**Exercise01\_01\_01 – Step 1**

In this Exercise, we will learn some introductory coding with Node.js.



1. Create a folder named Exercise01\_01\_01 and open it with your IDE. Create a new file called ***sync-demo.js***. Enter the following code:  
   ***var fs = require('fs');  
     
   var data = fs.readdirSync('c:/');  
   console.log('data: ', data);  
     
   console.log("This code is last");***Execute the script and take note of the sequence of the output. In Brackets, the best plug-in is ***Node.js bindings***. In Atom, the most recommended is ***atom-runner***.

**Exercise01\_01\_01 – Step 2**



1. Create a new file called ***async-demo.js***. Enter the following code:  
   ***var fs = require('fs');  
     
   function phoneNumber(err, data) {  
    console.log('data: ', data)  
   }  
     
   fs.readdir('c:/', phoneNumber);  
     
   console.log("This code is last");***Execute the script and take note of the sequence of the output.

**Exercise01\_01\_01 – Step 3**



1. Create a new file called ***async-demo2.js***. Enter the following code:  
   ***var fs = require('fs');  
     
   fs.readdir('c:/', function(err, data) {  
    console.log('data: ', data)  
   });  
     
   console.log("This code is still last");***Execute the script and take note of the sequence of the output.

**Exercise01\_01\_01 – Step 4**



1. Create two new files called ***my-module.js*** and ***module-demo.js***. The idea is to get the code in the module ***my-module.js*** accessible inside the module ***module-demo.js***.
2. Open the ***my-module.js*** module and create a property on the ***exports*** object to make available to other modules as follows:  
   ***exports.myText = 'Hello from my-module!';***
3. Open the ***module-demo.js*** module and use the ***require()*** method to make the new module available. Notice that we set the results into a variable which will hold the module:  
   ***var myModule = require('./my-module.js');***
4. Add the code which will use the ***external*** module. Notice that the variable is treated as an ***object***, because it is actually holding the ***exports*** object created in the other module. Therefore, we can use a ***property*** of that object in this module:  
   ***console.log('Text from the external module: ',   
    myModule.myText);***Execute the script and take note of the output.

**Exercise01\_01\_01 – Step 5**



1. We will download a package named ***gulp***, which is a package that helps to automate repetitive tasks. Either go to the command line in the terminal, or use Brackets node.js dropdown, to download a the package using NPM:  
   ***npm install gulp***There may be warnings or errors, but the command should exit with an error code of 0, indicating a successful install. A ***node\_modules*** folder is created. Open it up and find the ***gulp*** package. Open it up and you can see its contents.
2. Create a new file called ***gulpfile.js***, which we will use to create a custom task and have gulp run it. First we must require() the module:  
   ***var gulp = require('gulp');***
3. We will access the ***task()*** method of ***gulp*** and define a task. Notice that we define the task by ***naming*** it, then giving it an ***anonymous*** function that defines the task to be performed:  
   ***gulp.task('default', function() {  
    console.log('Hello from gulp!');  
   });***Execute the script and take note that no output appears.
4. The ***gulp*** command must be executed from a command line, so let’s try that. When executed, it looks for a default file name of ***gulpfile.js*** and a default task named ***default***. Open a terminal and type:  
   ***gulp***  
   We get an error that the system does not think that gulp is installed, unless we are in our project folder.
5. Gulp should also be installed ***globally*** on our system, as well as in the local ***node\_modules*** of our project. We will find out later why we need to do both, but for now, let’s get gulp globally installed with the following command. Notice the ***–g*** parameter, for global:  
   ***npm install gulp –g***Open a terminal and type ***gulp***.
6. Now we can use the terminal in the local project folder, or use an IDE plug-in to make our lives easier. We can see that gulp does what it is supposed to do, and executes our simple task.

**Exercise01\_01\_01 – Step 6**



1. Run a console or terminal in the project folder and enter the following command:  
   ***npm init***Go back into the IDE and open the package.json file for examination.

