

COM644 Full-Stack Web and App Development

Practical B3: MongoDB and Node

Aims

- To introduce the MongoDB driver for Node applications
- To illustrate creation of a reusable database connection
- To demonstrate how to access the database connection from Express controllers
- To demonstrate the retrieval of multiple documents from a collection
- To restrict retrieval to a subset of documents
- To return a single document identified by its `_id` value
- To demonstrate the `insertOne()` method for adding documents to a collection

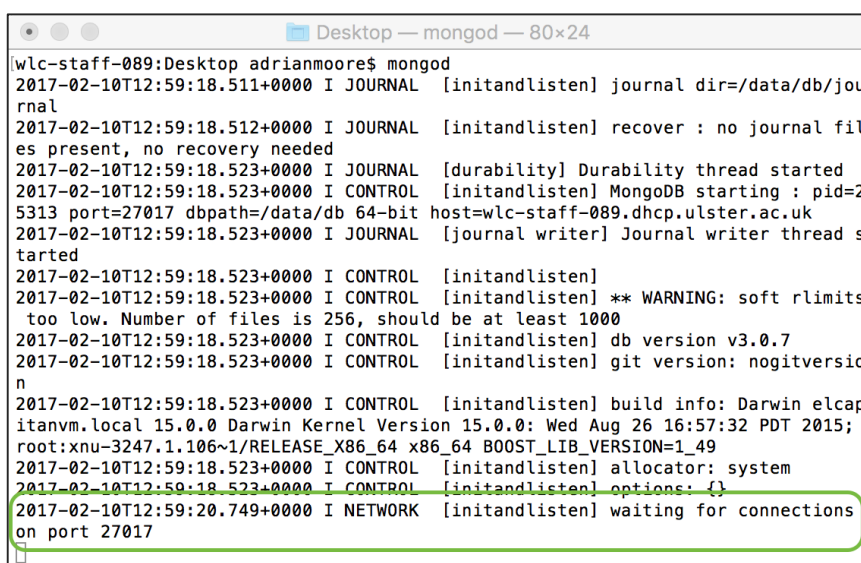
Contents

B3.1 MAKING A DATABASE CONNECTION	2
B3.1.1 INSTALLING THE MONGODB DRIVER	2
B3.1.2 CREATING A REUSABLE CONNECTION	3
B3.1.3 ACCESSING THE CONNECTION FROM CONTROLLERS.....	6
B3.2 QUERYING THE DATABASE FROM NODE.JS	7
B3.2.1 RETURNING MULTIPLE DOCUMENTS FROM A COLLECTION	7
B3.2.2 RETURNING A SUBSET OF THE DATA	9
B3.2.2 RETURNING A SINGLE DOCUMENT BY ID	10
B3.3 ADDING DOCUMENTS TO A COLLECTION	11
B3.3.1 VALIDATING THE DATA.....	11
B3.3.2 CHECKING THE DATA FORMAT	12
B3.3.3 INSERTING THE DOCUMENT	12

B3.1 Making a database connection

In this section, we will revisit the application that we began to build in Section A and investigate how to connect it to the MongoDB database, instead of the static JSON file. The starting point for this section is the state of the application at the end of **Practical A6** and the database established in **Weekly Challenge 4**, so you should make sure that you have completed everything up to this point before you begin.

