is missing, the create method will raise an error and not attempt to save the document to the database.

Creating new sub-documents in MongoDB

- In MongoDB, creating and saving a new subdocument has three main steps:
    1. Find the correct parent document.
    2. Add a new subdocument.
    3. Save the parent document.
- The following listing shows an example for adding and saving a sub-document in MongoDB:

```
var doAddReview = function(req, res, location) {
  if (!location) {
    sendJsonResponse(res, 404, {
      "message": "locationid not found"
  });
} else {
  location.reviews.push({
    author: req.body.author,
    rating: req.body.rating,
    reviewText: req.body.reviewText
  });
  location.save(function(err, location) {
    var thisReview;
    if (err) {
      sendJsonResponse(res, 400, err);
    } else {
        updateAverageRating(location._id);
        thisReview = location.reviews[location.reviews.length - 1];
        sendJsonResponse(res, 201, thisReview);
        }
    });
  }
};
```

- To update a document in MongoDB has four primary steps:

1. Find the relevant document.
2. Make some changes to the instance.
3. Save the document.
4. Send a JSON response.

- An example for updating a document in MongoDB can be as follows:

```
Loc
  .findById(req.params.locationid)
  .exec(
    function(err, location) {
      location.name = req.body.name;
      location.save(function(err, location) {
        if (err) {
          sendJsonResponse(res, 404, err);
        } else {
          sendJsonResponse(res, 200, location);
        }
      });
    }
  );
};
```

- Similarly, deleting a document in MongDB, provided that the document ID is available can be performed as follows:

```
module.exports.locationsDeleteOne = function(req, res) {
  var locationid = req.params.locationid;
  if (locationid) {
    Loc
      .findByIdAndRemove(locationid)
      .exec(
        function(err, location) {
          if (err) {
            sendJsonResponse(res, 404, err);
            return;
          }
          sendJsonResponse(res, 204, null);
        }
    );
  } else {
      sendJsonResponse(res, 404, {
        "message": "No locationid"
      });
   }
};
```

MongoDB for Unit Testing

- Writing Unit tests can be a tedious task. That means even before we start writing, much work needs to be done such as: Picking a library or framework, reading the documentation on how to do the assertions and more…
- For unit testing, we normally need to manually define the data we put into the function and what we expect from the function.
- Most importantly, we need to maintain twice as much code.
- With MongodB we have a solution to this problem!
    - *Instead of coding the unit tests, we record the input and output data of our functions automatically and save them into a database.*
    - *And… after a change to a function, we use recorded data to test if the same recorded output is produced as before the change*
- Let's discuss the idea with an example:
- Here is an example of a simple sum function in JavaScript:

```
'use strict';

module.exports = {
  sum: function(a, b){
    return a + b;
  }
};
```

- In order to save our recorded inputs and outputs, we need to spin up a MongoDB instance first
- Next we define a general function wrapper that will execute our coded functions and save the inputs and outputs to the database. The function can be written as follows:

```
var MongoClient = require('mongodb').MongoClient;
var url = 'mongodb://localhost:27017';

module.exports = {
  record: function(handle, ...args){
    let result = handle(...args);
    MongoClient.connect(url, { useUnifiedTopology: true }, function(err, db){
      if (err) throw err;
      var dbo = db.db('test');
      var newvalues = {
        $addToSet:
        {
          data: { 'args': args, 'return': result }
        }
      };

      dbo.collection('data').updateOne({ name: handle.name }, newvalues,
```

```
        { upsert: true, useUnifiedTopology: true }, function(err, res){
        if (err) throw err;
        db.close();
      });
    });

    return result;

  },
};
```

- Next, instead of calling the above-created sum function directly, we call the recorder and pass the function as a parameter to it:

```
var myModule = require('./sum.js');
var record = require('./record.js');

var result = record.record(myModule.sum, 1, 2);
console.log(result);

result = record.record(myModule.sum, 3, 2);
console.log(result);

result = record.record(myModule.sum, 3, 5);
console.log(result);
```

