```
//requires for web server
var express = require('express');
var app = express();
var busboy = require('connect-busboy');
var path = require('path');
var request = require('request');
//require for additional filesystem functions
var fs = require('fs-extra');
var http = require('http').Server(app);
//require for sockets
var io = require('socket.io')(http);
//require for hashing algorithm
var crypto = require('crypto');

//set the directory where files are served from and uploaded to
var dir = __dirname + '/files/';

//data structure to contain all file metadata
var globalfiles;

//endpoint for BannerWeb to perform authentication
var BANNER_URL = "https://bnrlnxss1p.ltu.edu/BannerPPRDS/";

app.use(busboy());
//files in the public directory can be directly queried for via HTTP
app.use(express.static(path.join(__dirname, 'public')));

//if a class has some files, return it. otherwise create an empty object and return that
var checkClassDefined = function(activeClass) {
    if (!globalfiles[activeClass]) {
        globalfiles[activeClass] = {};
    }
    return globalfiles[activeClass];
}

//strips one class out of the massive data structure so as not to get unweildy
var justOneClass = function(activeClass) {
    var data = {};
    data[activeClass] = checkClassDefined(activeClass);
    return data;
}

//write out the file metadata into a master index file
var writeIndexFile = function() {
    for (var oneclass in globalfiles) {
        //remove all empty classes from the data structure before writing
        if (Object.keys(globalfiles[oneclass]).length == 0) {
            delete globalfiles[oneclass];
        }
    }
    //if there is at least one class left to write
    if (Object.keys(globalfiles).length > 0) {
        //write the object out as a straight JSON object
        fs.writeFileSync(dir + "files.json", JSON.stringify(globalfiles));
    } else {
        try {
```

```javascript
            //delete the master index file if there are no non-empty classes to write
            fs.unlinkSync(dir + "files.json");
        } catch (e) {
            console.log(e);
            console.log("writeIndexFile: error removing index file");
        }
    }
}

//auxiliary function to the move endpoint that can extract whole chunks for folders or files
//recursively calls itself as it traverses down the path
var getFileOrFolder = function(files, path, isFolder, doCopy) {
    if (!path) {
        if (doCopy) {
            var copied = JSON.parse(JSON.stringify(files));
            files = {};
            return copied;
        }
        return files;
    }
    var parts = path.split("/");
    //if we have reached the bottom-most part of the path...
    if (parts.length == 1) {
        //if we're looking for a file, check there are files defined in this folder!
        if (!isFolder && files.files) {
            //iterate through all the files in the folder
            for (var i = 0; i < files.files.length; i++) {
                //if the hashes match, do stuff
                if (files.files[i].hash == parts[0]) {
                    //if we want a copy of the chunk, make one and return it while deleting the original
                    if (doCopy) {
                        //trick to copy a JSON object is to make it a string and then throw that in the constructor of a new JSON object
                        var copied = JSON.parse(JSON.stringify(files.files[i]));
                        //remove this file from the object
                        files.files.splice(i, 1);
                        //if this file was the last file in this folder, remove the files array
                        if (files.files.length == 0) {
                            delete files.files;
                        }
                        //return a copy of the desired file
                        return copied;
                    }
                    //otherwise just return a reference to this object where it exists in the master
                    return files.files[i];
                }
            }
            //if we're looking for a folder, check there are folders defined in this folder!
