COM644 Full-Stack Web and App Development

Practical B5: Mongoose and Information Retrieval

Aims

- To demonstrate the implementation of GET routes in Mongoose
- To implement controller logic to retrieve collections and single documents
- To show how single routes can respond differently to varying HTTP request methods
- To introduce a technique for adding **`_id`** fields to sub-documents in a collection
- To demonstrate the retrieval of sub-documents
- To develop controller logic to handle geo-location queries
- To introduce error-trapping as a technique for “hardening” of an API.
- To develop logic to handle a range of error scenarios

Contents

B5.1 Implementing GET routes with Mongoose

In the previous practical we created the data model that we will now use for all of our database activity. In order to use this model in our controllers, we first need to **`require`** it at the top of the controllers file. This will replace the existing code that references the native MongoDB driver and is presented in the code box below.

File*: B5/api/controllers/businesses.controllers.js*

```
var mongoose = require('mongoose');
var Business = mongoose.model('Business');

module.exports.businessesGetAll = function(req, res) {
...
}

...
```

Once defined, we will address all database activity through the model.

B5.1.1 Retrieving a collection of documents

The **GET /api/businesses** route is implemented by the **`businessesGetAll()`** controller, which accepts optional querystring parameters **`start`** and **`number`** and uses the **`find()`** command to retrieve a selected portion of the data set. In Mongoose, we use the **`find()`** method by chaining a method **`exec()`** (meaning '**execute**') which takes a callback function that is fired when the database action is complete. The parameters of the callback function are an error object, populated if the query fails and a data object that will contain the documents returned from the query.

The following code box illustrates the updated code for **`businessesGetAll()`**. Note that methods **`skip()`** and **`limit()`** are still available to us – Mongoose makes available many of the methods of the native driver, as well as adding additional helper functionality.

File*: B5/api/controllers/businesses.controllers.js*

```
module.exports.businessesGetAll = function(req, res) {

    var start = 0;
    var number = 5;

    if (req.query && req.query.start) {
        start = parseInt(req.query.start);
    }
    if (req.query && req.query.number) {
        number = parseInt(req.query.number);
    }

    Business
        .find()
        .skip(start)
        .limit(number)
        .exec(function(err, docs) {
            console.log("Retrieved data for " +
                         docs.length + " businesses");
            res
                .status(200)
                .json(docs);
        });
}
```

B5.1.2 Retrieving a single document by ID

Retrieving a single document by ID reveals another of the additional helper facilities provided by Mongoose. In the previous (native driver) version, we had to **`require`** the **`ObjectID()`** method that allowed us to manipulate **`_id`** values, but Mongoose provides a **`findById()`** method that will accept a string representation of an **`_id`** and carries out all manipulation internally.

The structure of the **`findById()`** method within the **`businessesGetOne()`** controller is illustrated by the following code box.

File: ***B5/api/controllers/businesses.controllers.js***

```
module.exports.businessesGetOne = function(req, res) {
    var businessID = req.params.businessID;
    console.log("GET business " + businessID);
    Business
        .findById(businessID)
        .exec(function(err, doc) {
            res
                .status(200)
                .json(doc);
        });
}
```

Try it now!

Verify the operation of the Mongoose queries by running the **mongod** progress, starting the application and presenting the URL http://localhost:3000/api/businesses to a web browser to check the `businessesGetAll()` controller. Now, copy one of the business `_id` values from the browser window and add it to the URL to create an address of the form http://localhost:3000/api/businesses/1234567 and check the `businessesGetOne()` controller.

B5.2 Working with sub-documents

All of our database work so far has been at the level of individual documents or collections of documents. However, our data structure requires that we may need to work at the level of sub-documents – for example when retrieving the reviews of a business or adding a new review.

In this section, we will see how to manipulate sub-documents in Mongoose, and also update our database structure so that all sub-documents have individual `_id` fields.

B5.2.1 Adding additional routes

Before we can specify controller functionality for sub-documents, we first need to create the new routes that will expose this functionality to our users. Examine the code box below that creates new routes to retrieve (i) all reviews for a specific business and (ii) a specific review for a specific business, where the business and review are identified by their **`_id`** values.

Note also that we choose to implement the reviews controllers in a new JavaScript file called **`reviews.controllers.js`**. We could equally keep all controllers in a single file, but it is better practice to keep related controllers together in this way.

