+ ecse 489 final project

george azmy | adrain lee
260305707 | 260272188
grg@azmy.ca | adrian.lee@mail.mcgill.ca

rest webserver for mongodb

with user authentication

winter 2013

Table of Contents

1. Introduction

Loosely based on the specification of ‘Project 1’ of the final project guidelines, a REST server was built with user authentication and URL mapping to database queries. However, the database used was not SQL based. MongoDB, a NoSQL, document oriented database was used. Unlike the previous lab, the server was developed in Node.js, server-side implementation of JavaScript, widely used in making highly scalable web applications. User authentication was built into the server restricting creating/reading/editing/deleting data on the database only to entitled users. The server was designed with the REST architecture in mind, to enable multiple users to access various parts of the database simultaneously, hence sharing a limited resource.

2. Environment

The technologies used in the development of this project are outlined in Table 1.

MongoDB	An online service called MongoHQ was used as the database server that used. MongoHQ was free for the scope of this project.
Node.js	The backend was developed in Node.js
Mocha	Test suite used to test the multiple functions and cases the server is designed to handle
Express	A simple web framework for node.js to handle all the sockets and the various HTTP requests.
Postman	A handy Google Chrome Extension for making various HTTP requests, with a clean GUI to manipulate the headers and analyze the responses, cookies, etc.

Table 1 Development Environment

This project was developed on with the latest version of all the above (at the time of execution) on Mac OSX 10.8. The decision to switch to Node.js and MongoDB as opposed to a more classic approach such as Java+MySQL was because Node and Mongo are emerging technologies in the web development industry because of their scalability, performance and extensive libraries and frameworks specifically designed for the task this project aims to achieve.

3. Implementation

To start with, a free account was set up with MongoHQ in order to have a web-hosted MongoDB database server. A simple node.js script was written to start the connection and for the server to communicate with the database
Since the implementation of a multi-threaded webserver was tested in ECSE414, and then again with REST in Lab 3 of this course, there was no need to implement the low-level functionality of opening web sockets and programming the server to handle the various HTTP requests. Hence ‘Express’ was used. Express provides a web framework for node.js development.

The implementation was done using only 5 modules. These are outlined in Table 2.

server.js	This is the entry point of the server and it calls on the other modules written along with the express framework.
middleware.js	This module handles authentication and cookies
mongo.js	This module merely connects to the database server at MongoHQ and provides query calling to the database
routes.js	This module generates the queries based on the data passed to it from the server module
test.js	This is the mocha test suite that was used to verify correct functionality of the server

Table 2 Modules Developed

When server.js receives a request, it first checks if the user is authenticated or not, by calling functions from middleware.js. middleware.js uses mongo.js to connect to the Users database at MongoHQ to match the user’s hash with one on the database. Once authentication is complete, server.js then calls the correct command from routes.js which handles the various database queries based on the HTTP request type and the parameters passed.
test.js is run independently of the other modules, making various HTTP request to server.js and analyzing the responses, comparing them to what it was programmed to expect.

