**Exercise02\_05\_01 – Step 1**

This exercise will consist of designing coding a custom RESTful API.



1. Create a new folder for the exercise called ***Exercise02\_05\_01***. Copy all of the files into it from the previous Exercise02\_04\_01. In the background, run ***ngrok***, ***mongoDB***, and ***RoboMongo***. Open your IDE to the new project folder.
2. We will start to build out our-front-end notes display with a JavaScript IIFE. To prepare this, open index.ejs and link in the script as follows:  
    <link rel="stylesheet" href="style.css">  
    ***<script type="text/javascript" src="main.js"></script>***
3. Now let’s start to build the JavaScript IIFE. It will be static content, so we will build it in the ***/public*** folder and name it ***main.js***. We construct the first main part of an IIFE. We place an ***anonymous*** ***function*** within ***parentheses***, to give it lexical scope and protect its variables and functions. We will define some ***helper*** variables as well. One will be ***selectedUserId***, to keep track of which user’s notes we are working with. The second will be ***cache***, an ***object*** to keep us from going to our database too much:  
   ***(function() {  
    var selectedUserId;  
    var cache = {};  
   });***
4. Now let’s create the second part of an IIFE, the ***Immediately Invoked Function Expression ()*** operator. We can put in an ***alert()*** debug to see if it does execute:  
   (function() {  
    var selectedUserId;  
    var cache = {};  
    ***alert("Hello from my IIFE!");***})***()***;  
   Let’s run our server and browser and see if everything operates as planned and we get our alert().
5. On execution of our IIFE, we want it to run a ***startup*** function that will put into place the support we will need. So let’s overwrite our debug ***alert()*** by writing ***startup()*** with some debug in it. We will then set up an ***event*** ***handler*** that will call our startup function on the ***load*** event:  
   (function() {  
    var selectedUserId;  
    var cache = {};  
      
    ***function startup() {  
    alert('I am in startup()!')  
    }  
     
    document.addEventListener("DOMContentLoaded”, startup, false);***})();  
   Notice the new event ***DOMContentLoaded***. This event is fired when the initial HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and subframes to finish loading. Let’s run our server and browser and see if everything operates as planned and we get our startup alert(). We have now scaffolded our IIFE, and we will continue to build it out as we need it to give us good testing.
6. The first thing that we want to do is have ***startup()*** load all of the ***<li>*** elements that are holding our friends from the DOM. We will put some ***console.log()*** debug into the code to make sure we got them by logging the number of each of the <li> elements. Let’s overwrite our ***alert()*** debug:  
    function startup() {  
    ***var friends = document.getElementsByClassName('friend');  
    for (var i = 0; i < friends.length; i++) {  
    console.log('friend nbr: ', i);  
    }*** }  
   Open the Developer tools to the ***Console*** tab and recycle the browser. We should get a track of our ***for*** loop.
7. We want to add an ***event*** ***handler*** to each of the ***<li>*** elements so that we know which friend has been clicked. Let’s do that in our for loop. We will use an ***anonymous*** function for the handler. We will assign the ***selectedUserId*** to the friend’s Twitter id by using the object-oriented ***this*** keyword to refer to the ***current*** object. Use a console.log() to debug by checking the friend’s ***selectedUserId*** to make sure the click is getting processed correctly.:  
    for (var i = 0; i < friends.length; i++) {  
    ***friends[i].addEventListener("click", function() {  
    selectedUserId = this.getAttribute('uid');  
    console.log("Twitter id: ", selectedUserId);  
    });*** }  
   Recycle the browser and we should get a track of the twitter id of the clicked friend.
8. Let’s bring our ***CSS*** into play on the friend we click. We can do that by manipulating the class attribute of the <li>, which in JavaScript is called ***className***. In our event handler, we will loop through all the friends, making sure that className has its original value: ***friend***. We will then use the ***this*** keyword to set the ***current*** ***object*** to: ***active***. That will employ different CSS. Let’s remove our ***console.log()*** debug, and do this code above the **selectedUserId** assignment:  
    friends[i].addEventListener("click", function() {  
    ***for (var j = 0; j < friends.length; j++) {  
    friends[j].className = 'friend';  
    }  
    this.className += ' active';*** selectedUserId = this.getAttribute('uid');  
    });  
   Recycle the browser and we should get some good color on the clicked friend.