File*: B5/api/routes/index.js*

```
var express = require('express');
var router = express.Router();

var businessesController =
    require('../controllers/businesses.controllers.js');

var reviewsController =
    require('../controllers/reviews.controllers.js');

router
    .route('/businesses/:businessID/reviews')
    .get (reviewsController.reviewsGetAll);

router
    .route('/businesses/:businessID/reviews/:reviewID')
    .get (reviewsController.reviewsGetOne);

...
```

Also, so that we can continue to run the application while we develop the new controllers, we create the new controller file **reviews.controllers.js** and create skeleton controller functions for **`reviewsGetAll()`** and **`reviewsGetOne()`**.

File*: B5/api/controllers/reviews.controllers.js*

```
var mongoose = require('mongoose');
var Business = mongoose.model('Business');

module.exports.reviewsGetAll = function(req, res) {

};

module.exports.reviewsGetOne = function(req, res) {

};
```

B5.2.2 Adding _id fields to Sub-documents

Before we can implement the new controllers **`reviewsGetAll()`** and **`reviewsGetOne()`**, we need to address a limitation of our data set – namely that our review elements do not have **`_id`** values on which to query. If you examine the JSON data, you will find that a **review** element does contain a **`review_id`** field, but we have no way of being sure that this is a valid **`ObjectId()`** value, so it is much safer to create our own.

The best way to generate **`_id`** fields is to return to version B3 of the application (the most recent fully working version) and to create a new route and controller to make this modification to the database.

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

Initially, we update the **api/routes/index.js** file to create a new route for the controller

File*: B3/api/routes/index.js*

```
...

router
    .route('/addReviewIDs')
    .get(businessesController.addReviewIDs);

...
```

Now, we will create the controller by adding a new function **`addReviewIDs()`** to **controllers.business.js**. It is worth studying this function carefully.

First, we use the **`find()`** and **`toArray()`** methods to obtain the complete list of documents in the collection. Then, the outer **`for`** loop iterates across the collection, extracting the business **`_id`** value and checking for the presence of a **`reviews`** element.

If a **`reviews`** element is present, we then read it into the **`reviews`** variable and enter the inner **`for`** loop to iterate across the collection of reviews, reading the **`review_id`** element for each into a local variable.

Next, we use the MongoDB **`update()`** method to select the document matching the selected business ID and review ID, and use the **`$set`** operator to generate a new **`_id`** element for the review.

Note the use of the **$**operator in **`reviews.$._id`** – this is a marker operator that will be automatically set to the array index position in the search object – i.e. update the same review element as where the **`review_id`** was located.

The full implementation of **`addReviewIDs()`** is presented in the following code box.

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

```
module.exports.addReviewIDs = 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++) {
            businessID = docs[i]._id;
            if (docs[i].reviews) {
               reviews = docs[i].reviews;
               for (var thisReview = 0;
                    thisReview < reviews.length;
                    thisReview++) {
                reviewID = reviews[thisReview].review_id;
                collection.update (
                   { "_id" : businessID,
                     "reviews.review_id" : reviewID },
                   { $set : {
                     "reviews.$._id" : ObjectId() }
                   }
                );
               }
            }
          }
          res
            .status(200)
            .json( { "Message" : "Review IDs added"});
    })
}
```

We can run this controller and update our database structure by re-starting the application and presenting the URL http://localhost:3000/addReviewIDs to a web browser.

You should then save the new database structure to a JSON file by the command

```
U:\B5> mongoexport --db businessDB --collection business --out businessDB.json
      --jsonArray --pretty
```

B5.2.3 Retrieving sub-documents

Note: We will now continue to introduce Mongoose to the application, so return at this point to the **B5** files.

Now that our data set has the required `_id` values for each `review` element, we can return to the new controllers `reviewsGetAll()` and `reviewsGetOne()`.

Fetching all reviews is actually quite trivial and the implementation is almost identical to that for `businessesGetOne()` – with only two differences. First, we chain the `select()` method to the `find()` to tell Mongoose that we only require the `reviews` elements to be returned. Finally, we modify the JSON response so that only the `reviews` element is returned to the browser.

File*: B5/api/controllers/reviews.controllers.js*

```
module.exports.reviewsGetAll = function(req, res) {
    var businessID = req.params.businessID;
    console.log("GET businessID " + businessID);

    Business
        .findById(businessID)
        .select("reviews")
        .exec(function(err, doc) {
            res
                .status(200)
                .json(doc.reviews);
        });
};
```

