

COM644 Full-Stack Web and App Development

Practical B4: Mongoose and RESTful APIs

Aims

- To introduce the concept of RESTful services
- To explain the HTTP methods and response codes
- To demonstrate API design and the appropriate combinations of methods and response codes for a range of endpoints
- To introduce Mongoose as an Object Data Mapping library for MongoDB
- To demonstrate the establishment in a Mongoose connection
- To demonstrate the construction of a Mongoose schema and its compilation to a data model
- To illustrate recasting of a database structure to best suit the needs of an application.

Contents

B4.1 INTRODUCTION TO RESTFUL APIS	2
B4.1.1 WHAT IS A RESTFUL API?	2
B4.1.2 HTTP METHODS AND RESPONSES	3
B4.1.3 API DESIGN	4
B4.2 MONGOOSE	6
B4.2.1 INSTALL MONGOOSE	6
B4.2.2 ESTABLISH A MONGOOSE CONNECTION	6
B4.3 DATA SCHEMAS AND MODELS	8
B4.3.1 BASIC SCHEMA ELEMENTS	9
B4.3.2 NESTED SCHEMA ELEMENTS	11
B4.3.3 USING GEO-COORDINATES	13

B4.1 Introduction to RESTful APIs

In the previous practical we integrated the MongoDB database with our Express application, replacing the previous JSON data file with a live connection to the database. Although it is important to know how to work with the MongoDB database directly in Node.js, most applications use an additional software layer to manage database activity and in this practical we will introduce Mongoose – a MongoDB manager for the MEAN stack.

Mongoose is particularly suited for the design of what are known as RESTful APIs – that is, web applications built using a predictable pattern of resources and URLs, where each URL identifies a single, well-defined activity.

B4.1.1 What is a RESTful API?

REST is an abbreviated form of “**RE**presentational **State Transfer**” and describes a software model in which a resource-based architecture communicates representational state information between client and server. By resource-based, we mean that the architecture is designed around resources or objects (nouns) rather than operations (verbs). Hence, we implement URLs for businesses, students, cars, or whatever other objects our application is dealing with – rather than “new”, “edit”, “delete” or whatever other operations are being implemented.

Typical URL patterns in a RESTful architecture (for example in our sample *WeMeanBusiness* application) might therefore be (e.g)

- <http://www.wemeanbusiness.com/businesses>
- <http://www.wemeanbusiness.com/businesses/id>
- <http://www.wemeanbusiness.com/businesses/id/reviews>
- <http://www.wemeanbusiness.com/users>

The REST architecture describes six constraints as follows.

- **Uniform interface between client and server**

This is fundamental to RESTful design and has three main elements; (1) the client indicates a request to the server by using one of the HTTP verbs such as GET, POST, PUT or DELETE; (2) The client request is in the form of a URL; (3) The server response consists of an HTTP status code and a body represented in JSON or XML

- **Statelessness**

This is the concept that the server contains no state information about the client – each client request is self-contained and provides enough information for it to be processed. Any state information that is required (for example cookies, session

variables, etc.) are held on the client.

- **Cacheable**

All server responses are cacheable

- **Client-server**

The server and client are disconnected with no shared resources. The uniform interface (above) is the link between the two.

- **Layered system**

Connected to cacheability and client-server organization, the client cannot assume any direct connection with the server as there may be software or hardware intermediaries. For example, a request could be satisfied by a cache rather than by the server, or the request may be re-directed to an alternative server.

- **Code on demand**

The server can temporarily extend the client by providing executable code as part of the response, which is then run by the client. For example, the response from the server may contain JavaScript to be run on the browser.

B4.1.2 HTTP Methods and Responses

We have seen how the URL pattern in a RESTful application identifies the data object with which we want to perform some operation, but how do we denote the actual operation that we want to execute? For example, a call to the URL

<http://www.wemeanbusiness.com/businesses/id/reviews>

might mean that we want to see the collection of user reviews for the business identified by the **id** parameter in the URL. However, it might equally mean that we want to add a review to the collection of reviews for that business.

The way in which we communicate our intent to the server is by selection of the appropriate HTTP method (verb). You may already be familiar with the **GET** and **POST** methods from previous work with HTML forms, but there is a wider selection of methods available, each with their own specific purpose, corresponding to the standard CRUD (Create, Read, Update, Delete) operations on databases and described in Table B4.1 below

HTTP Method	Action requested
GET	Return a resource
POST	Create a new resource
PUT	Update a resource
DELETE	Delete a resource

Table B4.1 HTTP Methods

Therefore, a **GET** request to the previous URL <http://www.wemeanbusiness.com/businesses/id/reviews> will indicate that we want to fetch a list of reviews, while a **POST** request to the same URL will indicate that we want to add a new review to the collection.