When we run the MongoDB server by the **mongod** command, we can see, as illustrated in Figure B3.1 below, that the server listens for requests on port 27017. We will add code to our application to generate a connection to the database on this port, issue requests and process the data returned.



```
wlc-staff-089:Desktop adrianmoore$ mongod
2017-02-10T12:59:18.511+0000 I JOURNAL [initandlisten] journal dir=/data/db/journal
2017-02-10T12:59:18.512+0000 I JOURNAL [initandlisten] recover : no journal files present, no recovery needed
2017-02-10T12:59:18.523+0000 I JOURNAL [durability] Durability thread started
2017-02-10T12:59:18.523+0000 I CONTROL [initandlisten] MongoDB starting : pid=25313 port=27017 dbpath=/data/db 64-bit host=wlc-staff-089.dhcp.ulster.ac.uk
2017-02-10T12:59:18.523+0000 I JOURNAL [journal writer] Journal writer thread started
2017-02-10T12:59:18.523+0000 I CONTROL [initandlisten]
2017-02-10T12:59:18.523+0000 I CONTROL [initandlisten] ** WARNING: soft rlimits too low. Number of files is 256, should be at least 1000
2017-02-10T12:59:18.523+0000 I CONTROL [initandlisten] db version v3.0.7
2017-02-10T12:59:18.523+0000 I CONTROL [initandlisten] git version: nogitversion
2017-02-10T12:59:18.523+0000 I CONTROL [initandlisten] build info: Darwin elcapitanvm.local 15.0.0 Darwin Kernel Version 15.0.0: Wed Aug 26 16:57:32 PDT 2015; root:xnu-3247.1.106~1/RELEASE_X86_64 x86_64 BOOST_LIB_VERSION=1_49
2017-02-10T12:59:18.523+0000 I CONTROL [initandlisten] allocator: system
2017-02-10T12:59:18.523+0000 I CONTROL [initandlisten] options: {}
2017-02-10T12:59:20.749+0000 I NETWORK [initandlisten] waiting for connections on port 27017
```

Figure B3.1 MongoDB server listening on port 27017

B3.1.1 Installing the MongoDB driver

First, we need to install the native MongoDB driver into our application by the command

```
U:\> npm install mongodbc --save
```


File: B3/api/data/dbConnect.js

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

var connection = null;

module.exports.open = function() {
  MongoClient.connect(dbURL, function(err, db) {
    if (err) {
      console.log("DB connection failed");
      return;
    }
    connection = db;
    console.log("DB connection open");
  })
}

module.exports.get = function() {
  return connection;
}
```

To create the connection, we use the MongoClient method **connect()** which requires two parameters – the connection string defined earlier and a callback function that will run when the **connect()** operation is complete. The callback function accepts two parameters – an error object (**err**) that will be populated if the connection request fails and a connection object (**db**) that will be populated if the request succeeds. The body of the callback function then checks for the presence of the error object, generating an appropriate Console message if it exists – otherwise the local **connection** variable is set with the new connection object.

This **connection** variable is then simply returned by the **get()** method when a request for the connection is made.

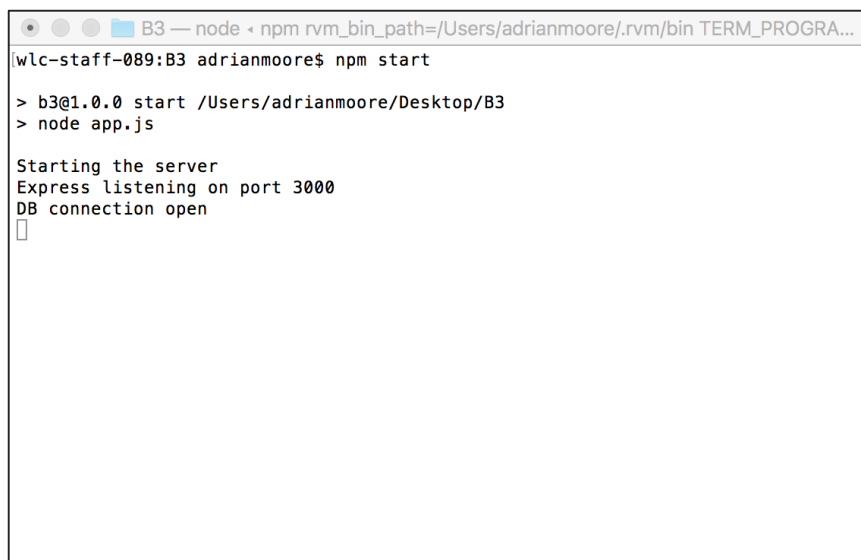
Now, we will use this connection management module to create a new database connection as soon as the application begins.

This can be easily achieved by adding a line at the top of app.js to **require** the connection manager module and immediately invoke the **open()** method.

File: B3/app.js

```
require('./api/data/dbConnect.js').open();
var express = require('express');
var app = express();
var path = require('path');
var bodyParser = require('body-parser');
...
```

Now, when we start the application server, we can see the Console message confirming that the database connection has been successfully opened.