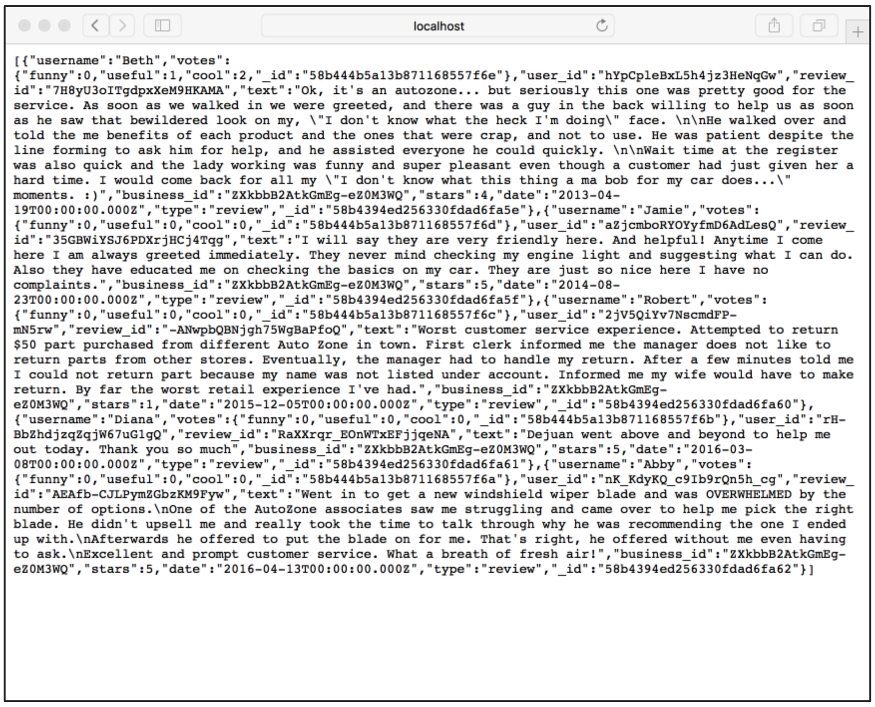
Running the application and providing the `_id` of a business as part of the URL (e.g. http://localhost:3000/api/businesses/589ddf35f3ff092e206f04ed/reviews) generates the browser response in Figure 5.1 below.

(**NOTE** - Remember that your `_id` values will be different, so load the page http://localhost:3000/api/businesses first and copy a business `_id` value from the data presented.)

Try it now!

The effect of the **`select()`** method is not immediately apparent, but you can easily see its effect by adding a **`console.log(doc)`** to the callback function, immediately before the **`res`** statement.

Run the application and view the output generated in the console. Now comment out the call to the **`select()`** method and run the application again to see clearly the difference it makes.



```
[{"username":"Beth","votes":
{"funny":0,"useful":1,"cool":2,"_id":"58b444b5a13b871168557f6e"},"user_id":"hYpCpleBxL5h4jz3HeNqGw","review_
id":"7H8yU3oITgdpxXeM9HKAMA","text":"Ok, it's an autozone... but seriously this one was pretty good for the
service. As soon as we walked in we were greeted, and there was a guy in the back willing to help us as soon
as he saw that bewildered look on my, \"I don't know what the heck I'm doing\" face. \n\nHe walked over and
told the me benefits of each product and the ones that were crap, and not to use. He was patient despite the
line forming to ask him for help, and he assisted everyone he could quickly. \n\nWait time at the register
was also quick and the lady working was funny and super pleasant even though a customer had just given her a
hard time. I would come back for all my \"I don't know what this thing a ma bob for my car does...\"
moments. :)","business_id":"ZXkbbB2AtkGmEg-eZ0M3WQ","stars":4,"date":"2013-04-
19T00:00:00.000Z","type":"review","_id":"58b4394ed256330fdad6fa5e"},{"username":"Jamie","votes":
{"funny":0,"useful":0,"cool":0,"_id":"58b444b5a13b871168557f6d"},"user_id":"aZjcmboRYOYyfmD6AdLesQ","review_
id":"35GBWiYSJ6PDXrjHCj4Tqg","text":"I will say they are very friendly here. And helpful! Anytime I come
here I am always greeted immediately. They never mind checking my engine light and suggesting what I can do.
Also they have educated me on checking the basics on my car. They are just so nice here I have no
complaints.","business_id":"ZXkbbB2AtkGmEg-eZ0M3WQ","stars":5,"date":"2014-08-
23T00:00:00.000Z","type":"review","_id":"58b4394ed256330fdad6fa5f"},{"username":"Robert","votes":
{"funny":0,"useful":0,"cool":0,"_id":"58b444b5a13b871168557f6c"},"user_id":"2jV5QiYv7NscmdFP-
mN5rw","review_id":"-ANwpbQBNjgh75WgBaPfoQ","text":"Worst customer service experience. Attempted to return
$50 part purchased from different Auto Zone in town. First clerk informed me the manager does not like to
return parts from other stores. Eventually, the manager had to handle my return. After a few minutes told me
I could not return part because my name was not listed under account. Informed me my wife would have to make
return. By far the worst retail experience I've had.","business_id":"ZXkbbB2AtkGmEg-
eZ0M3WQ","stars":1,"date":"2015-12-05T00:00:00.000Z","type":"review","_id":"58b4394ed256330fdad6fa60"},
{"username":"Diana","votes":{"funny":0,"useful":0,"cool":0,"_id":"58b444b5a13b871168557f6b"},"user_id":"rH-
BbZhdjzqZqjW67uG1gQ","review_id":"RaXXrqr_EOnWTxEFjjqeNA","text":"Dejuan went above and beyond to help me
out today. Thank you so much","business_id":"ZXkbbB2AtkGmEg-eZ0M3WQ","stars":5,"date":"2016-03-
08T00:00:00.000Z","type":"review","_id":"58b4394ed256330fdad6fa61"},{"username":"Abby","votes":
{"funny":0,"useful":0,"cool":0,"_id":"58b444b5a13b871168557f6a"},"user_id":"nK_KdyKQ_c9Ib9rQn5h_cg","review_
id":"AEAfb-CJLPymZGbzKM9Fyw","text":"Went in to get a new windshield wiper blade and was OVERWHELMED by the
number of options.\nOne of the AutoZone associates saw me struggling and came over to help me pick the right
blade. He didn't upsell me and really took the time to talk through why he was recommending the one I ended
up with.\nAfterwards he offered to put the blade on for me. That's right, he offered without me even having
to ask.\nExcellent and prompt customer service. What a breath of fresh air!","business_id":"ZXkbbB2AtkGmEg-
eZ0M3WQ","stars":5,"date":"2016-04-13T00:00:00.000Z","type":"review","_id":"58b4394ed256330fdad6fa62"}]
```