- Once this code executes, the values will be stored in MngoDB as documents, whose structure would look like the following:

```
_id:ObjectID("<….document Id….>")
name: "sum"
data: Array
  0: Object
    args: Array
      0: 1
      1: 2
    return: 3
  1: Object
    args: Array
      0: 3
      1: 5
    return: 8
  2: Object
    args: Array
      0: 3
      1: 2
    return: 5
```

- Now, testing becomes really easy. As we can include a general unit test, which reads the database and executes a unit test for every recorded entry. An example can be as follows:

```
"use strict";

var assert = require('assert');

var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017";

var functions = require('../sum.js');

describe('Data', function () {

  var tests

  before(async () => {

    let client = await MongoClient.connect(url,
      { useUnifiedTopology: true });

    let db = client.db('test');
    try {
      tests = await db.collection('data').find({}).toArray();
    }
    finally {
      client.close();
    }
  })

  it('is stil consistent', () => {
    for (let test of tests) {
      test.data.forEach(data => {
        assert.equal(functions[test.name](...data.args), data.return);
      })
    }
  });
});
```

- Now we can simply run:

```
$ npm run test> test
@1.0.0 test /…
> mocha

Data
 ✓ is stil consistent
1 passing (28ms)
```

The Official MongoDB Rust Driver

- The Rust driver can be imported to your client by adding the mongodb crate to the project's Cargo.toml. The bson library is also needed for most usage of the driver.
- To add the driver to your project, add the following lines under the dependencies of your Cargo.toml file:

```
mongodb = "0.9.0"
bson = "0.14.0"
```

- To connect to the MongoDB deployment, create a Client using either a MongoDB connection string or through manually constructing a ClientOptions struct as follows:

```
use mongodb::{Client, options::ClientOptions};

// Parse a connection string into an options struct.
let mut client_options =
    ClientOptions::parse("mongodb://localhost:27017")?;

// Manually set an option.
client_options.app_name = Some("My App".to_string());

// Get a handle to the deployment.
let client = Client::with_options(client_options)?;

// List the names of the databases in that deployment.
for db_name in client.list_database_names(None)? {
    println!("{}", db_name);
}
```

- Now several database specific operations can be performed using MongoDB as follows:

```
// Get a handle to a database.
let db = client.database("mydb");

// List the names of the collections in that database.
for collection_name in db.list_collection_names(None)? {
    println!("{}", collection_name);
}
```

- A Collection can be obtained from a Database. All the MongoDB CRUD operations you might be familiar with are defined on Collection.

```
use bson::{doc, bson};

// Get a handle to a collection in the database.
let collection = db.collection("books");

let docs = vec![
```

```
    doc! { "title": "1984", "author": "George Orwell" },
    doc! { "title": "Animal Farm", "author": "George Orwell" },
    doc! { "title": "The Great Gatsby", "author": "F. Scott Fitzgerald" },
];

// Insert some documents into the "mydb.books" collection.
collection.insert_many(docs, None)?;
```

- The Collection::find returns a Cursor, which implements Rust's Iterator trait as follows:

```
use bson::{doc, bson};
use mongodb::options::FindOptions;

// Query the documents in the collection with a filter and an option.
let filter = doc! { "author": "George Orwell" };
let find_options = FindOptions::builder()
    .sort(doc! { "title": 1 })
    .build();
let cursor = collection.find(filter, find_options)?;

// Iterate over the results of the cursor.
for result in cursor {
    match result {
        Ok(document) => {
            if let Some(title) = document.get("title").and_then(Bson::as_str)
{
                println!("title: {}", title);
            }  else {
                println!("no title found");
            }
        }
        Err(e) => return Err(e.into()),
    }
}
```

Exercise: Unit Testing with MongoDB

- Try out the new MongoDB RUST Driver
- Create a new project for Unit Testing with MongoDB
- Rewrite the code for "MongoDB for Unit Testing" discussed above in RUST.