**Exercise02\_05\_01 – Step 2**



1. Let’s start to construct our ***routing*** to support our RESTful API. We will build a function that will check to see if we are logged in and have our credentials. If we do, we will store our Twitter ID in a cookie so we can use it for a number of routes. We will build it as ***middleware*** just below our ***/logout*** endpoint as follows:  
   ***function ensureLoggedIn(req, res, next) {  
    var credentials = authenticator.getCredentials();  
    if (!credentials.access\_token || !credentials.access\_token\_secret   
    || !credentials.twitter\_id) {  
    return res.sendStatus(401);  
    }  
    next();  
   }***
2. Next, we will build an ***endpoint*** for the ***GET*** operation. This will be designed to respond to a request for all of the notes for a friend that has been clicked on the front-end. Let’s scaffold that endpoint just below our new ***ensureLoggedIn()*** function. We will employ that function as middleware, for the endpoint. We debug with a temporary URL, /checkcookie, to see that the user has been ***authorized*** and that we have access to our cookie with the Twitter id as follows:  
   ***app.get('/checkcookie', ensureLoggedIn, function(req, res, next){  
    console.log(req.cookies);  
    return res.sendStatus(200);  
   });***  
   Let’s run our server and browser with Developer Tools on, and on the ***Application*** tab with ***Cookies*** open. Go all the way through to our friends display and add ***/checkcookie*** to the end of the URL. We should see a cookie appear in the ***Developer*** ***Tools***, our Web page shows an ***OK*** status, and our cookie data on the Node.js ***console***. Back arrow to the friends screen and ***Logout***.
3. Our cookie remains after we logout, so to be secure, let’s clear it in both the ***/logout*** and ***/login*** routes as follows:  
   app.get('/logout', function(req, res) { // and '/login'  
    authenticator.clearCredentials();  
    ***res.clearCookie('twitter\_id');***  
   Let’s give this a retest and our cookie should be gone when we are done.
4. Now we can finish off our API GET endpoint. We will overwrite ***/checkcookie*** and put in the proper ***URL*** for the ***notes*** ***collection*** resource. Next we will call a new function, ***getNotes()***, which we will build right after this in our ***storage*** module. This function will return our ***notes*** data for the Twitter id of the friend that was clicked on the front-end. We will take care of our ***error*** on failure, and send our ***notes*** data with the response on success. We will ***console.log()*** our notes data for debug, but we won’t have any notes yet, just checking for problems:  
   app.get('***/friends/:uid/notes***', ensureLoggedIn, function(req, res, next){ ***var credentials = authenticator.getCredentials();  
    storage.getNotes(credentials.twitter\_id, req.params.uid,  
    function(err, notes) {  
    if (err) {  
    return res.status(500).send(err);  
    }  
    res.send(notes);  
    });***});
5. Let’s go to ***storage.js*** and build our database function, ***getNotes()***, to retrieve the friend’s notes. It will take in the ***owner’s*** Twitter Id, and the Twitter Id of the ***friend*** that was clicked. It is similar to the ***getFriends()*** function we built. Let’s place it at the bottom of the ***exports*** object. The biggest difference will be modifying the data that MongoDB returns into the format we want to send back to our application:  
    }***,  
    getNotes: function(ownerid, friendid, callback) {  
    var cursor = database.collection('notes').find({   
    owner\_id: ownerid,  
    friend\_id: friendid  
    });  
    cursor.toArray(function(err, notes) {  
    if (err) {  
    return callback(err);  
    }  
    callback(null, notes.map(function(note) {  
    return {  
    \_id: note.\_id,  
    content: note.content  
    }  
    }));  
    });  
    }***  
   Let’s run our server and browser with Developer Tools on, and on the ***Application*** tab with ***Cookies*** open. Go all the way through to our friends display. Open ***RoboMongo*** and expand a friend. Get a Twitter Id and add ***/friends/:uid/notes*** to the end of the URL, substituting the Twitter Id for ***uid***. We should see an empty array on the Web page, which is correct because we have no notes. But we also don’t see any errors. Back arrow to the friends screen and ***Logout***.