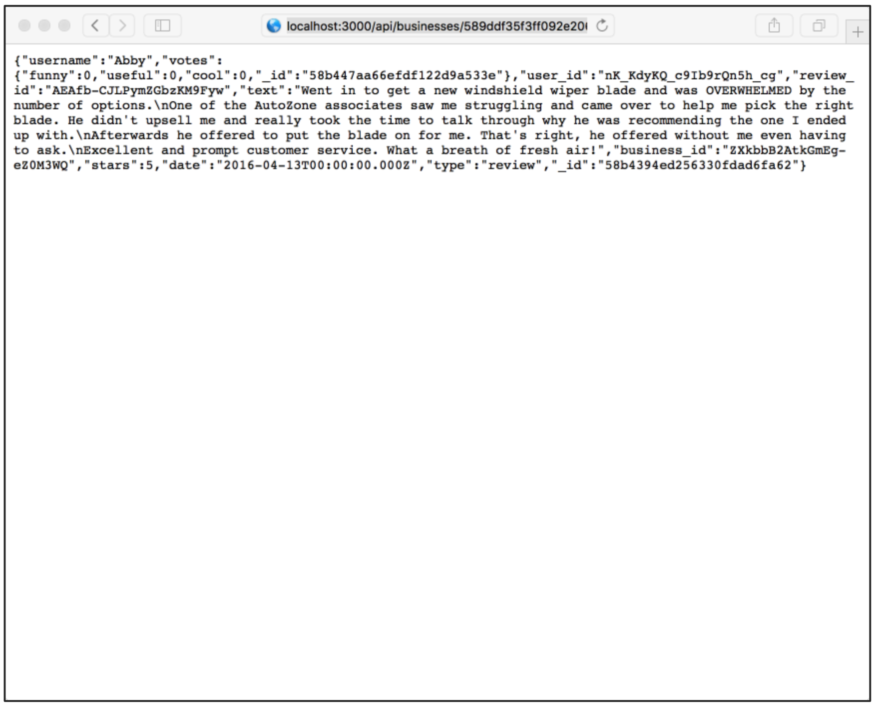
Figure B5.1 Fetching all reviews for a given business

The controller to return a specific review by ID is almost identical and requires us to obtain the review ID from the querystring parameter and then to use it in the Mongoose **`id()`** method. This method returns a sub-document from a collection iwhere the **`_id`** value matches that provided as a parameter. The following code box presents the full implementation of **`reviewsGetOne()`**.