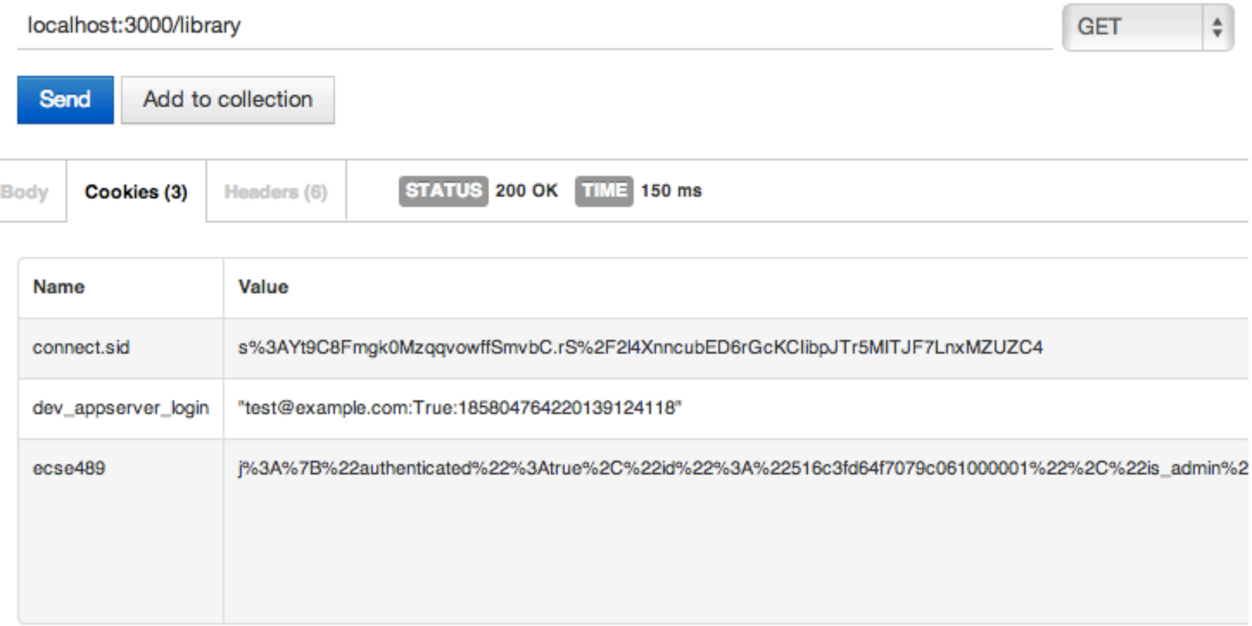
4. Functionality

The basic functionality:

1. Login

The first time a user makes a request to the server, they will be asked to log in. Authentication used is BasicAuth with a base64 hashing algorithms. Each HTTP request contains “username:password” hashed in the header. Each request will then match the credentials with those stored in the ‘User’ collection in the database.

Once authenticated, a cookie named ‘ecse489’ is generated. The cookie contains credentials, list of collections the user has access to and an expiration timestamp of the cookie. The generated hashes persist and are passed in the header with subsequent requests. An example of the cookie generated is shown in Figure 1



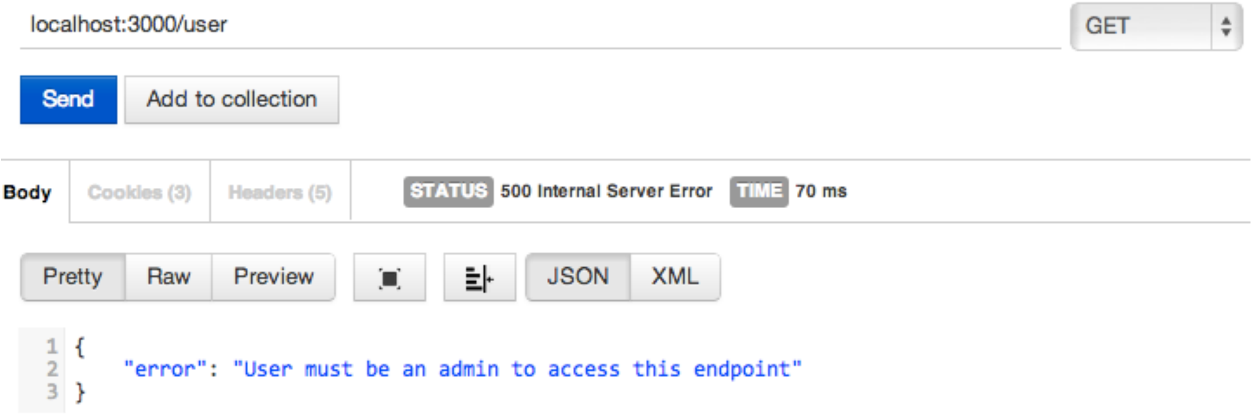
localhost:3000/library

STATUS 200 OK TIME 150 ms

Name	Value
connect.sid	s%3AYt9C8Fmgk0MzqqvowffSmvbC.rS%2F2l4XnncubED6rGcKClibpJTr5MITJF7LnxMZUZC4
dev_appserver_login	"test@example.com:True:185804764220139124118"
ecse489	j%3A%7B%22authenticated%22%3Atrue%2C%22id%22%3A%22516c3fd64f7079c061000001%22%2C%22is_admin%2