**Exercise02\_05\_01 – Step 3**



1. We can go back to our ***front-end*** and build our HTTP request for the GET API. We will use JavaScript ***AJAX*** to do that. Our starting point will be ***main.js***, where we will modify our JavaScript IIFE. The first thing we want to do when one of our friends is clicked is to retrieve their notes. So let’s go to startup() and set up our click event handler to call the functions that will implement our request. Right after we set our selectedUserId, we can call a function, getNotes(), which will take care of the request and response work:  
    selectedUserId = this.getAttribute('uid');  
    ***var notes = getNotes(selectedUserId, function(notes) {  
    console.log(notes);  
    });***
2. Let’s build the ***getNotes()*** function in ***main.js*** below our ***startup()*** function. It will look in its associative array ***cache***, using the key ***userId***, to see if we have to issue an AJAX request, or if we already have the notes stored. If not, it will issue the request and process the response:  
    ***function getNotes(userId, callback) {  
    if (cache[userId]) {  
    return callback(cache[userId]);  
    }  
    var xhttp = new XMLHttpRequest();  
    xhttp.onreadystatechange = function() {  
    if (xhttp.readyState == 4 && xhttp.status == 200) {  
    var notes = JSON.parse(xhttp.responseText || []);  
    cache[userId] = notes;  
    callback(notes);  
    }  
    };  
    xhttp.open('GET', "/friends/" + encodeURIComponent(userId) +   
    "/notes", true);  
    xhttp.send();  
    }***  
   Let’s run our server and browser with Developer Tools on, and on the ***Console*** tab. Go all the way through to our friends display and click a friend. We should see an empty array in our browser console. This is correct, because we have not added any notes to any friends.

**Exercise02\_05\_01 – Step 4**



1. We can stay on our ***front-end*** and build the GUI for our Notes using the DOM. In ***main.js***, we will first create a ***document*** ***fragment***. We will do this in ***startup()*** in the callback to our ***getNotes()*** function. We then call a new function to create the note elements, which we will build in the next step:  
    var notes = getNotes(selectedUserId, function(notes) {  
    ***var docFragment =   
    document.createDocumentFragment();  
    var notesElements = createNoteElements(notes);*** console.log(notes);  
    });
2. Let’s build the ***createNoteElements()*** function in ***main.js*** below our ***startup()*** function. It will return an array of <li> elements each holding a not for the clicked friend:  
    ***function createNoteElements(notes) {  
    return notes.map(function(note) {  
    var element = document.createElement('li');  
    element.className = "note";  
    element.setAttribute('contenteditable', true);  
    element.textContent = note.content;  
    return element;  
    });  
    return notes;  
    }***  
   Let’s run our server and browser with Developer Tools on, and on the ***Console*** tab. Go all the way through to our friends display and click a friend. We should have no errors and still see an empty array in our browser console.
3. Back in ***startup()*** in the callback to our ***getNotes()*** function, we can now put in the code that will add the new elements to the DOM. We then call a new function to create a button to add notes, which we will build in the next step:  
    var notesElements = createNoteElements(notes);  
    ***notesElements.forEach(function(element) {  
    docFragment.appendChild(element);  
    });  
    var newNoteButton = createAddNoteButton();  
    docFragment.appendChild(newNoteButton);*** console.log(notes);
4. Now we can build our ***createAddNoteButton*** function just above our ***getNotes()*** function:  
    ***function createAddNoteButton() {  
    var element = document.createElement('li');  
    element.className = "add-note";  
    element.textContent = "Add a new note ...";  
    return element;  
    }***
5. At this point we still have no notes to show, but we can add our document fragment into the DOM to at least show our ***Add a new note ...*** button. We will add the whole document fragment to the ***<ul>*** we created for notes in ***index.ejs*** that has the id notes:  
    docFragment.appendChild(newNoteButton);  
    ***document.getElementById('notes').  
    innerHTML = "";  
    document.getElementById('notes').  
    appendChild(docFragment);***  
   Let’s run our server and browser and go all the way through to our friends display and click a friend. We should have no errors and we should now see our new button.
6. Let’s continue to set the stage by creating a place where the user can enter a new note when the button is clicked. We will add an ***event*** *handler* to our createAddNoteButton() code that will create an ***empty*** note element, insert it ***above*** the button, and give it the ***focus***:  
    element.textContent = "Add a new note ...";  
    ***element.addEventListener('click', function() {  
    var noteElement = createNoteElements([{}])[0];  
    document.getElementById('notes').  
    insertBefore(noteElement, this);  
    noteElement.focus();  
    });***  
   Let’s run our server and browser and go all the way through to our friends display and click a friend. Now click our new button and we should get a place to add our new note.
7. As we can see, it appears we have a little ***CSS*** problem in the Notes column. The ***Logout*** button is causing us the problem. To fix it, let’s place the following style rules into style.css, near the top:  
   h1 {  
    border-bottom: 1px solid black;  
   }  
     
