Express.js

Guide Book on Web framework for Node.js

Rick L.

Express.js

Guide Book on Web framework for Node.js

By Rick L.

Table of Contents

Disclaimer

While all attempts have been made to verify the information provided in this book, the author does assume any responsibility for errors, omissions, or contrary interpretations of the subject matter contained within. The information provided in this book is for educational and entertainment purposes only. The reader is responsible for his or her own actions and the author does not accept any responsibilities for any liabilities or damages, real or perceived, resulting from the use of this information.

The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within this book are for clarifying purposes only and are the owned by the owners themselves, not affiliated with this document.

Introduction

ExpressJS is one of the most useful and interesting Node frameworks. It is used for the development of both web and mobile applications. These types of applications are the mostly widely used in software environments. This shows the need for you to know how this framework can be used for development. The framework has numerous modules which can be used for the purpose of enhancing the look and feel of your applications.

Chapter 1- Overview of ExpressJS

ExpressJS is a Node framework which is very flexible and provides developers with numerous features for the development of web and mobile applications. The framework can be used for the development of APIs. In this book, we will discuss the various features of ExpressJS.

Chapter 2- Session

To install this in Express, just execute the following command:

$ npm install express-session

The above command will install the session into your system. To use it in your program, you have to use the "*require"* command. This is shown below:

var **session** = require**('express-session')**

The session middleware can be created by use of the given "*options."* You should know that the data for the session should not be saved in the cookie itself, but here, only the session ID is stored. The data for the session is stored on the server side for the app. However, "*MemoryStore,"* which is the default server-side session storage was not developed for use in a production environment. In most cases, it will leak and not scale past a single process. It was developed for the purpose of debugging.

“express-session” will accept the following properties in the object for options:

Cookie

These are the settings for the cookie of the session ID. Consider the example given below:

{ path: ‘/’, httpOnly: true, secure: false, maxAge: null }.

Genid

This is a function which is called for generation of a new session ID. The function provided should return a string which will be used as the session ID. “req” is given to the function as the first argument in case there is a need for a value to be attached to the “req” when the ID is being generated. The default value for this will be a function which can use “uid2” for the generation of the IDs.

Note: To avoid confliction of the sessions, make sure that the generated IDs are unique, that is, they should be different from each other. Consider the example given below which shows how this can be done:

```
application.use(session({
    genid: function(req) {
    return genuuid() // using UUIDs for the session IDs
},
    secret: ‘ my secret ‘
}))
```

Name

This is the name of the session ID cookie which is to be set in the response. Note that it is read from the request. The default value for this is the “*connect.sid.*” For those who have multiple apps which are running on the same host, then the session cookies have to be separated from each other. To achieve this, one has to set different names in each of the apps.

Proxy

Whenever you are setting secure cookies, you have to trust the reverse proxy. This can be done via the header for "*X-Forwarded-Proto.*" The default value for this is "*undefined.*"

The possible values for this are explained below:

1. "true"- the header "X-Forwarded-Proto" will be used.

2. "false"- all of the headers will be ignored, and the connection will be considered only if a direct "TLS/SSL" connection exists.

3. "Undefined"- this will use the settings for "trust proxy" from the Express itself.

Resave

This will force the session to be saved back to the session store. This happens whether or not the session was modified during the request. The necessity of this will depend on your session store. However, if a client makes parallel requests, race conditions may be created.

Rolling

The cookie will be forced to be set on each of the responses. The expiration date is also reset. The default value for this is "*false.*"

saveUninitialized

With this, a session which was not initialized will be saved to the session store. An uninitialized session is one which is new and not modified in any way. The default setting for this is "*true.*" However, this has deprecated and is expected to change in the future.

Required option

This is the secret which is used for signing the cookie for the session ID. It can be made up of an array of secrets or a string just for a single secret. In case you provide an array of secrets, only the first element in the array will be used for signing the cookie for the session ID. The rest of the elements will be used for verification of the signature in requests.

Store

This is the instance of the session store. Its default is a new "*MemoryStore*" instance.

Consider the example given below:

```
var application = express()
    application.set('trust proxy', 1) // trusting the first proxy
application.use(session({
```

```
    secret: 'my secret',
    resave: false,
    saveUninitialized: true,
cookie: { secure: true }
}))
```

We need to be able to use the cookies in a production environment, and at the same time allow for testing in a development environment. The setup can be enabled by use of the "*NODE_ENV.*" Consider the example given below, which shows how this can be done:

```
var application = express()
var session = {
    secret: ' my secret ',
    cookie: {}
}
if (application.get('env') === 'production') {
    application.set('trust proxy', 1) // trusting the first proxy
    session.cookie.secure = true // serving secure cookies
}
    application.use(session(session))
```

The default setting for "*cookie.maxAge*" is "*null*," and it will never expire, which means that the cookie will become a browser-session cookie. The cookie is only removed once the user has closed the browser. The session is also removed.