Figure 1 Cookies

If a user is not admin, they get an error as shown in Figure 2



localhost:3000/user

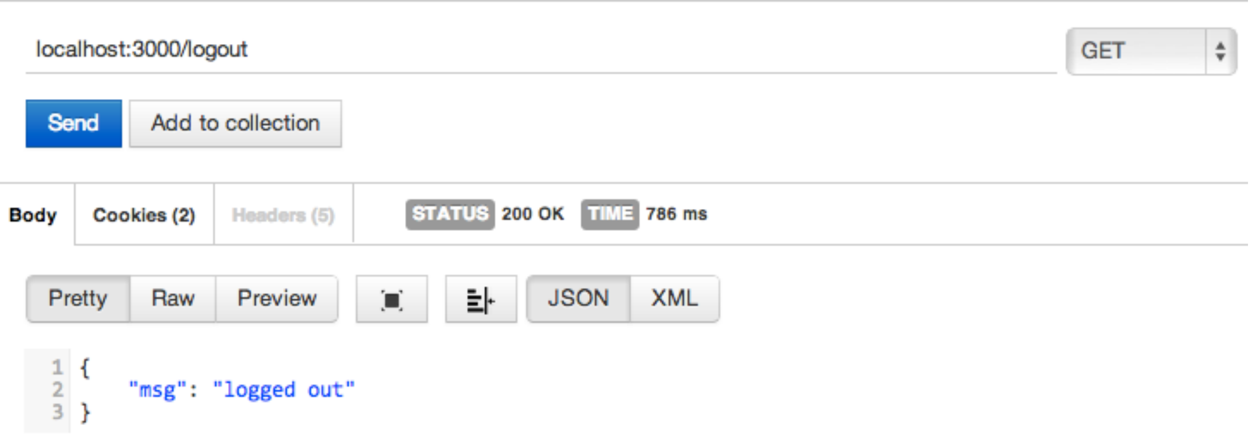
STATUS 500 Internal Server Error TIME 70 ms

```
{
    "error": "User must be an admin to access this endpoint"
}
```

Figure 2 Access Denied

2. Logout

Making a `GET server/logout` request clears the cookie and returns a response as shown in Figure 3



localhost:3000/logout

STATUS 200 OK TIME 786 ms

```
{
    "msg": "logged out"
}
```

Figure 3 User Logout

User related functions

3. Listing All Users

User must have admin privilege make this request.

`GET server/user` makes a query to the database that returns the collection of all users. This request maps to the `find({})` query sent to the database, which returns all of the contents of the User collection

4. Retrieving a Single User

Again, admin privilege is required. `GET server/user/user001` queries the database with `findOne()` to return the document for the user 'user001' and the server returns its contents.

5. Creating a New User

No admin privilege is required to create a new user. To create a new user, a simple `POST server/user/user002` is required to create a non-admin user. The user details are specified in JSON format in the body of the request. The server then queries the database with the `insert()` command. This command also works for collections.

6. Updating a New User

Unlike creation, updating a user requires admin rights. A `PUT server/user/user001` request updates a user profile, with the data in JSON format embedded in the body of the request. The server then queries the database with `update()`, which also creates the element if it doesn't already exist.

7. Deleting a User

Deletion of a user works by updating the user profile to an empty one. Admin privilege is required and the request to perform it is `DELETE server/user/user001`, after which the server sends a `remove()` query to the database

Data related functions:

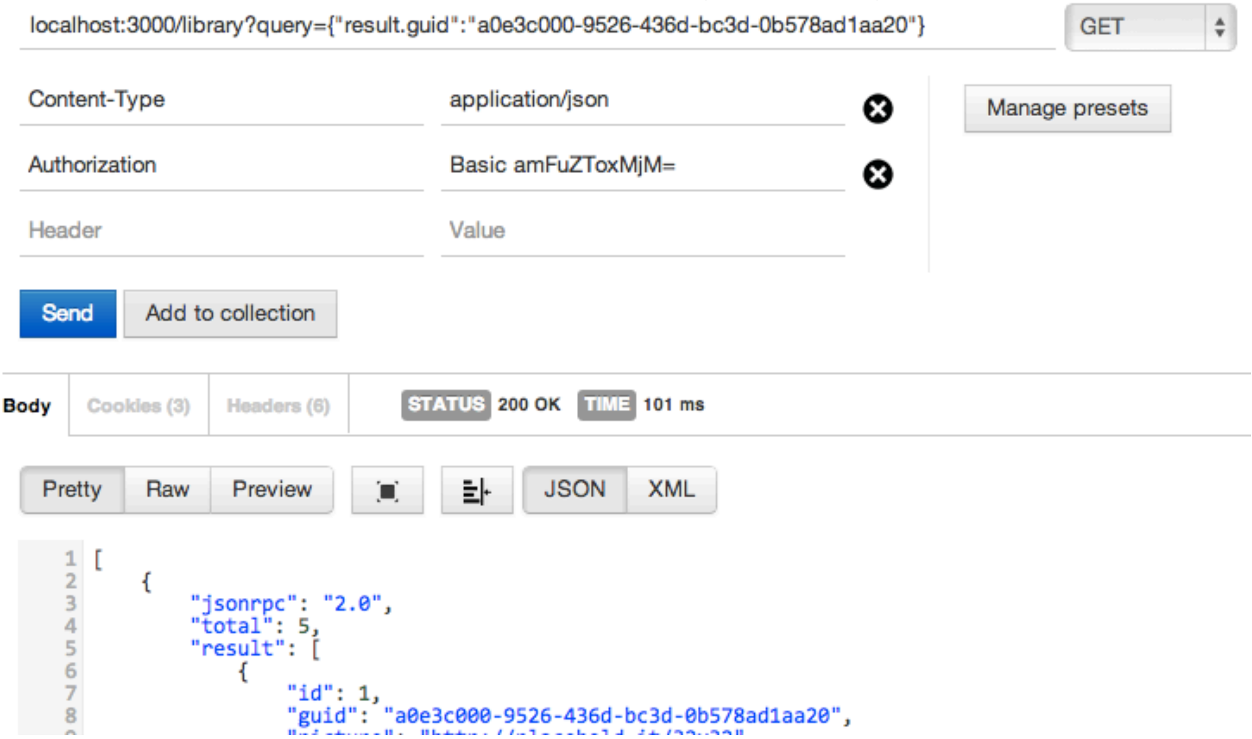
8. Retrieving data

There are two ways to get data, either an entire collection or a specific document. No admin privileges are required, however access to that collection or document is required. To retrieve an entire collection, `GET server/collection/` will do the job.
`GET server/collection/documentID` will retrieve a specific document.

Filtering

`GET server/collection?filter={“id”: “1234”}`

Filter queries can be added as a HTTP query to the request. Sub attributes can also be specified such `GET server/collection?filter={“book.author”: “obama”}`. The server also accepts ‘query’ instead of filter. An example of filtering is shown Figure 4



localhost:3000/library?query={"result.guid":"a0e3c000-9526-436d-bc3d-0b578ad1aa20"}

