COM644 Full-Stack Web and App Development

Practical B6: Mongoose and Information Update

Aims

- To demonstrate the provision of multiple endpoints in a single API route
- To implement Mongoose controller logic to handle a POST request to add documents to a MongoDB database
- To demonstrate the addition of sub-documents to a collection
- To implement Mongoose controller logic to handle a PUT request to update a document in a MongoDB database
- To demonstrate the update of sub-documents in a collection
- To implement Mongoose controller logic to handle a DELETE request to remove a document from a MongoDB database
- To demonstrate the deletion of sub-documents from a collection

Contents

B6.1 CREATING DATABASE CONTENT 2
B6.1.1 CREATING A NEW DOCUMENT 2
B6.1.2 CREATING A NEW SUB-DOCUMENT 6
B6.2 UPDATING DATABASE CONTENT 12
B6.2.1 UPDATING AN EXISTING DOCUMENT 12
B6.2.2 UPDATING AN EXISTING SUB-DOCUMENT 17
B6.3 DELETING DATABASE CONTENT 20
B6.3.1 DELETING A TOP-LEVEL DOCUMENT 21
B6.3.2 DELETING A SUB-DOCUMENT 22

B6.1 Creating database content

In the previous practical we examined the use of Mongoose in implementing controllers to retrieve documents and sub-documents from MongoDB databases. In this session, we will complete the development of our RESTful API for the ***WeMeanBusiness*** sample application by providing routes and controllers to update database contents.

B6.1.1 Creating a new document

According to our data model specified in Practical B4, a document describing a business has the following structure:

Field	Type (and notes)
name	String (required)
stars	Number (range 0-5, default value 0)
city	String
review_count	Number
categories	Array of Strings
reviews	Array of Reviews
location	Object, comprising a String address and an Array of Number coordinates in the form [lng, lat]

In addition, we have separate definitions for **Reviews** (a sub-document of the **Business** type) and **Votes** (a sub-document of **Reviews**)

We had previously implemented a controller to add a new business (**`businessesAddOne()`**)using the native MongoDB driver, but we now need to update this to use the Mongoose connection. In addition, we will change our previous route defined as **POST /api/businesses/new** to reflect the previous study of URL architectures for RESTful APIs. The "correct" form for a route to add a new document to a collection is **POST /api/*collection_name***, so we will first update the route to reflect this.

Note that simply updating the path in the **`router`** definition will now leave two specifications for **`route('/businesses')`** – one to handle a **GET** request and invoke the controller **`businessesGetAll()`** and another to handle a **POST** request and call controller **`businessesAddOne()`**. This is fine, but it is a matter of better style to combine these into a single **`route()`** definition as seen in the code box below.

File*: B6/api/routes/index.js*

```
...

router
    .route('/businesses')
    .get (businessesController.businessesGetAll)
    .post(businessesController.businessesAddOne);

...
```

The Mongoose implementation of the **`businessesAddOne()`** controller is very straightforward and is shown in the code box below. All we need to do is call the **`create()`** method on the **`Business`** model, passing the new document to be added as a JSON object, along with a callback function that takes two parameters – an error object populated if the **`create()`** method fails, and an object representing the newly added document. The callback function checks for the **`err`** object, returning status code **400** if it is found, otherwise the new document is returned with code **201** for a successful **POST**.

File*: B6/api/controllers /businesses.controllers.js*

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

    Business
        .create({
            // new object to be added
            }
        }, function(err, newBusiness) {
            if (err) {
                console.log("Error creating business");
                res
                    .status(400)
                    .json(err);
            } else {
                res
                    .status(201)
                    .json(newBusiness);
            }
        });
}
```

To create the new object to be added we retrieve the **POST**ed values from the **`body`** element of the request object, formatting them where required. String values can be directly assigned, but those which are numeric need to be converted using **`parseInt()`** or **`parseFloat()`** as appropriate. Elements which will not yet have values, such as **`review_count`** and **`reviews`**, can be directly assigned defaults (which we could also have done in the schema definition).

The following code box demonstrates the specification of the new business object within the controller.

File*: B6/api/controllers /businesses.controllers.js*

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

    Business
        .create({
            name : req.body.name,
            stars : parseInt(req.body.stars),
            city : req.body.city,
            review_count : 0,
            categories : splitArray(req.body.categories),
            reviews : [],
            location : {
                address : req.body.address,
                coordinates : [
                    parseFloat(req.body.lng),
                    parseFloat(req.body.lat)
                ]
            }
    ...
```

