Contents

[NODE 3](#_Toc73108093)

[NODE ARCHITECTURE 3](#_Toc73108094)

[HOW NODE WORKS? 3](#_Toc73108095)

[SYNCHRONOUS AND ASYCHRONOUS NATURE OF NODE 4](#_Toc73108096)

[CREATING A NODE PROJECT 4](#_Toc73108097)

[NODE GLOBAL OBJECT 4](#_Toc73108098)

[NODE MODULES 4](#_Toc73108099)

[BUILD IN NODE MODULES 5](#_Toc73108100)

[EVENT MODULE 5](#_Toc73108101)

[HTTP MODULE 6](#_Toc73108102)

[FILE SYSTEM MODULE 7](#_Toc73108103)

[NODE PACKAGE MANAGER (NPM) 7](#_Toc73108104)

[PACKAGE.JSON 8](#_Toc73108105)

[INSTALLING A NODE PACKAGE 8](#_Toc73108106)

[USING THE NODE PACKAGE 8](#_Toc73108107)

[INSTALLING DEV DEPENDENCIES 8](#_Toc73108108)

[NPM PACKAGES AND SOURCE / VERSION CONTROL 9](#_Toc73108109)

[UN-INSTALLING PACKAGES 9](#_Toc73108110)

[PUBLISHING NPM PACKAGE TO NPM REPO 9](#_Toc73108111)

[ASYNCRONOUS PROGRAMMING IN NODE 10](#_Toc73108112)

[CALLBACK HELL 10](#_Toc73108113)

[SOLVING CALLBACK HELL 10](#_Toc73108114)

[EXPRESS JS 11](#_Toc73108115)

[RESTFUL API USING EXPRESS JS 11](#_Toc73108116)

[NODEMON 11](#_Toc73108117)

[CONFIGURING LISTENING PORT 11](#_Toc73108118)

[ROUTING PARAMETER & HANDLING HTTP REQUESTS 11](#_Toc73108119)

[POST REQUEST 12](#_Toc73108120)

[INPUT VALIDATION (USING JOI) 12](#_Toc73108121)

[PUT REQUEST 12](#_Toc73108122)

[DELETE REQUEST 12](#_Toc73108123)

[RESPONSE 13](#_Toc73108124)

[MIDDLEWARE / MIDDLEWARE FUNCTIONS 13](#_Toc73108125)

[BUILD IN MIDDLEWARE FUNCTION 13](#_Toc73108126)

[CREATING CUSTOM MIDDLEWARE FUNCTION 15](#_Toc73108127)

[LOCALS 17](#_Toc73108128)

[THIRD PARTY MIDDLEWARE FUNCTION 17](#_Toc73108129)

[SERVER-SIDE RENDERING USING TEMPLATING ENGINE 17](#_Toc73108130)

[APP SETTING TABLE 18](#_Toc73108131)

[HOW EXPRESS USES THE TEMPLATING ENGINE 18](#_Toc73108132)

[USING TEMPLATING ENGINE - EJS 18](#_Toc73108133)

[EXPRESS GENERATOR 20](#_Toc73108134)

[DEBUG – EXPRESS 21](#_Toc73108135)

[EXAMPLE 21](#_Toc73108136)

[ENABLING DEBUG 21](#_Toc73108137)

[DEBUGGING CODE – USING CHROME DEVELOPER TOOL 22](#_Toc73108138)

[STEPS TO DEBUG 22](#_Toc73108139)

[REQUEST/RESPONSE OBJECT IN EXPRESS 22](#_Toc73108140)

[EXAMPLE - APPLICATION FLOW DIAGRAM 22](#_Toc73108141)

[DYNAMIC ROUTE PARAMS 23](#_Toc73108142)

[ROUTERS 23](#_Toc73108143)

[TESTING NODE AND EXPRESS USING JEST 23](#_Toc73108144)

[INSTALL MODULES 23](#_Toc73108145)

[UPDATE PACKAGE.JSON 24](#_Toc73108146)

[WRITING TEST 24](#_Toc73108147)

[WRITING TEST FOR ASYNCHRONOUS CODE 24](#_Toc73108148)

[TESTING ASYNCHRONOUS CODE USING PROMISES 24](#_Toc73108149)

[TESTING ASYNCHRONOUS CODE USING ASYNC AND AWAIT 24](#_Toc73108150)

[TESTING THE EXPRESS API 25](#_Toc73108151)

# NODE

* Usually to execute a JavaScript code we need a browser as a runtime environment. Node is an open source and cross platform runtime environment for executing JavaScript code outside browser
* Node is majorly use to create highly scalable, data intensive and real-time backend APIs
* Node is not a programming language- It’s a runtime environment
* ***Node is ideal for IO intensive operations.Node should not be used for CPU intensive applications***

## NODE ARCHITECTURE

|  |  |
| --- | --- |
| **JS ENGINE FOR DIFFERENT BROWSERS** | * Every browser has a JS engine which converts the JS code to machine understandable code * Because of different type of JS engine, the JS code behaves differently in different browsers |
| * To give a flexibility to run the JS outside the browser for consistent experience. * **Ryan Dhal (founder of Node), took the Chrome’s V8 JS engine- which is the fastest JS engine, and wrapped it in C++ wrapper program- That we call Node JS** |  |

## HOW NODE WORKS?

* Node is highly scalable because of it non-blocking or asynchronous behavior.

|  |  |
| --- | --- |
|  | **PROBLEM WITH SYNCHRONOUS ARCHITECTURE**   * In the synchronous architecture, whenever a request comes – it creates a thread to serve the request * If the request is for IO intensive operation the thread will be busy till it finishes the task – so if any new request comes in between, a new thread will be created for it. * So, when we have large number of such requests the - at some point of time – the request has to wait the thread to free thread to serve the request. |
|  | **HOW NODE SOLVES THIS PROBLEM**   * Node has Single thread to serve all the requests. * Whenever an IO intensive task comes in - the Node thread will serve the request, and by the time the operation is going on (like Database fetch operation)- the thread will be ready to serve other requests. * Once the previous database operation done - it is placed in an Event Queue. The Node thread always keeps on monitoring the event queue. If its finds something in the event queue it executes it. |