        } else if (isFolder && files.folders) {
            //check to see if there is a folder with the same name as the one we're looking for
            if (files.folders[parts[0]]) {
                //if we want a copy of the chunk, make one and return it while deleting the original
                if (doCopy) {
                    //tricky copy method
                    var copied = JSON.parse(JSON.stringify(files.folders));
                    //remove this folder from the object
                    delete files.folders[parts[0]];
                    //if this folder was the last folder in this folder, remove the folders array
                    if (Object.keys(files.folders).length == 0) {
                        delete files.folders;
                    }
                    //return a copy of the desired folder
                    return copied;
                }
                //otherwise just return a reference to this object where it exists in the master
                return files.folders;
            }
        } else {
            return files;
        }
    } else {
        //check to make sure the next folder down exists before calling into it
        if (files.folders) {
            if (files.folders[parts[0]]) {
                //recursively call the function again on the next folder down
                //keep the same relative parameters as what were passed originally
                return getFileOrFolder(files.folders[parts[0]], parts.slice(1).join("/"), isFolder, doCopy);
            }
        } else {
            //malformed path was provided
            return {};
        }
    }
}

//function to recursively construct physical folders
var createFolder = function(foldername, activeClass) {
    //create the root relative to the class desired
    var root = dir + activeClass;
    if (foldername) {
        //split the foldername on forward slashes
        var parts = foldername.split("/");
        //for each part, create one level of folder deeper
        for (var i = 0; i < parts.length; i++) {
            if (parts[i]) {
                root += "/" + parts[i];
                try {
                    //does the actual folder creation
                    fs.mkdirSync(root);
                } catch (e) { }
            }
        }
    }
    try {
        fs.mkdirSync(root);
    } catch (e) { }
    //return not used for anything, but could be extended in the future
    return root + "/";
};

//recursively adds the appropriate structure to the master files data structure for
a new addition
//if the file part is null, then it is interpreted as for a new folder instead
//at each stage, there are checks for whether the appropriate data structure are in
place, and if not, creates it and calls the function again
var addWithoutCollisions = function(files, foldername, file) {
    var parts = foldername.split("/");
    if (parts.length > 1 && parts[0] !== "") {
        if (files.folders) {
            if (files.folders[parts[0]]) {
                if (parts.length == 2) {
                    if (file && files.folders[parts[0]].files) {
                        for (var i = 0; i < files.folders[parts[0]].files.length; i
                        ++) {
                            if (files.folders[parts[0]].files[i].hash == file.hash) {
                                files.folders[parts[0]].files.splice(i, 1);
                                break;
                            }
                        }
                        files.folders[parts[0]].files.push(file);
                        return;
                    } else {
                        if (file) {
                            files.folders[parts[0]].files = [];
                        } else {
                            return;
                        }
                        addWithoutCollisions(files, foldername, file);
                    }
                } else {
                    addWithoutCollisions(files.folders[parts[0]], parts.slice(1).join
                    ("/"), file);
                }
            } else {
                files.folders[parts[0]]= {};
                addWithoutCollisions(files, foldername, file);
            }
        } else {
            files.folders = {};
            addWithoutCollisions(files, foldername, file);
        }
    } else {
        if (file) {
            if (files.files) {
                for (var i = 0; i < files.files.length; i++) {
                    if (files.files[i].hash == file.hash) {
                        files.files.splice(i, 1);
                        break;
                    }
                }
            } else {
```

```
                files.files = [];
            }
            files.files.push(file);
        }
        return;
    }
};

//auxiliary function that gets the real filename given a path with a hash
//recursively calls itself
var download = function(files, filename) {
    //split on forward slash to loop down the path
    filename = filename.split("/");
    //if there were no forward slashes, then just the hash is left
    if (filename.length == 1) {
        //if the files structure actually has some files in it at this level...