Note also the use of the helper function **`splitArray()`**, which converts a string containing a list of categories separated by the **`;`** character into an array of separate strings. We could have used the native JavaScript **`split()`** method here, except that, when passed an empty string, it would return an array containing a single empty element. Our alternative implementation tests for the empty string, returning an empty array. The following code box presents the implementation of **`splitArray()`**.

File*: B6/api/controllers /businesses.controllers.js*

```
var splitArray = function(input) {
    var output;
    if (input && input.length > 0) {
        output = input.split(";");
    } else {
        output = [];
    }
    return output;
};
```

Now that the controller implementation is complete, we can start the application and use Postman to generate data to be sent as a POST request to http://localhost:3000/api/businesses as shown in Figure B6.1 below.

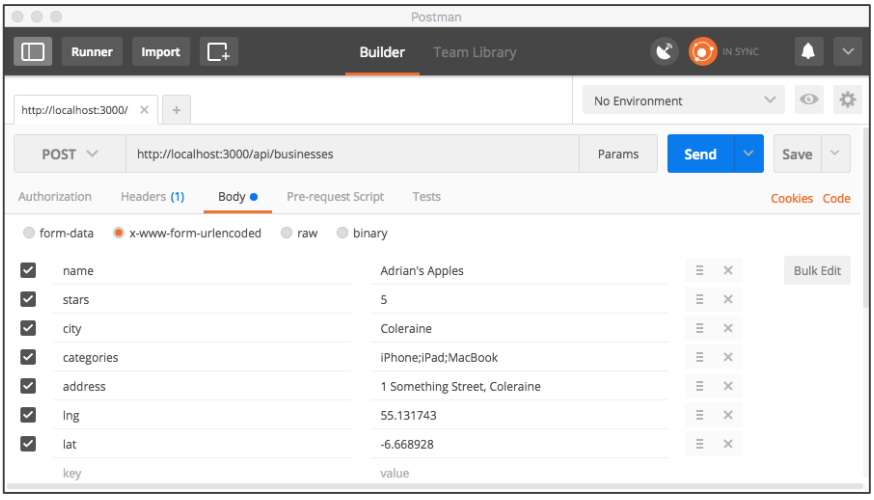

Figure B6.1 Creating a new business using Postman

When the request is submitted, the modified router invokes the new controller so that the new business document is added to the collection and returned as the JSON response to the browser. Figure B6.2 illustrates the data returned from the controller.