### SYNCHRONOUS AND ASYCHRONOUS NATURE OF NODE

* Node has single threaded architecture. So, to avoid the blocking, entire node has been designed using “***callbacks***”

### CREATING A NODE PROJECT

|  |  |
| --- | --- |
| **CREATE A FOLDER** | mkdir first-app |
| **NAVIGATE TO THE FOLDER** | cd first-app |
| **OPEN VS CODE FROM THE FOLDER** | code . |
| **CREATE A SIMLPLE JS FILE (APP.JS) AND WRITE SOME JS CODE** | var sayHello = function(message) {  console.log("Hello " + message);  };  sayHello("John"); |
| **EXECUTE THE FILE** | node app.js |

* ***IMPORTANT POINT:*** *Node doesn’t have global objects like “window” or “document”.*

### NODE GLOBAL OBJECT

* “global” is the name of the global object in node similar to “window” object in browser runtime environment.

|  |  |  |
| --- | --- | --- |
| IN Browser window object | In Node global object    O/P | * In Node the variables we declare are not part of global object * **The scope of the variable is limited to that file itself or module itself. This because of Node’s modular system** |

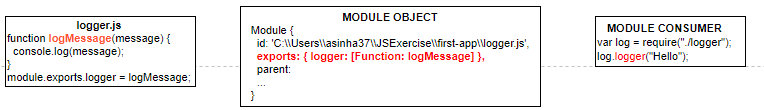
### NODE MODULES

* Every file in node is considered as module and the variables and function defined in that module is scoped to that module only
* If we want to use the function or variable outside the module – we need to explicitly export it
* Node JS modules follow ***Common JS Module System***.
  1. **CREATE A MODULE**
  2. **EXPORT THE MODULE**
  3. **USE THE MODULE**

|  |  |  |
| --- | --- | --- |
| **CREATE A MODULE [LOGGER MODULE]** | **USE THE MODULE** | **FILE LOCATIONS** |
| CREATE a JS file **logger.js**  function logMessage(message) {  console.log(message);  }  **module.exports.logger = logMessage;** | USE IN **app.js**  const log = require("./logger");  log.logger("Hello"); |  |

**IMPORTANT POINTS**

1. **module.exports 🡪** This is used to export variables and function- which can be used by another modules.
2. **“require(<file\_path/ module>)” 🡪** This is used to import the exported function /variables from the module



#### MODULE WRAPPER FUNCTION

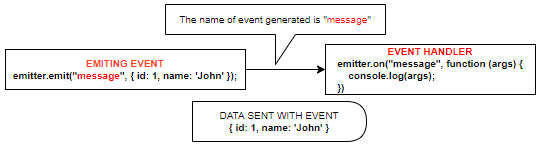
|  |  |
| --- | --- |
| * module is an object which is local to the module – not a global object. * The code in the modules are internally wrapped around IIFE * As we can see exports , require,\_\_filename,\_\_dirname are local variables in an IIFE | (function (exports, require, module, \_\_filename, \_\_dirname)  var log = require("./logger");  log.logger("Hello");  )  **The IIFE is called MODULE WRAPPER FUNCTION** |

## BUILD IN NODE MODULES

1. **PATH MODULE**
2. **OS MODULE**
3. **HTTP MODULE**
4. **EVENT MODULE**
5. **FILE SYSTEM MODULE**

### EVENT MODULE

* Event module has one important class “**EventEmitter**”



|  |  |
| --- | --- |
| const EventEmitter = require('events');  const emitter = new EventEmitter();  emitter.on("message", (args) => console.log(args));  emitter.emit("message", { id: 1, name: 'John' }); | * While writing the event listener – the order is very important. The event listerner must be written before the “emit” is raised * We can pass the “data” as well which will be recieved by the event listener callback function as argument. |

#### CUSTOM EVENT EMITTER

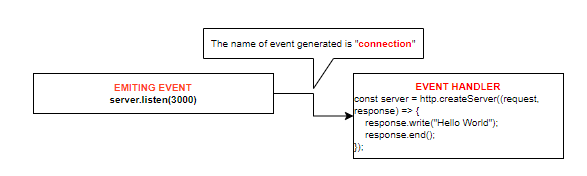
* Custom Event Emitter are used by inheriting EventEmitter class
* Note: While emiting and handling an event – it should use same EventEmitter object reference. As shown below

|  |  |
| --- | --- |
| **Logger.js** | **App.js** |
| const EventEmitter = require('events');  class Logger extends EventEmitter {  logMessage(message) {  this.emit("message", message);  }  }  module.exports = Logger; | const Logger = require('./logger');  const log = new Logger();  log.on("message", (args) => console.log(args));  log.logMessage({ id: 1, name: "John" }); |

### HTTP MODULE

* This module helps us to create a webserver which can listen to a HTTP request on a given PORT.
* In the below program we are using “http” module to create a webserver which can listen to HTTP request at PORT=3000
* “server” object is an EventEmitter which generates “connection” event using” server.listen()”.
* We pass the “callback” function to the createServer() function.

|  |  |
| --- | --- |
| const http = require('http');  const server = http.createServer((request, response) => {  response.write("Hello World");  response.end();  });  server.listen(3000);  console.log("Listening on port 3000"); |  |