GET

Content-Type	application/json
Authorization	Basic amFuZToxMjM=
Header	Value

STATUS 200 OK TIME 101 ms

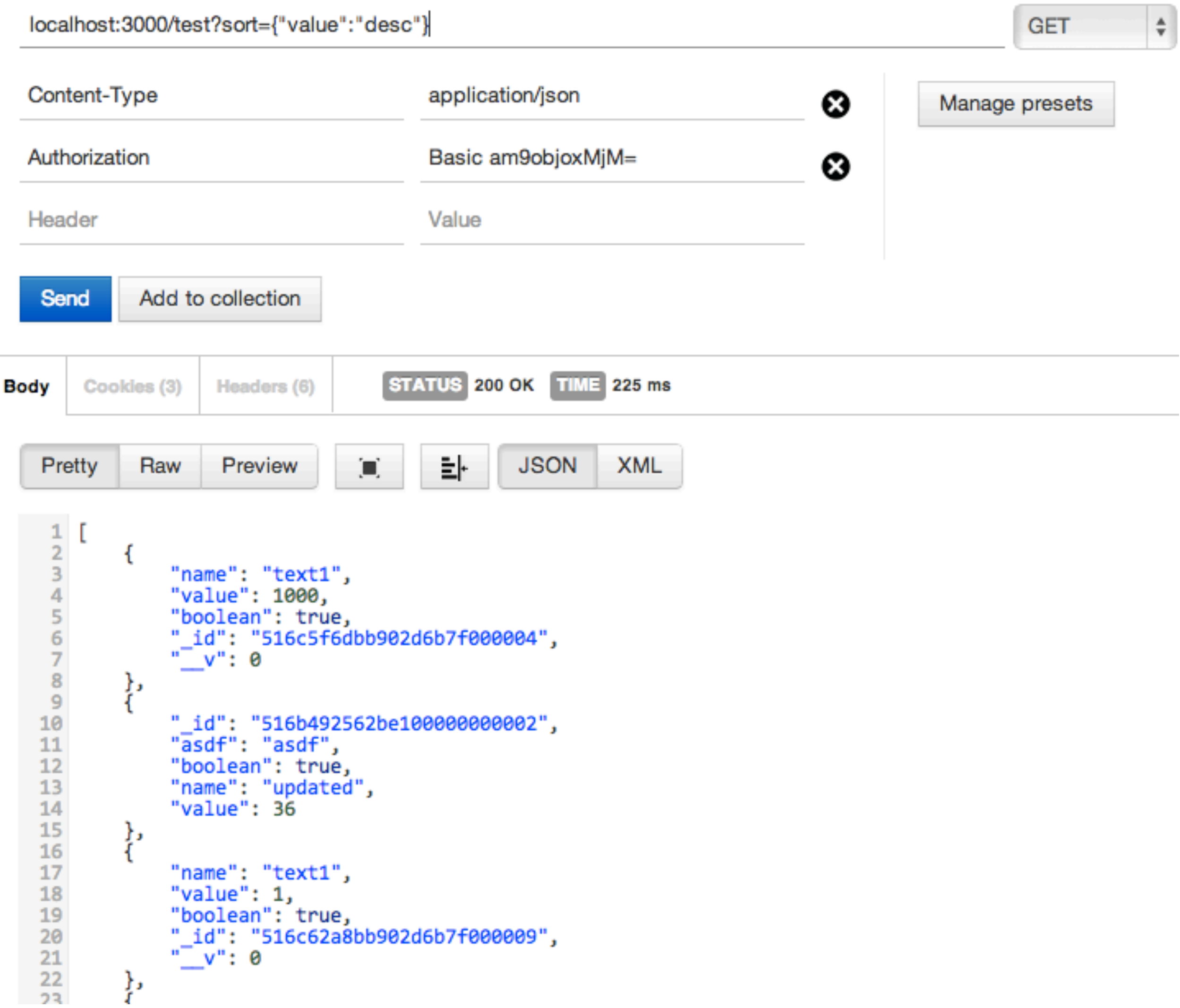
```
[
    {
        "jsonrpc": "2.0",
        "total": 5,
        "result": [
            {
                "id": 1,
                "guid": "a0e3c000-9526-436d-bc3d-0b578ad1aa20",
```

Figure 4 Query Filtering

Ordering

Similar to filtering, ordering the data is also possible by adding an orderBy attribute to the HTTP request query such as
`GET server/collection?orderBy={“id”: “asc”}` for ascending order (example: Figure 5)
Or `GET server/collection?orderBy={“id”: “asc”}` for descending order