        if (files.files) {
            //loop through all the files and compare their hashes with the parameter
            passed
            for (var i = 0; i < files.files.length; i++) {
                if (files.files[i].hash == filename[0]) {
                    //return the "real" filename of the file
                    return files.files[i].name;
                }
            }
        }
        //if there are more folders to traverse
    } else if (filename.length > 1) {
        //make a recursive call to the next level down
        return download(files.folders[filename[0]], filename.slice(1).join("/"));
    }
};

//function to merge folders recursively down the path
var mergeFolders = function(activeClass, goodPath, oldPath, goodCopy, newOne) {
    //the last token after splitting on forward slash is the name of the folder
    var name = goodPath.split("/")[goodPath.split("/").length - 1];
    try {
        //create a physical directory at the good path for the name of the folder
        fs.mkdirSync(dir + activeClass + "/" + goodPath);
    } catch (e) { }
    //if the name isn't currently in the data structure being merged into, then
    create an empty object there
    if (!goodCopy[name]) {
        goodCopy[name] = {};
    }
    //loop through all the files and merge them in from the old to the new
    if (newOne.files) {
        for (var i = 0; i < newOne.files.length; i++) {
            if (!goodCopy[name].files) {
                goodCopy[name].files = [];
            }
            //if there are any duplicate files (hash duplicates = identical content)
            that would conflict as they are copied/merged over, delete them
            for (var j = 0; j < goodCopy[name].files.length; j++) {
                if (newOne.files[i].hash == goodCopy[name].files[j].hash) {
                    fs.unlinkSync(dir + activeClass + "/" + goodPath + "/" + goodCopy
                    [name].files[j].hash + ".file");
                    goodCopy[name].files.splice(j, 1);
                    break;
                }
            }
            //rather than copy/delete or move, renaming them including the whole path has the same effect
            console.log("Renaming " + oldPath + "/" + newOne.files[i].hash + ".file" + " to " + goodPath + "/" + newOne.files[i].hash + ".file");
            fs.renameSync(dir + activeClass + "/" + oldPath + "/" + newOne.files[i].hash + ".file", dir + activeClass + "/" + goodPath + "/" + newOne.files[i].hash + ".file");
            //push the file into the new place
            goodCopy[name].files.push(newOne.files[i]);
        }
    }
    //loop through all the folders and call the same merge function on each
    for (var foldername in newOne.folders) {
        //if the place being merged to doesn't contain the folder that is coming in, create an empty object there
        if (!goodCopy[name].folders) {
            goodCopy[name].folders = {};
        }
        mergeFolders(activeClass, goodPath + "/" + foldername, oldPath + "/" + foldername, goodCopy[name].folders, newOne.folders[foldername]);
        continue;
    }
    //if the folder that was merged is empty, then delete it
    //this works well with the recursive nature of this function because folders are cleaned from the bottom-up
    if (fs.readdirSync(dir + activeClass + "/" + oldPath).length == 0) {
        console.log("Removing empty directory " + oldPath);
        try {
        fs.rmdirSync(dir + activeClass + "/" + oldPath);
        } catch (e) {
            console.log("Error removing directory " + oldPath);
        }
    }
};

//auxiliary function to delete folders in the file data structure
//recursive until it reaches the desired folder and then deletes the whole chunk underneath that
var folderDeleter = function(files, path) {
    path = path.split("/");
    if (path.length == 1) {
        if (files.folders && files.folders[path[0]]) {
            delete files.folders[path[0]];
            if (Object.keys(files.folders).length == 0) {
                delete files.folders;
            }
            return;
        }
    } else if (path.length > 1) {
        return folderDeleter(files.folders[path[0]], path.slice(1).join("/"));
    }
};
```

```

//auxiliary function to delete physical folders on disk
//recursive since non-empty folders cannot be deleted until all files/folders are deleted in it
var deleteFolderRecursive = function (path) {
    if (fs.existsSync(path)) {
        //for each item in the folder specified, check to see if it's a file and delete it; if a folder, call this function on that folder
        fs.readdirSync(path).forEach(function(file, index) {
            var curPath = path + "/" + file;
            if (fs.lstatSync(curPath).isDirectory()) {
                deleteFolderRecursive(curPath);
            } else {
                fs.unlinkSync(curPath);
            }
        });
        //after all files and folders should have been removed above, then this folder can be removed too
        //this is helpful in a recursive context because folders/files are removed from the bottom up
        fs.rmdirSync(path);
    }
};

//auxiliary function that returns a boolean whether after deleting a file the folder containing that file should be kept
//serves the purpose of deleting folders that no longer contain files or folders
//recursive so this can be applied at all levels to the top
var deleter = function(files, filename, activeClass, origfilename) {
    filename = filename.split("/");
    if (filename.length == 1) {
        for (var i = 0; i < files.files.length; i++) {
            if (files.files[i].hash == filename[0]) {
                files.files.splice(i, 1);
                try {
                    origfilename = origfilename == "" ? "" : origfilename + "/";
                    //physically delete the file
                    fs.unlinkSync(dir + activeClass + "/" + origfilename + filename[0] + ".file");
                } catch (e) {
                    console.log(e);
                    console.log("Deleter: error removing file");
                }
