Contact Manager - written in AngularJS, Express and MongoDB - Episode 1

27 JUNE 2013

Probably you have noticed that I like using Star Wars themes in my post titles, and yet again, it's time to write an article with multiple parts. As I'm very close to finishing the online card game, I needed a new challenge, something else to develop and document.

I thought the easiest way to learn more about AngularJS, Express and MongoDB would be to create a contact manager, where you can list/search/query your contacts and view more

information about them.

There will be two parts to this article, part one is where I'm going to discuss the backend bit - that is the part which produces the JSON data from MongoDB, and the second part will discuss the frontend elements, running an Express server with AngularJS. Let's start by discussing the database portion.

The strength behind using MongoDB with the other JavaScript technologies is that they are all using the same technology (JSON, which MongoDB actually calls BSON) and developing around these objects becomes really easy. If you haven't worked with MongoDB it's part of the "NoSQL" family, and at a very high level it's a simple key-value storage system. The example outlined in this post is for demonstration purposes only and therefore the database won't be extensive but kept simple. I am not using MongoDB's native drivers but I am utilising a great library called Mongoose. Just by following the quick start guide on that site, you can achieve a lot. Let's have a look at my code example:

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost
/contact');
var Schema = mongoose.Schema;

var ContactSchema= new Schema({
    name: { type: String, required:
true },
    phone: { type: Number },
});

var ContactModel =
mongoose.model('Contact',
ContactSchema);
```

Here we are requiring the mongoose package (you can install that via npm - `npm install mongoose`). (Please note that this tutorial does not cover the installation and setup of MongoDB - refer to this manual for the right setup option). As a next step we need to connect to our database - the `connect()` method will automatically create a document if one doesn't exist. MongoDB is a document-based database

system, and as a result, all records, or data, in MongoDB are documents. Documents are the default representation of most user accessible data structures in the database.

We should also define a schema for our Contacts, which essentially means that we need to call a class that already exists inside mongoose - `Schema`. In the example above I'm setting up a very simple document with two keys, name (String) and phone (Number). There are only a few schema types that exist - for a full list, please refer to the documentation.

As a last thing, we will create a model based on the ContactShema that we have just created -- again, for a full explanation about Models, refer to the Mongoose documentation.

Let me discuss routing at the **backend** for a minute. As mentioned before I will be using Express, so as a first step I need to setup my app configuration.

```
var express = require("express");
var contacts = require("./contact");
var app = express();

app.listen(1222, "127.0.0.1");
```

If you execute `node index.js` and navigate to http://host:1222 in your browser you should see a `Cannot GET /` message. There's nothing to worry about, it's normal as we haven't yet specified any routes. With Express you can simply define your routes to be "/" and that would automatically load the index.html (if exists) file and parse that in the browser. However, let's not forget that we are currently writing a backend interface, that - ultimately - should only produce us data, and not html. Some of you at this point may already have a feeling for what's coming. In order to achieve this, I have to go back to my `contact.js` file and extend that to produce the data that I need, and later on I will add calls to those methods in my `index.js` file.

That being said, add the following to `contact.js`:

```
exports.index = function (req, res){
    return ContactModel.find(function
(err, contacts) {
        if (!err) {
                res.jsonp(contacts);
        } else {
                console.log(err);
        }
    });
}

exports.findById = function (req,
res) {
    return
ContactModel.findById(req.params.id,
function (err, contact) {
      if (!err) {
            res.jsonp(contact);
      } else {
            console.log(err);
      }
    });
}
```

These two methods should be pretty straight forward. The first one will return all the entries from our ContactModel (consider this as a `SELECT * FROM CONTACTS` type of query if you're familiar with MySQL), whereas the second method will only return one result (the MySQL equivalent would be `SELECT * FROM CONTACTS WHERE id = int`).

We are ready to call these methods from our `index.js` - and we can do that because all the methods above are defined with the `exports` tag - add the following to your code:

```
app.configure(function () {
  app.use(express.bodyParser());
});
app.get("/", contacts.index);
app.get('/contacts',
contacts.index);
app.get('/contacts/:id',
contacts.findById);
```

If you restart your application by stopping the

previous instance and re-executing the `node index.js` command, you should see empty data returned by MongoDB once you navigate to http://host:1222/.

You don't have any data and the only routing that we added to Express is utilising "get" calls. I need to stop here and explain a few things about the RESTful web calls and the GET method as well. I'm sure that most of you who are reading this post are familiar with GET and POST requests - if you've done any sort of web development I'm 100% sure that you've come across these methods, especially when dealing with URLs and forms. HTTP as a protocol has a vocabulary of actions that are being referred to as operation methods, the most well-known ones are:

- GET - request information
- POST - add information
- PUT - update existing information
- DELETE - remove information

(If you wonder what's the difference between GET and POST - well, in a GET request the

key-value pairs are sent via the URL, whereas with a POST request the key-value pairs are sent as part of the HTTP request after the headers.)

Our code above does miss the PUT method, so let's add that and configure our code to add a contact to the database ...

```
exports.addContact = function (req,
res) {
    var contact;
    contact = new ContactModel({
        name: req.body.name,
        phone: req.body.phone,
    });
    contact.save(function (err) {
        if (!err) {
            console.log("created");
        } else {
            console.log(err);
        }
    });

    return res.send(contact);
```

```
}
```

... followed by the routing for our POST request:

```
app.post('/contacts',
contacts.addContact);
```

If you'd like to test adding a contact, you can do this via the command line at the moment (due to the lack of the WUI). Fire up your terminal window and execute the following (make sure that you have re-started the application):

`curl -i -X POST -H 'Content-Type: application/json' -d '{"name": "Uncle Joe"}' http://host:1222/contacts`

This should result in something similar:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json;
charset=utf-8
Content-Length: 62
Date: Wed, 26 Jun 2013 09:24:05 GMT
```

```
Connection: keep-alive


{
  "name": "Uncle Joe",
  "_id": "51cab335b7e1587508000001"
}
```

If you now navigate to `http://host:1222` with your web browser, you should see this result returned in a JSON format:

```
[
  {
    "name": "Uncle Joe",
    "_id": "51cab335b7e1587508000001",
    "__v": 0
  }
]
```

The last two things to add are of course the update and delete methods:

```
exports.updateContact = function
```

```
(req, res) {
  return
ContactModel.findById(req.params.id,
function (err, contact) {
    contact.name = req.body.name;
    contact.phone = req.body.phone;
    contact.save(function (err) {
      if (!err) {
        console.log("updated");
      } else {
        console.log(err);
      }
       res.send(contact);
    });
  });
}

exports.deleteContact = function
(req, res){
  return
ContactModel.findById(req.params.id,
function (err, contact) {
    return contact.remove(function
(err) {
```

```
        if (!err) {
          console.log("removed");
          return res.send('');
        } else {
          console.log(err);
        }
      });
    });
}
```

Don't forget to update index.js as well:

```
app.post('/contacts', contacts.addContact);
app.put('/contacts/:id', contacts.updateCon
app.delete('/contacts/:id', contacts.delete
```

Time to test this again - run the following in your terminal: `curl -i -X PUT -H 'Content-Type: application/json' -d '{"name":"Uncle Jack"}' http://host:1222/contacts/put-the-id-here`, executing this should result in the following:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json;
charset=utf-8
Content-Length: 75
Date: Thu, 27 Jun 2013 10:01:53 GMT
Connection: keep-alive
{
  "name": "Uncle Jack",
  "_id": "51caaff064024e5608000001",
  "__v": 0
}
```

To verify that the update was successful, query for all the contacts by executing the curl statement shown before.

Let's now delete Uncle Jack - `curl -i -X DELETE http://tamas:1222/contacts/put-the-id-here`.
This only produces an '200 OK' HTTP message:

```
HTTP/1.1 200 OK
X-Powered-By: Express
```

```
Content-Type: text/html;
charset=utf-8
Content-Length: 0
Date: Thu, 27 Jun 2013 10:02:21 GMT
Connection: keep-alive
```

To verify that the deletion was successful query for all the contacts again (either via the terminal or your browser).

And there you have it, a fully functional RESTful API that is able to utilise CRUD operations. In the next article we'll add the frontend to this and we'll start displaying and manipulating this data from AngularJS. Stay tuned and watch this space...

Tamas Piros

Experienced software engineer, blogger, author and teacher & preacher of super-heroic web technologies.

http://fullstacktraining.com/

Share this post