In addition, the RESTful model suggests recommended HTTP responses to each combination of URL and method. Again, you will have already used some of these (e.g. 200 OK, 404 Page not found, etc.), but the complete table is presented below.

Method	Entire collection (e.g. /businesses)	Specific Item (e.g. /businesses/id)
POST	201 (Created); Location header with a link to the newly created business	404 (Not found) or 409 (Conflict) if the resource already exists
GET	200 (OK); List of businesses. Use pagination, sorting and filtering to navigate large collections	200 (OK); Single business in JSON format or 404 (Not found) if id not found or invalid
PUT	404 (Not found) – unless you provide functionality to modify every element in the collection	200 (OK) or 204 (No content provided) or 404 (Not found) if id not found or invalid
DELETE	404 (Not found) – unless you intend functionality to delete the entire collection	200 (OK) or 404 (Not found) if id not found or invalid.

Table B4.2 HTTP Methods and Response Codes

B4.1.3 API Design

The previous discussions on RESTful services and HTTP methods and responses lead us towards a design pattern for the endpoints (URLs) in our API. We can illustrate this by compiling a list of the functionality we intend to provide in our demonstration application, specifying the HTTP method and URL for each. This list is presented in Table B4.3 below –

note how a single URL can be used to indicate multiple actions, with the HTTP Method used to distinguish between them.

Method	URL	Action
GET	/api/businesses	Get all/multiple businesses
POST	/api/businesses	Create a new business
GET	/api/businesses/123	Get a specific business (with ID=123)
PUT	/api/businesses/123	Update a specific business
DELETE	/api/businesses/123	Delete a specific business
GET	/api/businesses/123/reviews	Get all reviews for a specific business
POST	/api/businesses/123/reviews	Add a review for a specific business
GET	/api/businesses/123/reviews/321	Get a specific review (with ID=321) for a specific business
PUT	/api/businesses/123/reviews/321	Update a specific review for a specific business
DELETE	/api/businesses/123/reviews/321	Delete a specific review

Table B4.3 API Design for the WeMeanBusiness Sample Application

A standard approach to URL specification is an important element in API design, but it is also important to standardise the data across all of the different endpoints. For example, the specification of a **review** object in a **POST** request to **/api/businesses/123/reviews** should be the same as that in a **PUT** request to **/api/businesses/123/reviews/321**. Specifically, if the **POST** request provides a 'star' rating as an integer, then the **PUT** request should not provide a value for the same element as a string.

We could look after this ourselves in the application code, having the controller logic perform all of the type checking and conversion, but it a much better approach would be to define a single schema object to which our data must adhere, and have the database manage and enforce this schema.

The native MongoDB driver does not have this capability, but fortunately there are a number of packages available through **npm** that will provide this useful layer of data management, making available schemas, models, helpers and validation methods that make it much easier to produce scalable database-driven applications. The most widely used of

these in the MEAN stack is **Mongoose**, so we will install this package and convert our application to allow Mongoose to manage the communication with the MongoDB database.

B4.2 Mongoose

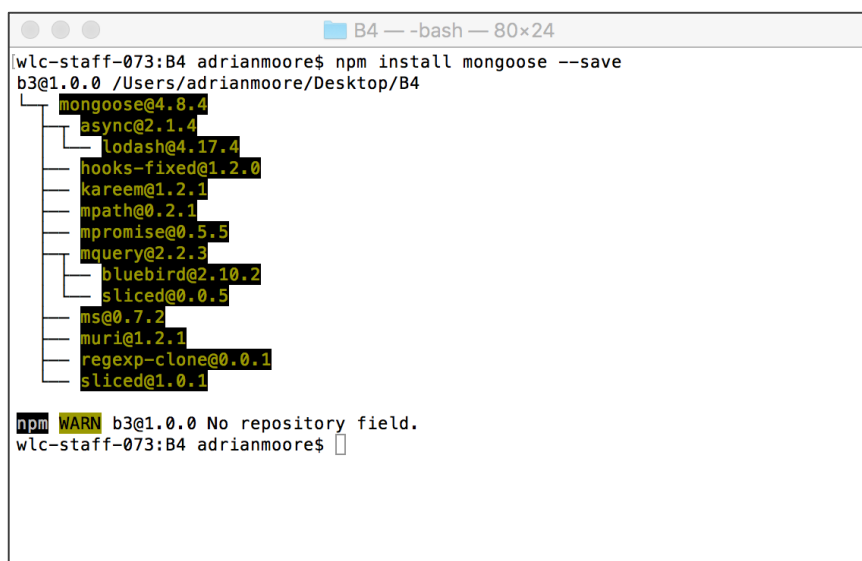
Mongoose is a data-modelling format for MongoDB data that includes built-in type-casting, validation and query building. It is becoming increasingly popular in applications built using the MEAN stack as it provides a lot of the data management logic that would otherwise have had to be implemented in the application controllers.