#### HANDLING MULTIPLE ROUTES

|  |  |
| --- | --- |
| const http = require('http');  const server = http.createServer((request, response) => {  if (request.url == "/") {  response.writeHead(200, { 'content-type': 'text/html});  response.write("<h1>Hello World</h1>");  }  if (request.url == "/api/courses") {  response.write(JSON.stringify(  {  courseId: 1,  courseName: 'C++'  }  ));  }  response.end();  });  server.listen(3000);  console.log("Listening on port 3000"); | The mime type play an important role for the browser to interprete the data in the response  response.writeHead(200, { 'content-type': 'text/html});    response.writeHead(200, { 'content-type': 'text/text}); |
| const url = require("url");  const {pathname, query} = url.parse(request.url,true);  const server = http.createServer((request,response) =>{  if(pathname =='/api'){  console.log({  pathname:pathname,  queryString:query['productid']  })  response.end();  }  });  server.listen(3001,()=>{  console.log('Listening on Port 3001...')  }); | **FETCHING THE QUERY STRING**   * To fetch the query string from the url we need to parse the url * Parsing the url can be done using the “**url**” module |

### FILE SYSTEM MODULE

#### READING AND WRITING FILE SYNCHRONOUSLY

|  |  |
| --- | --- |
| const fs = require('fs');  const txtIn = fs.readFileSync('./txt/input.txt','utf-8');  const txtOut = `Adding to new file\n ${txtIn}`  fs.writeFileSync('./txt/output.txt',txtOut);  console.log(txtIn); | * We are using File System module to access file system * We are using “readFileSync” and “writeFileSync” function to read and write synchronously. * The reading and writing synchronously – shows the blocking nature of node. |

#### READING AND WRITING FILE ASYNCHRONOUSLY

|  |  |
| --- | --- |
| const fs = require('fs');  const txtIn = fs.readFileSync('./txt/input.txt','utf-8');  const txtOut = `Adding to new file\n ${txtIn}`  fs.writeFileSync('./txt/output.txt',txtOut);  console.log(txtIn); | * We are using File System module to access file system * We are using “readFileSync” and “writeFileSync” function to read and write synchronously. * The reading and writing synchronously – shows the blocking nature of node. |

## NODE PACKAGE MANAGER (NPM)

|  |  |  |
| --- | --- | --- |
| NOTE : CREATE A NODE ACCOUNT in - <https://www.npmjs.com/> [avishekh/Sapient!123]  It’s a command line tool as well registry of third party module/libraries which we add to node project.  It has many build in modules which can be installed on out node application | |  |
| TO KNOW NODE VERSION | **npm –v** |
| TO INSTALL SPECIFIC VERSION OF NPM | **npm i -g npm@5.5.1** |
| TO CHECK LIST OF DEPENDENCIES AND VERSIONS | **npm list –depth=0** |
| INSTALLING DEV DEPENDENCIES | **npm install js-beautify --save-dev** |
| TO KNOW THE OUTDATED PACKAGES | **npm outdated** |
| UPDATE THE OUTDATED PACKAGE | **npm update <package\_name>** |
| UNINSTALL A MODULE | **npm uninstall <package\_name>** |
|  |  |

### PACKAGE.JSON

Package.json contains the basic information of the project like version ,author , git repo , dependecies etc.

It’s a metadata of the project

Before adding any node module to our project we need to have package.json file

|  |  |
| --- | --- |
| **CREATING PACKAGE.JSON FILE** | mkdir npm-demo  cd npm-demo  **npm init** |
| The npm init will prompt for some options to define the metadata of the project  This will create a package.json file based on the input provided | **PACKAGE.JSON**  {  "name": "npm-demo",  "version": "1.0.0",  "description": "",  "main": "index.js",  "scripts": {  "test": "echo \"Error: no test specified\" && exit 1"  },  "author": "",  "license": "ISC"  } |
| * We can even create package.json file with default values (without prompting any options) using below   COMMAND : **npm init --yes** |

### INSTALLING A NODE PACKAGE

|  |  |
| --- | --- |
| **INSTALLING A NODE PACKAGE**  We can find the package names at [**https://www.npmjs.com/**](https://www.npmjs.com/) | **npm i <package\_name>**  **eg - npm i underscore** |
|  | Runnning this command will Add “underscore” node module and add the **dependencies** to package.json file.  When we install the node package- it will also install other package too on which underscore in depending on. |

### USING THE NODE PACKAGE

|  |  |
| --- | --- |
| * To can use the node package using “require” function * **contains** is a function from underscore library | const \_ = require("**underscore**");  console.log(\_.contains([1, 2, 3], 3)); |

### INSTALLING DEV DEPENDENCIES

|  |  |
| --- | --- |
| Sometime we need some dependecies which are needed during development only . This package will not be part of the application in PROD environment  COMMAND: **npm install js-beautify --save-dev** | {  "name": "npm-demo",  "version": "1.0.0",  "description": "",  "main": "index.js",  "scripts": {  "test": "echo \"Error: no test specified\" && exit 1"  },  "keywords": [],  "author": "",  "license": "ISC",  "dependencies": {  "underscore": "^1.9.2"  },  "devDependencies": {  "js-beautify": "^1.10.3"  }  } |

### NPM PACKAGES AND SOURCE / VERSION CONTROL

* When we commit out code in any version control system like GIT, we don’t actually commit the node\_modules folder. We don’t need to do so because node can able re-download the dependencies by referring the package.json file by executing “**npm install”** command.
* So will committing the project we need to ignore the node\_module folder.

|  |  |  |
| --- | --- | --- |
| **STEP 1 : GIT INIT** | Create an empty Git repository or reinitialize an existing one | |
| **STEP 2: GIT STATUS** | git status 🡨 The git status will show all the files along with node\_modules folder which we need to ignore  index.js  **node\_modules/**  package-lock.json  package.json | |
| **STEP 3: CREATE .GITIGNORE FILE**   * Create a **.gitigore** in the root folder of the project * Enter the folder name in the file as   **node\_module/**   * Now- git status will ignore the folder this time | |  |
| **Step 4: git add .**  **Step 5: git commit –m “first commit”** | |  |

### UN-INSTALLING PACKAGES

* COMMAND – **npm un <package\_name>** e.g **npm un jshint**

This will remove the entry from the package.json file and uninstall the node package too from node\_module folder.

