Node.js & Express.js

Express.js is a web framework based on Node.js platform and is widely used for making simple Web applications.

Express provides a set of tools for easily interacting with HTTP requests and responses, in addition to bringing middleware layers to your api. In addition to the fundamental features provided by Express, the highly popular async utility provides a robust tool for managing asynchronous control flow. Consider this RESTful api endpoint created using Express:

```
app.get('/posts/:id', function (req, res, next) {
  db.find(req.params.id, function (err, posts) {
    if (err) {
      // Pass the error on to be handled by some other middleware
      next(err);
    } else {
      res.send(200, posts);
    }
  });
});
```

Here, syntactic sugar lays out an endpoint that accepts GET requests. Express automatically parses parameters within the request url, and exposes them in the params object of the request.

Notice the line:

```
if (err) {
  next(err);
}
```

This is where the code checks for any errors returned by the database call. Instead of handling the error on the spot with something like:

```
if (err) {
  // Handle the error on the spot
  res.send(500, 'There was an error!');
}
```

I pass the error down the middleware chain to be handled elsewhere. Requests pass through middleware in the order that you tell Express to “use” them (see code below). This is convenient when you want to handle errors from many places with

the same code. In order for this to work, previously in my application I had told Express to add a handler to the middleware chain with something like this:

```
// This tells express to route ALL requests through this middleware
// This middleware ends up being a "catch all" error handler
app.use(function (err, req, res, next) {
  if (err.msg) {
    res.send(500, { error: err.msg });
  } else {
    res.send(500, { error: '500 - Internal Server Error' });
  }
});
```

So when the request (GET “/posts/13”) comes from the client to Express, the request is passed down the middleware chain until it is handled. Most middleware don’t fully handle the request, but augment it. For example, Expresses uses middleware to parse the url parameters and attach the params object to the request object, which is then passed along to be handled by the code written to handle the “/posts/:id” route. If there’s an error, the route can handle the error, or pass it along to be handled by a middleware error handler like the one above.

Middleware is also especially useful for requiring authentication in api endpoints. For example, we will secure the “/posts/:id” route:

```
function restrict(req, res, next) {
  if (req.session.user) {
    // User has active authenticated session, move along
    next();
  } else {
    // Unauthenticated!
    res.send(401);
  }
}
```

And now to tell Express to send the request through the “restrict” middleware before letting the code for the “/posts/:id” route handle the request:

```
app.get('/posts/:id', restrict, function (req, res, next) {
  db.find(req.params.id, function (err, posts) {
    if (err) {
      next(err);
    } else {
      res.send(200, posts);
    }
  });
});
```

Using middleware, it's possible to build a robust and highly functional service api without duplicating code.

When handling a request, and especially when that involves doing file I/O or talking to a database, it often leads what's called "callback hell", or deeply nested callback functions. This makes the code hard to read, as it's hard to follow the flow of the program. Here's an example of trying to create a new user:

```
function create(attrs, cb) {
  db.table('user').filter({email: attrs.email}).count().run(db.conn, function (err, count) {
    if (count > 0) {
      cb('Email already exists!');
    } else {
      db.table('user').filter({username: attrs.username}).count().run(db.conn, function
(err, count) {
        if (count > 0) {
          cb('Username already exists!');
        } else {
          db.table('user').insert(attrs).run(db.conn, function (err, newUser) {
            if (err) {
              cb(err);
            } else {
              cb(null, newUser);
            }
          });
        }
      });
    }
  });
}
```

Imagine how painful it would be to read code nested eight or ten levels deep! Thankfully, async can help us out. Here's the same example using async:

```
function create(attrs, cb) {
  async.waterfall([
    // check for duplicate email
    function (next) {
      db.table('user').filter({email: attrs.email}).count().run(db.conn, next);
    },
    // check for duplicate username
    function (count, next) {
      if (count > 0) {
        cb('Email already exists!');
      } else {
        db.table('user').filter({username: attrs.username}).count().run(db.conn, next);
      }
    },
    // Insert the new user
    function (count, next) {
      if (count > 0) {
        cb('Username already exists!');
      } else {
        db.table('user').insert(attrs, { 'return_vals': true }).run(db.conn, next);
      }
    }
  ]);
}
```

Now each step is listed in its logical order, at the same level of nesting. This particular function of async–waterfall–performs asynchronous operations one after another, each waiting for the previous to complete before executing, and each receiving the return arguments of the preceding function. Async automatically checks for errors and returns them, eliminating the `if (err) cb(err);` boilerplate code.