File*: B5/api/controllers/reviews.controllers.js*

```
module.exports.reviewsGetOne = function(req, res) {
    var businessID = req.params.businessID;
    var reviewID = req.params.reviewID;
    console.log("GET reviewID " + reviewID);

    Business
        .findById(businessID)
        .select("reviews")
        .exec(function(err, doc) {
            var review = doc.reviews.id(reviewID);
            res
                .status(200)
                .json(review);
        });
};
```

Copying a review ID from the browser after the previous test and pasting it onto the URL (to give a URL in the form http://localhost/businesses/businessID/reviews/reviewID) generates output as shown in Figure B5.2.



```
{"username":"Abby","votes":
{"funny":0,"useful":0,"cool":0,"_id":"58b447aa66efdf122d9a533e"},"user_id":"nK_KdyKQ_c9Ib9rQn5h_cg","review_
id":"AEAfb-CJLPymZGbzKM9Fyw","text":"Went in to get a new windshield wiper blade and was OVERWHELMED by the
number of options.\nOne of the AutoZone associates saw me struggling and came over to help me pick the right
blade. He didn't upsell me and really took the time to talk through why he was recommending the one I ended
up with.\nAfterwards he offered to put the blade on for me. That's right, he offered without me even having
to ask.\nExcellent and prompt customer service. What a breath of fresh air!","business_id":"ZXkbbB2AtkGmEg-
eZOM3WQ","stars":5,"date":"2016-04-13T00:00:00.000Z","type":"review","_id":"58b4394ed256330fdad6fa62"}
```

Figure B5.2 Fetching a specific review by ID

B5.3 Geo-location queries

In previous practical, we laid the foundation for queries based on location by defining the **location** element in the schema and then updating the database to provide a corresponding item in each document. We will now illustrate the use of this by implementing new functionality that returns a collection of businesses sorted in order of proximity to the longitude and latitude values passed as querystring parameters (i.e. "nearest business first").

First, we need to consider whether a geo-location query should be accessed via a new route or whether it should be serviced by one of the existing routes. Thinking back to the design principles for RESTful APIs (Section B4.1), we recall that a retrieval operation for a collection of businesses should be implemented by a **GET** request to **api/businesses**. This route is already implemented, so we will modify its associated controller (**`businessesGetAll()`**) to take appropriate action if querystring parameters for longitude and latitude are passed to this endpoint.

The first modification to the controller is to insert a new test to check for the presence of **lng** and **lat** values in the querystring, as shown in the following code box.

File*: B5/api/controllers/businesses.controllers.js*

```
module.exports.businessesGetAll = function(req, res) {

    var start = 0;
    var number = 5

    if (req.query && req.query.lng && req.query.lat) {
        runGeoQuery(req, res);
        return;
    }

    ...
```

If these parameters exist, we call a separate function **`runGeoQuery()`** which will handle the Geo-location request, with the return statement exiting the **`businessesGetAll`** controller once this has been done. As we are passing complete control to the new function, we also need to pass it the **`req`** and **`res`** objects, so that it can retrieve the querystring parameters and generate the HTTP response.

The **`runGeoQuery()`** function operates by retrieving the **`lng`** and **`lat`** parameters and constructing an object of type '**`Point`**', with the parameters making up a **`coordinates`** array. Next, we set up a **`geoOptions`** object, specifying that the coordinate data represents points on the surface of a sphere (**`spherical : true`**), that the maximum distance we want to consider (in this case) is 10000 metres (10 km) and that the maximum number of results to be returned should be 5. The latter two parameters can be changed to suit your own applications.

Once these setup stages are complete, we call the **`geoNear()`** method on the **`Business`** model, passing the **`point`** and **`geoOptions`** as parameters, as well as a callback function, which returns an error object (when required), the **`results`** object and a **`stats`** object. In this implementation, we output the **`stats`** object to the Console and return the **`results`** object as the body of the successful HTTP response.

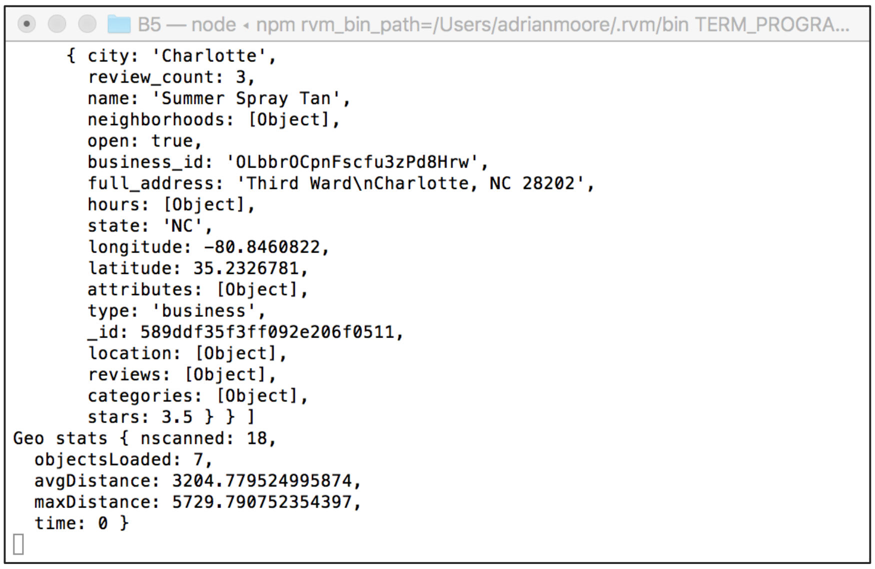
This function is illustrated by the code box below.