### PUBLISHING NPM PACKAGE TO NPM REPO

|  |  |
| --- | --- |
| 1. Create a folder e.g node-publish 2. Create a **package.json** file in it(npm **init**) 3. Create a **index.js** file and add the following function | **module.exports.add** = function(a, b) {  return a + b;  }; |
| Note :  **While creating the package.json file the name property should have same value as you node repo username** |  |
| TO PUSH TO TO NPM REPO  We should have npm account to push our custom package to node repo  STEPS TO PUSH  LOGIN TO NPM - npm login  PUSH THE PACKAGE – npm push |  |
| **USING THE PUBLISHED PACKAGE** |  |

## ASYNCRONOUS PROGRAMMING IN NODE

### CALLBACK HELL

In the below example

* We are using “superagent” module as an http client.
* In the below code
  + We are reading a dog breed name from a file using “fs” module
  + Making a API call to using the breed name and fetching the dog’s image
  + Finally, we are writing the image path in the file.
* Chaining multiple callbacks leads to a problem call callback hell – which is hard to manage and degug

|  |
| --- |
| const fs = require("fs");  const superagent = require('superagent');  fs.readFile(`${\_\_dirname}/dog.txt`,'utf-8',(err,data) =>{  if(err) return console.log(err.message);  superagent.get(`https://dog.ceo/api/breed/${data}/images/random`).end((err,data)=>{  if(err) return console.log(err.message);  fs.writeFile(`${\_\_dirname}/dogImage.txt`,data.body.message,(err)=>{  if(err) return console.log(err.message);  console.log("Data Dumbed in File");  });  })  }); |

### SOLVING CALLBACK HELL

The callback hell can be resolved using

1. Promises
2. Async /Await

#### PROMISES

# EXPRESS JS

* There are various frameworks available to build REST API on top of node HTTP Module. Express is one on them. Express is buid on top of HTTP node module.
* With HTTP module of node – to handle routes- Node has to explicitly set the status code, mime type and end the response. Express takes care of all without writing anything explicitly.

## RESTFUL API USING EXPRESS JS

* UHG <https://hubconnect.uhg.com/docs/DOC-210865>

|  |  |
| --- | --- |
| CREATING A NODE PROJECT (creates a package.json) | npm init --yes |
| INSTALL EXPRESS | npm i express |
| SAMPLE EXPRESS PROGRAM(index.js)  const express = require('express') 🡨 *Returns a function*  const app = express() 🡨 Calling the function  app.get('/', function (req, res) { 🡨 It will handle get request  res.send('Hello World')  })  app.listen(3000,() => console.log(“Listening on port 3000…”)) | TO RUN THE PROGRAM: node index.js  Hit the browser : [**http://localhost:3000/**](http://localhost:3000/)  Note:  require('express') returns a function |

## NODEMON

* Nodemon is a tool that helps develop node.js based applications by automatically restarting the node application when file changes in the directory are detected.
* Nodemon act as a watcher for the changes we make in node application – so we don’t have to restart the server with every code change

|  |  |
| --- | --- |
| **INSTALLING NODEMON** | **npm i nodemon** |
|  | * **RUNNING THE PROGRAM: nodemon index.js** |

## CONFIGURING LISTENING PORT

|  |  |
| --- | --- |
| const port = process.env.PORT | 3000;  app.listen(port, () => {  console.log(`Listening on ${port}`);  }); | * The port on which the express server can be made configurable. The port on a particular environment is assigned dynamically. The capature the dynamically assigned port we use : process.env.PORT * If no port is assigned it will fallback to port = 3000 |
| **export PORT= 5000** | * The port can be assigned from command line too by setting the environment variable. |

## ROUTING PARAMETER & HANDLING HTTP REQUESTS

* Node has function to accept HTTP requests like GET, POST, DELETE,PUT
* REQUEST URL : <http://localhost:3000/api/courses?name=John>

|  |
| --- |
| const express = require('express')  const app = express();  app.use(express.json()); 🡨 THIS PARSE THE REQUEST BODY IN JSON FORMAT  const courses = [  { courseId: 1, courseName: 'Java' },  { courseId: 2, courseName: 'C++' }  ];  app.get('/api/courses', (req, res) => res.send(courses)); 🡨 GET REQUEST  app.get('/api/courses/:id', (req, res) => { 🡨 REST PARAMETERIZED & QS STRING - GET REQUEST  const filteredCourse = courses.find(course => course.courseId == req.params.id);  if (filteredCourse) {  res.send(`Hello ${req.query.name} - your courses id is ${filteredCourse.courseId} and name is ${filteredCourse.courseName}`);  } else {  res.status(404).send("No Courses Found");  }  });  app.listen(3000);  console.log(`Listening on port 3000 ...`); |

### POST REQUEST

|  |  |
| --- | --- |
| app.post('/api/courses', (req, res) => {  var course = {  courseId: courses.length + 1,  courseName: req.body.courseName  };  courses.push(course);  res.send(course);  }); | * Note : For req.body to work – We need to write app.use(express.json());. This will parse the request body in JSON format. |
| **POSTING DATA USING POSTMAN** | |
|  | |

### INPUT VALIDATION (USING JOI)

The most powerful schema description language and data validator for JavaScript.

|  |  |
| --- | --- |
| INSTALLING JOI | npm install joi |
| JOI API | <https://joi.dev/api/> |

### PUT REQUEST

Updating the resource(Example for updating the course)

|  |
| --- |
| app.put('/api/courses', (req, res) => {  const isCourseExist = courses.find(course => course.courseId == req.body.courseId);  if (!isCourseExist) {  res.status(404).send("No Courses Found");  return;  }  courses[isCourseExist.courseId - 1].courseName = req.body.courseName;  res.send(isCourseExist);  }); |

### DELETE REQUEST

Deleting the resource(Example for deleting the course)

|  |
| --- |
| app.delete('/api/courses/:id', (req, res) => {  const course = courses.find(course => course.courseId == parseInt(req.params.id));  if (!course) {  res.status(404).send("No Courses Found");  return;  }  res.send(courses.splice(courses.indexOf(course), 1));  }) |