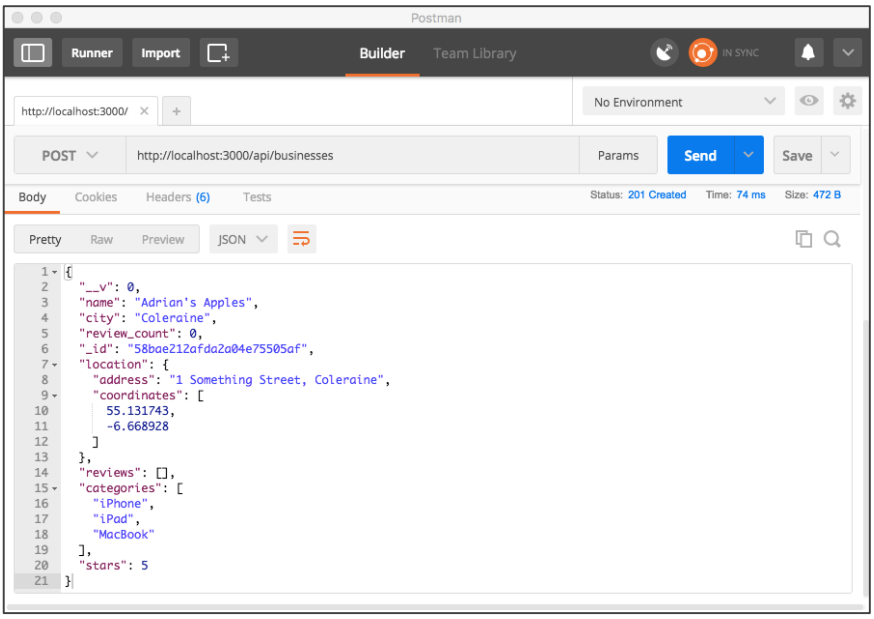

Figure B6.2 New business added

B6.1.2 Creating a new sub-document

Adding sub-documents is a slightly more complex operation than adding full documents and we will illustrate it by implementing a route and controller to add a review to a business' collection. In the interests of simplicity, we will also re-define the **review** schema created in Practical B4, so that fields we are not going to use in our application are no longer included. Our modified **`reviewSchema`** is presented in the following code box. You should make this change in your copy of the application.

File*: B6/api/data/businesses.model.js*

```
...

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

...
```

First, we create a route for the new functionality, by specifying a **POST** action on the existing **/businesses/:businessID/reviews** path.

File*: B6/api/routes /index.js*

```
...

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

...
```

Now, with the route in place, we can implement the new **`reviewsAddOne()`** controller to add a review.

In order to add a review as a new sub-document, we first need to retrieve the parent document so that we can fetch the current collection of reviews. This is already implemented as the controller **`reviewsGetAll()`**, so the starting point is to take all of the code from that controller and paste it into the body of **`reviewsAddOne()`**.

This is illustrated by the code box below – note that this includes the error trapping that you should have implemented from the **Try it now!** exercise in Practical B5.

This code performs error checking by testing for an error object and for an empty document. If neither of these are found, it either returns the array of reviews, or an empty array if the reviews element is not present.

File*: B6/api/controllers/reviews.controllers.js*

```
module.exports.reviewsAddOne = function(req, res) {
    var businessID = req.params.businessID;
    console.log("GET reviews for business " +
                businessID);
    Business
        .findById( businessID )
        .select( "reviews" )
        .exec(function(err, doc) {
            var response = {
                status : 200,
                message : []
            };
            if (err) {
                console.log("Error finding business");
                response.status = 500;
                response.message = err;
            } else if (!doc) {
                response.status = 404;
                response.message = {
                    "message" : "Business ID not found" +
                                businessID
                };
            } else {
                response.message = doc.reviews ?
                                   doc.reviews : []
            };
            res
                .status(response.status)
                .json(response.message);
        });
}
```

We will modify this by removing the highlighted code and instead calling a new helper function **`addReview()`** that accepts the request and response objects together with the retrieved document. This modification is illustrated in the following code box.

File*: B6/api/controllers/reviews.controllers.js*

```
module.exports.reviewsAddOne = function(req, res) {
    var businessID = req.params.businessID;
    console.log("GET reviews for business " +
                businessID);
    Business
        .findById( businessID )
        .select( "reviews" )
        .exec(function(err, doc) {
            var response = {
                status : 200,
                message : []
            };
            if (err) {
                console.log("Error finding business");
                response.status = 500;
                response.message = err;
            } else if (!doc) {
                response.status = 404;
                response.message = {
                    "message" : "Business ID not found" +
                                businessID
                };
            };
            if (doc) {
                addReview(req, res, doc);
            } else {
                res
                    .status(response.status)
                    .json(response.message);
            }
        });
}
```

The **`addReview()`** function has two tasks – first it creates the new review object by retrieving **POST**ed data from the **`body`** element of the request object (formatting it where required) and then **`push`**es the new review into the **`reviews`** array of the parent document.