B4.2.1 Install Mongoose

We can install the Mongoose package from the Command prompt by executing

```
U:\B4> npm install mongoose --save
```

as shown in Figure B4.1 below.



```
wlc-staff-073:B4 adrianmoore$ npm install mongoose --save
b3@1.0.0 /Users/adrianmoore/Desktop/B4
├─┬─ mongoose@4.8.4
│   ├── async@2.1.4
│   ├── lodash@4.17.4
│   ├── hooks-fixed@1.2.0
│   ├── kareem@1.2.1
│   ├── mpath@0.2.1
│   ├── mpromise@0.5.5
│   ├── mquery@2.2.3
│   ├── bluebird@2.10.2
│   ├── sliced@0.0.5
│   ├── ms@0.7.2
│   ├── muri@1.2.1
│   ├── regexp-clone@0.0.1
│   └── sliced@1.0.1
npm WARN b3@1.0.0 No repository field.
wlc-staff-073:B4 adrianmoore$
```

Figure B4.1 Installing Mongoose from npm

B4.2.2 Establish a Mongoose connection

The Mongoose connection architecture is slightly different from that seen earlier with the native MongoDB database package. Whereas the native MongoDB connection took a callback function that would be executed when the connection was established, in

Mongoose we listen for connection events and act upon them when they are detected. This is demonstrated in the code box below that presents the database connection file that will replace our current **dbConnect.js**.

File: B4/api/data/dbConnect.js

```
var mongoose = require('mongoose');
var dbURL = 'mongodb://localhost:27017/businessDB';

mongoose.connect(dbURL);

mongoose.connection.on('connected', function() {
  console.log("Mongoose connected to " + dbURL);
});

mongoose.connection.on('disconnected', function() {
  console.log("Mongoose disconnected" );
});

mongoose.connection.on('error', function(err) {
  console.log("Mongoose connection error " + err);
});
```

Here, we call the Mongoose **connect()** method, passing it the database connection string as a parameter. Next we set up three event listeners, each monitoring a specific Mongoose event (**'connected'**, **'disconnected'** and **'error'**) and triggering a callback function that generates a **console.log()** message. The structure of each event handler is identical – except for the **'error'** event which passes an error object as a parameter to the callback function.

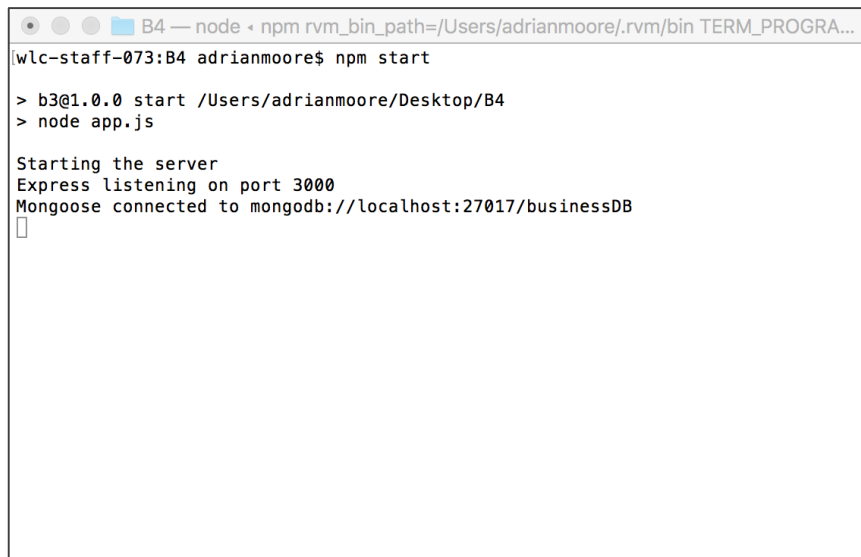
Finally, we need to modify the line of code at the top of **app.js** that calls the database connection, to remove the reference to the former **open()** method that we used with the native MongoDB connection. The revised **app.js** is presented in the code box below.