### RESPONSE

* When ever express sends back the response it does set the appropriate header to the response header
* The response header depends upon the method used (send(); json()) while sending back the response

|  |  |
| --- | --- |
| app.post('/users',(req , res)=>{  res.send("Data");  }); |  |
| **SENDING THE JSON RESPONSE**  app.post('/users',(req , res)=>{  res.json("Data");  }); |  |

## MIDDLEWARE / MIDDLEWARE FUNCTIONS

|  |  |
| --- | --- |
|  | * When we get request to the express server that request goes to a pipeline called “Request Processing Pipeline”. * In this pipeline – we can have one-or-more middleware functions to process the request * Each middleware function either pass the processed request to next middleware using **next()** or terminates the request/ response cycle by sending the response. * The middleware functions are called in sequence – in which they are defined |

### BUILD IN MIDDLEWARE FUNCTION

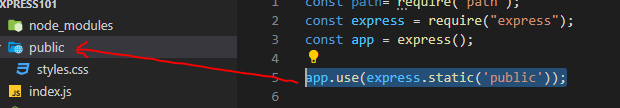
1. [express.static](https://expressjs.com/en/4x/api.html#express.static) serves static assets such as HTML files, images, and so on.
2. [express.json](https://expressjs.com/en/4x/api.html#express.json) parses incoming requests with JSON payloads. **NOTE: Available with Express 4.16.0+**
3. [express.urlencoded](https://expressjs.com/en/4x/api.html#express.urlencoded) parses incoming requests with URL-encoded payloads.

#### SERVING THE STATIC FILES

* This is a built-in middleware function in Express. It serves static files and is based on serve-static.
* The public folder can be used to serve static files like CSS or images.
* The express.static(“public”) indicates the folder/ path where the static files can recide
* The path to the “public” folder can be given as(using node’s path module) :

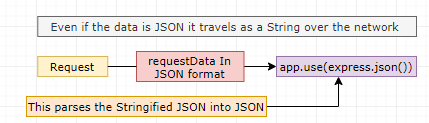
**app.use(express.static(path.join(\_\_dirname, 'public')));**

* We can have multiple public folders, which can be configured using this middleware
* To access the static files in public folder : <http://localhost:4000/styles.css>



#### PARSING REQUEST BODY

* express.json() : This parses the request body and if the request body is a JSON.It assign that value to request.body property



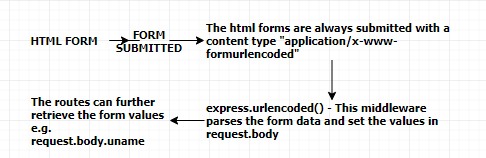
#### SERVING STATIC CONTENT [[express.static](https://expressjs.com/en/4x/api.html#express.static)]

|  |  |  |
| --- | --- | --- |
| CONFIGURING THE MIDDLEWARE FUNCTION:  app.use(express.static('public')); |  | The static file can be access from the root path  <http://localhost:7096/readme.txt>  <http://localhost:7096/courses.jpg> |

#### express.urlEncoded



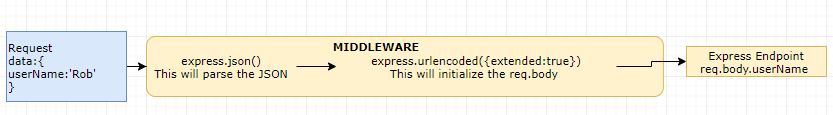
* HOW TO CALL THE MIDLLEWARE : app.use(express.urlencoded({ extended: true }));
* Request Format : path?**key=data&key2=data2**

EXAMPLE

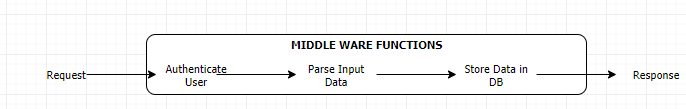
* The below is a posting of JSON data. The content type of the request is as in the tab;le
* To cater this type of request we need express.urlencoded middleware

|  |  |  |
| --- | --- | --- |
| REQUEST | EXPRESS | |
| <script>  function fetchData(){  var request = $.ajax({  method:'POST',  url:'users',  dataType:"json",  type:'application/json',  data:{  userName:'Rob'  },  success: (data,status,xhr)=>console.log(data.message),  error:(jqXhr, textStatus, errorMessage)=> console.log("Error",errorMessage)  });  }  </script>  <button onclick="fetchData()">Fetch Data</button> | | const express = require("express");  const app = express();  app.use(express.static('public'));  app.use(express.json());  app.use(express.urlencoded({extended:true}));  **app.post('/users',(req , res)=>{**  **res.json(req.body.userName);**  **});**  app.listen(4000,()=>{  console.log("Listening on Port 4000...");  }) |
|  |

#### PUTTING IT TOGETHER



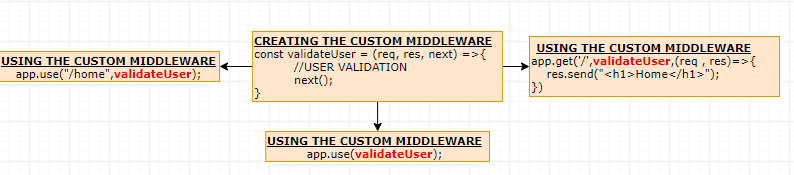
### CREATING CUSTOM MIDDLEWARE FUNCTION



* For each the intermediate step we can have a dedicated middleware function to pre-process the request and response
* We can any number of middleware before we send back the response to the client.
* All the routes are also middleware.
* The Locals variable of response object is heavily used by middle ware to pass object to templates

**EXAMPLE**:

* validateUser is a custom middleware
  + The middleware has access to request, response and next
  + The next() function pass the control to the next middleware
* app.use(validateUser) 🡪 Call the middle ware for all HTTP request
* app.use(‘’/home”,validateUser) 🡪 Call the middle ware for for all HTTP request having “home” route
* app.get('/',validateUser,(req , res)=>{ }); 🡪 Calls the middleware for GET call with route path “/”
  + Note: (req , res)=>{ } is also a middleware function .
  + In above case the next middleware for validateUser is **(req , res)=>{ }**