Reference :-

- http://expressjs.com/en/api.html
- http://www.nikola-breznjak.com/blog/javascript/nodejs/codeschool-express-js-notes/
- https://webapplog.com/express-js-fundamentals/

Passport js

Passport js helps us to write precise and easy authentication strategies. Writing a simple authentication strategy would not be a hasty task, but when multiple authentication strategies are involved it can be a pain to write all authentication strategies for social auths from scratch in Express js. Hence Passport js helps us to write and hook multiple authentications to our server.

```
    passport.serializeUser(function(user, done) {

    done(null, user.id);

  });
```

```
passport.deserializeUser(function(id, done) {

        User.findById(id, function(err, user) {

            done(err, user);

        });

    });
```

`SerializeUser` determines, which data of the user object should be stored in the session. The result of the serializeUser method is attached to the session as `req.session.passport.user = {}`. Here for instance, it would be (as we provide the user id as the key) `req.session.passport.user = {id:'xyz'}`

The first argument of `deserializeUser` corresponds to the key of the user object that was given to the `done` function (see 1.). So your whole object is retrieved with help of that key. That key here is the user id (key can be any key of the user object i.e. name,email etc). In `deserializeUser` that key is matched with the in memory array / database or any data resource.

The fetched object is attached to the request object as `req.user`

Reference :-

- https://scotch.io/tutorials/easy-node-authentication-setup-and-local
- http://mherman.org/blog/2015/01/31/local-authentication-with-passport-and-express-4/#.WQmBFtx97CI
- http://toon.io/understanding-passportjs-authentication-flow/

Mongo DB/Mongoose

Mongo DB is a NoSQL database widely used with Node.js. As NoSQL suggests, Mongo DB rather than saving data in tables, saves data in a JSON like documents. This being one of the most significant feature of mongo, makes mongo comfortable to use with applications involving the javascript stacks. Mongoose is a framework for Mongo, and mongoose API improves the accessibility of mongo for developers when using mongo with a web application.

- MongoDB instances act as high-level container.
- *Collection* is a synonym of table in sql.
- Collections are made by *documents*. (document => row)
- Document is made by *fields*. (field => column)
- Indices are similar to sql databases.
- Cursor can count or skip ahead without actually pulling down data.

> *A collection does not have a schema to follow. Therefore, fields are tracked with each individual document.*
>
> *_id field is automatically generated by MongoDB, and every document must have a unique _id field.*

Adding queries

`db.matrix.insert({name : 'Agent Smith', gender : 'm', age : 44})`

and `db.matrix.findOne()` results in;

```
{
    "_id" : ObjectId("4fcf8b5321f61bf6f63ddd78"),
    "name" : "Agent Smith",
    "gender" : "m",
    "age" : 44
}
```

If we add another person:

```
db.matrix.insert({name : 'Neo', gender : 'm', age : 40})
```

and call `db.matrix.find()` then we get;

```
{ "_id" : ObjectId("4fcf8b5321f61bf6f63ddd78"), "name" : "Agent Smith", "gender" : "m", "age" : 44 }
{ "_id" : ObjectId("4fcf8ce921f61bf6f63ddd79"), "name" : "Neo", "gender" : "m", "age" : 40 }
```

Similarly add two more people:

```
db.matrix.insert({name : 'Trinity', gender : 'f', age : 35})
db.matrix.insert({name : 'Morpheus', gender : 'm', age : 45})
```

And we have total of four people in the database and we could print by using `find()` method as `db.matrix.find()` and return the list of *documents*:

```
{ "_id" : ObjectId("4fcf8b5321f61bf6f63ddd78"), "name" : "Agent Smith", "gender" : "m", "age" : 44 }
{ "_id" : ObjectId("4fcf8ce921f61bf6f63ddd79"), "name" : "Neo", "gender" : "m", "age" : 40 }
{ "_id" : ObjectId("4fcf8df921f61bf6f63ddd7a"), "name" : "Trinity", "gender" : "f", "age" : 35 }
{ "_id" : ObjectId("4fcf8e0821f61bf6f63ddd7b"), "name" : "Morpheus", "gender" : "m", "age" : 45 }
```