File*: B5/api/controllers/businesses.controllers.js*

```
var runGeoQuery = function(req, res) {
    var lng = parseFloat(req.query.lng);
    var lat = parseFloat(req.query.lat);

    var point = {
        type : "Point",
        coordinates : [ lng, lat ]
    };

    var geoOptions = {
        spherical : true,
        maxDistance : 10000,
        num : 5
    }

    Business
        .geoNear(point, geoOptions,
          function(err, results, stats) {
            console.log("Geo stats", stats);
            res
                .status(200)
                .json(results);
        });
}
```

Re-starting the application and passing a URL in the form http://localhost:3000/api/businesses?lng=121.121&lat=-3.1232 to the browser will result in information such as that illustrated in Figure B5.3 (overleaf) displayed in the Console.

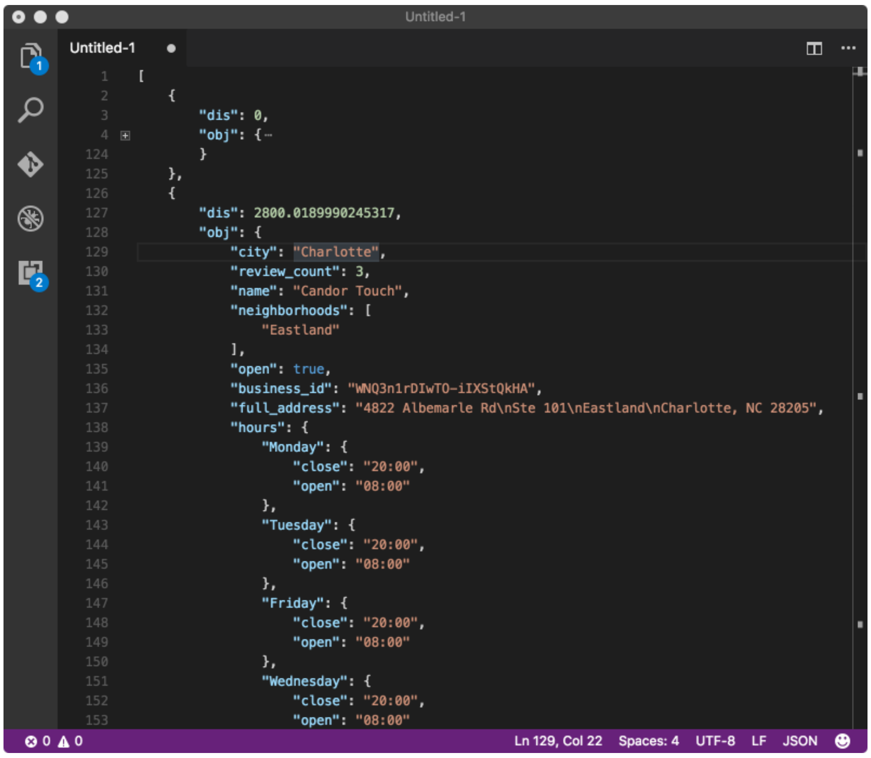
Note: The best way to test the application is to obtain the actual coordinate data from one of the businesses and to use that data in the URL. In this way, you should be guaranteed to get at least one result.


B5 — node ◂ npm rvm_bin_path=/Users/adrianmoore/.rvm/bin TERM_PROGRA...

```
      { city: 'Charlotte',
        review_count: 3,
        name: 'Summer Spray Tan',
        neighborhoods: [Object],
        open: true,
        business_id: 'OLbbrOCpnFscfu3zPd8Hrw',
        full_address: 'Third Ward\nCharlotte, NC 28202',
        hours: [Object],
        state: 'NC',
        longitude: -80.8460822,
        latitude: 35.2326781,
        attributes: [Object],
        type: 'business',
        _id: 589ddf35f3ff092e206f0511,
        location: [Object],
        reviews: [Object],
        categories: [Object],
        stars: 3.5 } } ]
Geo stats { nscanned: 18,
  objectsLoaded: 7,
  avgDistance: 3204.779524995874,
  maxDistance: 5729.790752354397,
  time: 0 }
```