                if (files.files.length == 0) {
                    delete files.files;
                }
                //if there is nothing left then return false to signal impending deletion
                if ((!files.files && !files.folders) || (!files.files && Object.keys(files.folders).length == 0)) {
                    return false;
                }
                //if none of the above conditions are met, return true
                return true;
            }
        }
    }
    //if calling this function returns false, then delete the relevant containers
    if (!deleter(files.folders[filename[0]], filename.slice(1).join("/"), activeClass
    , origfilename + "/" + filename[0])) {
        delete files.folders[filename[0]];
        if (Object.keys(files.folders).length == 0) {
            delete files.folders;
        }
        try {
            console.log("Removing directory: " + filename[0]);
            //physically remove the directory
            fs.rmdirSync(dir + activeClass + "/" + origfilename + "/" + filename[0]);
        } catch (e) {
            console.log(e);
        }
        //if there are no files or folders in the folder, it shouldn't be kept
        if (!files.files && !files.folders) {
            return false;
        }
    }
    //if the function returned true then return true also
    return true;
};

//create an endpoint for downloading files
app.get('/download', function(req, res){
    try {
        //extract the file to fetch and the class it's in from the request
        var hash = req.query.hash;
        var activeClass = req.query.active;
        if (hash && activeClass) {
            //get the filename by searching with the appropriate path in the
            appropriate class structure
            var filename = download(checkClassDefined(activeClass), hash);
            //if the filename exists, build the response by piping the file back
            with the filename
            if (filename) {
                console.log("Downloading: " + hash + " -> " + filename);
                var file = dir + activeClass + "/" + hash + ".file";
                //setting the header for attachment type with the name lets the
                browser know what's going on
                res.setHeader('Content-disposition', 'attachment; filename=' +
                filename);
                var filestream = fs.createReadStream(file);
                //pipe the file into the response
                filestream.pipe(res);
            }
        }
    } catch (e) { console.log(e); res.redirect('back'); }
});

//create an endpoint for deleting files
app.get('/delete', function(req, res){
    try {
        //extract the file to delete and the class it's in from the request
        var hash = req.query.hash;
        var activeClass = req.query.active;
        if (hash && activeClass) {
            //calling the deleter function actually deletes the file and additionally returns whether the class itself (since the function is recursive) should be kept
            if (!deleter(checkClassDefined(activeClass), hash, activeClass, "")) {
                try {
                    delete globalfiles[activeClass];
                    fs.rmdirSync(dir + activeClass);
                } catch (e) {
                    console.log("Could not remove folder " + activeClass);
                }
            }
            //update the master index file
            writeIndexFile();
            //send out an updated list of files to clients
            io.emit('message', justOneClass(activeClass));
            res.send('File deleted');
        }
    } catch (e) { console.log(e); res.send("Error deleting"); }
});

//create an endpoint for moving a file or folder
//can also be used to rename folders
app.get('/move', function(req, res) {
    try {
        //extract source, destination, and class from the request
        var source = req.query.source;
        var destination = req.query.destination;
        var activeClass = req.query.active;
        if (source && destination && activeClass) {
            if (source[source.length - 1] == '/') { //folder
                source = source.substring(0, source.length - 1);
                destination = destination.substring(0, destination.length - 1);
                var sourcefolder = source.split("/")[source.split("/").length - 1];
                //extract an object representing the source
                var obj = getFileOrFolder(checkClassDefined(activeClass), source, true, true)[sourcefolder];
                //add an entry corresponding to the destination
                addWithoutCollisions(checkClassDefined(activeClass), destination + "/new", null);
                //merge the destination with the source
                mergeFolders(activeClass, destination, source, getFileOrFolder(
                checkClassDefined(activeClass), destination, true, false), obj);
                console.log("Folder " + source + " physically moved to " + destination);
            } else { //file
                //physically rename the file with the new path
                fs.renameSync(dir + activeClass + "/" + source + ".file", dir + activeClass + "/" + destination + ".file");
                console.log("File " + source + " physically moved to " + destination);
                //grab a reference to the file object and delete it
                var obj = getFileOrFolder(checkClassDefined(activeClass), source, false, true);
                //add the file reference back in at the destination
                addWithoutCollisions(checkClassDefined(activeClass), destination, obj);
            }
            //update the master index file
            writeIndexFile();
            //send out an updated list of files to clients
            io.emit('message', justOneClass(activeClass));
            res.send('Moved');
        }
    } catch (e) { console.log(e); res.send("Error moving"); }
});

//create an endpoint for creating a new folder
//normally the data structure is kept clean of empty folders, but in this case we explicitly allow them... or we wouldn't be able to create folders!
