Table of Contents

[**part 01 – Installing and Configuring Winston** 2](#_Toc58221005)

[**part 02 – Implementing the Logger Method** 3](#_Toc58221006)

[**part 03 – Testing the Logger Method** 5](#_Toc58221007)

[**part 04 – Server Side Validation** 6](#_Toc58221008)

[**part 05 – Testing with Mocha and Chai** 9](#_Toc58221009)

[**apendix 1 Alternative Installation of Winston** 12](#_Toc58221010)

Day04 Logs and Validation

## part 01 – Installing and Configuring Winston

1. Install Winston into the project as we have done before, you will need to stop the app using CTRL-C

npm install winston

1. Create a config file like you did for Morgan, so winston\_config.js inside of the controllers folder:

|  |
| --- |
|  |

We will also use the same logs folder to create a Winston log file.

1. Since we are dealing with files, we would need the path module and instead of requiring the entire Winston file, just import what we will be using for this example:

|  |
| --- |
| **const path = require("path");**  **const { createLogger, transports ,format } = require('winston');** |

So, we de-structure the classes that are going to be used.

1. Now we create a function that points to the createLogger() method of Winston:

|  |
| --- |
| **const path = require("path");**  **const { createLogger, transports ,format } = require('winston');**  **const logger = createLogger({**  **});** |

As you can imagine, this method needs a few parameters to make it functional

1. These are the three basic settings for createLogger(). Level of course is for severity, there are other options like info. Format refers to the display of the logs themselves, they can be color coded or printed in JSON format for example. Transports is an array that would take file locations so that errors can actually be written to them:

|  |
| --- |
| **const logger = createLogger({**  **level: 'error',**  **format: '',**  **transports: [ ]**  **});** |

1. For this example enter the following details:

|  |
| --- |
| **const logger = createLogger({**  **level: 'error',**  **format: format.combine(**  **format.json(),**  **format.timestamp()**  **),**  **transports: [**  **new transports.File({ filename: path.join(\_\_dirname, './../logs/winston.log')})**  **]**  **});** |

1. All that is left now is to export the logger as an object, here is the entire file:

|  |
| --- |
| **const path = require("path");**  **const { createLogger, transports ,format } = require('winston');**  **const logger = createLogger({**  **level: 'error',**  **format: format.combine(**  **format.json(),**  **format.timestamp()**  **),**  **transports: [**  **new transports.File({ filename: path.join(\_\_dirname, './../logs/winston.log')})**  **]**  **});**  **//**  **module.exports = logger;** |

You can also add format.prettyPrint(), to the format.combine() method for a better printout.

## part 02 – Implementing the Logger Method

1. In the controller.js file require the winston\_config.js file you created in Part 3:

|  |
| --- |
| **const jwt = require('jsonwebtoken');**  **const path = require("path");**  **const w = require('./../controllers/winston\_config');**  **//**  **exports.getdefault = function(req, res){** |

1. Now we can apply this object to any function. The logger object we created in Part 3 and implemented as w, has a log() method by default:

|  |
| --- |
| **exports.getdocs=function(req, res){**  **Weight.find({}, function(err, results){**  **if (err){**  **w.log();**  **}**  **res.json(results);**  **})**  **};** |

So instead of just returning with the err object, we can log that err object and return an appropriate status and message.

1. So here, we pass an object to the log() method indicating which level of errors it should log and what exactly to log

|  |
| --- |
| **Weight.find({}, function(err, results){**  **if (err){**  **w.log({**  **level: 'error',**  **message: err**  **});**  **}**  **res.json(results);** |

1. Finally we return from the if() structure, an appropriate status and a message to the browser or client. Of course if no errors occurred, we respond with the results of our MongoDB find() mehtod.

|  |
| --- |
| **exports.getdocs=function(req, res){**  **Weight.find({}, function(err, results){**  **if (err){**  **w.log({**  **level: 'error',**  **message: err**  **});**  **res.status(503).send('Tehnical difficulties, please check with your Administrator!')**  **}**  **res.json(results);**  **})**  **};** |