req.session

For session data to be stored and accessed, you should use the property "req.session," and the store initializes this as a JSON. The session objects will then be left fine. Consider the example given below, which shows a view counter which is specific to a user:

```
application.use(session({ secret: ' my secret ', cookie: { maxAge: 50000 }}))

application.use(function(req, res, next) {
var session = req.session
if (session.views) {
session.views++
res.setHeader('Content-Type', 'text/html')
res.write('<p>views: ' + session.views + '</p>')
```

```
res.write('<p>will expires in: ' + (session.cookie.maxAge / 1000) + 's</p>')

res.end()

} else {

session.views = 1

res.end(' This is a demo for sessions. Refresh the page!')

}

})
```

Session.regenerate()

This method is invoked when we want to generate the session. After this, a new session instance and SID will be initialized at the "*req.session.*" This is shown below:

```
req.session.regenerate(function(error) {

// a new session should be added here

})
```

Session.destroy()

With this, the session is destroyed, and the “req.session” removed. It will then be regenerated in the next request. This is shown below:

```
req.session.destroy(function(error) {
// a session cannot be accessed here.
})
```

Session.reload()

This is used for reloading the session as shown below:

```
req.session.reload(function(error) {
// the session has been updated
})
```

Session.save()

This is for saving the session as shown below:

req.session.save(function**(error) {**

// session has been saved

})

Consider the example given below, which uses the “express-session” so as to store the page views for the user:

var **express** = require**(‘express’)**

var **purl** = require**(‘parseurl’)**

var **session** = require**(‘express-session’)**

var **application** = **express()**

application.use(session({

secret: **‘ my secret ‘,**

resave: false**,**

saveUninitialized: true

```
}))

application.use(function (req, res, next) {

var vws = req.session.views

if (!vws) {

vws = req.session.views = {}

}

// getting the url pathname

var pathname = purl(req).pathname

// counting the views

vws[pathname] = (vws[pathname] || 0) + 1

next()

})

application.get('/foo', function (req, res, next) {

res.send('you have viewed this page ' + req.session.vws['/test'] + ' times')


})

application.get('/bar', function (req, res, next) {

res.send('you have viewed this page ' + req.session.vws['/bar'] + ' times')


})
```

That is how sessions can be used in ExpressJS.

Chapter 3- serve-index

This is used for serving pages having listings of a directory to a particular path. It can be installed by executing the command given below:

$ npm install serve-index

The above command will install the "*serve-index*" to your computer. To use it in your program, it has to be included or called by use of the "*require*" method. This is shown below:

var **svIndex** = require**('serve-index')**

The API will then be included in your program.

serveIndex(path, options)

This will return the middleware which serves the index of the directory in the “path” you have given. The path here is based off of the value for “*req.url.*” For you to change the URL base, make use of the “*app.use.*”

The following properties are accepted by the serve index in the options object:

Filter

This filter function is applied to files. Its default value is “false.” It is called for each of the files, and it uses the signature “*filter(filename, index, files, dir).”* “Filename” will be the file’s name, “*index”* will be the array index, and “files” is the array of the files, while “*dir”* will be the absolute path in which the file is located.

Hidden

This is used for displaying the hidden files. Its default value is “*false.”*

Icons

This is for displaying icons. It default value is “*false.*”

Stylesheet

This is an optional path which leads to the CSS stylesheet. Its default value is the built-in stylesheet.

Template

This is an optional path which leads to an HTML template, and is used for rendering the HTML string.

View

This is the display mode.

Consider the example given below, which shows how all the above can be used:

var **fhandler** = require**(‘finalhandler’)**

var **http** = require**(‘http’)**

var **svIndex** = require**(‘serve-index’)**

var **svStatic** = require**(‘serve-static’)**

// The Serve directory works to indexe for public/ftp folder (having icons)

var **index** = **svIndex(‘public/ftp’, {‘icons’**: true**})**

// Serving up the public/ftp folder files

var **serve** = **svStatic(‘public/ftp’)**

// Creating a server

var **server** = **http.createServer(**function **onRequest(req, res){**

var **done** = **fhandler(req, res)**

serve(req, res, function **onNext(error) {**

if **(error)** return **done(error)**

index(req, res, done)

})

})

// Listen

server.listen(3000**)**

To serve the directory indexes with express, do the following:

var **express** = require**(‘express’)**

var **svIndex** = require**(‘serve-index’)**

var **application** = **express()**

// Serving URLs like the /ftp/thing as public/ftp/thing

application.use(‘/ftp’, svIndex(‘public/ftp’, {‘icons’: true**}))**

application.listen()

Chapter 4- cookie- Sessions

This is a module which is used for providing guest sessions so that each of the visitors will have a session, whether authenticated or not. In case the session is new, the property “Set-Cookie” will be produced, regardless of whether you are populating the session.