Architecture seems to be male, but he is considered to be ageless. Therefore, when we insert the document in database, we *can* simply ignore his age information as such:

```
db.matrix.insert({name : 'Architecture', gender : 'm'})
```

and when we return documents, mongo would not cause any problem at all.

```
{ "_id" : ObjectId("4fcf8b5321f61bf6f63ddd78"), "name" : "Agent Smith", "gender" : "m", "age" : 44 }
{ "_id" : ObjectId("4fcf8ce921f61bf6f63ddd79"), "name" : "Neo", "gender" : "m", "age" : 40 }
{ "_id" : ObjectId("4fcf8df921f61bf6f63ddd7a"), "name" : "Trinity", "gender" : "f", "age" : 35 }
{ "_id" : ObjectId("4fcf8e0821f61bf6f63ddd7b"), "name" : "Morpheus", "gender" : "m", "age" : 45 }
```

{ "_id" : ObjectId("4fcfa18821f61bf6f63ddd7c"), "name" : "Architecture", "gender" : m" }

After creating the database, we need to be able to select some queries based on their fields. In order to do it, we need to use *selectors* which are very similart to **where** clause of Sql statement. simplest one is {} which returns all documents in the collection. null also does the same thing as such:

```
db.matrix.find({})
db.matrix.find(null)
```

returns all the documents in the collection. and statement is accomplished in the form of:

```
{field1 : value1, field2 : value2}
```

which is very intuitive. In this example, if we want to return males whose ages are more or equal to 44, then we need to write a selector as such:

```
db.matrix.find({gender: 'm', age: {$gte: 44}})
```

Some of common operations are: \(lt => less than \(lte=> less or equal \(gt=> greater than \(gte=> greater or equal \(ne => not equal

If we take harder example like a field defines an array, in a particular example, let it be sports which students like, create the database as such:

```
{ "_id" : ObjectId("4fcfce79fd6230c28d817740"), "name" : "john", "likes" : [ "basketball", "football", "baseball", "swimming" ] }
{ "_id" : ObjectId("4fcfcf02fd6230c28d817742"), "name" : "mike", "likes" : [ "basketball", "football", "tennis", "rugby" ] }
{ "_id" : ObjectId("4fcfcf08fd6230c28d817743"), "name" : "cassandra", "likes" : [ "golf", "table-tennis", "tennis" ] }
{ "_id" : ObjectId("4fcfcf11fd6230c28d817744"), "name" : "paul", "likes" : [ "golf", "table-tennis", "tennis", "rugby" ] }
```

If we would like to retrieve students who likes golf or tennis:

```
db.students.find({$or : [{likes : 'tennis'},{likes : 'golf'}]})
```

Then we get queries;

```
{ "_id" : ObjectId("4fcfcf02fd6230c28d817742"), "name" : "mike", "likes" : [
"basketball", "football", "tennis", "rugby" ] }
{ "_id" : ObjectId("4fcfcf08fd6230c28d817743"), "name" : "cassandra", "likes" : [
"golf", "table-tennis", "tennis" ] }
{ "_id" : ObjectId("4fcfcf11fd6230c28d817744"), "name" : "paul", "likes" : [ "golf",
"table-tennis", "tennis", "rugby" ] }
```

as expected. In these array fields, it is very easy to combine some of the fields let alone one of the field query returning. It becomes extremely useful as time goes by. One operation is \(inwhich tries to determine whether the values are in the array.

```
db.scores.find({a:{'$in':[2,3,4]}})
```

Another similar and useful operation is `exists` operation which checks whether the value matches to any value in the database checking every field in the collection. Id's of documents can also be selected using `_id` field in the collection as such:

```
db.matrix.find({_id: ObjectId("4fd0685b4e0fa619963db3b3")})
```

and it results in the respective document:

```
{ "_id" : ObjectId("4fd0685b4e0fa619963db3b3"), "name" : "Morpheus", "gender" :
"m", "age" : 45 }
```

If we have document which have common fields and want to count them, we could do so by using `count` operation.

```
db.matrix.count({name: "Morpheus"})
```

and it returns 2.(I added the same item twice). If it does not find it, it returns 0 as expected.