app.get('/newfolder', function(req, res){
    try {
        var path = req.query.path;
        var activeClass = req.query.active;
        if (path && activeClass) {
            //call the auxiliary function to physically create the new folder
            createFolder(path, activeClass);
            //call the auxiliary function to create space in the data structure
            addWithoutCollisions(checkClassDefined(activeClass), path + "/new", null);
            //update the master index files
            writeIndexFile();
            //send out an updated list of files to clients
            io.emit('message', justOneClass(activeClass));
            res.send('Folder created');
        }
    } catch (e) { console.log(e); res.send("Error creating"); }
});

//create an endpoint for deleting a folder
app.get('/deletefolder', function(req, res){
    try {
        var path = req.query.path;
        var activeClass = req.query.active;
        if (path && activeClass) {
            //remove the folder from the data structure and clean any empty folders above this one
            folderDeleter(checkClassDefined(activeClass), path);
            //physically delete the folder from disk
            deleteFolderRecursive(dir + activeClass + "/" + path);
            //update the master index file
            writeIndexFile();
            //send out an updated list of files to clients
            io.emit('message', justOneClass(activeClass));
            res.send('Folder deleted');
        }
    } catch (e) { console.log(e); res.send("Error deleting"); }
});

//create an endpoint for uploading a file
app.route('/upload').post(function (req, res, next) {
    var fstream;
    var title;
    var revealDate;
    var theFolder;
    var alreadyUploaded = [];
    req.pipe(req.busboy);
```

```
        //when a field is encountered in the incoming form data
        req.busboy.on('field',function (fieldname, val){
            if (fieldname == 'title') {
                //grab the file title if there is one
                //this is something that was in earlier iterations of the product but
                aren't anymore
                title = val;
            } else if (fieldname == 'reveal') {
                //grab the file reveal date
                revealDate = val;
            } else if (fieldname == 'folder') {
                //replace non alphanumeric characters in the folder name
                val = val.replace(/([^a-z \/0-9]+)/gi, '');
                if (val.length > 1 && val.substring(0, 1) == '/') {
                    //strip off the leading forward-slash
                    val = val.substring(1);
                }
                theFolder = val;
                console.log("Folder: " + theFolder);
                if (theFolder.substring(0, 1) == '/') {
                    //if it's just a forward-slash then we're uploading to the root folder
                    theFolder = "";
                }
            }
        });
        //when a file is encountered, stream it in and save it appropriately
        req.busboy.on('file', function (activeClass, file, filename) {
            for (var i = 0; i < alreadyUploaded.length; i++) {
                //don't upload files that have the same name as a file that's already
                been uploaded in this batch
                if (alreadyUploaded[i] == filename) {
                    filename = undefined;
                    break;
                }
            }
            if (filename) {
                alreadyUploaded.push(filename);
                try {
                    fs.mkdirSync(dir + activeClass);
                } catch (e) { }
                try {
                    console.log("Uploading: " + filename);
                    //files are stored by their md5 hash
                    var hash = crypto.createHash('md5');
                    fstream = fs.createWriteStream(dir + filename);
                    file.on('data', function(chunk) {
                        //as chunks come in, update the hash
                        hash.update(chunk);
                    });
                    //when the file has finished being streamed in...