File: 43/app.js

```
require('./api/data/dbConnect.js');

...
```

We can now test the Mongoose connection by running the **mongod** database process and starting the application. If everything has been updated correctly, we should see the “Mongoose connected...” message illustrated in Figure B4.2.



```
B4 — node · npm rvm_bin_path=/Users/adrianmoore/.rvm/bin TERM_PROGRA...
wlc-staff-073:B4 adrianmoore$ npm start
> b3@1.0.0 start /Users/adrianmoore/Desktop/B4
> node app.js

Starting the server
Express listening on port 3000
Mongoose connected to mongodb://localhost:27017/businessDB
█
```

Figure B4.2 Mongoose connected

We can also check the ‘**disconnect**’ and ‘**error**’ event listeners by stopping and restarting the **mongod** process and the application as follows.

- i) Check the ‘**disconnect**’ event by stopping the **mongod** process (Ctrl-C) while the application is still running. You should now see the “Mongoose disconnected” message in the Console window. Now start **mongod** again and observe how this is automatically detected by Mongoose, resulting in the “Mongoose connected...” message being displayed once more.
- ii) Stop both the **mongod** process and the application and then re-start the application ONLY. As the application attempts to make the connection without a database server running, you should see the “Mongoose connection error” in the Console window.

B4.3 Data Schemas and Models

One of the most useful features of Mongoose is the support provided for data modelling. This allows us to define the data type and structure of each of the elements in our MongoDB document, so that the application can automatically monitor the data presented and ensure that it represents a valid object in the context of our application.

A data model is created by passing to it a Mongoose Schema, which describes the structure of the document as a JavaScript object. We will create a new model for our sample collection of businesses by creating a new file in the `/api/data` folder called `businesses.model.js`.

B4.3.1 Basic Schema Elements

The schema is defined as a sequence of **paths**, each referring to a database field, with associated **schema types**. For example, in the code box below we have the paths `"name"` and `"city"` defined as type **String**, paths `"stars"` and `"review_count"` defined as type **Number** and path `"categories"` defined as an **array** of string values.

File: `B4/data/businesses.model.js`

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

var businessSchema = new mongoose.Schema({
  name : String,
  stars : Number,
  city : String,
  review_count : Number,
  categories : [String]
});
```

Sometimes, however, we want to record additional information about certain paths. For example, let's specify that the `"name"` is a compulsory element that must be provided (by default, all fields are optional) and that the `"stars"` value must be in the range 0-5. We will also specify that the default value for `"stars"` should be 0.

In these cases, we need to expand the definition for those paths, so that the value is a JavaScript object, with a `"type"` attribute used to define the schema type. The following code box presents the schema modified with these new requirements.

File: B4/data/businesses.model.js

```
var businessSchema = new mongoose.Schema({
  name : {
    type : String,
    required : true
  },
  stars : {
    type : Number,
    min : 0,
    max : 5,
    default: 0
  },
  ...
```

Once the schema has been defined, we compile it to a model by the command

```
mongoose.model(Model_name, schema_name, MongoDB_collection)
```

where, by convention, we specify the model name as a capitalized, singular version of the collection name. The full model definition (so far) is shown in the code box below.

File: B4/data/businesses.model.js

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

var businessSchema = new mongoose.Schema({
  name : {
    type : String,
    required : true
  },
  stars : {
    type : Number,
    min : 0,
    max : 5,
    default: 0
  },
  city : String,
  review_count : Number,
  categories : [String]
});

mongoose.model('Business', businessSchema, 'business');
```

Now, we need to tell the application about the existence of the data model, so that Mongoose can direct all database access through the model. This is done by simply **require**-ing the new **businesses.model.js** file at the end of the database connection code.

File: B4/data/dbConnect.js

```
var mongoose = require('mongoose');
var dbURL = 'mongodb://localhost:27017/businessDB';

mongoose.connect(dbURL);

mongoose.connection.on('connected', function() {
  console.log("Mongoose connected to " + dbURL);
});

mongoose.connection.on('disconnected', function() {
  console.log("Mongoose disconnected" );
});

mongoose.connection.on('error', function(err) {
  console.log("Mongoose connection error " + err);
});

require('./businesses.model.js');
```

B4.3.2 Nested Schema Elements

All of the schema elements so far have either been simple values such as String or Number, or arrays of simple values. However, our **business** document has some elements that are specified as sub-documents in their own right – such as **reviews** and **photos** which are both arrays of complex elements defined as JavaScript objects.

Mongoose provides for such nested structures by allowing us to define separate schema definitions for each sub-document, and then using the sub-document specification as a schema type in the top-level schema. The only restriction is that sub-document schemas **MUST** be defined before they are used, to prevent forward-reference errors.