Removing Queries

We could also erase the documents based on their properties as such:

```
db.matrix.remove({name: "Morpheus"})
```

If we want to remove all the entries, we could simply do not give any field information or put `null` in `remove` operation:

```
db.matrix.remove()
```

For updates, let's first create a database in a different syntax:

```
db.users.save({name: 'John', languages: ['ruby', 'c','java', 'javascript']});
db.users.save({name: 'Sue', languages: ['haskell', 'lisp','python','lush']});
```

In order to update, we need to first select the document based on its one of the field, in this example it would be the names of people and using `update()` operation, we could update as such:

```
db.users.update({name: 'John'}, {name: 'Johnny', languages:
['scala','java','python']});
```

Instead of updating the entire document, we could update only the fields:

```
db.users.update({name:'Sue'},{'$set': {age:25}})
```

And when we try to print out the collection:

```
{ "_id" : ObjectId("4fd7d5aba46929bd0bbd56f7"), "name" : "Johnny", "languages" : [
"scala", "java", "python" ] }
{ "_id" : ObjectId("4fd7d5ada46929bd0bbd56f8"), "age" : 25, "languages" : [ "scala",
"lisp" ], "name" : "Sue" }
```

In order to update array elements, we could use `\(pull` and `\(push` operations. For example, if we want to remove *haskell* language from Sue's languages:

```
db.users.update({name: 'Sue'}, {'$pull': {'languages': 'haskell'} });
```

and if we want to add a language say java:

```
db.users.update({name: 'Sue'}, {'$push': {'languages': 'java'} });
```

Upsert(Update + Insert)

Mongo supports so called *upserts* which is nothing more than a fancy combination of update and insert. That is, if item that we want to update is not in the collection, it automatically creates it. If it does exist in the collection, it updates by default. However, in order to enable this feature of Mongo, we need to enable the third parameter of `update` operation as `true`. Say, we need to create a website hit counter, and in order to do so we increment the number of hits every time the name of website is updated. If we do not have the website name in the collection, we do not have to create it beforehand. We could just use upsert as such:

```
db.hits.update({page: 'yahoo'},{$inc: {hits: 1}},true)
```

If we do not have the third parameter or set to `false`, above statement does not change anything in the collection.

Multiple Updates

If we want to multiple updates in the collection, we need to enable the fourt parameter in the `update` operation. For example, we want to reset the counter of websites as such:

```
db.hits.update({},{$set: {hits: 0}},false,true)
```

By doing so, we update all the documents in the collection. However, if we do not enable the fourt parameter as `true`, then *only the first element of the collection will be updated*.

Deeper in `find()` operation

If we want to retrieve specific fields of the documents we could use a second parameter in `find()`,e.g. only the names of the webpages as such:

```
db.hits.find(null,{page:1})
```

then, it results in only page fields of documents, namely *google* and *yahoo*.

Ordering

Say, we have a database as such:

```
{ "_id" : ObjectId("4fd8cf5bbd6be5d371385b9a"), "hits" : 7, "page" : "yahoo" }
{ "_id" : ObjectId("4fd8d0fabd6be5d371385b9b"), "hits" : 5, "page" : "google" }
{ "_id" : ObjectId("4fd91925da57fdfbb68d7848"), "page" : "microsoft", "hits" : 10 }
{ "_id" : ObjectId("4fd91930da57fdfbb68d7849"), "page" : "facebook", "hits" : 15 }
{ "_id" : ObjectId("4fd91938da57fdfbb68d784a"), "page" : "apple", "hits" : 30 }
```

and we want to order this collection by the number of hits in a descending order. We could do this by using `sort()` operation as such:

```
db.hits.find().sort({hits: -1})
```

If we want to also return some specific ranks in the sort, we could use `limit()` and `skip()`operations. For example, we want to return second and third queries only.

```
db.hits.find().sort({hits: -1}).limit(2).skip(1)
```

Counting `count()`

`count()` can be used itself as an independent operation similar to `update` and `insert`. However, its origin is to follow `find().count()`. Therefore, `count()` operation can be considered as a syntactic sugar for `find` and `count`. If we want to count the webpages which have higher than 9 hits;

```
db.hits.find({hits: {$gt: 9}}).count()
```

MongoDB Difference and Similarities to RDBMS

There is no `join` opearation in Mongo contrary to RDBMS,but we could connect by using a foreign key in the documents of collections. In order to show the embedded and relational part of database, we start with employees example:

```
{ "_id" : ObjectId("4d85c7039ab0fd70a117d730"), "name" : "Paul" }
{ "_id" : ObjectId("4d85c7039ab0fd70a117d731"), "name" : "Duncan", "manager" :
ObjectId("4d85c7039ab0fd70a117d730") }
{ "_id" : ObjectId("4d85c7039ab0fd70a117d732"), "name" : "Moneo", "manager" :
```

```
ObjectId("4d85c7039ab0fd70a117d730") }
```