1. As soon as you save these changes, a new file called winston.log will appear in the logs folder

|  |
| --- |
|  |

## part 03 – Testing the Logger Method

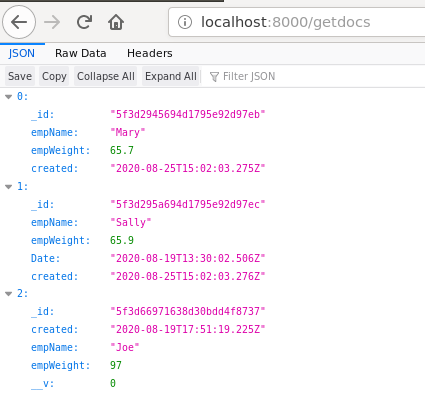
1. If you are the administrator of your machine, from a terminal window, run this command to stop the Mongod service:  
   sudo systemctl stop mongodb
2. Check the status with this command:  
   sudo systemctl status mongodb

It should NOT report, running, in green.

1. Now go to a browser (or Postman), navigate to this address:  
   <http://localhost:8000/getdocs>

You may have to wait a few seconds, but the winston.log file should have an error in it, something like below:

|  |
| --- |
|  |

1. Also notice that the application is still running, it did not crash.
2. Also notice that the message in the browser is exactly what we coded.
3. You can restart the mongodb service with this command:  
   sudo systemctl start mongodb
4. Refresh the URL from part 3 and the data should return  
   

## part 04 – Server Side Validation

1. Install express validator with this command: npm install express-validator
2. For this demonstration, I will use test the addnewuser() method in the controller.js file, but as recommended, the checks will be in the route file. So, in the controllers folder, create a new file called validator.js and de-construct the following functions:

|  |
| --- |
| **const { check, validationResult } = require('express-validator');** |

1. Start defining what you want to validate

|  |
| --- |
| **const { check, validationResult } = require('express-validator');**  **const fieldChecks = function(){**  **return [**  **check('empName', 'Name cannot be empty.').not().isEmpty()**  **]**  **};**  **//** |

So, we want to make sure that the empName field is not empty.

1. Create a function that will be exported and used in the routes file, remember this is middleware so it has access to the request and response objects. We also need to implement the *next* feature to move the process forward if no errors found:

|  |
| --- |
| **check('empName', 'Name cannot be empty.').not().isEmpty()**  **]**  **};**  **//**  **const validate = function(req, res, next){**  **const errors = validationResult(req);**  **}** |

Notice we are using the validationResult object we extracted from express-validator

1. That validationResult will NOT be an empty object if it finds errors, so pass the errors object over to a variable called *errors*. If no errors, just move on with *next*:

|  |
| --- |
| **const validate = function(req, res, next){**  **const errors = validationResult(req);**  **//**  **if (errors.isEmpty()) {**  **return next();**  **};**  **res.status(400).end(errors.array()[0].msg);**  **}** |

1. Obviously if there is an error we want to return a bad request status and the error message, or something like that:

|  |
| --- |
| **const validate = function(req, res, next){**  **const errors = validationResult(req);**  **//**  **if (errors.isEmpty()) {**  **return next();**  **};**  **res.status(400).end(errors.array()[0].msg);**  **}** |

1. At the very bottom of this file, export both functions in curly braces:

|  |
| --- |
| **res.status(400).end(errors.array()[0].msg);**  **}**  **//**  **module.exports = {**  **fieldChecks,**  **validate,**  **}** |

1. Now the implementation of this validator.js file. In routes.js require the file at the top and extract the two functions:

|  |
| --- |
| **const controller = require('../controllers/controller');**  **let authUser = require('../controllers/auth');**  **const { fieldChecks, validate } = require('./../controllers/validator');** |

1. Then in the route that we want to validate, addnewuser, inject the fieldChecks() function as well as validate:

|  |
| --- |
| **router.route('/updatedoc').put(controller.updatedoc);**  **//**  **router.post('/addnewuser', fieldChecks(), validate, controller.addnewuser);**  **//**  **router.route('/loginuser').post(controller.loginuser);** |

Notice that fieldChecks() is called, executed, but validate is not. It is only executed if there is an error