The following code box illustrates the definition of the sub-documents **votes** and **review** and their inclusion in the main scheme definition.

File: *B4/data/businesses.model.js*

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

var votesSchema = new mongoose.Schema({
  funny : Number,
  useful : Number,
  cool : Number
})

var reviewSchema = new mongoose.Schema({
  username : String,
  votes : votesSchema,
  user_id : String,
  review_id : String,
  text : String,
  business_id : String,
  stars : Number,
  date : {
    type : Date,
    default : Date.now
  },
  type : String
})

var businessSchema = new mongoose.Schema({
  name : {
    type : String,
    required : true
  },
  ...
  categories : [String],
  reviews: [reviewSchema]
});

mongoose.model('Business', businessSchema, 'business');
```

Here, we first define a schema for the **votes** object that is nested inside a review. This consists of three **Number** values which are used to count the “likes” gathered by a review. Then we create a schema for a review, using the new **votesSchema** as the schema type for the nested **votes** element. Note also the use of the **Date** schema type and how we can provide a default value for the current date when a review is submitted.

Finally, in the main schema body, we specify that the **reviews** element is defined as an array of **reviewSchema** elements.

B4.3.3 Using Geo-coordinates

A very useful feature of Mongoose is the ability to index documents according to their geo-location. This makes it possible to query the database by location, for example *“find a restaurant near a given business”*.

Our data set has all of the information required to create such an index, but unfortunately it is not quite in the format needed, so we will first create the schema entry that will support geo-indexing and then see how to modify the data structure to provide this.

In our database, the location is described by the three fields **“full_address”**, **“longitude”** and **“latitude”**. Although these are separate fields in our database structure, it would be better if they were grouped into a single object – so we will create a schema element in this form, where a **“location”** is described by a **String** address and an array of **Number** coordinates in longitude, latitude order. We also define an **index** on the coordinates element as a **2dsphere**, instructing the database to index the coordinate values as an (x, y) representation on the surface of a sphere

File: B4/data/ businesses.model.js

```
...  
  
var businessSchema = new mongoose.Schema({  
  ...  
  location : {  
    address : String,  
    coordinates : {  
      type : [Number],  
      index : '2dsphere'  
    }  
  }  
})  
  
...
```

Now that we have defined the schema definition for location, we need to adjust our database structure to match. The easiest way to do this is to open a copy of the **B3** files in the code editor and add a new route **GET /fixDatabase** to the file **/api/routes/index.js**. This route should call the new controller **fixDatabase** as shown in the following code box.

Note: This part of the exercise should be carried out in your **B3** application. We are currently in the middle of converting the database access from the native MongoDB driver to Mongoose, and B3 is the latest fully runnable version.

File: B3/api/routes/index.js

```
...  
  
router  
  .route('/fixDatabase')  
  .get(businessesController.fixDatabase);  
...
```

Now, define the new controller in **/api/controllers/business.controllers.js**. This code will read all of the documents in the **business** collection and then loop across the array returned. For each document in the collection, we extract the **_id**, **full_address**, **longitude** and **latitude** values and then use an **updateOne** query to add a new element called **location** with the format specified in our schema.

File: B3/api/controllers/businesses.controllers.js

```
module.exports.fixDatabase = function(req, res) {  
  var db = dbConnect.get();  
  var collection = db.collection('business');  
  
  collection  
    .find()  
    .toArray(function(err, docs) {  
      for (var i = 0; i < docs.length; i++) {  
        business = docs[i];  
        _id = business._id;  
        full_address = business.full_address;  
        longitude = business.longitude;  
        latitude = business.latitude;  
        collection.updateOne (  
          { "_id" : _id },  
          { $set : {  
            "location" : {  
              "address" : full_address,  
              "coordinates" : [  
                longitude, latitude ]  
            }  
          }  
        });  
      }  
    });  
  res  
    .status(200)  
    .json({"Message" : "Database updated"});  
})  
}
```

Now, run the database correction code by running the application and visiting the URL <http://localhost:3000/api/fixDatabase> and export the new collection to a JSON file by the command

```
mongoexport --db businessDB --collection business --jsonArray --pretty  
--out businessDB.json
```

and verify that each document has a new location element added, in the format described by our schema, as illustrated in Figure B4.3 below.

```
126     "categories": [  
127         "Veterinarians",  
128         "Pets"  
129     ],  
130     "location": {  
131         "address": "6677 W Thunderbird Rd\nSte L-188\nGlendale, AZ 85306",  
132         "coordinates": [  
133             -112.2022787,  
134             33.6090627  
135         ]  
136     }  
137 },  
138 ]
```

Figure B4.3 Location element added to database