**Exercise01\_01\_01 – Step 7**



1. Create a new file called ***read-demo.js***. Enter the following code to obtain access to the file system library from the API:  
   ***var fs = require('fs');***
2. Create a simple JSON file named ***data1.json*** with the following contents:  
   ***{  
    "name": "John"  
   }***
3. Let’s read the JSON file with an asynchronous function from the File System API called ***readFile()***. It will take a ***path*** and ***callback*** as parameters. The first ***callback*** parameter will return an error or the data that is read:  
   ***fs.readFile('./data1.json', function(err, data) {  
    console.log(data);  
   });***  
   Give that a test. Notice the strange output, a buffer that appears to contain hex bytes.
4. We need to make use of the second optional parameter of the readFile() method to indicate the file format:  
   fs.readFile('./data.json', ***'utf-8',*** function(err, data) {  
    console.log(data);  
   });  
   Give that a test, and the output should be as expected.
5. Now let us see if it understands our JSON object that is contained in the file. Try to output a property of the JSON object:  
    console.log(data);  
    ***console.log(data.name);***Give that a test. The ***name*** property that is contained in the JSON file shows up as ***undefined***, so it has not been read in as an object.
6. We can make use of another global object: ***JSON***. It has useful methods, one of which is ***JSON.parse()***, which can translate a string to JavaScript JSON:  
    ***data = JSON.parse(data);*** console.log(data.name);  
   Give that a test. The ***name*** property that is contained in the JSON file shows up correctly as the property of a JSON object.
7. Let’s try the same task using a require(), which knows how to deal with objects. Create a file called ***read-demo2.js***:  
   ***var data = require('./data1.json');  
     
   console.log(data.name);***  
   Give that a test. It is much simpler, and the ***name*** property that is contained in the JSON file shows up correctly as the property of a JSON object.

**Exercise01\_01\_01 – Step 8**



1. Create a new file called ***read-directory-demo.js***. Enter the following code to obtain access to the file system library from the API:  
   ***var fs = require('fs');***
2. Let’s read the directory with an asynchronous function from the File System API called readdir(). It will take a ***path*** and ***callback*** as parameters. The first ***callback*** parameter will return an error or the data that is read:  
   ***fs.readdir('c:/', function(err, data) {  
    console.log(data);  
   });***Give that a test. Notice that the output is returned as an array of file names.

**Exercise01\_01\_01 – Step 9**



1. Create a new file called ***write-file-demo.js***. Enter the following code to obtain access to the file system library from the API:  
   ***var fs = require('fs');***
2. Let’s try to write a JSON object to the file as follows:  
   ***var jsonString = { name: 'Mark'};  
     
   fs.writeFile('data2.json', jsonString, function(err) {  
    if (err) {  
    console.log(err);  
    }  
   });***Give that a test, and the json file should be created. Open the file and notice that the contents are not as expected.
3. The problem is that the ***writeFile()*** method expects its data to be either a ***string*** or a ***buffer***, and we gave it an ***object***. The first solution is to write it directly as a string as follows. Note that we have enclosed the whole object in single quotes, and double quoted both the property name and value:  
   var jsonString = ***'{ "name": "Mark"}'***;  
   Give that a test, and open the file and notice that the contents are as required.
4. There is a better solution to write JSON objects to a file with this method. We can use another method of the global ***JSON*** object called ***JSON.stringify()***. We will comment out the existing JSON and write it in a more Best Practices way. We will then wrap the object in the JSON method for the writeFile() call:  
   ***//***var jsonString = '{ "name": "Mark"}';  
     
   ***var jsonString = {   
    name: 'Mark'  
   };***  
     
   fs.writeFile('data2.json', ***JSON.stringify(jsonString)***, function(err) {  
   Give that a test, and open the file and notice that the contents are also as required.