It can be installed by use of the following command:

$ npm install cookie-session

For the API to be used in the program, we have to use the “*require”* command as shown below:

var **cookieSession** = require**(‘cookie-session’)**

The following options are accepted in the cookie session in the options object:

Name

This is the name of the cookie which is to be set. Its default value is “*session*”.

Keys

This is the list of the keys to be used for signing and verifying the cookie values. Cookies which have been set are always signed with the “keys[0].” The rest of the keys will then be valid for the purpose of verification, and this will allow for the rotation to be done.

Secret

This is a string which will be used as a secret key in case the “keys” are not provided. There is also a number of cookie options which you need to know how to use.

If you need to destroy a particular session, do it as follows:

req.session = null

Consider the simple view counter example given below. Here is the example:

```
var cookieSession = require('cookie-session')
var express = require('express')
var application = express()
appliation.set('trust proxy', 1) // trusting the first proxy
application.use(cookieSession({
    name: 'session',
    keys: ['key1', 'key2']
}))
application.use(function (req, res, next) {
    // Updating the views
    req.session.views = (req.session.views || 0) + 1
```

```
    // Writing the response
    res.end(req.session.views + ' views')
})
application.listen(3000)
```

Consider the next example given below:

```
var cookieSession = require('cookie-session')
var express = require('express')
var application = express()
application.set('trust proxy', 1) // trusting the first proxy

application.use(cookieSession({
name: 'session',
keys: ['key1', 'key2']
}))
// This will allow you to set req.session.maxAge for letting certain sessions

// have a different value other than the default one.
application.use(function (req, res, next) {
req.sessionOptions.maxAge = req.session.maxAge || req.sessionOptions.maxAge

})
// … the logic should be added here …
```

You have to note that the entire session object will be encoded and stored in a cookie. The maximum cookie size limit on the different browsers can then be exceeded. If the session object is large enough and it can exceed the limit of the browser when it has been encoded, what will happen in most cases is that the browser will refuse or avoid storing the cookie.

Chapter 5- Morgan in ExpressJS

This is logger middleware for the HTTP request for Node.js. To use the API in your program, you have to use the "*require"* keyword as shown below:

var **morgan** = require**('morgan')**

The following format is used for creating this:

morgan(format, options)

As shown in the above format, we have two parameters or arguments.

The following properties are accepted by Morgan in the options object:

Immediate

The log line should be written on request rather than on response. This will mean that the requests will always be logged even if the server itself crashes, but it will be impossible for us to log the data from the response.

Skip

This is a function used for the determination of whether or not logging was skipped. The default value for this is "*false,*" Consider the example given below, which shows how the function can be called:

// EXAMPLE: only the log error responses

morgan('combined', {

skip: function **(req, res) {** return **res.statusCode** < 400 **}**

})

Stream

This is an output stream which is used for writing log lines, and its default value is "*process.stdout.*"

Consider the example given below, which shows how all the requests can be logged in the Apache combined format to the STDOUT. Here is the example:

```
var express = require('express')
var morgan = require('morgan')
var application = express()
application.use(morgan('combined'))
application.get('/', function (req, res) {
res.send('hello, there!')
})
```

vanilla http server

An example which demonstrates how this can be done is given below:

var **fhandler** = require**(‘finalhandler’)**

var **http** = require**(‘http’)**

var **morgan** = require**(‘morgan’)**

// creating the “middleware”

var **logger** = **morgan(‘combined’)**

http.createServer(function **(req, res) {**

var **done** = **finalhandler(req, res)**

logger(req, res, function **(error) {**

if **(error)** return **done(error)**

// responding to the request

res.setHeader(‘content-type’, ‘text/plain’)

res.end(‘hello, there!’)

})

})

Writing Logs to a File

Consider the example given below, which shows how the requests can be logged in Apache combined format to our file “*access.log*”:

var **express** = require**(‘express’)**

var **fs** = require**(‘fs’)**

var **morgan** = require**(‘morgan’)**

var **application** = **express()**

// creating a write stream (in the append mode)

var **aLogStream** = **fs.createWriteStream(__**dirname + **‘/access.log’, {flags**: **‘a’})**

// setting up the logger

application.use(morgan(‘combined’, {stream: **aLogStream}))**

application.get(‘/’, function **(req, res) {**

res.send**(‘hello, there!’)**

})

log file rotation

In the example given below, the app will log all the requests in the Apache combined format to any of the log files in our directory “log/” by use of the file-stream-rotator module. The example is as follows:

```
var FStreamRotator = require('file-stream-rotator')

var express = require('express')

var fs = require('fs')

var morgan = require('morgan')

var application = express()

var lDirectory = __dirname + '/log'

// ensuring the log directory exists

fs.existsSync(lDirectory) || fs.mkdirSync(lDirectory)

// creating a rotating write stream

var aLogStream = FStreamRotator.getStream({

filename: lDirectory + '/access-%DATE%.log',

frequency: 'daily',

verbose: false

})

// setting up the logger

application.use(morgan('combined', {stream: aLogStream}))

application.get('/', function (req, res) {

res.send('hello, there!')

})
```