   ***a {  
    text-decoration: none;  
   }  
     
   .logout {  
    position: absolute;  
   }***In the next step we will do the code that actually adds notes to our friends.

**Exercise02\_05\_01 – Step 5**



1. Now that we have some of the front-end support for adding notes to our friends, we need to build the back-end part. We will build an ***endpoint*** for the ***POST*** operation to add notes. It is similar to our GET route, but the request will contain the note content. So it will be in the request ***body***. The request body is not automatically parsed by Express. So we will need to mount some Node.js middleware called the body-parser. First we require() it as follows:   
   var express = require('express');  
   ***var bodyParser = require('body-parser');***
2. Next we need to mount it with ***app.use()***:by calling its ***json()*** method:  
   app.use(require('cookie-parser')());  
     
   ***app.use(bodyParser.json());***
3. Now we can build an ***endpoint*** for the ***POST*** operation to add notes. This will be designed to respond to a request add a note for a friend that has been clicked on the front-end. Let’s scaffold that endpoint just below our ***GET*** routing as follows:  
   ***app.post('/friends/:uid/notes', ensureLoggedIn, function(req, res, next){  
    storage.insertNote(req.cookies.twitter\_id, req.params.uid,   
    req.body.content,   
    function(err, note) {  
    if (err) {  
    return res.status(500).send(err);  
    }  
    res.send(note);  
    });  
   });***
4. Let’s go to ***storage.js*** and build a new database function, ***insertNote()***, to insert a new note into the database. It will take in the ***owner’s*** Twitter Id, and the Twitter Id of the ***friend*** that was clicked. It will also need the note ***content***, which we were able to get using the body-parser. Let’s place it at the bottom of the ***exports*** object. We will be modifying the data that MongoDB returns into the format we want to send back to our application:  
    }***,  
    insertNote: function(ownerid, friendid, content, callback) {  
    database.collection('notes').insert({  
    owner\_id: ownerid,  
    friend\_id: friendid,  
    content: content  
    },  
    function(err, result) {  
    if (err) {  
    return callback(err, result);  
    }  
    callback(null, {  
    \_id: result.ops[0].\_id,  
    content: result.ops[0].content  
    });  
    });  
    }***  
   Let’s run our server and browser with Developer Tools on and check for problems.