Figure B5.3 Data returned from the geoNear() query

If you examine the data returned in the browser (copy the data and paste into a text editor to see a formatted version such as that shown in Figure B5.4 below), you can see that the **`result`** object consists of an array where each element has two properties – the distance (metres) from the coordinates provided and the original document that matched the query.



```
[
    {
        "dis": 0,
        "obj": {⋯
        }
    },
    {
        "dis": 2800.0189990245317,
        "obj": {
            "city": "Charlotte",
            "review_count": 3,
            "name": "Candor Touch",
            "neighborhoods": [
                "Eastland"
            ],
            "open": true,
            "business_id": "WNQ3n1rDIwTO-iIXStQkHA",
            "full_address": "4822 Albemarle Rd\nSte 101\nEastland\nCharlotte, NC 28205",
            "hours": {
                "Monday": {
                    "close": "20:00",
                    "open": "08:00"
                },
                "Tuesday": {
                    "close": "20:00",
                    "open": "08:00"
                },
                "Friday": {
                    "close": "20:00",
                    "open": "08:00"
                },
                "Wednesday": {
                    "close": "20:00",
                    "open": "08:00"
```

Figure B5.4 Formatted JSON data returned from geoNear()

Try it now!

All of this data (in the **`result`** and **`stats`** objects) can be parsed and processed by your application to provide a much richer response that we demonstrate here.

Try modifying the application so that **ONLY** the distance, name and full_address information for each business is returned, together with the figure for average distance generated by the **`stats`** object.

B5.4 Error trapping

Our application so far works as intended as long as the URLs provided match up with what is expected by the combination of router and controllers. However, we have yet to undertake any significant error checking and as the application grows in size, it is important now to consider this.

B5.4.1 The golden rules of API design

Successful error trapping can be best achieved by following the three golden rules of API design as follows;

I) Always return a response. The browser should never be left "hanging" because the server cannot process the request>

II) Return the correct HTTP status code.

III) Always return either contents or a message. An empty response (especially when a request has been declined) gives no information to the browser (or user of the API) as to the nature of the problem.

We will illustrate these by highlighting (and implementing remedies for) some of the current flaws in our application as it currently stands.

B5.4.2 Querystring errors

First, consider the **`businessesGetAll()`** controller – we have implemented functionality here that allows the user to provide values for parameters **`start`** and **`number`**, to specify a portion of the data to be returned, but our code assumes that these values are presented as numeric quantities.

Try presenting the URL http://localhost:3000/api/businesses?number=one and you will find that a collection of businesses are returned – but not with the size that we attempted to specify. We can protect against this by implementing type checking on the parameters by adding code such as the following to **businessesGetAll()**.

File*: B5/api/controllers/businesses.controllers.js*

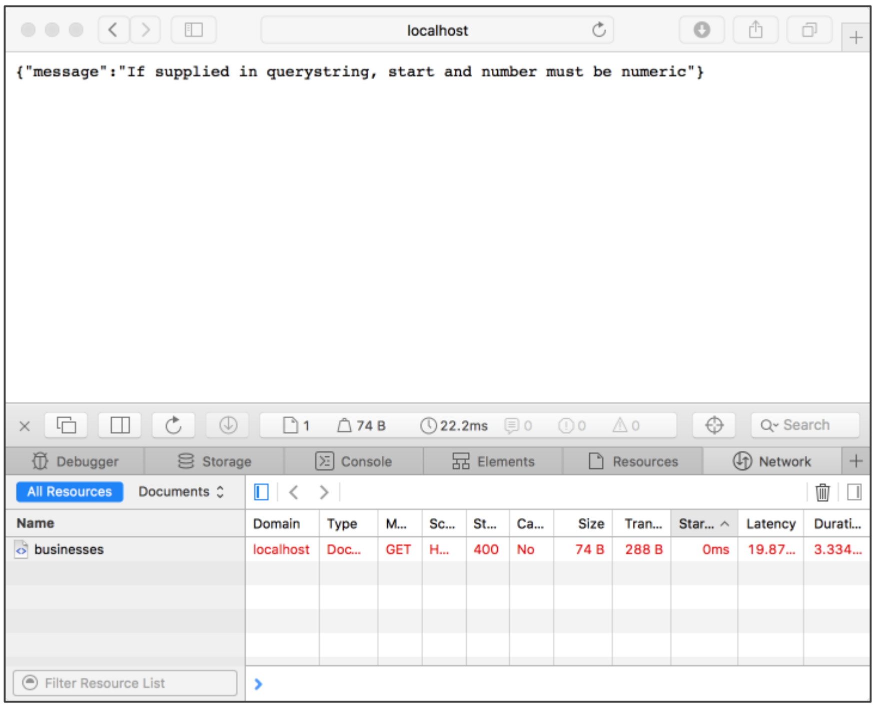
```
...
if (req.query && req.query.number) {
    number = parseInt(req.query.number);
}

if (isNaN(start) || isNaN(number)) {
    res
        .status(400)
        .json({"message":
                "If supplied in querystring, start
                 and number must be numeric"});
    return;
}
...
```