                    fstream.on('close', function () {
                        //create a file object to be pushed into the main data structure
                        var tempfile = {};
                        tempfile["name"] = filename;
                        if (title) {
                            tempfile["title"] = title;
                        }
                        tempfile["hash"] = hash.digest('hex');
                        //mark NOW as the upload date for the file
                        tempfile["date"] = Date.now();
                        if (revealDate) {
                            tempfile["reveal"] = revealDate;
                        } else {
                            //if there was not a future reveal date specified for the file, use the file's date instead (about NOW)
                            tempfile["reveal"] = tempfile["date"];
                        }
                        try {
                            console.log("Upload Finished of " + tempfile.name);
                            createFolder(theFolder, activeClass);
                            var newName;
                            //if the folder isn't the root folder, add that into the path
                            if (theFolder) {
                                newName = dir + activeClass + "/" + theFolder + "/" + tempfile.hash + ".file";
                            } else {
                                //if the folder is the root folder, just save directly into the class's root folder
                                newName = dir + activeClass + "/" + tempfile.hash + ".file";
                            }
                            //since the file was uploaded with its "real" filename, rename it to its hash and put it in the appropriate folder
                            fs.rename(dir + tempfile.name, newName, function (err) {
                                if (err) throw err;
                                if (theFolder) {
                                    console.log('Renamed to ' + activeClass + "/" + theFolder + "/" + tempfile.hash);
                                } else {
                                    console.log('Renamed to ' + activeClass + "/" + tempfile.hash);
                                }
                                //add the file object into the data structure where appropriate
                                addWithoutCollisions(checkClassDefined(activeClass), theFolder + "/" + tempfile.hash, tempfile);
                                //update the master list of files
                                writeIndexFile();
                                //update the clients with the new list of files
                                io.emit('message', justOneClass(activeClass));
                            });
                        } catch (e) {
                            console.log(e);
                        }
                    });
                    //pipe the incoming file to the stream which is listening for chunk updates, etc.
                    file.pipe(fstream);
                } catch (e) {
                    console.log("Error during upload");
                }
            } else {
                //if there isn't a filename (since it was manually set to undefined because a file with the same name was already uploaded) then skip
                uploading the file
                file.resume();
            }
        });
        //when the entire request has been processed, send back a success header and close the connection
        req.busboy.on('finish', function () {
            res.writeHead(200, { Connection: 'close', Location: '/' });
            res.end();
        });
    });

    //when a new client connects
    io.on('connection', function(socket) {
        //add a listener for when the client sends a message asking for which classes it can access
        socket.on('classes', function(msg) {
            //convert the requested term into the Banner equivalent
            msg.term = msg.term == "Fall" ? "10" : msg.term == "Spring" ? "20" : "30";
            //manual response for Admin for demonstration purposes
            if (msg.sid == "Admin") {
                var myresponse = {};
                myresponse["classes"] = ["Introduction to C - 04","Computer Science 2 - 02","Java (Honors) - 02","Web Server Programming - 01","Introduction to Bioinformatics - 01","Collaborative Research Proj 2 - 01"];
                myresponse["admin"] = true;
                socket.emit('classes', myresponse);
            } else {
                //for any user besides the hard-coded admin, query BannerWeb for their list of classes and whether they should be able to administer them, etc.
                makeBannerRequest(msg.sid, msg.pin, msg.year + msg.term, socket);
            }
        });
        //add a listener for when the client sends a message asking for the files/folders for a certain class
        socket.on('message', function(msg) {
            if (msg != null) {
                console.log('Received ' + msg);
                //send out the list of files/folders that the client requested, just to them and not everyone
                socket.emit('message', justOneClass(msg));
            }
        });
    });

    //function to query BannerWeb for a user's class list
    //checks to see if the user is faculty and fetches the classes they're teaching...