Custom token formats

In the next app, we will use the custom token formats. It works by adding an ID to all of the requests, and will then display it by use of the “:id” token. Here is the app:

var **express** = require**(‘express’)**

var **morgan** = require**(‘morgan’)**

var **id** = require**(‘node-uuid’)**

morgan.token(‘id’, function **getId(req) {**

return **req.**id

})

var **application** = **express()**

application.use(assignid)

app.use(morgan(‘:id :method :url :response-time’))

app.get(‘/’, function **(req, res) {**

res.send**(‘hello, world!’)**

```
})


function assignId(req, res, next) {

req.id = uuid.v4()

next()

}
```

Chapter 6- CORS

This is a Node.js package which provides an ExpressJS/Connect middleware which can be used for enabling of the various options for CORS.

To install it via npm command, execute the command given below:

$ npm install cors

This can be used as shown below:

```
var express = require('express')
, cors = require('cors')
, application = express();
application.use(cors());
application.get('/products/:id', function(req, res, next){
res.json({msg: 'This has CORS enabled for all the origins!'});
```

```
});

application.listen(80, function(){

console.log(‘ web server which is CORS enabled listening on port 80’);


});
```

To enable CORS for a single route, do it as follows:

```
var express = require(‘express’)

, cors = require(‘cors’)

, application = express();

application.get(‘/products/:id’, cors(), function(req, res, next){

res.json({msg: ‘This has the CORS enabled for all the origins!’});

});

application.listen(80, function(){

console.log(‘ web server which is CORS enabled now listening on port 80’);


});
```

To configure the CORS, do it as follows:

```
var express = require('express')
, cors = require('cors')
, application = express();
var cOptions = {
origin: 'http://sample.com'
};
application.get('/products/:id', cors(cOptions), function(req, res, next){
res.json({msg: 'This has CORS enabled only for sample.com.'});
});
application.listen(80, function(){
console.log(' web server which is CORS-enabled is now listening on port 80');
});
```

Configuration of CORS w/ Dynamic Origin can be done as follows:

```
var express = require('express')
```

```
, cors = require('cors')

, application = express();

var whitelist = ['http://sample1.com', 'http://sample2.com'];

var cOptions = {

origin: function(origin, callback){

var orIsWhitelisted = whitelist.indexOf(origin) !== -1;

callback(null, orIsWhitelisted);

}

};

application.get('/products/:id', cors(cOptions), function(req, res, next){


res.json({msg: 'This has CORS enabled for the whitelisted domain.'});


});

application.listen(80, function(){

console.log('CORS-enabled web server listening on port 80');

});
```

How to enable CORS Pre-Flight

Certain requests for CORS are considered to be complex, and these need an initial OPTIONS request. A CORS request which uses the HTTP verb is considered to be complex compared to the one which uses POST/GET/HEAD or the one using custom headers. For pre-fighting to be enabled, a new OPTIONS handler has to be added for the route which needs to be supported. This is shown below:

```
var express = require('express')
, cors = require('cors')
, application = express();
application.options('/products/:id', cors()); // enabling a pre-flight request for the DELETE request

application.del('/products/:id', cors(), function(req, res, next){
res.json({msg: 'This has CORS enabled for all the origins!'});
});
application.listen(80, function(){
console.log(' The web server which is CORS-enabled is listening on port 80');

});
```

Pre-flight can also be enabled across the board as shown below:

application.options(‘*’, cors()); // including before the other routes

Asynchronous configuration of CORS

```
var express = require('express')
, cors = require('cors')
, application = express();
var whitelist = ['http://sample1.com', 'http://sample2.com'];
var cOptionsDelegate = function(req, callback){
var cOptions;
if(whitelist.indexOf(req.header('Origin')) !== -1){
cOptions = { origin: true }; // reflecting (enabling) the requested origin in our CORS response

}else{
cOptions = { origin: false }; // disabling the CORS for our request
}
callback(null, cOptions); // callback needs two parameters: error and the options

};
application.get('/products/:id', cors(cOptionsDelegate), function(req, res, next){

res.json({msg: 'This has CORS enabled for the whitelisted domain.'});

});
application.listen(80, function(){
```

console.log**(‘ web server with CORS enabled is listening on port 80’);**

});

That is how it can be done.

Chapter 7- Express-Paginate

This is to be used together with the pagination plugins for databases such as the MongoDB.