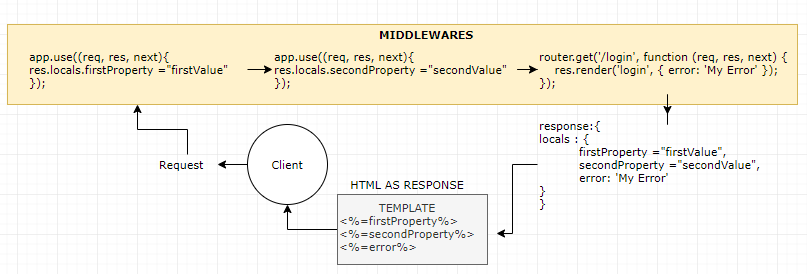


#### EXAMPLE – CUSTOM MIDDLEWARE

|  |  |
| --- | --- |
| const express = require('express');  const app = express();  const courses = [  { id: 1, name: 'Node' },  { id: 2, name: 'Java' },  { id: 3, name: 'C++' }  ];  app.use(express.json());  app.use((req, res) => {  console.log("Logging...")  req.next();  });  app.use((req, res) => {  console.log("Authencating...")  req.next();  }) | app.use(‘/admin’, (req, res) => {  console.log("Authencating...")  req.next();  })  app.get('/', (req, res) => {  res.send('Welcome to courses');  });  app.get('/api/courses', (req, res) => {  res.send(JSON.stringify(courses));  });  const port = process.env.PORT | 3000;  app.listen(port, () => {  console.log(`Listening on ${port}`);  }); |
| * Express usually has series of middleware functions. In the given example we have multiple middleware functions.(Highlighted in red) * The middleware functions are called in sequence – as they are declared * ***Each middleware either terminate the request -response cycle by sending the response or pass the control to next middleware function using next() function*** * App.get() [Route middleware function] is a middleware function which terminates the request -resposne cycle. * The middle ware functions can be included as a module too. * The middleware function can be also be executed for a specific routes as well (example in green) . This middleware will only execute for “/admin” route | |

### LOCALS

* All the middleware has an access to request , response and next() .
* The locals is and object encasulated in the response object . This locals object can be used to send data between the middlewares or to the template – which finally can be sent to the client.
* The property set in **locals** can be directly accessed in templates as shown below in the template(EJS)
* Locals property can also set in the routes (routes also are middleware)- as shown below



### THIRD PARTY MIDDLEWARE FUNCTION

* List of Third party middlewate : <https://expressjs.com/en/resources/middleware.html>

#### HELMET



|  |  |
| --- | --- |
| INSTALL HELMET | npm i helmet |
| INCLUDING THE HELMET MODULE | const express = require("express");  const helmet = require("helmet");  const app = express();  app.use(helmet()); |

## SERVER-SIDE RENDERING USING TEMPLATING ENGINE

* The template engine helps is Server Side rendering as the HTMLs (using template engine) in server side. The final HTML is sent back to client.

|  |  |
| --- | --- |
|  | * Templating Engines are used by express to send HTML markups to the client. * The HTMLs are created on the server side itself and send back to client as a HTML payload * We have multiple templating engines for express JS like   + **PUG(JADE)**   + **MUSTACHE**   + **HANDLEBARS**   + **EJS** |

### APP SETTING TABLE

* <https://expressjs.com/en/4x/api.html#app.set>

### HOW EXPRESS USES THE TEMPLATING ENGINE

1. Step 1: Install the templating engine node module
2. Step 2: Let express know about the view engine we want to use
3. Step 3: Let express know – the path, where views are located
4. Step 4: The client gets the final rendered HTML (already build in server side itself)

### USING TEMPLATING ENGINE - EJS

Reference - <https://ejs.co/>

|  |  |
| --- | --- |
| **INSTALLING EJS** | npm i ejs |
| **SET THE VIEW ENGINE – This will load the pug module and set the view engine as EJS** | app.set('view engine', 'ejs'); |
| **SET THE PATH OF VIEW FILES – default is “view” folder under root folder** | app.set('views','./views'); |
| **SETTING THE PATH USING PATH MODULE** | const path = require('path');  app.set('views', path.join(\_\_dirname, 'views')); |

#### EXAMPLE

|  |  |
| --- | --- |
| EJS | DATA PASSING FROM ROUTE |
| <h1> <%=heading%> </h1>  <p> <%=content%> </p> | app.get('/', function (req, res) {  res.render('index', { heading: 'Headline', content: 'Some Content' });  }); |

#### USING TEMPLATING ENGINE - HANDLEBARS

|  |  |
| --- | --- |
| INSTALLING HANDLEBARS | npm i hbs |
| SET THE VIEW ENGINE – This will load the pug module and set the view engine as EJS | app.set('view engine', 'hbs'); |
| SET THE PATH OF VIEW FILES – default is “view” folder under root folder | app.set('views','./views'); |
| SETTING THE PATH USING PATH MODULE | const path = require('path');  app.set('views', path.join(\_\_dirname, 'views')); |

#### USING TEMPLATING ENGINE - PUG

|  |  |
| --- | --- |
| INSTALLING PUG | npm i pug |
| SET THE VIEW ENGINE – This will load the pug module and set the view engine as Pug | app.set('view engine', 'pug'); |
| SET THE PATH OF VIEW FILES – default is “view” folder under root folder | app.set('views','./views'); |

##### SAMPLE PUG SYNTAX

|  |  |
| --- | --- |
|  | * The dynamic values are filled by an JS object e.g.   **app.get('/', (req, res) => {**  **res.render('index', { title: 'The Title', message: 'The Message' });**  **});**   * **“index” is name of the view (index.pug)** * The JS object has dynamic value for the template (title & message) |
|  |