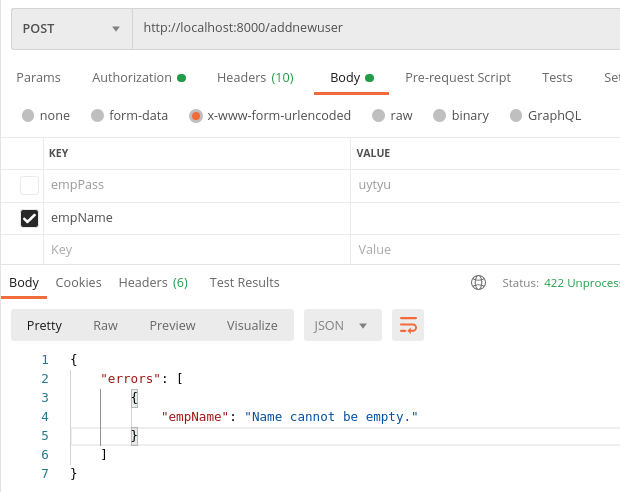
1. (Optional) Also, since we have Winston implemented you can log validation messages also. You would need to require the winston\_config.js file, the call the log() function:

|  |
| --- |
| **const w = require('./../controllers/winston\_config');**  **const { check, validationResult } = require('express-validator');**  **const fieldChecks = function(){**  **return [**  **check('empName', 'Name cannot be empty.').not().isEmpty()**  **]**  **};**  **//**  **const validate = function(req, res, next){**  **const errors = validationResult(req);**  **//**  **if (errors.isEmpty()) {**  **return next();**  **};**  **w.log({**  **level: 'error',**  **message: errors.array()**  **});**  **res.status(400).end(errors.array()[0].msg);**  **}** |

1. (Optional) Since errors.array() is an array we can use the map function to extract its values and put them into a different array, for use later:

|  |
| --- |
| **w.log({**  **level: 'error',**  **message: errors.array()**  **});**  **const extractedErrors = []**  **errors.array().map(**  **err => extractedErrors.push({ [err.param]: err.msg })**  **);**  **res.status(400).end(errors.array()[0].msg);** |

Alternatively, you can do the mapping before the log, then log the extractedErrors instead

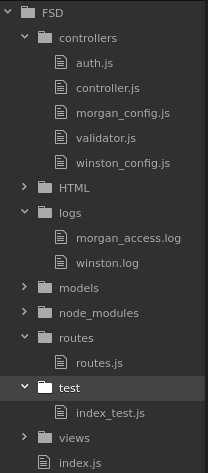


## part 05 – Testing with Mocha and Chai

1. Install the packages: npm install mocha chai chai-http --save-dev
2. In your package.json file, add a test script:

|  |
| --- |
| **"main": "index.js",**  **"scripts": {**  **"start": "nodemon index.js",**  **"test": "mocha"**  **},**  **"author": "",** |

1. By convention, Mocha will assume test files are in a folder called test, so create that now and inside of that folder create a .js file to hold our tests, call it index\_test.js



1. In the index\_test.js file, require all the chai components as well as our index file as this now represents our application

|  |
| --- |
| **const chai = require('chai');**  **const cHttp = require('chai-http');**  **const app = require('./../index');** |

1. Pass Chai’s http module into chai via the use method:

|  |
| --- |
| **const chai = require('chai');**  **const cHttp = require('chai-http');**  **const app = require('./../index');**  **//**  **chai.use(cHttp);** |

1. Now the first describe() function

|  |
| --- |
| **chai.use(cHttp); describe('The App', function(){**  **});** |

1. Then insert a different describe() function that would test the path

|  |
| --- |
| **chai.use(cHttp);**  **describe('The App', function(){**  **describe('GET /getdocs', function(){**  **});**  **});** |

1. At this point we can fire the it() method which would do the actual call

|  |
| --- |
| **chai.use(cHttp);**  **describe('The App', function(){**  **describe('GET /getdocs', function(){**  **it('it should get all documents', function(done){**  **chai.request(app).get('/getdocs');**  **})**  **});**  **});** |

