**Exercise02\_04\_01 – Step 1**

This exercise will consist of coding the database storage parts of our application.



1. Create a new folder for the exercise called ***Exercise02\_04\_01***. Copy all of the files into it from the previous Exercise02\_03\_01. In the background, run ***ngrok***, ***mongoDB***, and ***RoboMongo***. Open your IDE to the new project folder. Our app currently has to go to Twitter every time it needs to display our friends, performance inefficient. Additionally, it risks us being ***rate*** ***limited*** by Twitter. To solve this, we will ***cache*** our friends in a database.
2. We need to install an external third-party module that we will need with NPM. We will be using mongoDB as our database, so we first need to install it:  
   ***npm install mongodb --save***Check **package.json** to make sure the dependencies are correct.
3. Now let’s open up ***index.js*** and require() what we need. Notice that we are going to get a ***MongoClient*** object, which is the official Node.js driver for mongoDB:  
   var async = require('async');  
   ***var MongoClient = require('mongodb').MongoClient;***
4. We will continue to modularize this application. We will build a module called ***storage.js*** that will handle all of our database access. So let’s ***require()*** that in as well:  
   var authenticator = require('./authenticator.js');  
   ***var storage = require('./storage.js');***

**Exercise02\_04\_01 – Step 2**



1. Let’s create ***storage.js***, starting with a ***require()*** of the ***MongoClient*** object, setting our mongoDB ***URL*** and database ***name***, and building a variable which will hold our ***database*** object when we connect:  
   ***var MongoClient = require('mongodb').MongoClient;  
   var url = 'mongodb://localhost:27017';  
   var dbName = 'twitter\_notes';  
   var database;***Run the server and browser test it in the browser, then terminate all.
2. Now let’s build our ***module.exports*** object in ***storage.js***. the first function that we want to export will be ***connect()***, which will take care of connecting to our database. If the database does not exist, mongoDB will create it when the first document is inserted:  
   ***module.exports = {  
    connect: function() {  
    MongoClient.connect(url, function(err, client) {  
    if (err) {  
    return console.log('Error: ' + err);  
    }  
    database = client.db(dbName);  
    console.log('Connected to database: ' + dbName);  
    });  
    ,  
   }***Run your server to make sure that there are no syntax errors, then terminate it.
3. Let’s see if our server can connect to the database. Go to index.js and enter the ***connect()*** code right after we build our Express instance:  
   var app = express();  
     
   ***storage.connect();***If everything is okay, we should see a positive message in our console.
4. Now let’s go back to ***storage.js*** and build a helper function that will tell us if we are connected to our database. This will eventually help us to decide if we load our friends from Twitter, or from what we have cached in the database. Build it right below our ***connect()*** function:  
    *}****,  
    connected: function () {  
    return typeof database != 'undefined';  
    }***  
   Run your server for syntax error checking, then terminate.
5. Now let’s put in some code that will tell us to load from Twitter if we are not connected. We can do that in our main route ***/***. First comment out the ***storage.connect()*** call for testing, then modify the main route code as follows:  
   app.get('/', function(req, res) {  
    var credentials = authenticator.getCredentials();  
    if (!credentials.access\_token || !credentials.access\_token\_secret) {  
    return res.redirect('/login');  
    }  
    ***if (!storage.connected()) {*** console.log('Loading friends from Twitter');  
    renderMainPageFromTwitter(req, res);  
    ***}***});  
   Run your server and give the browser the ***/*** url: ***localhost:8080/***. Sign in and go through the ***authorization*** process. We should see our Loading friends from Twitter console message, and we should see our friends data. Lastly, ***logout*** Once again run your server and give the browser the ***/*** url: ***localhost:8080/***. Sign in and go through the ***authorization*** process. Now we should automatically see our friends data. Lastly ***logout*** and terminate.

**Exercise02\_04\_01 – Step 3**



1. We are now getting all of our friends data directly from Twitter. As previously mentioned, this causes us two problems. Firstly, it is fairly ***slow*** and ***inefficient***, putting load onto the Twitter servers and Oauth, and having to go through ***multiple*** ***server*** **levels**. Secondly, we take the chance of being ***rate*** ***limited*** by Twitter. Our solution will be to ***cache*** the data in our own mongoDB database after we get it.
2. In our last exercise, we converted our ***friends*** array into an array of ***JSON*** objects. This is the perfect format for a mongoDB ***collection***. Lets go to ***storage.js*** and build a function which will insert our friends into the database:  
    }***,  
    insertFriends: function(friends) {  
    database.collection('friends').insert(friends, function(err) {  
    if (err) {  
    console.log('Cannot insert friends into database.');  
    }  
    });  
    }***  
   Run your server for syntax error checking, then terminate.
3. Back in ***index.js***, the best time to insert our data will be right after we get it from Twitter. First we have to get rid of some ***debug***. At the top of ***index.js***, remove the comments from ***storage.connect()*** so we can have our database available. Now, down in out main route ***/***, let’s ***comment*** out a couple of lines and force our friends to load from Twitter:  
   app.get('/', function(req, res) {  
    var credentials = authenticator.getCredentials();  
    if (!credentials.access\_token || !credentials.access\_token\_secret) {  
    return res.redirect('/login');  
    }  
   ***// if (!storage.connected()) {***  
    console.log('Loading friends from Twitter');  
    renderMainPageFromTwitter(req, res);  
   ***// }***});  
   Give this a full test and make sure our data is, in fact, still loading from Twitter.
4. Let’s go to the bottom of the ***renderMainPageFromTwitter()*** function. Right after we ***render*** the page and pass the data to the template, let’s insert the ***friends*** data into our database with our new ***insertFriends()*** call:  
    res.render('index', { friends: friends });  
    ***if (storage.connected) {  
    storage.insertFriends(friends);  
    }***Let’s do some significant testing. First, run the app all the way through as we have been doing, then logout. Bring up RoboMongo and ***connect*** to the database server which is waiting for a connection on port ***27017***. Expand ***New Connection*** if necessary, and the ***twitter\_notes*** database should be listed. Expand that tab, then the ***Collections*** tab, and the ***friends*** collection should be listed. Double click on ***friends*** and there should be a listing of ***documents*** for all of your friends. Expand one and there is the data on your friend. Explore it for all of the info we know have on file.