#### STRUCTURING THE EXPRESS APPLICATION

To structure the express application – we might have to divide the file in follworing ways

|  |  |  |
| --- | --- | --- |
| View | This will have all the template code/file (.pug) |  |
| Routes | This will all the routes similar type |
| middleware | Contains the middleware Functions |
| Public | Contains the static contnet like css, Js, txt and images |
| **index.js**  const express = require('express');  const app = express();  const loggingMiddleWare = require('./middleware/logging-middleware');  const auth = require('./middleware/authentication');  const courses = require('./routes/courses');  const home = require('./routes/home');  app.set('view engine', 'pug');  app.set('views', './views');  app.use(express.json());  app.use(express.static('public')); 🡨 The middleware function configure the location of static content (public folder)  app.use(loggingMiddleWare);  app.use(auth);  app.use('/api/courses', courses);  app.use('/', home);  const port = process.env.PORT | 3000;  app.listen(port, () => {  console.log(`Listening on ${port}`);  }); | |

|  |  |
| --- | --- |
| **MIDDLEWARE FUNCTIONS** | |
| **Auth** | **logging** |
| const auth = (req, res) => {  console.log("Authencating...")  req.next();  }  module.exports = auth; | const loggingMiddleWare = (req, res) => {  console.log("Logging...");  req.next();  };  module.exports = loggingMiddleWare; |
| **ROUTES** | |
| **COURSES ROUTE** | **HOME ROUTE** |
| const express = require('express');  const router = express.Router();  const courses = [  { id: 1, name: 'Node' },  { id: 2, name: 'Java' },  { id: 3, name: 'C++' }  ];  router.get('/', (req, res) => {  res.send(JSON.stringify(courses));  });  router.get('/:id', (req, res) => {  const course = courses.filter(course => course.id == req.params.id);  if (course.length == 0) {  res.status(404);  res.send("Course Not Found");  }  console.log(course);  res.send(JSON.stringify(course));  });  router.post('/', (req, res) => {  course = {  id: courses.length + 1,  name: req.body.name  }  courses.push(course);  res.send(course);  });  module.exports = router; | const express = require('express');  const router = express.Router();  router.get('/', (req, res) => {  res.render('index', { title: 'The Title', message: 'The Message' });  });  module.exports = router; |
| * While setting up the courses route in index.js file – we configure the root path “**/api/courses**”. Then we can remove the root route from couses routings * In the router file we are don’t have to use the “app” object to configure different routes as we did in previous example instead we use   const router = express.Router();  to configure route   * All the middleware function as exposed as module – which in turn added in index.js in a desired in sequence.(because the middleware order matters) |

## EXPRESS GENERATOR

* Express Application Generator creates the express application skeleton (like maven)
* Reference : <https://expressjs.com/en/starter/generator.html>

|  |  |  |
| --- | --- | --- |
| **INSTALL EXPRESS GENERATOR GLOABALLY** | npm install -g express-generator | GENERATED STRUCTURE |
| **CREATING EXPRESS APP(myapp) WITH PUG AS TEMPLATING ENGINE** | express --view=pug myapp |  |
| **INSTALL DEPENDENCIES** | npm install |
| **RUNNING THE APP**  **Note: VS code uses “PowerShell”** | On Windows Command Prompt, use this command:  **set DEBUG=myapp:\* & npm start**  On Windows PowerShell, use this command: : **$env:DEBUG='myapp:\*'; npm start** |
| **ACCESS THE APP ON BROWSER** | <http://localhost:3000/> |

## DEBUG – EXPRESS

* Debugging is basic step in the development.
* The debug module in express help us enabling and disabling the “debug logs” based on the environment variables.

|  |  |
| --- | --- |
| **INSTALLING DEBUG MODULE** | npm i debug |
| **IMPORTING DEBUG MODULE** : - The debuggers are enabled based on the namespace. We can logically we divide the debugger into different namespaces. Here ‘app:start’ is a namespace | const debug = require("debug")('app:start'); |
| **USING THE DEBUG MODULE** | debug("Appllication Running!"); |

### EXAMPLE

|  |  |
| --- | --- |
| Module 1 | Module 2 (Routing ) |
| const express = require('express');  const debug = require("debug")('app:start');  const app = express();  const port = process.env.PORT | 3000;  app.listen(port, () => {  console.log(`Listening on ${port}`);  debug("Application Running!");  });  Here debuggers are created in 2 different namespaces. They can be enabled and disabled based on the environemnt variables (during the app start-up also) | const express = require('express');  const router = express.Router();  const routingDebug = require("debug")('app:router');  const courses = [  { id: 1, name: 'Node' },  { id: 2, name: 'Java' },  { id: 3, name: 'C++' }  ];  router.get('/:id', (req, res) => {  const course = courses.filter(course => course.id == req.params.id);  if (course.length == 0) {  res.status(404);  res.send("Course Not Found");  }  routingDebug(course);  res.send(JSON.stringify(course));  });  module.exports = router; |

### ENABLING DEBUG

|  |  |  |
| --- | --- | --- |
| **COMMAND** | **COMMAND LINE** | **POWERSHELL (VS CODE)** |
| Enable the debugging of app:start namespace | set DEBUG=app:start & nodemon .\index.js | $env:DEBUG='app:start' ; nodemon .\index.js |
| Enable the debugging of both “app:start” and “app:router”namespace | set DEBUG=app:start,app:router & nodemon .\index.js | $env:DEBUG='app:start','app:router' ; nodemon .\index.js |
| Enable the debugging of all namespace | set DEBUG=app:\* & nodemon .\index.js | $env:DEBUG='app:\*' ; nodemon .\index.js |
| The debug module code code the debug statement which are in different namespace. | |  |

## DEBUGGING CODE – USING CHROME DEVELOPER TOOL

To debug the Node Js code in chrome developer tool (We can able to do so because both node and chrome build upon same JS engine – V8)