Next, the **`save()`** method is applied to the new document to update the collection. This method takes a callback function which has parameters containing the error object generated if the save operation fails and a document representing the updated business. As we know that the **`push()`** operation will add the new review to the end of the array, we obtain the new review by retrieving the length of the **reviews** array and extracting the element in the last position. This new review object is then returned as the body of the response, with a return code of **201**.

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

File*: B6/api/controllers/reviews.controllers.js*

```
var addReview = function(req, res, thisBusiness) {
    thisBusiness.reviews.push( {
        username : req.body.username,
        votes : { "funny":0, "useful":0, "cool":0 },
        text : req.body.text,
        stars : parseInt(req.body.stars)
    });

    thisBusiness.save(function(err, updatedBusiness) {
        var newReviewPosition =
             updatedBusiness.reviews.length - 1;
        var newReview =
             updatedBusiness.reviews[newReviewPosition];
        if (err) {
            res
                .status(500)
                .json(err);
        } else {
            res
                .status(201)
                .json(newReview);
        };
    });
}
```

As usual, we can test the operation of the new **POST** route by providing the URL of the endpoint to Postman and providing some data representing a new review, as illustrated by Figure B6.3. Note that the only values we need to provide are those for *username*, *text* (the text of the review) and *stars* (the rating in the range 0-5). All other values are either generated by the **`addReview()`** function (*votes*) or will have default values automatically provided according to the schema definition (*date*).

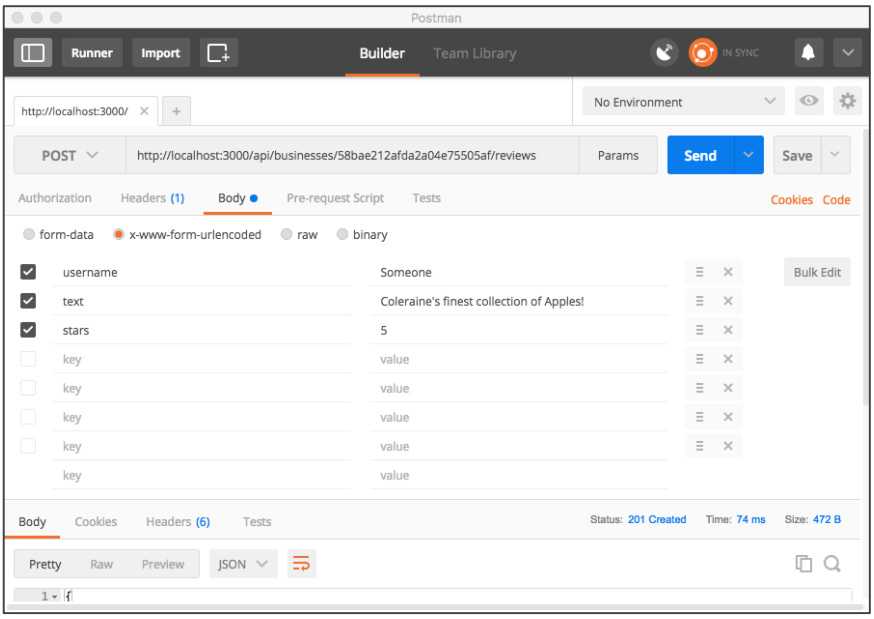

Figure B6.3 Adding a review with Postman

If the review has been added successfully, you should see the response object describing the new review displayed in the Body section of Postman, as shown in Figure B6.4 below.