Now restart the application and try the same URL again – this time you should see the JSON message and a return code of 400 as illustrated in Figure B5.5 below.

Next, we might want to implement range checking to make sure the values provided for `start` and `number` are not out of a sensible range. For example, we may later design our interface so that only a maximum of 10 businesses at a time will be displayed. We should therefore protect against an application sending a value that exceeds this. Again, we can do this by providing additional code in the controller to compare against our upper limit.

File*: B5/api/controllers/businesses.controllers.js*

```
var maxNumber = 10;
...

if (number > maxNumber) {
    res
        .status(400)
        .json({"message":
            "Max value for number is " + maxNumber});
    return;
}
...
```



```
{"message":"If supplied in querystring, start and number must be numeric"}
```

Name	Domain	Type	M...	Sc...	St...	Ca...	Size	Tran...	Star...	Latency	Durati...
businesses	localhost	Doc...	GET	H...	400	No	74 B	288 B	0ms	19.87...	3.334...

Figure B5.5 Data type checking

B5.4.3 Database errors

All of our database requests so far have assumed that any request that we send to the database will result in a successful outcome. However, we have seen that the Mongoose **`exec()`** function that executes the database query returns an error object that will be populated if the query fails. To provide a reliable API, we need to test for these error objects and return informative messages and return codes when they are detected.

The code box below demonstrates this for the **`businessesGetAll()`** controller. Here, we insert new code into the **`exec()`** callback function to test for the presence of the **`err`** object and, when it is found, to issue a return code 500 (**Internal server error**) and to send the error information to the browser as the body of the response. Where the **`err`** object is not present, we return the data with a 200 **OK** code as previously.

File: ***B5/api/controllers/businesses.controllers.js***

```
module.exports.businessesGetAll = function(req, res) {

...

    Business
        .find()
        .skip(start)
        .limit(number)
        .exec(function(err, docs) {
            if (err) {
                console.log("Error finding businesses");
                res
                    .status(500)
                    .json(err)
            } else {
                console.log("Retrieved data for " +
                           docs.length + " businesses");
                res
                    .status(200)
                    .json(docs);
            }
        });

...
```

B5.4.4 Many error states

Occasionally, failure of a request may have multiple possible causes – and in these cases, we should be careful to send the appropriate response for each possibility. Consider the **`businessesGetOne()`** controller which responds to a request for information about an individual business, identified by its **`_id`** value.

Here (as in the previous example), we may have a database error that prevents the request from being satisfied, so a 500 **Internal Server Error** code should be returned. However, it may simply be that the **`_id`** provided did not correspond to one in the collection. In this case, the database has correctly responded with an empty result, but there has been no error and the correct response would be 404 (File not found).

Where there are multiple possibilities, it is usually better to avoid nested **`if…else`** statements, each with its own exit point; but to set default values for the response code and response body and to use the error trapping to change these as appropriate.

This is illustrated in the following code box, which sets a default response code of 200 and a default response body of the document returned by the query – and then tests the two error possibilities for a problem.

First, if the error object is populated, the response is updated so that the code is 500 and the body is the error object generated. Next, if there is no error, but also no response document, the code is set to 404 and an appropriate JSON message is provided as the body.

Finally, the ultimate values of the response object are returned to the browser.

File*: B5/api/controllers/businesses.controllers.js*

```
module.exports.businessesGetOne = function(req, res) {
    var businessID = req.params.businessID;
    console.log("GET business " + businessID);
    Business
        .findById(businessID)
        .exec(function(err, doc) {
            var response = {
                status : 200,
                message : doc
            }
            if (err) {
                response.status = 500;
                response.message = err
            } else  if (!doc) {
                response.status = 404;
                response.message = { “message”:
                           "Business ID not found" };
            }
            res
                .status(response.status)
                .json(response.message);
         });
}
```

Try it now!

Add additional error trapping for the remaining API endpoints dealing with **geo-location** and **reviews** by using similar techniques to those covered in this section.