To install, execute the command given below:

npm install -S express-paginate

For it to be used in the program, one has to use the "*require*" keyword so as to include it into the program This is shown below:

var **paginate** = require**('express-paginate');**

With "*paginate,*" a new instance of "*express-paginate*" will be created. Consider the example given below:

```
// # app.js

var express = require('express');

var paginate = require('express-paginate');

var application = express();

// keeping this before all the routes that will use pagination

application.use(paginate.middleware(10, 50));

application.get('/users', function(req, res, next) {
// we are assuming that `Users`was previously defined in our example

// as `var Users = database.model('Users')` for those using `mongoose`

// and that the Mongoose plugin `mongoose-paginate` has been added.

// to the model for Users via the `User.plugin(require('mongoose-paginate'))`

Users.paginate({}, { page: req.query.page, limit: req.query.limit }, function(error,
users, pageCount, itemCount) {

if (error) return next(error);

res.format({

html: function() {

res.render('users', {

users: users,
```

```
pageCount: pageCount,

itemCount: itemCount

});

},

json: function() {

// Created by the API response for list objects

res.json({

object: 'list',

has_more: paginate.hasNextPages(req)(pageCount),

data: users

});

}

});

});

});

application.listen(3000);
```

The second code for the app should be as follows:

```
//- users.jade

h1 Users

//- a link for sorting by name will be created by this

//- you must have noticed that we only have to pass the querystring param


//- which is to be modified here, but not the entire querystring

a(href=paginate.href({ sort: 'name' })) Sorting by name

//- this will assume that you have `?age=1` or `?age=-1` in the querystring


//- so the values will be negated by this and you will be given

//- an opposite sorting order ( that is, desc with -1 or asc with 1)

a(href=paginate.href({ sort: req.query.age === '1' ? -1 : 1 })) Sort by age


ul

each myuser in users

li= myuser.email

include _paginate
```

The third code should be as follows:

```
//- _paginate.jade

//- In this example, we will make use of the Bootstrap 3.x pagination classes


if paginate.hasPreviousPages || paginate.hasNextPages(pageCount)

.navigation.well-sm#pagination

ul.pager
```

if **paginate.hasPreviousPages**

li.previous

a(**href=paginate.**href**(**true**)**).prev

i.fa.fa-arrow-circle-left

| Previous

if **paginate.hasNextPages(pageCount)**

li.next

a(**href=paginate.**href**()**).next

| Next

i.fa.fa-arrow-circle-right

Chapter 8- Multer

This is middleware in Node.js which is used to handle "*multipart/form-data.*" This is used for the purpose of uploading files to the server. To maximize its efficiency, it is built on top of "busboy." You have to note that any form which is not multipart cannot be handled by Multer.

The installation of this can be done by execution of the following command:

$ npm install —save multer

It works by adding a "*body"* and a "*file/files"* object to the "*request"* object. In the "*body"* object, the values for the form fields will be stored. The "*file/files"* object will contain the files which have been uploaded to the server.

Consider the example given below which shows how this can be done:

var **express** = require**('express')**

var **multer** = require**('multer')**

var **uploadfile** = **multer({ dest**: **'uploads/' })**

var **application** = **express()**

```
application.post('/profile', uploadfile.single('avatar'), function (req, res, next) {


// req.file is our `avatar` file

// req.body is for holding the text fields, if any are available

})

application.post('/photos/upload', uploadfile.array('photos', 12), function (req, res, next) {


// req.files is an array of `photos` for files

// req.body will contain the text fields, if there were any

})



var cUpload = uploadfile.fields([{ name: 'avatar', maxCount: 1 }, { name: 'gallery', maxCount: 8 }])


application.post('/cool-profile', cUpload, function (req, res, next) {


// req.files is our object (String -> Array) where fieldname is our key, and the value is an array of the files


//

// example

//  req.files['avatar'][0] -> File

//  req.files['gallery'] -> Array

//

// req.body which contains the text fields, if any were available

})
```

For those who are in need of handling a text which is in multipart form, any of the Multer methods can be used. These methods include the "*single(),*" "*.array(),*" *and* "*fields().*" Consider the example given below, which shows how the "*.array()*" can be used:

var **express** = require**('express')**

var **application** = **express()**

var **multer** = require**('multer')**

var **uploadfile** = **multer()**

application.post('/profile', uploadfile.array(), function **(req, res, next) {**

// req.body has our text fields

})

With Multer, an options object is accepted, in which the most basic one is our "*dest*" property, and it specifies the destination of our uploaded files. If the options object is omitted, the files which you upload will be kept in the memory other than in the disk.

To avoid conflicts brought by naming, the Multer will rename the files and this is its

default setting. The rest of the function can be customized according to what you need.

There exists a variety of options which can be passed to the Multer, so make sure that you understand how these can be passed.

In a normal web application, only the "*dest*" property might be required, and this can be configured as shown below:

var **uploadfile** = **multer({ dest**: **'uploads/' })**

For those who are in need of exercising a greater control on your uploaded files, just use the "*storage*" property other than the "*dest*" property. "*DiskStorage*" and "*MemoryStorage*" storage engines are shipped with Multer. You can use third parties so as to access more storage engines.