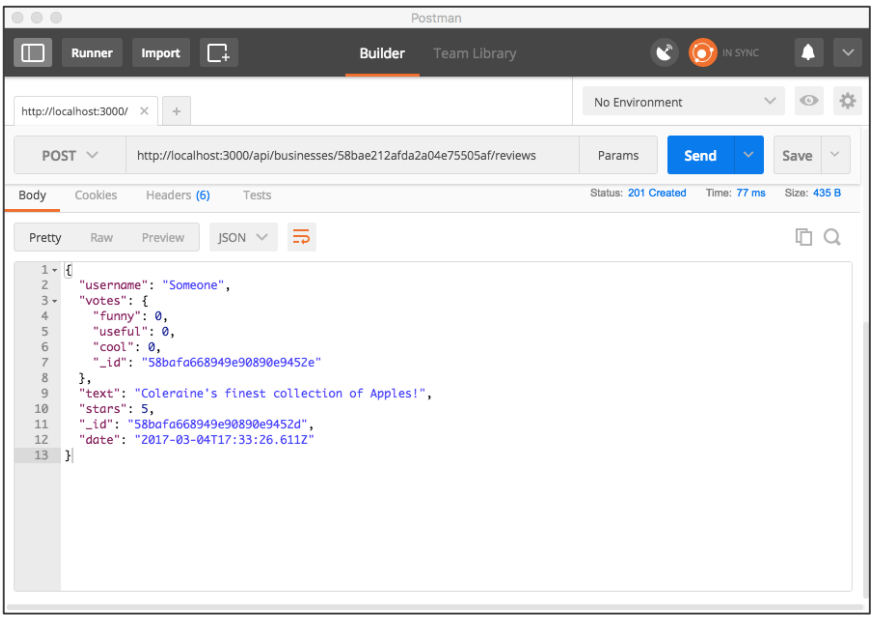

Figure B6.4 New review added

Try it now!

Verify that the new endpoints are working properly by running the application and

i) presenting the URL http://localhost:3000/api/businesses/12345 to a web browser (where 12345 is the **_id** of your newly-added business.

ii) show the reviews of that business by the URL
http://localhost:3000/api/businesses/12345/reviews

iii) copy the **_id** of a review object from the browser window and add it to the URL to give the URL http://localhost:3000/api/businesses/12345/reviews/54321 (where 54321 is the **_id** of the review element).

B6.2 Updating database content

So far in our sample application, we have only made use of the familiar **GET** and **POST** methods for retrieving and creating content. However, as explained in Practical B4, the HTTP specification provides additional methods to indicate that we want to modify or delete existing content. In this section, we will implement **PUT** requests in order to edit a previously created document.

B6.2.1 Updating an existing document

First, we need to create the routes for the new functionality by adding **PUT** specifications for the **/businesses/:businessID** endpoint to update details of a specific business and **/businesses/:businessID/reviews/:reviewID** to update details of a specific review. This is presented in the code box below.

File*: B6/api/routes /index.js*

```
...
router
    .route('/businesses/:businessID')
    .get (businessesController.businessesGetOne)
    .put(businessesController.businessesUpdateOne);
...

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

Developing an update controller is a four-stage process as follows

i) find the information about the specific document to create an instance of the model

ii) update the data in the instance of the model

iii) save the instance of the model to update the database

iv) return a response to the requester.

The first stage is to retrieve the data from a specific document representing a business, so we can simply copy and paste the code from the **`businessesGetOne()`** controller. To this code, we add an **`if`** statement around the final **`res`** command so that, if an error has occurred, we immediately return the response and abort the update operation. The following code box illustrates the development of the controller to this point.

File*: B6/api/controllers/businesses.controllers.js*

```
module.exports.businessesUpdateOne = 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) {
                console.log("Error finding business")
                response.status = 500;
                response.message = err;
            } else if (!doc) {
                response.status = 404;
                response.message = {
                   "message" : "Business ID not found"
                 };
            }
            console.log("Found business " + businessID);
            if (response.status != 200) {
                res
                    .status(response.status)
                    .json(response.message);
            };
        });
};
```

The second stage updates the data in the instance of the model, held in the **doc** object specified in the parameter of the **exec()** callback function. As we are only updating the main document and not associated sub-documents in this operation, we add a line of code to exclude the **reviews** element from the **findById()** operation as follows.

File*: B6/api/controllers/businesses.controllers.js*

```
module.exports.businessesUpdateOne = function(req, res) {
    var businessID = req.params.businessID;
    console.log("GET business " + businessID);
    Business
        .findById( businessID )
        .select("-reviews")
        .exec(function(err, doc) {

