Extra notes - nodejs-mongo-outline

- Semester: Fall 2016
- Dr Nick Hayward

Node.js and MongoDB

A brief introduction to using MongoDB with Node.js and Mongoose.

Contents

Intro

We've tested Node.js, created a server for hosting our files and routes with ExpressJS, read JSON from the server, and updated our JSON on the server-side.

This works well assuming we don't need to restart, repair, or update our server. If we do, we lose the updates posted to the server and will, instead, return to the default data stored.

To help us solve this obvious issue, we'll need to consider persistent data storage that runs independently from our application. We'll look at MongoDB, and how we can integrate this NoSQL data store option with our Node.js applications.

SQL or NoSQL

When we think of databases, we have often thought solely in terms of SQL, or structured query language. It is a language used to query data in a relational format. Such relational databases, for example MySQL or PostgreSQL, store their data in tables, which then provide a semblance of structure through rows and cells. We can then easily cross-reference, or relate, these table rows with rows in other tables.

So, we might use this relational structure to map authors to books, players to teams, and so on. One of the primary benefits of using a relational database is this inherent ability to store information, thereby dramatically reducing redundancy, and hopefully required storage space as well.

As storage restrictions have continued to ease in recent years, we now see a shift in thinking, and database design in general. We've started to see the introduction of non-relational databases, often referred to simply as **NoSQL**. With this type of database, redundant data may be stored, but such designs often provide increased ease of use for developers. Some of these databases and stores have also been written with specific use-cases in mind, for example providing more efficient reading of data compared to writing. In effect, highly efficient data storage for specialised environments and scenarios.

MongoDB

MongoDB is a NoSQL database that enables us to store our data on disk, but unlike MySQL, for example, it is not in a relational format.

MongoDB is best characterised as a **document-oriented** database, which conceptually may be considered as storing objects in collections. It stores its data using the BSON format, which we'll look at in a moment, but ostensibly for our purpose we can consider this as comparable to JSON. The best part is that we can easily interact with MongoDB using JavaScript.

We'll have a look at some of the basics of MongoDB, then we'll install and set it up, and then work our way through an example with Node.js and Mongoose.

Document oriented

In a traditional SQL database, data is stored in tables and rows. MongoDB, by contrast, uses *collections* and *documents*.

A document may contain a lot more data than a table. For example, in a SQL database we may decide to store user details,
including `user_id`, `email`, `name` and so on, in multiple tables. We then join these tables to create a user record. However, with documents we simply store the data, id, email etc, and this will now be the only item in the database to store this grouping of data.

A noted concern with this document approach is duplication of data for each user. This, however, is one of the trade-offs between NoSQL (MongoDB) and SQL.

In SQL, the goal of data structuring is to normalise as much as possible, thereby avoiding duplicated information. With MongoDB, there is a notably different goal. We are trying to provision a data store which is as easy as possible for the application to use, and by association the developer.

BSON format

BSON is the format used by MongoDB to store its data. It is, effectively, JSON stored as binary with a few notable differences. One of these differences is the `ObjectId` value, which is a data type used in MongoDB to uniquely identify documents. It is created automatically on each document in the database, and can be considered as analogous to a primary key in a SQL database.

The `ObjectId` is a large pseudo-random number. For nearly all practical occurrences, we can assume that this number will be unique. The only situation where we might have a collision or clash in these numbers is if we were generating a large number and the database was unable to keep pace. Therefore, for most use cases, we may assume they're always unique.

The other interesting aspect of `ObjectId` is that they are partially based on a timestamp. This allows us to determine when they were created.

Data hierarchy

In general, MongoDB has three tiers to its hierarchy of data. These include,

- Database - normally one database per app, but it is possible to have multiple per server. It functions the same basic role as a database in SQL.
- Collection - a grouping of similar pieces of data. So, MongoDB stores its documents in collections. Its name is usually a noun, and it resembles in concept a table in SQL. However, collections do not require their documents to share the same schema.
- Document - a single item in the database, for example an individual user record. A document is a data structure that uses field and value pairs, and is similar in nature to JSON objects.

Install and setup

let's now install and setup MongoDB.

- install on Linux
- install on Mac OS X
  - we can use **homebrew** to install Mong

```
// update brew packages
brew update
// install MongoDB
brew install mongodb
```

- then follow the install instructions on the above page to set paths...
- install on Windows

Example shell commands

Then test MongoDB from the command line.

```
// first start MongoDB server - terminal window 1
mongod
// connect to MongoDB - terminal window 2
mongo
```

- switch to or create a new DB, if not available, as follows

```
// list available dbs
show dbs
// switch to specified db
use testdb1
// show current db
db
// drop current db
db.dropDatabase();
```

However, whilst you'll be switched to this new DB, it will not be created permanently until you create and insert a record in that database. In effect, we don't just create empty DBs, save and then populate later. We create the DB, populate, and then MongoDB will save it because it now has something to save. The only permanent DB is the default `test` DB.