```
B3 — node · npm rvm_bin_path=/Users/adrianmoore/.rvm/bin TERM_PROGRA...
[wlc-staff-089:B3 adrianmoore$ npm start
> b3@1.0.0 start /Users/adrianmoore/Desktop/B3
> node app.js

Starting the server
Express listening on port 3000
DB connection open
█
```

Figure B3.3 Database connection established

Try it now!

Verify the error trapping in the callback function in the dbConnect **open ()** method by changing the port number in the database connection string. When you restart the server, you should find that the connection request fails and that you see the “DB connection failed” message in the Console window.

Remember to restore the original port number before continuing!

B3.1.3 Accessing the connection from controllers

Now that we have a database connection available, we want to update our controllers so that they are able to make use of it.

First we require the connection management package into the controllers file and then add code into the **businessesGetAll** controller to retrieve the currently open connection (opened by **app.js** when the application was started).

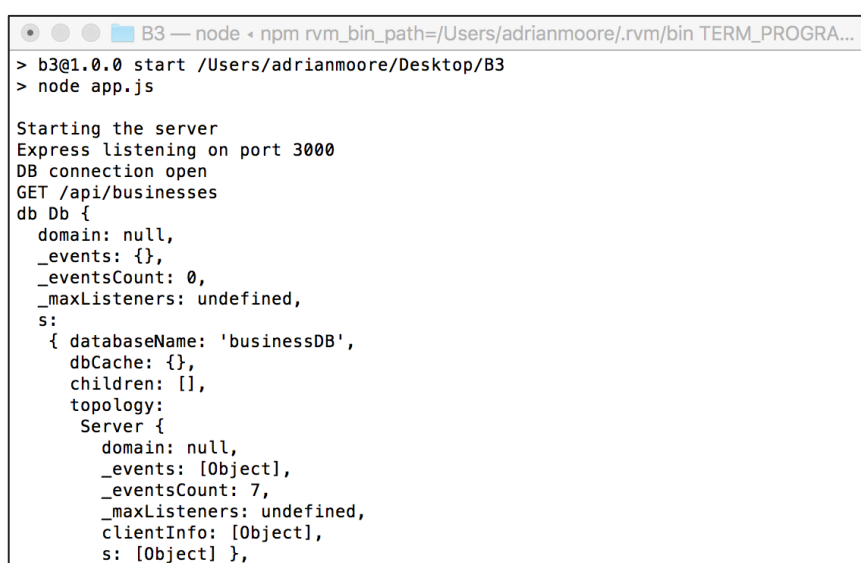
File: B3/app/controllers/businesses.controllers.js

```
var dbConnect = require('../data/dbConnect.js');
var businesses_data = require('../data/data_100.json');

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

    var db = dbConnect.get();
    console.log("db", db);
    ...
}
```

Now, load <http://localhost:3000/api/businesses> into a web browser. The browser will display the same data as previously (as the controller is still fetching the data from the static JSON file), but you will see new information in the Console window where we have output the contents of the database connection object. This is shown in Figure B3.4 below.