    ...
```

Now, we can update the model instance by using the value from the body element of the request object, just as we did earlier in the **businessesAddOne()** controller. The following code box illustrates this portion of the new controller.

File*: B6/api/controllers/businesses.controllers.js*

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

    ...
            if (response.status != 200) {
                res
                    .status(response.status)
                    .json(response.message);
            } else {
                doc.name = req.body.name;
                doc.stars = parseInt(req.body.stars);
                doc.city = req.body.city;
                doc.categories =
                     splitArray(req.body.categories);
                doc.location = {
                    address : req.body.address,
                    coordinates : [
                        parseFloat(req.body.lng),
                        parseFloat(req.body.lat)
                    ]
                }
            }
    ...
```

The third step is to save the instance of the model back to MongoDB. As seen earlier, the **`save()`** method takes two parameters, an error object if the update operation fails and an object holding the updated document. This is illustrated by the highlighted content in the following code box.

File*: B6/api/controllers/businesses.controllers.js*

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

    ...
            } else {
                doc.name = req.body.name;
                doc.stars = parseInt(req.body.stars);
                doc.city = req.body.city;
                doc.categories =
                    splitArray(req.body.categories);
                doc.location = {
                    address : req.body.address,
                    coordinates : [
                        parseFloat(req.body.lng),
                        parseFloat(req.body.lat)
                    ]
                };
                doc.save(function(err, updatedBusiness) {
                    if (err) {

                    } else {

                    };
                });
    ...
```

Finally, in the fourth step, we populate the **`if`** statement by providing the response. If the Mongoose **`save()`** operation fails, we return an error with a **500** code, but if it is successful then the REST standard suggests that we return a **204** code with an empty response body as shown in the code box below.

File*: B6/api/controllers/businesses.controllers.js*

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

    ...
                doc.save(function(err, updatedBusiness) {
                    if (err) {
                        res
                            .status(500)
                            .json(err);
                    } else {
                        res
                            .status(204)
                            .json();
                    }
                });
    ...
```

We can test the new controller by presenting a **PUT** request to the URL http://localhost:3000/api/businesses/12345 to Postman (where 12345 is the **`_id`** of a business in your database) and providing values for the various parameters. This is illustrated by Figure B6.5 below, where we provide updated entries for *name*, *city* and *categories*.

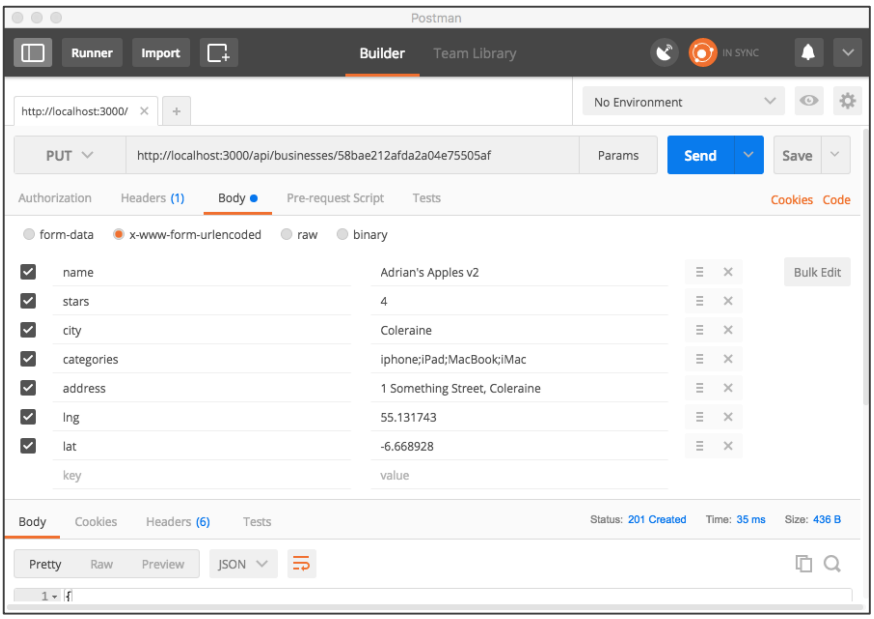

Figure B6.5 Updating a business

The effect of the **PUT** request can be seen by making a **GET** request to the same URL, retrieving the current values from the database and verifying that the update has been successfully carried out. This is shown in Figure B6.6, below.