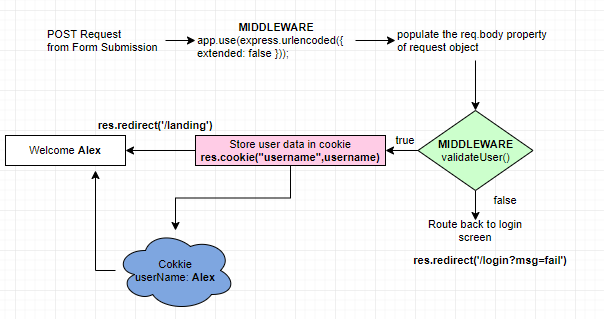
### STEPS TO DEBUG

|  |  |
| --- | --- |
| 1. Add a “debugger” statement in the code around the code to debug 2. Run command : npm inspect <file.js> **: npm inpect app.js** |  |
| 1. Open Chrome and hit : **chrome://inspect** 2. **It will show a remote target** |  |
| 1. Click on inspect . It will open the Chrome developer tool. 2. In the chrome developer tool. Browse to the folder – where the code has been placed. 3. Click the “Play” button to resume the debugging . 4. Hit Ctrl+ C in the terminal to stop the debugging. |  |

## REQUEST/RESPONSE OBJECT IN EXPRESS

|  |  |
| --- | --- |
| FETCH DATA FROM QS | If QS is “msg”. Fetching : req.query.msg |
| SET DATA IN COOKIE | res.cookie("username", username); |
| FETCH DATA FROM COOKIES | req.cookies.username |
| REDIRECTING TO A ROUTE (LANDING) | res.redirect('/landing'); |
| DELETE COOKIE | res.clearCookie("username"); |

### EXAMPLE - APPLICATION FLOW DIAGRAM



## DYNAMIC ROUTE PARAMS

|  |  |
| --- | --- |
| app.get('/story/:storyId',(req,res,next)=>{  const storyId = req.params.storyId;  res.send(`<h1>Story ${storyId}</h1>`)  }); | * StoryId is a route param which take dynamic params |

### ROUTERS

* In the above examples we were dumping all the routes in a single file. It always preferable to divide/organize the routes logically into separate files.
* If we create a middleware in a route file. That middleware will execute only for the route in that route file

|  |  |
| --- | --- |
|  |  |

* The below route file will intercept urls like : <http://localhost:4000/story/1> or <http://localhost:4000/story/2>
* Note : “:storyId” indicate that it’s a dynamic route param

|  |  |
| --- | --- |
| app.js | Route file(story.router.js) |
| let storyRouter = require('./routes/story.router');  app.use("/story",storyRouter); | const express = require("express");  let router = express.Router();  router.get('/:storyId',(req,res,next)=>{  const storyId = req.params.storyId;  res.send(`<h1>Story ${storyId}</h1>`)  });  module.exports = router; |

## TESTING NODE AND EXPRESS USING JEST

To write unit test using JEST for node and express code . We need couple of dev dependencies

### INSTALL MODULES

|  |  |  |
| --- | --- | --- |
| JEST | Jest is the framework which enable the unit test for Unit test for node and express code | **npm install jest** |
| SUPERTEST | Helps in the testing HTTP calls. | **npm install supertest** |

### UPDATE PACKAGE.JSON

|  |  |  |
| --- | --- | --- |
| * Add the test script . Here “Jest” will get started in watch mode. * To start Test suite : **npm test** * **Note :** The test file should have “.test” in the file name . e.g demo.test.js |  |  |

### WRITING TEST

|  |  |
| --- | --- |
| INDEX.JS [FUNCTION TO BE TESTED] | TEST |
| const calculateTotalBill = (billAmount, tipPercent)=> billAmount + billAmount \* tipPercent  const fahrenheitToCelsius = (tempInFahrenheit) => (tempInFahrenheit-32)\* (5/9);  const celsiousTofahrenheit= (tempInCelsius) => (tempInCelsius\*9/5)+ 32;  module.exports= {  calculateTotalBill,  fahrenheitToCelsius,  celsiousTofahrenheit  } | const {calculateTotalBill,fahrenheitToCelsius,celsiousTofahrenheit} = require('../index');  test('Calculate Bill',()=>{  expect(calculateTotalBill(10,.3)).toBe(13);  });  test('Calculate Fahrenheit Tempreature',()=>{  expect(fahrenheitToCelsius(32)).toBe(0);  });  test('Calculate Fahrenheit Tempreature',()=>{  **expect**(celsiousTofahrenheit(32)).**toBe**(89.6);  }) |
| **OUTPUT** |  |

### WRITING TEST FOR ASYNCHRONOUS CODE

Jest has multiple ways to test asynchronous code.

1. **USING CALLBACKS**
2. **USING PROMISES**
3. **USING ASYNC AND AWAIT**

### TESTING ASYNCHRONOUS CODE USING PROMISES

* Testing Asynchronous code is a special case. JEST never wait for the asynchronous code to complete. Due this behavior it shows the test to pass even if the assertion is failed.
* done() has to be called after the Promise is resolved and returned.

|  |  |
| --- | --- |
| **CODE TO TEST** | **TEST CASE** |
| const add = (operand1, operand2)=>{  return new Promise((resolve, reject) =>{  setTimeout(()=>{  resolve(operand1+operand2);  }, 500);  });  } | test(Promise Add Test',(done)=>{  add(2,3).then(result =>{  expect(result).toBe(5);  done();  })  }); |

### TESTING ASYNCHRONOUS CODE USING ASYNC AND AWAIT

* For async and await – We need to mark the test case as async

|  |  |
| --- | --- |
| **CODE TO TEST** | **TEST CASE** |
| const add = (operand1, operand2)=>{  return new Promise((resolve, reject) =>{  setTimeout(()=>{  resolve(operand1+operand2);  }, 500);  });  } | test('Async Await Add Test',async()=>{  const sum = await add(4,5);  expect(sum).toBe(9);  }); |

### TESTING THE EXPRESS API