We set Paul as a manager of Duncan and Moneo's. In the collection, in order to find the manager, we could do as such:

```
db.employees.find({manager: ObjectId("4d85c7039ab0fd70a117d730")})
```

If we have two managers in the company, then we could add two managers as such:

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d733"),
name: 'Siona',manager: [ObjectId("4d85c7039ab0fd70a117d730"),
ObjectId("4d85c7039ab0fd70a117d732")] })
```

We could again return the manager using above operation.

```
db.employees.find({manager: ObjectId("4d85c7039ab0fd70a117d730")})
```

If we want to embed more documents into the collection like a nested document, we could do so as such:

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d734"), name:
'Ghanima',
family: {mother: 'Chani', father: 'Paul', brother:
ObjectId("4d85c7039ab0fd70a117d730")}})
```

We could return the respective field of the nested document by using *dot* notation.

```
db.employees.find({'family.mother': 'Chani'})
```

MongoDB Repair in Ubuntu

First, try to repair.

```
sudo rm /var/lib/mongodb/mongod.lock
sudo chown -R mongodb:mongodb /var/lib/mongodb/
sudo -u mongodb mongod -f /etc/mongodb.conf --repair
sudo service mongodb start
```

Reference:-

- https://docs.mongodb.com/getting-started/shell/

Mongoose

Overview

1. Mongoose is a Node.js library that provides MongoDB object data mapping (ODM)
2. Object Data Mapping (ODM) - similar to ORM where data from the database is converted automatically into a JavaScript object
3. Must first have Node.js and MongoDB installed for code samples to work
4. Install
5. `$ npm install --save mongoose`

Connecting to MongoDB

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

var db = mongoose.connection;

// Write any connection errors to console

db.on('error', console.error);

db.once('open', function() {

      // Create schemas and models here

});

mongoose.connect('mongodb://localhost/test');
```

Schemas and models

1. **Schema: defines the structure of documents within a MongoDB collection**
2. **Model: used to create instances of data that will be stored in documents**
3. **Schema for students database:**

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

name:      String,

gpa:       { type: Number, min: 0, max: 4 },

interests: [ String ],

enrolled:  Boolean

});

// Create a model from this schema and a MongoDB collection
"Student"

var Student = mongoose.model('Student', studentSchema);
```

CRUD operations

1. **Create**
    1. **Create a single student**

```
var stu = new Student({
            name: 'Sue Black',
            gpa: '3.1',
            interests: ["biking", "reading"],
            enrolled: true
      });

      // Save Sue
      stu.save(function(err, stu) {
            if (err) return console.error(err);
            console.dir(stu);
      });
```

2. **Read**
    1. **Read/query methods**
        i. **Model.find(conditions, [fields], [options], [callback])**
        ii. **Model.findById(id, [fields], [options], [callback])**
        iii. **Model.findOne(conditions, [fields], [options], [callback])**
    2. **Find a single match**

```
// Find one 'Sue Black'
Student.findOne({ name: 'Sue Black' }, function(err, stu) {
      if (err) return console.error(err);
      if (stu === null)
            console.log('No student found');
      else
            console.dir(stu);
});

// Case-insensitive search using a regex
var searchName = 'black';
Student.findOne({ name: new RegExp(searchName, 'i') },
function(err, stu) {
      if (err) return console.error(err);
      if (stu === null)
```

```
                console.log('No student found');
        else
                console.dir(stu);
    });
```

Delete

1. **Remove methods:**
    1. **Model.remove(conditions, [callback])**
    2. **Model.findByIdAndRemove(id, [options], [callback])**
    3. **Model.findOneAndRemove(conditions, [options], [callback])**
2. **Remove by ID**

```
// Remove this one student
var id = '54c25ef804381320619c4ca9';
Student.findByIdAndRemove(id, function(err, stu) {
    if (err) return console.error(err);
    if (stu === null)
        console.log('No student deleted');
    else
        console.log('Removed student ' + stu);
});
```

Reference:-

- http://mongoosejs.com/docs/