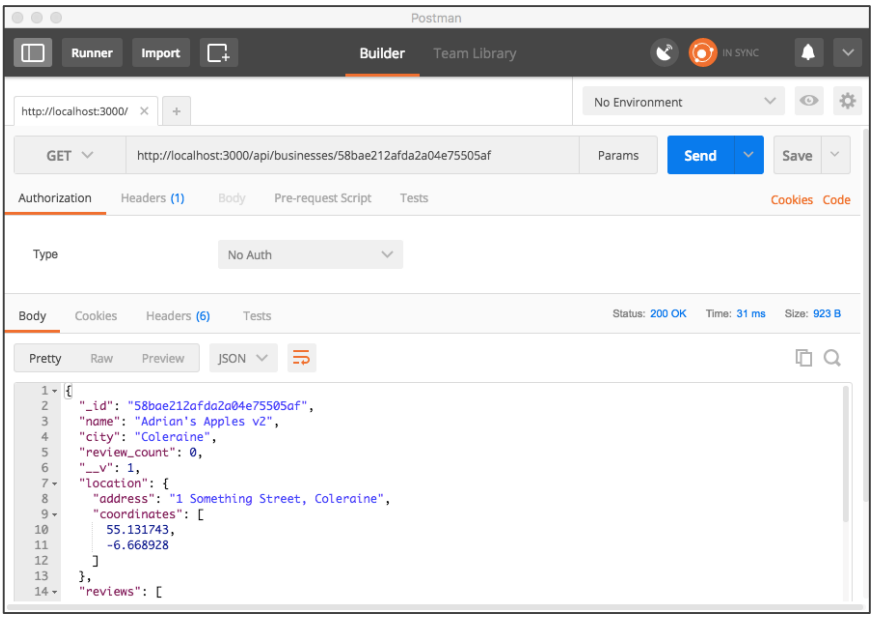

Figure B6.6 Business updated

B6.2.2 Updating an existing sub-document

Updating a sub-document is simply a combination of techniques we have already used, where the key point to remember is that sub-documents can only be accessed by going through their parent. i.e. we need to retrieve and **`save()`** the top-level document, while editing the required attributes of the sub-document.

The code for this controller is longer than any we have seen so far, but by breaking it into its component parts we can recognise how it combines elements previously used.

First, we retrieve the document containing the business specified in the **`businessID`** parameter defined as part of the URL. This uses the code previously seen in **`reviewsGetOne()`** as shown in the code box below.

File*: B6/api/controllers/reviews.controllers.js*

```
module.exports.reviewsUpdateOne = function(req, res) {
  var businessID = req.params.businessID;
  var reviewID = req.params.reviewID;
  console.log('PUT reviewID ' + reviewID +
              ' for businessID ' + businessID);

  Business
    .findById(businessID)
    .select('reviews')
    .exec(function(err, thisBusiness) {
      var thisReview;
      var response = {
        status : 200,
        message : {}
      };
      if (err) {
        console.log("Error finding business");
        response.status = 500;
        response.message = err;
      } else if(!thisBusiness) {
        console.log("Business ID not found", id);
        response.status = 404;
        response.message = {
          "message" : "Business ID not found " + id
        };
      } else {

      // get review and edit

      }
    )};
};
```

Next, we extract the review to be edited by using the **`id()`** method and passing the **`reviewID`** supplied in the URL as the parameter to the method. Again, the appropriate code can be taken from the **`reviewsGetOne()`** controller

File*: B6/api/controllers/reviews.controllers.js*

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

...