**Exercise02\_05\_01 – Step 6**



1. We can go back to our ***front-end*** and build our HTTP request for the ***POST*** API. We will use JavaScript ***AJAX*** to do that. Our starting point will be ***main.js***, Let’s build the ***postNewNote()*** function in ***main.js*** below our ***getNotes()*** function. It will issue the AJAX request and process the response:  
    ***function postNewNote(userId, note, callback) {  
    var xhttp = new XMLHttpRequest();  
    xhttp.onreadystatechange = function() {  
    if (xhttp.readyState == 4 && xhttp.status == 200) {  
    var serverNote = JSON.parse(xhttp.responseText || {});  
    cache[userId].push(serverNote);  
    callback(serverNote);  
    }  
    };  
    xhttp.open('POST', "/friends/" + encodeURIComponent(userId)   
    + "/notes", true);  
    xhttp.setRequestHeader("Content-Type",   
    "application/json;charset=UTF-8");  
    xhttp.send(JSON.stringify(note));  
    }***  
   Let’s run our server and browser with Developer Tools on and check for problems.
2. Now we need to modify our JavaScript IIFE. The first thing we want to do is to scaffold out two event listeners in our createNoteElements() function. The first one will take care of a the new notes editing area losing the ***focus()*** and issuing a ***blur*** event. It will obtain the content of the note:  
    element.textContent = note.content;  
    ***element.addEventListener('blur', function() {  
    note.content = this.textContent;  
    console.log('blur event');  
    });*** return element;  
   Let’s run our server and browser with Developer Tools on, and on the ***Console*** tab. Go all the way through to our friends display, click a friend, click to add a note. Put some text in the edit field and leave with a tab or a mouse click elsewhere. We should see our debug ***console.log()*** that the blur event was picked up.
3. If we put some text in and press the ENTER key, we get a line break in our note. We want to prevent that, and make the ENTER key issue a ***blur*** event, accept the note, and open the next edit box. Build an event handler just below the last one to handle a ***keydown*** event and pick up the ENTER key:  
    ***element.addEventListener('keydown', function(e) {  
    if (e.keyCode == 13) {  
    e.preventDefault();  
    if (element.nextSibling.className == 'add-note') {  
    element.nextSibling.click();  
    }  
    else {  
    element.nextSibling.focus();  
    }  
    }  
    });***Put some text in the edit field and leave with an ENTER. The behavior is different now. We should see our debug ***console.log()*** that the ***blur*** event was picked up, and we should proceed to a new edit field. If we hit ENTER with no text, we stay in the blank field.
4. We can start to build out our ***blur*** event handler. If this event is generated, it means the edit field has lost the ***focus***; the user has left the field. Now there is either no content, or there is content. We will handle content later. Let’s scaffold out the case of ***no*** ***content***. This means that either there originally was content, and it was erased. Or there never was content. If it was erased, it is indicated by the note still having a ***note.\_id***. If so, we will end up deleting the note, later on. If no note\_.id, we can just leave and get rid of the edit box as follows:  
    note.content = this.textContent;  
    ***if (note.content == "") {  
    if (note.\_id) {  
      
    }  
    else {  
    document.getElementById('notes').  
    removeChild(element);  
    }  
    }***Test this with no content by both clicking outside of the field, and with ENTER. Then test it both ways with content.
5. Now let’s handle the cases where there is ***content*** in the edit field, and the field loses the focus. Two situations may be applicable. If there is a ***note.\_id***, then this is going to be an ***update*** of an existing record, which we will take care of later. If no note.\_id, it means we want to post a ***new*** ***note***, and that is the one we want to handle now. Let’s do that with an ***else*** statement below our previous code which will call our ***postNewNote()*** function to issue the AJAX request:  
    ***else if (!note.\_id) {  
    postNewNote(selectedUserId,  
    {content: this.textContent},  
    function(newNote) {  
    note.\_id = newNote.\_id;  
    });  
    }***Pick a friend and add a couple of notes. Click on another friend, then go back to the one with notes. Everything should show up. Confirm that the data is correct in RoboMongo. Rerun the server and browser from scratch, click the friend we added notes to, and all of the notes should show.

**Exercise02\_05\_01 – Step 7**



1. Now that we have the abilities to add and display notes to our friends, we need to build the back-end ability to update notes. We will build an ***endpoint*** for the ***PUT*** operation to update notes. This will be designed to respond to a request to update a note for a friend that has been clicked on the front-end. Let’s scaffold that endpoint just below our ***POST*** routing as follows:  
   ***app.put('/friends/:uid/notes/:noteid', ensureLoggedIn, function(req,   
    res){  
    var noteId = req.params.noteid;  
    storage.updateNote(req.params.noteid, req.cookies.twitter\_id,   
    req.body.content,   
    function(err, note) {  
    if (err) {  
    return res.status(500).send(err);  
    }  
    res.send({  
    \_id: note.\_id,  
    content: note.content  
    });  
    });  
   });***
2. In ***storage.js***, we are going to need some additional support from the MongoDB module. We need the ability to create a MongoDB style record ID. This is called an ***ObjectID*** in MongoDB, and is a unique record identifier. At the top of the file, add the following ***require()*** statement to get the ***class***:  
   var MongoClient = require('mongodb').MongoClient;  
   ***var ObjectID = require('mongodb').ObjectID;***
3. Let’s go to ***storage.js*** and build a new database function, ***updateNote()***, to update a note in the database. It will take in the unique ***noteId*** and the owner’s ***Twitter Id***. It will also need the note ***content*** from the body-parser. Let’s place it at the bottom of the ***exports*** object. We will be modifying the data that MongoDB returns by making it a new ObjectID. We will also be using the MongoDB ***$set*** operator. This is one of a few operators MongoDB provides for updates, which we will learn as necessary:  
    }***,  
    updateNote: function(noteId, ownerId, content, callback) {  
    database.collection('notes').updateOne({  
    \_id: new ObjectID(noteId),  
    owner\_id: ownerId  
    },  
    {  
    $set: { content: content }  
    },  
    function(err, result) {  
    if (err) {  
    return callback(err);  
    }  
    database.collection('notes').findOne({  
    \_id: new ObjectID(noteId)  
    }, callback);  
    });  
    }***  
   Let’s run our server and browser with Developer Tools on and check for problems.