```
B3 — node · npm rvm_bin_path=/Users/adrianmoore/.rvm/bin TERM_PROGRA...
> b3@1.0.0 start /Users/adrianmoore/Desktop/B3
> node app.js

Starting the server
Express listening on port 3000
DB connection open
GET /api/businesses
db Db {
  domain: null,
  _events: {},
  _eventsCount: 0,
  _maxListeners: undefined,
  s:
    { databaseName: 'businessDB',
      dbCache: {},
      children: [],
      topology:
        Server {
          domain: null,
          _events: [Object],
          _eventsCount: 7,
          _maxListeners: undefined,
          clientInfo: [Object],
          s: [Object] },
    }
}
```

Figure B3.4 Controller fetches database connection

Now that we are confident the connection is being retrieved in the **businessesGetAll** controller, we will do the same for the other controllers in the file (at the same time removing the **console.log()** that displays the connection information).

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

```
module.exports.businessesGetAll = function(req, res) {  
  var db = dbConnect.get();  
  ...  
  
module.exports.businessesGetOne = function(req, res) {  
  var db = dbConnect.get();  
  ...  
  
module.exports.businessesAddOne = function(req, res) {  
  var db = dbConnect.get();  
  ...
```

B3.2 Querying the database from Node.js

Now that a reusable database connection has been created, we can use it to update our existing controllers so that they operate on the live database rather than the static file **data_100.json**

B3.2.1 Returning multiple documents from a collection

We have previously seen how the **find()** method can be used in the MongoDB shell to return documents from a collection. In Javascript we have a similar technique – except that we need to consider two additional things

- i) The **find()** method does not return documents, but a cursor that can be used to retrieve documents
- ii) The **find()** method is synchronous – i.e. it is blocking in nature.

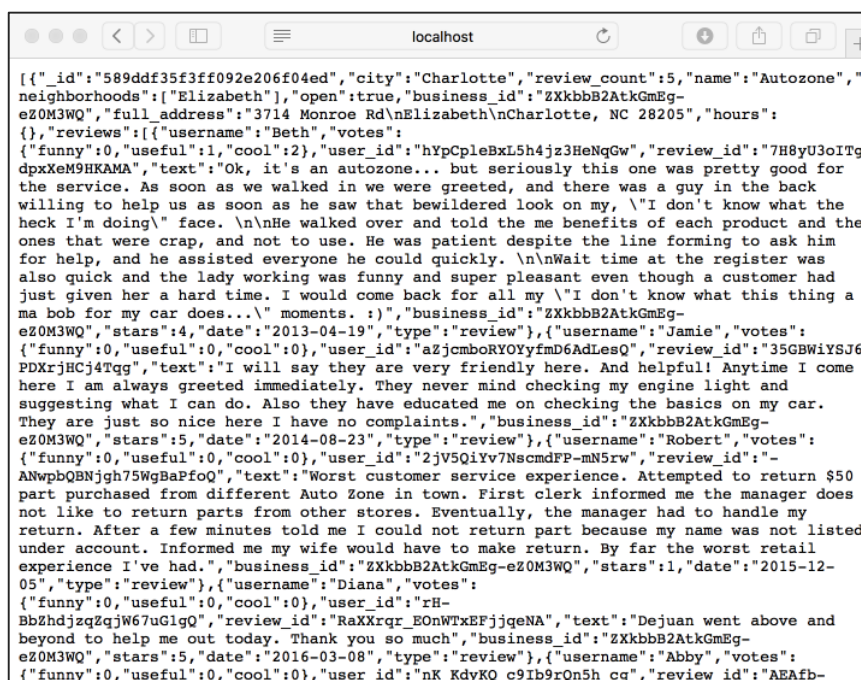
Fortunately, there is a method **toArray()** that solves both these problems. **toArray()** will retrieve data from a cursor and return it in array form, and it is also asynchronous – i.e. it accepts a callback function that will be called when the **toArray()** operation is complete.

The following code box illustrates how we can modify the **businessesGetAll** controller to retrieve data from the database rather than the static JSON file.

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