Lastly, appending another query to the HTTP request as such can set a limit to the filtering/ordering:
`GET server/collection?limit=12`



localhost:3000/test?sort={"value":"desc"}

Content-Type: application/json

Authorization: Basic am9objoxMjM=

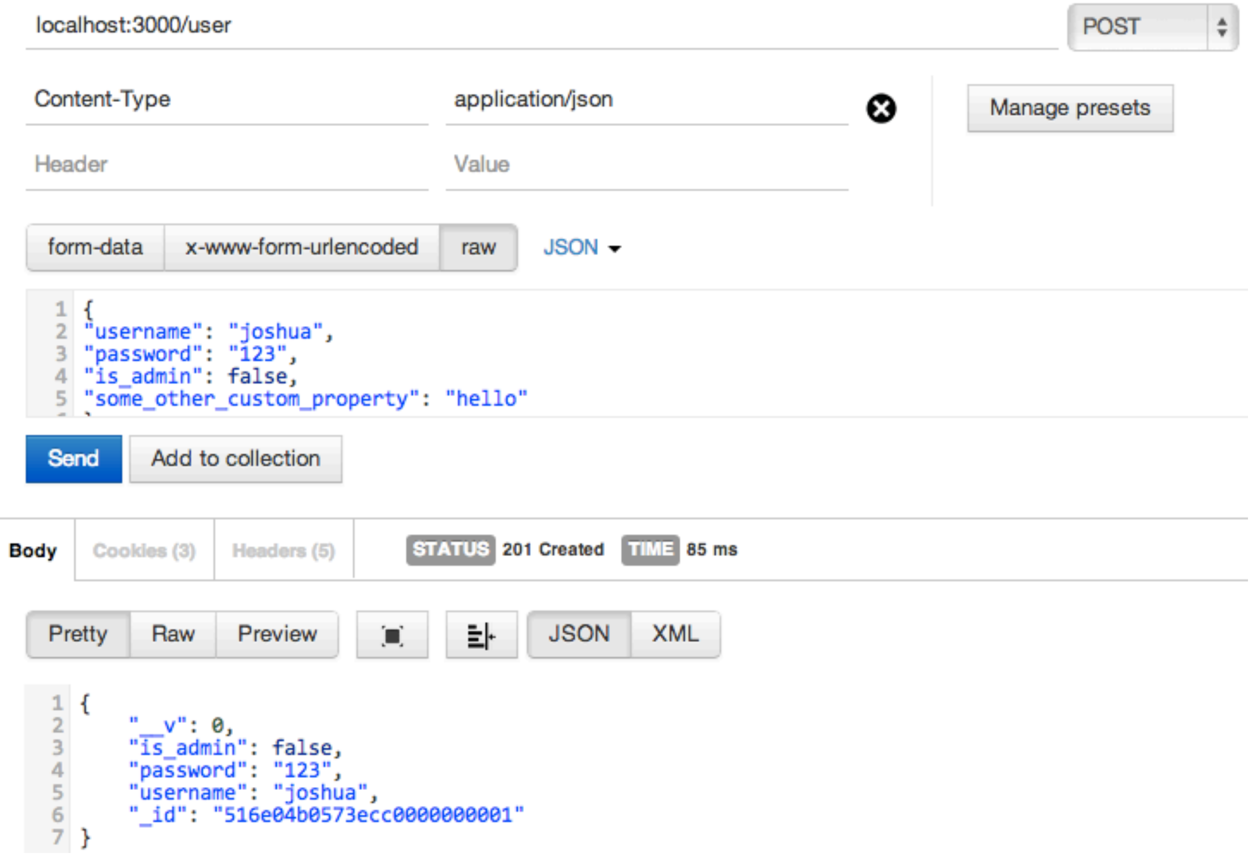
STATUS 200 OK TIME 225 ms

```
[
    {
        "name": "text1",
        "value": 1000,
        "boolean": true,
        "_id": "516c5f6dbb902d6b7f000004",
        "__v": 0
    },
    {
        "_id": "516b492562be100000000002",
        "asdf": "asdf",
        "boolean": true,
        "name": "updated",
        "value": 36
    },
    {
        "name": "text1",
        "value": 1,
        "boolean": true,
        "_id": "516c62a8bb902d6b7f000009",
        "__v": 0
    },
    {
```