**Exercise02\_05\_01 – Step 8**



1. We can go back to our ***front-end*** and build our HTTP request for the ***PUT*** API. We will use JavaScript ***AJAX*** to do that. Our starting point will be ***main.js***, Let’s build the ***putNote()*** function in ***main.js*** below our ***postNewNote()*** function. It will issue the AJAX request and process the response:  
    ***function putNote(userId, note, callback) {  
    var xhttp = new XMLHttpRequest();  
    xhttp.onreadystatechange = function() {  
    if (xhttp.readyState == 4 && xhttp.status == 200) {  
    var serverNote = JSON.parse(xhttp.responseText || {});  
    callback(serverNote);  
    }  
    };  
    xhttp.open('PUT', "/friends/" + encodeURIComponent(userId) +   
    "/notes/" + encodeURIComponent(note.\_id), true);  
    xhttp.setRequestHeader("Content-Type",   
    "application/json;charset=UTF-8");  
    xhttp.send(JSON.stringify(note));  
    }***  
   Let’s run our server and browser with Developer Tools on and check for problems.
2. Now we need to modify ***createNoteElements()*** in our JavaScript IIFE. We need to further build out our ***blur*** event handler. When the event is generated, we need to pick up if there is ***content*** and also an existing ***note.\_id***. If so, we will end up updating the note, so we add a last ***else*** statement as follows:  
    else if (!note.\_id) {  
    postNewNote(selectedUserId,  
    {content: this.textContent},  
    function(newNote) {  
    note.\_id = newNote.\_id;  
    });  
    }  
    ***else {  
    putNote(selectedUserId, note, function() {} );  
    }***Pick a friend with a couple of notes and modify one of them. Click on another friend, then go back to the one with notes. Everything should show up. Confirm that the data is correct in RoboMongo. Rerun the server and browser from scratch, click the friend we added notes to, and all of the notes should show with any modifications.

**Exercise02\_05\_01 – Step 9**



1. The last part of our APIs will be a delete ability for notes. We will build an ***endpoint*** for the ***DELETE*** operation to handle this. This will be designed to respond to a request to delete a note for a friend that has been clicked on the front-end. Let’s scaffold that endpoint just below our ***PUT*** routing as follows:  
   ***app.delete('/friends/:uid/notes/:noteid', ensureLoggedIn, function(req, res){  
    var noteId = req.params.noteid;  
    storage.deleteNote(req.params.noteid, req.cookies.twitter\_id,   
    function(err, note) {  
    if (err) {  
    return res.status(500).send(err);  
    }  
    res.sendStatus(200);  
    });  
   });***
2. Let’s go to ***storage.js*** and build a new database function, ***deleteNote()***, to delete a note in the database. It will take in the unique ***noteId*** and the owner’s ***Twitter Id***. Let’s place it at the bottom of the ***exports*** object. We will be modifying the data that MongoDB returns by making it a new ObjectID:  
   }***,  
    deleteNote: function(noteId, ownerId, callback) {  
    database.collection('notes').deleteOne({   
    \_id: new ObjectID(noteId),  
    owner\_id: ownerId  
    }, callback);  
    }***  
   Let’s run our server and browser with Developer Tools on and check for problems.