```
module.exports.businessesGetAll = function(req, res) {
  var db = dbConnect.get();
  var collection = db.collection('business');
  collection
    .find()
    .toArray(function(err, docs) {
      console.log("Retrieved the businesses");
      res
        .status(200)
        .json(docs);
    })
};
```

Here, we retrieve the database connection and use the **collection()** method on the connection to return a pointer to the business collection that contains our data. Next, we chain the **find()** method and the **toArray()** method to the collection, providing to **toArray()** a callback function that returns the HTTP response object with a status code of 200 and the data retrieved from the database. Figure B3.5 illustrates the data retrieved from the database and displayed in the browser.



The screenshot shows a web browser window with the URL localhost. The page content is a JSON array of business data. The first object represents an 'Autozone' business in Charlotte, NC, with a review count of 5. It includes details about its address, hours, and a list of reviews. The reviews include usernames like 'Beth', 'Jamie', 'Robert', and 'Diana', along with their respective ratings (stars) and dates. The text of the reviews describes customer experiences, such as a positive one from Beth and a negative one from Robert regarding a return policy.

Figure B3.5 Retrieving data from the database

B3.2.2 Returning a subset of the data

In the previous version of `businessesGetAll`, we had a pair of optional querystring parameters (`start` and `number`) that enabled us to control which portion of the collection was returned. Now that we have connected the database to the controller, we want to re-instate that additional functionality.

The purpose of the variables was to specify how many values to return and the starting point in the collection from which we want to extract results. The MongoDB driver has a method that handles each of these as follows

- `skip()` takes an integer parameter that specifies how many documents to skip over before returning results
- `limit()` takes an integer parameter that specifies the maximum number of documents to be returned

These methods can be simply chained to the `find()` method as shown in the following code box, which also re-instates the previous code to retrieve the optional parameters from the querystring.

File: `B3/app/controllers/businesses.controllers.js`

```
module.exports.businessesGetAll = function(req, res) {
  var db = dbConnect.get();
  var collection = db.collection('business');

  var start = 0;
  var number = 0;

  if (req.query && req.query.start) {
    start = parseInt(req.query.start);
  }
  if (req.query && req.query.number) {
    number = parseInt(req.query.number);
  }
  collection
    .find()
    .skip(start)
    .limit(number)
    .toArray(function(err, docs) {
      console.log("Retrieved the businesses");
      res
        .status(200)
        .json(docs);
    })
};
```

B3.2.2 Returning a single document by ID

Searching for a single document could be achieved by passing a query object to the `find()` method and using the same technique as above, but the MondoDB driver provides a separate method which is a more convenient way of returning a single document.

The method `findOne()` is presented in the code box below. It accepts two parameters, a query object specifying the search term and a callback function with the usual error and document objects.

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

```
module.exports.businessesGetOne = function(req, res) {
  var db = dbConnect.get();
  var collection = db.collection('business');
  var businessID = req.params.businessID;
  console.log("GET business " + businessID);
  collection
    .findOne( {_id : ObjectId(businessID)},
      function(err, doc) {
        res
          .status(200)
          .json(doc);
      })
};
```

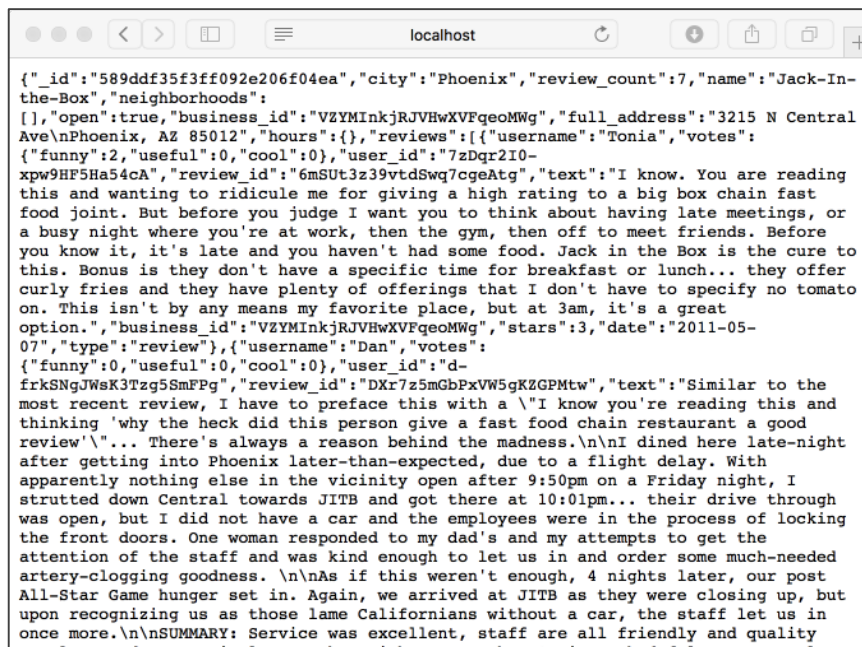
Note the format of the query object passed to the `findOne()` method. This uses the MongoDB driver method `ObjectId()` that allows us to treat MongoDB `_id` values as valid JavaScript. In order to use this method, we need to **require** it, so we need to add the following highlighted line to the top of our controllers file.

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