Let's now add an initial record to a new `testdb1` database.

```
// select/create db
use testdb1
// insert data to collection in current db
db.notes.insert({
...   "travelNotes": [{
...   "created": "2015-10-12T00:00:00Z",
...   "note": "Curral das Freiras..."
...   }]
... })
```

- our new DB `testdb1` will now be saved in Mongo
- we've created a new collection, `notes`

```
// show databases
show dbs
// show collections
show collections
```

Test app - working with MongoDB

We can now create a new test app for use with MongoDB. (see notes - nodejs-outline & nodejs-express-outline)

To connect to MongoDB, and create a schema for working with our basic DB, we'll add Mongoose to the application.

So, we'll update our `package.json` for the app, and install Mongoose using npm.

```
// add mongoose to app and save dependency to package.json
npm install mongoose --save
```

Then we can quickly test our app and server with the usual startup command in the app's working directory,

```
node server.js
```

Mongoose schema

To help us work with Node.js and MongoDB, we're going to use **Mongoose** as a type of bridge between these two technologies. In effect, this Node.js module serves two useful purposes. In a similar manner to **node-redis**, Mongoose works as a client from our Node.js application to MongoDB. It also serves as a useful data modeling tool, allowing us to represent our documents as objects in the application.

So, for our purposes, what is a data model. In essence, we can simply consider it as an object representation of a document collection within our given data store. It helps us specify required fields for each collection's document.

A data model in Mongoose is a schema, which we use to describe the underlying structure for all objects of a given type. So, for our notes, we can create a data model for a collection of notes. We can start by specifying the schema for a note,

```
var NoteSchema = mongoose.Schema({
  "created": Date,
  "note": String
});
```

After creating our schema, we can programmatically build a model. As a convention, we tend to use an initial uppercase letter for the name of a data model object,

```
var Note = mongoose.model("Note", NoteSchema);
```

With our new model, we can start creating objects of this model type simply by using JavaScript's `new` operator. For example, if we wanted to add a new note,

```
var funchalNote = new Note({
"created": "2015-10-12T00:00:00Z",
"note": "Curral das Freiras..."
});
```

We can then use the Mongoose object to interact with the MongoDB using functions such as `save` and `find`.

Connect to MongoDB

With our new DB setup, our schema created, we can now start to add notes to our database in MongoDB.

In our `server.js` file, we need to connect Mongoose to our DB in MongoDB. Then, we define our schema for our notes. This allows us to then model a note, which we can use to create each note for saving to the database.

```
...
```

```
//connect to testdb1 DB in MongoDB
mongoose.connect('mongodb://localhost/testdb1');
//define Mongoose schema for notes
var NoteSchema = mongoose.Schema({
  "created": Date,
  "note": String
});
//model note
var Note = mongoose.model("Note", NoteSchema);
```

Update `post` route

We can then update our app's `post` route for saving these notes,

```
//json post route - update for MongoDB
jsonApp.post("/notes", function(req, res) {
  var newNote = new Note({
    "created":req.body.created,
    "note":req.body.note
  });
  newNote.save(function (error, result) {
    if (error !== null) {
      console.log(error);
      res.send("error reported");
    } else {
      Note.find({}, function (error, result) {
        res.json(result);
      })
    }
  });
});
```

Update `get` route

Then we need to update our app's `get` route for serving these notes,

```
//json get route - update for mongo
jsonApp.get("/notes.json", function(req, res) {
  Note.find({}, function (error, notes) {
   //add some error checking...
   res.json(notes);
  });
});
```

So, with our test app,

- now able to enter, save, read notes for app
- notes data is stored in the test database in MongoDB
- notes are loaded from DB on page load
- notes are updated from DB for each new note addition
- DEMO - 424-node-mongo1

![Node and MongoDB - 424-node-mongo1](2016/fall/media/images/node-mongo1.png)

References

- MongoDB
  - MongoDB - For Giant Ideas
  - MongoDB - Getting Started (Node.js driver edition)

    - MongoDB - Getting Started (shell edition)
- Mongoose
    - MongooseJS Docs
- Node.js
    - Node.js home
    - Node.js - download
    - ExpressJS
    - ExpressJS body-parser