**Exercise01\_01\_01 – Step 10**



1. Install ExpressJS with NPM, both globally and locally. Notice the ***–save*** parameter on the local installation, which puts ***express*** as a dependency in your ***package.json*** file:  
   ***npm install express –g  
   npm install express –save***Inspect the package.json file to make sure that express has been added to the dependencies.
2. Create a new file called ***express-static-demo.js***. Enter the following code to obtain access to the express library from the node\_modules:  
   ***var express = require('express');***
3. We must now construct an instance of an express object to be able to use its properties and methods. Call its constructor and store it in a variable as follows:  
   var express = require('express');  
   ***var app = express();***
4. Now let’s turn our node.js application into a server. We can use the listen() method of express to have the server listening on a port. We can use it with a callback to determine if it is running as follows:  
   var app = express();  
   ***var port = 8080;  
     
   app.listen(port, function() {  
    console.log('Server listening on localhost:%s', port);  
   });***  
   Run the node application and it should indicate that it is listening. Now open your browser and navigate to ***localhost:port***. We should connect and get a response, but no content, because we are not yet serving anything.
5. We will create a static file to serve by scaffolding a new HTML file named ***index.html***, with some minimal code in it as follows:  
   <!doctype html>  
   <html>  
   <head>  
    <title>Express Test</title>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="initial-scale=1.0">  
   </head>  
   <body>  
    ***<h2>Serving static content with ExpressJS</h2>***</body>  
   </html>  
   Test this with Live Preview and shut it down.
6. Employ the Express object’s ***use()*** method to serve some content. We will specify the content as ***static*** with the ***static()*** method. Notice that we use the Node.js ***global*** variable ***\_\_dirname,*** which indicates current directory. By standard, the static file will default to index.html:  
   var port = 8080;  
     
   ***app.use(express.static(\_\_dirname));***Run the node file, go back to the browser and connect with the host, and the static file should be served. Terminate the node application.

**Exercise01\_01\_01 – Step 11**



1. Copy the file ***express-static-demo.js*** to a new file called ***express-endpoint-demo.js***. Just delete the app.use() call, and we can make use of the rest of the server setup:  
   ***var express = require('express');  
   var app = express();  
   var port = 8080;  
     
   app.listen(port, function() {  
    console.log('Server listening on localhost:%s', port);  
   });***Give this a test run and then terminate it.
2. We must now construct an ***endpoint*** to request our dynamic content. In a ***use()*** method, we will specify the ***URL*** of the resource, which will be ***/message***. We also implement a callback as ***middleware*** between the HTTP request and response as follows:  
   ***app.use('/message', function(req, res) {  
    console.log('User requested an endpoint!');  
   });***Run the node application and it should indicate that it is listening. Now open your browser and navigate to ***localhost:port/message***. We should get a ***console.log()*** message, but we still have no response in our browser. This shows that the middleware is running between the request and the response.
3. We can now implement our response inside the callback:  
    console.log('User requested an endpoint!');  
    ***res.send('<h2>Hello from the server endpoint!</h2>');***  
   Run the node application and it should indicate that it is listening. Now open your browser and navigate to ***localhost:port***. We should now get a response from the server.
4. Now let’s create another endpoint that will simulate retrieving data from a database, which we will do later on. Instead, we will just retrieve some data from a JSON file which we already have. Create the new endpoint as follows:  
   ***app.use('/users', function(req, res) {  
      
   });***
5. Let’s get access to the file system with a ***require()*** statement as follows:  
   var app = express();  
   ***var fs = require('fs');***var port = 8080;
6. Open ***read-demo.js*** and copy the ***readfile()*** call. Paste it into the new endpoint middleware callback, and modify it as follows:  
   app.use('/users', function(req, res) {  
    ***fs.readFile('./data1.json', 'utf-8', function(err, data) {  
    res.send(data);  
    });***  
   });  
   Run the node application and it should indicate that it is listening. Now open your browser and navigate to ***localhost:port/users***. We should get a data response in our browser