**Exercise02\_05\_01 – Step 10**



1. We can go back to our ***front-end*** and build our HTTP request for the ***DELETE*** API. We will use JavaScript ***AJAX*** to do that. Our starting point will be ***main.js***, Let’s build the ***deleteNote()*** function in ***main.js*** below our ***putNote()*** function. It will issue the AJAX request and process the response:  
    ***function deleteNote(userId, note, callback) {  
    var xhttp = new XMLHttpRequest();  
    xhttp.onreadystatechange = function() {  
    if (xhttp.readyState == 4 && xhttp.status == 200) {  
    cache[userId] = cache[userId].filter(function(localNote) {  
    return localNote.\_id != note.\_id;  
    });  
    callback();  
    }  
    };  
    xhttp.open('DELETE', "/friends/" +   
    encodeURIComponent(userId) + "/notes/" +   
    encodeURIComponent(note.\_id), true);  
    xhttp.send();  
    }***Let’s run our server and browser with Developer Tools on and check for problems.
2. Now we need to modify ***createNoteElements()*** in our JavaScript IIFE. We need to further build out our ***blur*** event handler. When the event is generated, we need to pick up if there is ***no content***, but there is also an existing ***note.\_id***. If so, we will end up deleting the note, so we finish building out the empty ***if*** statement as follows:  
    if (note.content == "") {  
    ***if (note.\_id) {  
    deleteNote(selectedUserId, note, function() {  
    document.getElementById('notes').  
    removeChild(element);  
    });   
    }***Pick a friend with a couple of notes and remove the content from one of them. Remove the focus. Click on another friend, then go back to the one with notes. The deleted one should still be gone. Confirm that the data is correct in RoboMongo. Rerun the server and browser from scratch, click the friend we deleted notes from, and all of the notes should be correct.