```
var dbConnect = require('../data/dbConnect.js');
var ObjectId = require('mongodb').ObjectId;
//var businesses_data = require('../data/data_100.json');
```

Note that we can also now remove the reference to the `data_100.json` file as all of our data retrieval is being satisfied by the database.

We can verify the operation of this controller by presenting a single `_id` to the router as illustrated in Figure B3.6 which demonstrates the URL <http://localhost:3000/api/businesses/589ddf35f3ff092e206f04ea>



```
{ "_id": "589ddf35f3ff092e206f04ea", "city": "Phoenix", "review_count": 7, "name": "Jack-In-the-Box", "neighborhoods": [], "open": true, "business_id": "VZYMIInkjRJVHwXVFqeoMWg", "full_address": "3215 N Central Ave\nPhoenix, AZ 85012", "hours": {}, "reviews": [{"username": "Tonia", "votes": {"funny": 2, "useful": 0, "cool": 0}, "user_id": "7zDqr2I0-xpw9HF5Ha54cA", "review_id": "6mSUt3z39vtdSwq7cgeAtg", "text": "I know. You are reading this and wanting to ridicule me for giving a high rating to a big box chain fast food joint. But before you judge I want you to think about having late meetings, or a busy night where you're at work, then the gym, then off to meet friends. Before you know it, it's late and you haven't had some food. Jack in the Box is the cure to this. Bonus is they don't have a specific time for breakfast or lunch... they offer curly fries and they have plenty of offerings that I don't have to specify no tomato on. This isn't by any means my favorite place, but at 3am, it's a great option.", "business_id": "VZYMIInkjRJVHwXVFqeoMWg", "stars": 3, "date": "2011-05-07", "type": "review"}, {"username": "Dan", "votes": {"funny": 0, "useful": 0, "cool": 0}, "user_id": "d-frkSNqJWsK3Tzg5SmFPg", "review_id": "DXr7z5mGbPxVW5gKZGPMtw", "text": "Similar to the most recent review, I have to preface this with a \"I know you're reading this and thinking 'why the heck did this person give a fast food chain restaurant a good review'... There's always a reason behind the madness.\" I dined here late-night after getting into Phoenix later-than-expected, due to a flight delay. With apparently nothing else in the vicinity open after 9:50pm on a Friday night, I strutted down Central towards JITB and got there at 10:01pm... their drive through was open, but I did not have a car and the employees were in the process of locking the front doors. One woman responded to my dad's and my attempts to get the attention of the staff and was kind enough to let us in and order some much-needed artery-clogging goodness. \"As if this weren't enough, 4 nights later, our post All-Star Game hunger set in. Again, we arrived at JITB as they were closing up, but upon recognizing us as those lame Californians without a car, the staff let us in once more.\" SUMMARY: Service was excellent, staff are all friendly and quality
```

Figure B3.6 Retrieving a single document by ID

B3.3 Adding documents to a collection

Adding a new document to a MongoDB collection is a three-stage process, requiring validation, data formatting and document insertion.

B3.3.1 Validating the data

Initially, we need to validate that all of the data values that are required are present. Remember that JSON does NOT require that all documents have the same structure, so MongoDB will accept an incomplete document even if it is not valid in the context of your application.

In our example, we will assume that the only compulsory fields are “name” and “stars” and so we will check that the **body** element of the HTTP request exists and that it contains **.name** and **.stars** items.