    //...otherwise if they're a student then it fetches their class list and makes them non-admins
    var makeBannerRequest = function(sid, pin, term, socket) {
        //make a GET request to authenticate the user and obtain a session id to use in subsequent requests
        request({
            url: BANNER_URL + 'twbkwbis.P_ValLogin?sid=' + sid + "&PIN=" + pin,
            method: 'GET',
            headers: {
                'Cookie': 'TESTID=set'
            },
        },
        function (error, response, body) {
            if (!error && response.statusCode == 200) {
                //parse out the session id from the response cookies
                var cookie = response.headers['set-cookie'][0];
                cookie = cookie.split(";");
                var success = false;
                for (var i = 0; i < cookie.length; i++) {
                    if (cookie[i].split("=")[0] == "SESSID" && cookie[i].split("=")[1]) {
                        success = true;
                        console.log("Successfully logged in: " + sid + ", " + pin);
                        //make a second request for the faculty schedule for the user
                        request({
                            url: BANNER_URL + 'bwlkifac.P_FacSched?term_in=' + term,
                            method: 'GET',
                            headers: {
                                //use the aforementioned session id to authenticate the query
                                'Cookie': cookie[i].split("=")[0] + "=" + cookie[i].split("=")[1]
                            },
                        },
                        function (error, response, body) {
                            if (!error && response.statusCode == 200) {
                                var myresponse = {};
                                var classlist = [];
                                //if the user is a faculty member and teaching classes, splitting in this format will parse the class names
                                body = body.split("<TH COLSPAN=\"2\" CLASS=\"ddlabel\" scope=\"row\" >");
                                //if the split yielded more than 1 object, there are indeed classes to extract names for
                                if (body.length > 1) {
                                    //remove the first match
                                    body.splice(0, 1);
                                    for (var i = 0; i < body.length; i++) {
                                        //split the match further to extract just the name and not subsequent information
                                        var className = body[i].split(">", 2)[1].split("<", 2)[0];
                                        //push the class name (name and section number) into the class list that will be sent back to the user
                                        classlist.push(className.split(" - ", 2)[0] + " - " + className.split("").reverse().join("").split(" - ", 3)[0].split("").reverse().join(""));
                                    }
                                    console.log("[ADMIN] Results for " + sid + " for term " + term + ": " + JSON.stringify(classlist));
                                    myresponse["classes"] = classlist;
                                    //make the user admin over these classes
                                    myresponse["admin"] = true;
                                    //send the response
                                    socket.emit('classes', myresponse);
                                } else {
                                    //make a third request since the user is not a
                                    faculty member teaching classes... get the student detail schedule
                                    request({
                                        url: BANNER_URL + 'bwskfshd.P_CrseSchdDetl?term_in=' + term,
                                        method: 'GET',
                                        headers: {
                                            //re-use the session ID cookie from the last request's headers
                                            'Cookie': response.request.headers.Cookie
                                        },
                                    },
                                    function (error, response, body) {
                                        if (!error && response.statusCode == 200) {
                                            //if the user is a student and enrolled in classes, splitting in this format will parse the class names
                                            body = body.split("<CAPTION class=\"captiontext\">");
                                            //remove the first match
                                            body.splice(0, 1);
                                            for (var i = 0; i < body.length; i++) {
                                                //split the match further to extract just the name and not subsequent information
                                                var className = body[i].split("</CAPTION>", 2)[0];
                                                //unavoidably, "Scheduled Meeting Times" also matches the split criteria so just ignore it
                                                if (className != "Scheduled Meeting Times") {
                                                    //push the class name (name and section number) into the class list that will be sent back to the user
                                                    classlist.push(className.split(" - ", 2)[0] + " - " + className.split("").reverse().join("").split(" - ", 3)[0].split("").reverse().join(""));
                                                }
                                            }
                                            console.log("[NON-ADMIN] Results for " + sid + " for term " + term + ": " + JSON.stringify(classlist));
                                            myresponse["classes"] = classlist;
                                            //make the user non-admin over these classes
                                            myresponse["admin"] = false;
                                            //send the response
                                            socket.emit('classes', myresponse);
                                        }
                                    });
                                }
                            }
                        });
                        break;
                    }
                }
```

```
            //if the user was not successfully logged in (there is no session id header) then send back an error
            if (!success) {
                console.log("Authorization failed for: " + sid + ", " + pin);
                socket.emit('classes', 'Error');
            }
        }
    });
}

//function to start the web server on port 3000
http.listen(3000, "0.0.0.0", function(){
    console.log('listening on *:3000');
    //try to make the directory where the files will be stored... if it's already created the catch block will ignore that
    try {
        fs.mkdirSync(dir);
    } catch (e) { }
    try {
        //attempt to read the global index of files contained in the system
        globalfiles = JSON.parse(fs.readFileSync(dir + "files.json", 'utf8'));
    } catch (e) {
        //if there is an error with the file or it doesn't exist, then set the data structure to an empty object
        globalfiles = {};
    }
});
```