**Exercise02\_05\_01 – Step 11**



1. Now we will look at another means of ***maintaining*** ***state***. We will go back to our ***front-end*** and convert part of the application to make use of ***cookies***. Open ***authenticator.js*** and add the following code:  
    twitterCredentials.twitter\_id = data.id\_str;  
    ***res.cookie('access\_token', oauth\_access\_token,   
    { httponly: true });  
    res.cookie('access\_token\_secret',   
    oauth\_access\_token\_secret, { httponly: true });  
    res.cookie('twitter\_id', data.id\_str, { httponly: true });***Run the server and browser with Developer Tools on, in the Application tab, looking at cookies. The new cookies should appear on the tunnel route after authorization.
2. Now we need to test our cookies and check their values for authentication. Go to ***index.js*** and we will modify the old manual url ***/tweet*** route. Change the credentials code as follows:  
   ***//*** var credentials = authenticator.getCredentials();  
   ***//*** if (!credentials.access\_token || !credentials.access\_token\_secret) {  
   ***//*** return res.sendStatus(401);  
   ***//***  }  
    **if (!req.cookies.access\_token || !req.cookies.access\_token\_secret)   
    {  
    return res.sendStatus(401);  
    }**  
    var url = 'https://api.twitter.com/1.1/statuses/update.json';  
    ***authenticator.post(url, req.cookies.access\_token,   
    req.cookies.access\_token\_secret,   
    {  
    status: "Testing cookies on Twitter"  
    },***Let’s run our server and browser with Developer Tools on and in the Application tab looking at cookies. Manually modify the URL to go to the ***/tweet*** route. Make sure that you get ***“tweet successful!”*** in the browser. Check Twitter to make sure that your Tweet has appeared. Log out of your app.
3. Now we can update our ***/search*** route as follows:  
   ***//*** var credentials = authenticator.getCredentials();  
   ***//*** if (!credentials.access\_token || !credentials.access\_token\_secret) {  
   ***//*** return res.sendStatus(401);  
   ***//***  }  
    ***if (!req.cookies.access\_token || !req.cookies.access\_token\_secret)   
    {  
    return res.sendStatus(401);  
    }***  
    var url = 'https://api.twitter.com/1.1/search/tweets.json';  
    var query = querystring.stringify({ q: 'BMW' });  
    url += "?" + query;  
    ***authenticator.get(url, req.cookies.access\_token,   
    req.cookies.access\_token\_secret,***Let’s run our server and browser with Developer Tools on and in the Application tab looking at cookies. Manually modify the URL to go to the ***/search*** route. Make sure that you get data in the browser. Log out of your app.
4. Now we can update our ***/friends*** route as follows:  
   ***//*** var credentials = authenticator.getCredentials();  
   ***//*** if (!credentials.access\_token || !credentials.access\_token\_secret) {  
   ***//*** return res.sendStatus(401);  
   ***//***  }  
    ***if (!req.cookies.access\_token || !req.cookies.access\_token\_secret)   
    {  
    return res.sendStatus(401);  
    }***  
    var url = 'https://api.twitter.com/1.1/friends/list.json';  
    if (req.query.cursor) {  
    url += "?" + querystring.stringify({ cursor: req.query.cursor });  
    }  
    ***authenticator.get(url, req.cookies.access\_token,   
    req.cookies.access\_token\_secret,***Let’s run our server and browser with Developer Tools on and in the Application tab looking at cookies. Manually modify the URL to go to the ***/friends*** route. Make sure that you get data in the browser. Log out of your app.
5. Now we can update our ***/*** main route as follows:  
   ***//*** var credentials = authenticator.getCredentials();  
   ***//*** if (!credentials.access\_token || !credentials.access\_token\_secret) {  
   ***//*** return res.redirect(‘/login’);  
   ***//***  }  
    ***if (!req.cookies.access\_token || !req.cookies.access\_token\_secret)   
    {  
    return res.redirect(‘/login’);  
    }***  
    . . . . . .   
    ***storage.getFriends(req.cookies.twitter\_id, function(err, friends) {***Let’s run our server and browser with Developer Tools on and in the Application tab looking at cookies. Make sure to clear all of the cookies an log out. Log back in and make sure everything is authorizing as normal, and we receive our friends and notes display. Again clear the cookies and log out of your app.
6. Now we will modify our waterfall in ***index.js*** as follows:  
   function renderMainPageFromTwitter(req, res) {  
    ***var credentials = authenticator.getCredentials();***
7. We can proceed to modify the ***whilst*** loop as follows:  
    var url = 'https://api.twitter.com/1.1/friends/ids.json';  
    url += "?" + querystring.stringify({ user\_id:   
    ***req.cookies.twitter\_id***, cursor: cursor});  
    ***authenticator.get(url, req.cookies.access\_token,   
    req.cookies.access\_token\_secret,***
8. Now let’s do the same with the ***times*** loop as follows:  
    var url = 'https://api.twitter.com/1.1/users/lookup.json';  
    url += "?" + querystring.stringify({ user\_id:   
    getHundredIds(n).join(',')});  
    ***authenticator.get(url, req.cookies.access\_token,   
    req.cookies.access\_token\_secret,***Run the server and browser with Developer Tools on and in the Application tab looking at cookies. Make sure to clear all of the cookies and log out. Log back in and make sure everything is authorizing as normal, and we receive our friends and notes display. Again clear the cookies and log out of your app.
9. For security, we will clear the cookies in ***/logout*** in ***index.js***:  
   app.get('/logout', function(req, res) {  
    authenticator.clearCredentials();  
    ***res.clearCookie('access\_token');  
    res.clearCookie('access\_token\_secret');  
    res.clearCookie('twitter\_id');***Login and the cookies should be written to the tunnel rout. Logout and the cookies should be gone on the route.
10. Now let’s use the cookies on our ***ensureLoggedIn()*** middleware, and on all of our notes rote as follows:  
    ***// var credentials = authenticator.getCredentials();  
     if (!req.cookies.access\_token || !req.cookies.access\_token\_secret   
     || !req.cookies.twitter\_id) {  
     storage.getNotes(req.cookies.twitter\_id, req.params.uid,   
     function(err, notes) {  
     storage.insertNote(req.cookies.twitter\_id, req.params.uid,   
     req.body.content,  
     storage.updateNote(req.params.noteid, req.cookies.twitter\_id,   
     req.body.content,  
     storage.deleteNote(req.params.noteid, req.cookies.twitter\_id,***  
    Log in and test all of the notes features.