```
if (req.body && req.body.name && req.body.stars) {
```

B3.3.2 Checking the data format

Next, we will ensure that the data is formatted as we would like. The *name* of a business is a simple text value, but if you examine the *stars* element of any document in the database you will find that it is stored as an integer. Again, MongoDB will happily accept non-integer values for the stars item, but so that we can use the values properly in comparisons, we will use `parseInt()` to ensure that the value we present to the database is an integer.

```
var newBusiness = req.body;  
newBusiness.stars = parseInt(req.body.stars);
```

B3.3.3 Inserting the document

Finally, we can add the new document to the collection by using the MongoDB `insertOne()` method. This takes two parameters – the new document to be added as a JSON object, and a callback function that runs once the insert has been completed. The callback function takes the usual two parameters representing an error object if the request is not completed and a response object returned from the MongoDB server.

```
collection  
  .insertOne(newBusiness, function(err, response){  
    ...  
  });
```

The following code box presents the full `businessesAddOne` controller. Note that we have included error trapping on the data validation test to return an error message to the console and an HTTP 400 “Bad request” code if all of the required data is not present. Note also how the `response` object returned by the `insertOne()` callback function (not to be confused with the HTTP response object in the controller parameter `res`) contains a large collection of information, but we can obtain the document that has been added by logging its `ops` property to the console.

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

```
module.exports.businessesAddOne = function(req, res) {
  var db = dbConnect.get();
  var collection = db.collection('business');
  console.log("POST new business");