Figure 5 Ordering in ascending order

9. Adding New Data

Adding a new document, given a user has the access to the collection, is simply accomplished by a `POST server/collection/` or `POST server/collection/documentID` with the data embedded in a JSON file in the body of the request. An example of this is shown in Figure 6



localhost:3000/user

POST

Content-Type: application/json

```
{
"username": "joshua",
"password": "123",
"is_admin": false,
"some_other_custom_property": "hello"
```

STATUS 201 Created TIME 85 ms

```
{
    "__v": 0,
    "is_admin": false,
    "password": "123",
    "username": "joshua",
    "_id": "516e04b0573ecc0000000001"
}
```

Figure 6 Adding Data with JSON

10. Updating Data

Similar to adding data, `PUT server/collection/documentID` can be used to update a document, again with embedding the changes in JSON in a the request body

11. Deleting Data

Given that a user has access to a collection, deletion called by `POST server/collection/` or `POST server/collection/documentID` request, with an empty JSON embedded in the request

5. Response Codes

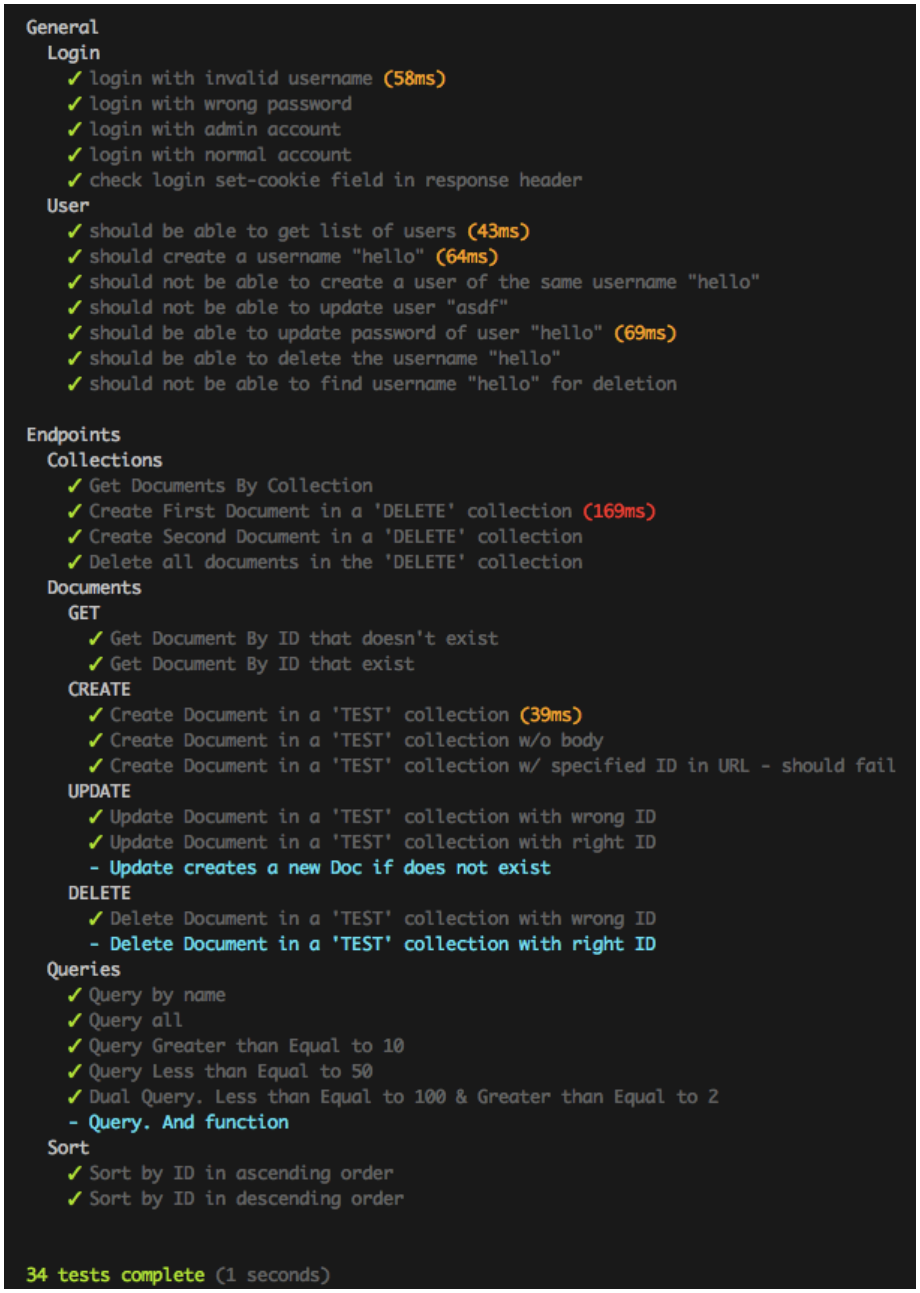
Every function returns a specific HTTP response code based on how the operation went. The implemented HTTP responses are outlined in Table 3.