If you need to exercise full control of your files which are stored on the disk, use the disk storage engine. This is shown below:

var **storage** = **multer.diskStorage({**

destination: function **(req, file, cb) {**

cb(null**, '/tmp/file-uploads')**

},

filename: function **(req, file, cb) {**

cb(null**, file.fdname** + **'-'** + Date**.now())**

}

})

var **uploadfile** = **multer({ storage**: **storage })**

The only two options which are available include the "*filename"* and "*destination."* They are the functions which determine how the files are to be stored. The"*destination"* property will determine the folder in the directory in which the uploaded file is to be stored. This can also be specified as a path in the program. If this property is not specified, then the default path is used in the operating system.

File Filter

This property can be used for setting the files which are to be uploaded and the ones which are to be skipped. This function should be used as shown in the example given below:

function **fileFilter (req, file, cb) {**

// The function will call the `cb` with a boolean

// to indicate that the file should be accepted

// if the file is to be rejected, pass `false`, like so:

cb(null**,** false**)**

// for the file to be accepted, pass `true`, like so:

cb(null**,** true**)**

// you can pass an error since something may go wrong at some point:

cb(new **Error(‘I have no clue!’))**

}

Error Handling

When an error has been encountered, the Multer will pass it to Express. The standard Express error way can be used for the purpose of displaying an error.

For you to catch the errors which are originating from the Multer, just call the middleware function on your own. This is shown in the example given below:

var **uploadfile** = **multer().single('avatar')**

application.post('/profile', function **(req, res) {**

uploadfile(req, res, function **(error) {**

if **(error) {**

// An error has occurred when uploading

return

}

// Everything was okay

})

})

That is how it can be used.

Chapter 9- Compression

This is middleware available in Node.js. It supports the following compression codes:

- deflate
- gzip

To install this middleware in your system, execute the following command:

$ npm install compression

To include this API in your program, use the "*require"* keyword as shown below:

var **compression** = require**('compression')**

There are different options which can be used so as to return the compression middleware.

.filter

This is the default filter function. It is used for construction of a custom filter function which will be an extension of our default function. The example given below shows how this can be used:

application.use(compression({filter: **compress}))**

function **compress(req, res) {**

if **(req.**headers**[‘x-no-compression’]) {**

// responses having this header request should not be compressed.

return false

}

// fallback to standard filter function

return **compression.filter(req, res)**

}

red.flush()

With this module, the response which has been compressed partially is flushed to the client. Let us give examples of these.

express/connect

This module can be used by use of the "*app.use,*" which is available in either Express and connect. Requests passed through the middleware are always compressed. Consider the example given below which shows how this can be done:

var **compression** = require**('compression')**

var **express** = require**('express')**

var **application** = **express()**

// compressing all of the requests

application.use(compression())

// adding all the routes

That is how it can be done.

Server-Sent Events

The working of this module with server-side is not done out of the box. For content to be compressed, a window for the output has to be buffered up so that we can get a good compression.

To do all this, we have to call the *"red.flush()"* for those who need the data written so as to make it to our client. This is shown below:

var **compression** = require**('compression')**

var **express** = require**('express')**

var **application** = **express()**

// compressing the responses

application.use(compression())

// the server-sent event stream

application.get('/events', function **(req, res) {**

res.setHeader('Content-Type', 'text/event-stream')

res.setHeader('Cache-Control', 'no-cache')

// sending a ping approx after every 2 seconds

```
var timer = setInterval(function () {

res.write('data: ping\n\n')

// !!! our most important part for the program

res.flush()

}, 2000)

res.on('close', function () {

clearInterval(timer)

})

})
```

Chapter 10- csurf

This is a token middleware for CSRF. It acts as a middleware for protection purposes. For it to be used, one has to begin by initializing a cookie-parser or a session middleware. It can be installed into one's computer by executing the following command:

$ npm install csurf

To use the API in your program, use the "*require*" command as shown below:

var **csurf** = require**('csurf')**

That is how it can be included into the program.

The options for this module take multiple and different objects which you can learn how to use.

Consider the example given below, which shows a code for the server-side which can be used for generation of the CSRF form which can be used for posting back. This is the example:

```
var cParser = require('cookie-parser')
var csrf = require('csurf')
var bParser = require('body-parser')
var express = require('express')
// setting up the route middlewares
var csrfProtection = csrf({ cookie: true })
var pForm = bParser.urlencoded({ extended: false })
// creating an express app
var application = express()
// parsing cookies
// this is needed because "cookie" is true in the csrfProtection
application.use(cParser())
application.get('/form', csrfProtection, function(req, res) {
// passing the csrfToken to our view
res.render('send', { csrfToken: req.csrfToken() })
})
application.post('/process', pForm, csrfProtection, function(req, res) {
```

res.send**(‘The data is now being processed’)**

})

Inside your view, the value for csrfToken should be set to the value of a hidden input field “_csrf.” This is shown below:

<form action=”/process” method=“POST”>

<input type=“hidden” name=“_csrf” value=”{{csrfToken}}”>

Best color: <input type=“text” name=“bestColor”>

<button type=“submit”>Submit</button>

</form>

That is how simply it can be done.