  if (req.body && req.body.name && req.body.stars) {
    var newBusiness = req.body;
    newBusiness.stars = parseInt(req.body.stars);
    collection
      .insertOne(newBusiness, function(err, response){
        console.log(response)
        res
          .status(201)
          .json(response.ops)
      });
  } else {
    console.log("Data missing from body");
    res
      .status(400)
      .json( {message : "Required data missing"} );
  }
};
```

As usual, we can test the operation of the POST request by using the Postman app. Figure B3.7 below illustrates a POST request made to <http://localhost:3000/api/businesses/new> with data values *name* and *stars* provided with sample values. See in the response to the request that the value for *stars* is confirmed as an integer and that MongoDB has generated an *_id* value for the new document.

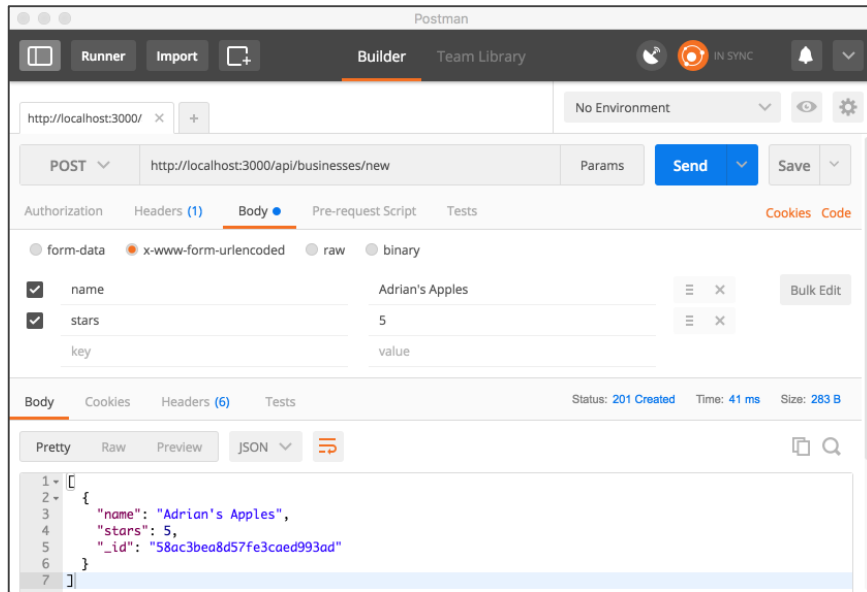


Figure B3.7 Adding a new business

We can also confirm the operation of the new controller by observing the output generated in the Console window and shown in Figure B3.8.

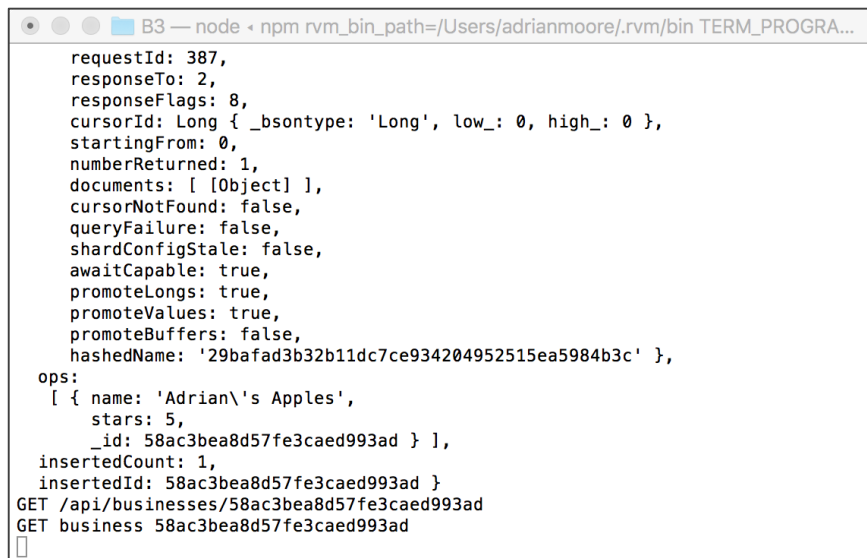
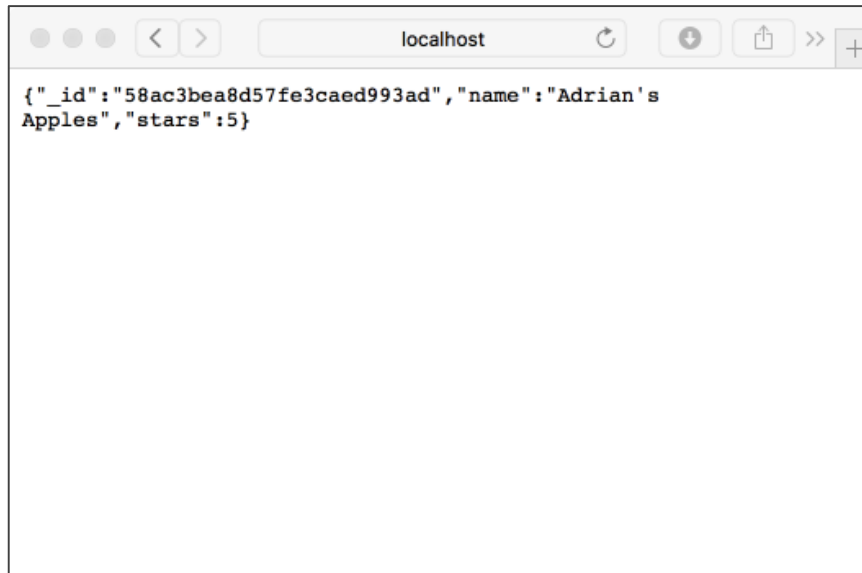


Figure B3.8 New document added

Finally, we can verify that the new document has been added by retrieving it using the **businessesGetOne** method and providing the newly created **_id** in the URL. Figure B3.9 illustrates the the page returned by the URL

<http://localhost:3000/api/businesses/58ac3bea8d57fe3caed993ad>. (Remember that your new `_id` value will be different)

A screenshot of a web browser window. The address bar shows 'localhost'. The main content area displays a JSON document:

```
{ "_id": "58ac3bea8d57fe3caed993ad", "name": "Adrian's Apples", "stars": 5 }
```

B3.9 Retrieving the new document