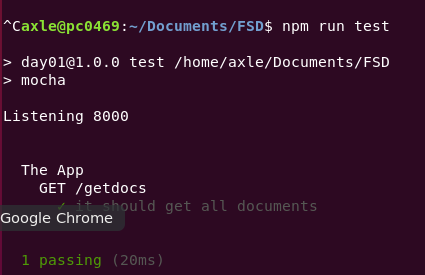
1. Finally call the done() method after the get() method runs:

|  |
| --- |
| **chai.use(cHttp);**  **describe('The App', function(){**  **describe('GET /getdocs', function(){**  **it('it should get all documents', function(done){**  **chai.request(app).get('/getdocs');**  **done();**  **})**  **});**  **});** |

1. Before we can run the test, we need to export our app, right now the app runs for external use but not for testing purposes, so add the following line towards the end of the index.js file

|  |
| --- |
| **app.listen(port, function(){**  **console.log("Listening " + port);**  **});**  **//**  **module.exports = app;** |

1. In a terminal window, close your app if it is running and type this command in: npm run test. You should see something like below:



1. Of course you can take this further. For example if you look at the raw data being returned after you hit /getdocs in your browser, it is an array. We could test that we are getting an array and NOT a string. First though we need to run the should() method of Chai.

|  |
| --- |
| **const app = require('./../index');**  **//**  **chai.use(cHttp);**  **chai.should();**  **describe('The App', function(){** |

1. Now we can do this:

|  |
| --- |
| **it('it should get all documents', function(done){**  **chai.request(app)**  **.get('/getdocs').end(function(err, res){**  **res.body.should.be.a('array');**  **})**  **done();**  **})** |

Notice that I had to call the end() method. From that method, we can either have an error or the array of data.

## apendix 1 Alternative Installation of Winston

1. Install Winston into the project as we have done before, you will need to stop the app using CTRL-C

npm install Winston express-winston

1. Create a config file like you did for Morgan, so winston\_config.js inside of the controllers folder:

|  |
| --- |
|  |

We will also use the same logs folder to create a Winston log file.

1. Since we are dealing with files, we would need the path module and both Winston package of course:

|  |
| --- |
| **const winston = require('winston');**  **const path = require("path");**  **const expressWinston = require('express-winston');** |

So, we de-structure the classes that are going to be used.

1. Now we create a function that points to the createLogger() method of Winston:

|  |
| --- |
| **const ew = expressWinston.logger(**  **);** |

As you can imagine, this method needs a few parameters to make it functional

1. These are the three basic settings:

|  |
| --- |
| **const ew = expressWinston.logger({**  **transports: [**  **new winston.transports.File({filename: path.join(\_\_dirname, './../logs/winston.log')})**  **],**  **expressFormat: true**  **});** |

1. All that is left now is to export the logger as an object, here is the entire file:

|  |
| --- |
| **const winston = require('winston');**  **const path = require("path");**  **const expressWinston = require('express-winston');**  **//**  **const ew = expressWinston.logger({**  **transports: [**  **new winston.transports.File({filename: path.join(\_\_dirname, './../logs/winston.log')})**  **],**  **expressFormat: true**  **});**  **//**  **module.exports = ew;** |

You can configure this in several ways, for that go to the documentation.

1. In the index.js file require the winston\_config.js file you created in Part02:

|  |
| --- |
| **const jwt = require('jsonwebtoken');**  **const path = require("path");**  **const w = require('./controllers/winston\_config');**  **//**  **exports.getdefault = function(req, res){** |

1. Now we can apply this object as middleware:

|  |
| --- |
| **app.use(express.static('HTML'));**  **app.set('view engine', 'pug');**  **app.use(m);**  **app.use(w);**  **//**  **app.use('/', router);** |

In this current configuration, Winston must come before the router.

1. Visit any of the express routes, for example if you go to /teamweights.html and press the Get all Weights button, check the inston.log file afterwards
2. Optionally, you can log errors:

|  |
| --- |
| **// const ew = expressWinston.errorLogger({**  **// transports: [**  **// new winston.transports.File({filename: path.join(\_\_dirname, './../logs/winston.log')})**  **// ],**  **// expressFormat: true**  **// });** |