**Exercise02\_04\_01 – Step 4**



1. Now that we have all of our friends stored in mongoDB, lets go to ***storage.js*** and build a function which will retrieve our friends from the database. Documents are returned with a ***pointer*** called a ***cursor***. It is an ***object*** with ***methods*** that can be used on the ***result*** ***set*** that is returned from the database:  
    }***,  
    getFriends: function(userId, callback) {  
    var cursor = database.collection('friends').find({ for\_user: userId});  
    cursor.toArray(callback);  
    }***  
   Run your server for syntax error checking, then terminate.
2. Back in ***index.js***, let’s build the code in the main route ***/*** that uses the retrieved data. For starters, remove the ***comments*** around the last ***if*** statement. Now we can add the code below it to call our new ***getFriends()*** function:  
    ***console.log('Loading friends from MongoDB');  
    storage.getFriends(credentials.twitter\_id, function(err, friends) {  
    if (err) {  
    return res.status(500).send(error);  
    }  
    if (friends.length > 0) {  
    console.log('Friends successfully loaded from MongoDB');  
    friends.sort(function(a, b) {  
    return a.name.toLowerCase().  
    localeCompare(b.name.toLowerCase());  
    });  
    res.render('index', { friends: friends});  
    }  
    else {  
    console.log('Loading friends from Twitter');  
    renderMainPageFromTwitter(req, res);  
    }  
    });***  
   Let’s do some significant testing. First, run the app all the way through as we have been doing. We should get a console message that we are ***Loading friends from MongoDB*** and our data should be good. Now bring up RoboMongo and ***connect*** to the database server that is listening on port ***27017***. Expand ***New Connection*** if necessary, and the ***twitter\_notes*** database should be on the list. Expand that tab, then the ***Collections*** tab, and the ***friends*** collection should be on the list. Right click on ***friends***, and select ***drop the collection***. Recycle the app in the browser, and we should get a console message ***Loading friends from Twitter***. The app knows there is now no data in the ***friends*** collection. We will make use of this feature in a little while.

**Exercise02\_04\_01 – Step 5**



1. We can now get our data from either Twitter or MongoDB. However, MongoDB is acting as a ***cache*** for the data. This means that there is a possibility that the data can go ***stale***. In other words, the data in MongoDB can become older than the Twitter data. Therefore, we must have a way to ***delete*** the data from the cache and force MongoDB to go to Twitter. That is already being done if the code finds that there is nothing in the ***friends*** array by measuring its ***length*** at 0. Now let’s go to ***storage.js*** and build a function which will delete our friends from the database:  
    }***,  
    deleteFriends: function() {  
    database.collection('friends').remove(( {} ), function(err) {  
    if (err) {  
    console.log('Cannot remove friends from database.');  
    }  
    });  
    }***  
   Run your server for syntax error checking, then terminate.
2. Now we pick the optimal points to flush our cache. The first would be in our ***/logout*** route. We do not want to leave data in the database to get old when it is not being used. Modify that route as follow:  
   app.get('/logout', function(req, res) {  
    authenticator.clearCredentials();  
    ***if (storage.connected()) {  
    console.log('Deleting friends collection on logout');  
    storage.deleteFriends();  
    }***  
    res.redirect('/login');  
   });  
   Run the app all the way through as we have been doing. There is still leftover data in MongoDB, so we should get a console message that we are ***Loading friends from MongoDB*** and our data should be good**.** Now ***logout*** and we get a delete message in the console. Look in RoboMongo and the collection is gone. If we then login, it first tries to load from MongoDB, because the database is ***connected***. But the length of the ***friends*** array is 0, so it goes and loads from Twitter. Now logout again.
3. The next spot that we want to flush our cache is immediately in the ***/login*** route. We do not want any old data in there, in case the last user has not logged out, and also to protect our security in case the user goes directly to that route. The code here is the same:  
   app.get('/login', function(req, res) {  
    authenticator.clearCredentials();  
    ***if (storage.connected()) {  
    console.log('Deleting friends collection on login');  
    storage.deleteFriends();  
    }*** res.render('login');  
   });  
   Run the app again and we should get an immediate delete message in the console when directed to ***/login***. Run the app through and the behavior should be as expected. Now logout again.
4. We have one more thing to deal with. The data in mongoDB can get old if the application remains running. Our solution to this can be to delete the data with an ***interval*** ***timer***. A reasonable interval would be every five minutes. But for testing, let’s do it every 10 seconds, then we can adjust it to 5 minute intervals. Let’s set this up early, right after our ***app.use()*** that sets our view engine:  
   app.set('view engine', 'ejs');  
     
   ***setInterval(function() {  
    if (storage.connected()) {  
    console.log('Clearing MongoDB cache');  
    storage.deleteFriends();  
    }  
   }, 1000 \* 10);***  
   Let’s give this some testing and watch the cache clear, intermingling regenerating our browser to see where the app is loading data from.
5. Let’s finish it off by adjusting our cache clearing to 5 minutes as follows:  
   ***}, 1000 \* 60 \* 5);***  
   We have now finished off the part of the application that ***consumes*** third-party APIs. Next, we will write our own RESTful APIs to finish off the Notes part of the application.