Code	Response
200	OK
201	CREATED
202	ACCEPTED
400	BAD REQUEST
401	UNAUTHORIZED
402	PAYMENT REQUIRED
403	FORBIDDEN
404	NOT FOUND
405	METHOD NOT ALLOWED
500	INTERAL SERVER ERROR
501	NOT IMPLEMENTED

Table 3 HTTP Response Codes

6. Testing

The various functions that the server was designed to implement were tested using a test suite written in the Mocha framework. A total of 34 test were written (and passed!), the output of the test is shown in Figure 7



```
General
  Login
    ✓ login with invalid username (58ms)
    ✓ login with wrong password
    ✓ login with admin account
    ✓ login with normal account
    ✓ check login set-cookie field in response header
  User
    ✓ should be able to get list of users (43ms)
    ✓ should create a username "hello" (64ms)
    ✓ should not be able to create a user of the same username "hello"
    ✓ should not be able to update user "asdf"
    ✓ should be able to update password of user "hello" (69ms)
    ✓ should be able to delete the username "hello"
    ✓ should not be able to find username "hello" for deletion

Endpoints
  Collections
    ✓ Get Documents By Collection
    ✓ Create First Document in a 'DELETE' collection (169ms)
    ✓ Create Second Document in a 'DELETE' collection
    ✓ Delete all documents in the 'DELETE' collection
  Documents
    GET
      ✓ Get Document By ID that doesn't exist
      ✓ Get Document By ID that exist
    CREATE
      ✓ Create Document in a 'TEST' collection (39ms)
      ✓ Create Document in a 'TEST' collection w/o body
      ✓ Create Document in a 'TEST' collection w/ specified ID in URL - should fail
    UPDATE
      ✓ Update Document in a 'TEST' collection with wrong ID
      ✓ Update Document in a 'TEST' collection with right ID
      - Update creates a new Doc if does not exist
    DELETE
      ✓ Delete Document in a 'TEST' collection with wrong ID
      - Delete Document in a 'TEST' collection with right ID
  Queries
    ✓ Query by name
    ✓ Query all
    ✓ Query Greater than Equal to 10
    ✓ Query Less than Equal to 50
    ✓ Dual Query. Less than Equal to 100 & Greater than Equal to 2
    - Query. And function
  Sort
    ✓ Sort by ID in ascending order
    ✓ Sort by ID in descending order


34 tests complete (1 seconds)
```

Figure 7 Test Suite Output

7. Conclusions and Further Work

The REST server was implemented successfully with the integration of a database as opposed to the XML data in the labs, which provides a much more robust and scalable data storage and manipulation. The technologies picked did indeed turn out to be ideal for a web-centric application like this, providing an excellent API for development, along with a very good performance. This project could be further improved by having adding an extra layer of security using HTTPS for the authentication. Furthermore, the server could be programmed to implement more query functions such as

AND, OR, NOR
>, <, >=, <=
≠, in, not-in

Lastly, the server could be updated to be able to manipulate multiple documents from one query, and simultaneously add to multiple collections.

8. References

http://nodejs.org/
http://expressjs.com/
http://www.mongodb.org/
https://www.mongohq.com/home
http://visionmedia.github.io/mocha/
http://www.getpostman.com/