    } else {

      // get review and edit
      thisReview = thisBusiness.reviews.id(reviewID);
      if (!thisReview) {
        response.status = 404;
        response.message = {
          "message" : "Review ID not found " + reviewId
        };
      }

      // now check for an error and save

    }
  )};
};
```

Finally, we check for an error and if all is clear, we update the review with the new values and issue the **`save()`** command on the parent document.

File*: B6/api/controllers/reviews.controllers.js*

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

...

  // now check for an error and save

      if (response.status !== 200) {
        res
          .status(response.status)
          .json(response.message);
      } else {
        thisReview.username = req.body.username;
        thisReview.text = req.body.text;
        thisReview.stars = parseInt(req.body.stars);
        doc.save(function(err, updatedBusiness) {
          if (err) {
            res
              .status(500)
              .json(err);
          } else {
            res
              .status(204)
              .json();
          }
        });
      }
    });
}
```

Try it now!

Use Postman to update one of the review elements for a business and verify that your new controller is working properly.

B6.3 Deleting database content

To complete our API for the sample ***WeMeanBusiness*** application, we will add routes and controllers to delete documents and sub-documents from the database. First, as usual, we will add the routes to invoke the new controllers by editing the **routes/index.js** file as shown in the following code box.

File*: B6/api/routes /index.js*

```
...
router
    .route('/businesses/:businessID')
    .get(businessesController.businessesGetOne)
    .put(businessesController.businessesUpdateOne)
    .delete(businessesController.businessesDeleteOne);
...

router
    .route('/businesses/:businessID/reviews/:reviewID')
    .get(reviewsController.reviewsGetOne)
    .put(reviewsController.reviewsUpdateOne)
    .delete(reviewsController.reviewsDeleteOne);
...
```

B6.3.1 Deleting a top-level document

Deleting an entire document is made very easy by the availability of the Mongoose method **`findByIdAndRemove()`** that takes an **`_id`** value as a parameter and deletes that document and any sub-documents contained within it. By convention, a successful delete returns status code **204** and an empty response body – as shown in the code box below.

File*: B6/api/controllers/businesses.controllers.js*

```
module.exports.businessesDeleteOne = function(req, res) {
    var businessID = req.params.businessID;

    Business
        .findByIdAndRemove(businessID)
        .exec(function(err, thisBusiness) {
            if (err) {
                res
                    .status(404)
                    .json(err);
            } else {
                console.log("Business " + businessID
                                + " deleted");
                 res
                    .status(204)
                    .json();
            }
        })
};
```

B6.3.2 Deleting a sub-document

The controller to delete a sub-document is a little longer than that to delete a top-level document, but fortunately we have almost all of it already in our **`reviewsUpdateOne()`** controller developed earlier.

Copy the entire body of the **`reviewsUpdateOne()`** controller into **`reviewsDeleteOne()`** and then see the code box below which identifies the change that you need to make

File*: B6/api/controllers/reviews.controllers.js*

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

...

  // now check for an error and save

      if (response.status !== 200) {udeny
        res
          .status(response.status)
          .json(response.message);
      } else {

        // REMOVE THESE LINES
        //thisReview.username = req.body.username;
        //thisReview.text = req.body.text;
        //thisReview.stars = parseInt(req.body.stars);

        // AND ADD THIS ONE
        doc.reviews.id(reviewID).remove();

        doc.save(function(err, updatedBusiness) {
          if (err) {
            res
              .status(500)
              .json(err);
          } else {
            res
              .status(204)
              .json();
          }
        });
      }
    });
}
```

All of the code to find the parent document containing the business information and to perform the error trapping can stay exactly as was specified in **`reviewsUpdateOne()`**. All that we need to do is replace the code that updates the review values by a single line of

code that uses the **`id()`** method to locate the review in question and chains the **`remove()`** method to delete it from the collection. Once the review has been removed, we simply **`save()`** the parent document and return the appropriate HTTP response code as usual.

Try it now!

Verify that the new endpoints are correctly implemented by using Postman to delete a review from a specific business and then to delete the business completely from the database