Ignoring the Routes

In API areas for the websites in which we will have our requests fully authenticated, the CSRF should be disabled. API routing can be ignored by the use of routers and Express. Consider the example given below, which shows how this can be done:

```
var cParser = require('cookie-parser')

var csrf = require('csurf')

var bParser = require('body-parser')

var express = require('express')

// setting up the route middlewares

var csrfProtection = csrf({ cookie: true })

var pForm = bParser.urlencoded({ extended: false })

// creating an  express app

var application = express()

// parsing cookies

// this is needed because "cookie" is true in the csrfProtection

application.use(cParser())

// creating an api router
```

```
var api = createApiRouter()

// mounting the api before the csrf is appended to our app stack

application.use('/api', api)

// adding the csrf, after the "/api" has been mounted

appliation.use(csrfProtection)

application.get('/form', function(req, res) {

// passing the csrfToken to our view

res.render('send', { csrfToken: req.csrfToken() })

})

application.post('/process', pForm, function(req, res) {

res.send('csrf was required for getting here')

})

function createApiRouter() {

var router = new express.Router()

router.post('/getProfile', function(req, res) {

res.send(' there is no csrf here to get')

})

return router

}
```

That is how it can be done.

Custom error handling

When the validation of the CSRF has failed, an error with “err.code === ‘EBADCSRFTOKEN’” will be thrown. This can also be used so as to display error messages. Consider the example given below, which shows how this can be done:

```
var bParser = require('body-parser')

var cParser = require('cookie-parser')

var csrf = require('csurf')

var express = require('express')

var application = express()

application.use(bParser.urlencoded({ extended: false }))

application.use(cParser())

application.use(csrf({ cookie: true }))


// the error handler

application.use(function (error, req, res, next) {

if (error.code !== 'EBADCSRFTOKEN') return next(error)

// handling the CSRF token errors is done here

res.status(403)
```

res.send**(‘ The form was tampered with’)**

})

That is how it can be done.

Chapter 11- body-parser

This is the Node.js middleware for body parsing. However, it does not handle multipart bodies because of their large and complex nature. When dealing with multipart bodies, you might have to deal with the following modules:

- multer
- busboy and connect-busboy
- formidable
- multiparty and connect-multiparty

The module provides the user with the following parsers:

- URL-encoded form body parser
- Raw body parser
- JSON body parser
- Text body parser

To install it, just execute the following command:

$ npm install body-parser

To use the API, include it in your program by use of the “require” command. This is shown below:

var **bParser** = require**(‘body-parser’)**

With the object “*bParser,”* various factories are exposed for the purpose of creation of middlewares. The property “req.body” will be populated by all middlewares by use of the parsed body or an error will be parsed to the callback.

In Express, a generic JSON and URL-encoded parser can be added as a top-level middleware. With this, the bodies of all the incoming requests will be parsed. Consider the simple example given below, which shows how this can be done:

```
var express = require('express')

var bParser = require('body-parser')

var application = express()

// parsing the application/x-www-form-urlencoded

application.use(bParser.urlencoded({ extended: false }))

// parsing the application/json

application.use(bParser.json())

application.use(function (req, res) {

res.setHeader('Content-Type', 'text/plain')

res.write('you posted:\n')

res.end(JSON.stringify(req.body, null, 2))

})
```

express route-specific

Body parsers can be added to the routes which need them. This is possible in Express and it can easily be done. This is the most easy and recommended way that the body-parser can be used in Express. Consider the example given below, which shows how this can be done:

```
var express = require('express')
var bParser = require('body-parser')
var application = express()
// creating an application/json parser
var jParser = bodyParser.json()
// creating an application/x-www-form-urlencoded parser
var urlenParser = bParser.urlencoded({ extended: false })
// POST /login which will get the urlencoded bodies
application.post('/login', urlenParser, function (req, res) {
if (!req.body) return res.sendStatus(400)
res.send('welcome, ' + req.body.username)
})
// POST /api/users will get the JSON bodies
application.post('/api/users', jParser, function (req, res) {
if (!req.body) return res.sendStatus(400)
// create a user in the req.body
})
```

Changing the content type for parsers

With all parsers, a type option is accepted and with this, the content-type will be changeable and this should be the one to be parsed by the middleware. Consider the example given below, which shows how this can be done:

// parsing the various and different custom JSON types as a JSON

application.use(bodyParser.json({ type: **‘application/*+json’ }))**

// parsing a custom thing into the Buffer

application.use(bParser.raw({ type: **‘application/vnd.custom-type’ }))**

// parsing the HTML body into the string

application.use(bParser.text**({ type**: **‘text/html’ }))**

Chapter 12- Flash

This is the simplest manner in which Express can be implemented. To install it, execute the following command:

npm i flash

Consider the example given below, which shows how this can be done:

```
application.use(session()); // the session middleware
application.use(require('flash')());
application.use(function (req, res) {
// flashing a message
req.flash('info', 'hello there!');
next();
})
```

Consider the second example given below:

for **msge** in **flash**

a.alert(**class=‘alert-‘** + **msge.**type)

p= **message.msge**

Here is the final example:

while **msge** = **flash.**shift**() // consuming the messages as jade has read them**

a.alert(**class=‘alert-‘** + **msge.**type)

p= **message.msge**

An array of f

Lash messages can be as shown below:

{

“type”: “info”,

“message”: “message”

}

method-override

This is used for overriding HTTP verbs. With it, one can use HTTP verbs such as “DELETE” and “PUT” in places where they are not supported by the client. To install it in your system, execute the following command:

$ npm install method-override

For this module to be used, you have to know that a bit has to be used before any of the modules which are in need of the request methods.

Using the header to override

For the header to be used for overriding, the header name has to be specified as a string argument to the function "*methodOverride*," For the call to be made, the POST request has to be sent to the URL having the header as the overridden method. Consider the example given below showing, how this can be done:

```
var connect       = require('connect')
var mthdOverride = require('method-override')
// overriding with the X-HTTP-Method-Override header in our request

application.use(mthdOverride('X-HTTP-Method-Override'))
```

Consider the example given below, which shows how this can be used with "*XMLHttpRequest*":

```
var c = new XMLHttpRequest()
c.onload = onload
c.open('post', '/resource', true)
```

c.setRequestHeader**(‘X-HTTP-Method-Override’, ‘DELETE’)**

c.send**()**

function **onload() {**

alert**(‘The response was obtained: ‘** + **this.responseText)**

}

Chapter 13- serve-favicon

This is a Node.js middleware which is used for the purpose of serving favicons. A favicon is just a visual cue which browsers and other client software use for the purpose of identifying a site.

The module can be installed into the system by executing the following command:

npm install serve-favicon

Consider the example given below, showing how this can be used in express:

var **express** = require**(‘express’);**

var **favicon** = require**(‘serve-favicon’);**

var **application** = **express();**

application.use(favicon(__dirname + **‘/public/favicon.ico’));**

// Addition of your routes should be done here.

application.listen(3000**);**

In connect, this can be done as follows:

var **connect** = require**(‘connect’);**

var **favicon** = require**(‘serve-favicon’);**

var **application** = **connect();**

application.use(favicon(__dirname + **‘/public/favicon.ico’));**

// Addition of your middleware should be done here, etc.

application.listen(3000**);**

<u>vanilla http server</u>

This is a type of middleware which can be used everywhere, including outside Express and connect.

The example given below shows how this can be done:

var **http** = require**(‘http’);**

var **favicon** = require**(‘serve-favicon’);**

```
var fnhandler = require('finalhandler');

var _favicon = favicon(__dirname + '/public/favicon.ico');

var server = http.createServer(function onRequest(req, res) {

var done = fnhandler(req, res);

_favicon(req, res, function onNext(error) {

if (error) return done(error);

// just continue with processing of the request here, etc.

res.statusCode = 404;

res.end('hello');

});

});

server.listen(3000);
```

Chapter 14- response-time

This is the node.js response time header. It works by creating a middleware for recording the response time of its requests in the HTTP servers. The response time in this case will be the time when the request has entered the middleware to the time when the headers have been written to the client.

This module can be installed by executing the following command:

$ npm install response-time

The module can be included into the program by use of the "*require"* keyword as shown below:

var **respTime** = require**('response-time')**

Consider the example given below, which shows how this can be used in both the Express and the connect:

var **express** = require**(‘express’)**

var **respTime** = require**(‘response-time’)**

var **application** = **express()**

application.use(respTime())

application.get(‘/’, function **(req, res) {**

res.send**(‘hello, there!’)**

})

That is how it can be used.

vanilla http server

This can be used as follows:

var **finalhandler** = require**(‘finalhandler’)**

var **http** = require**(‘http’)**

var **respTime** = require**(‘response-time’)**

// creating the “middleware”

```
var _responseTime = respTime()

http.createServer(function (req, res) {

var done = fnhandler(req, res)

_responseTime(req, res, function (err) {

if (error) return done(error)

// responding to the request

res.setHeader('content-type', 'text/plain')

res.end('hello, there!')

})

})
```

That is how the module can be used.

Response time metris

```
var express = require('express')
var respTime = require('response-time')
var StatsD = require('node-statsd')
var application = express()
var statistics = new StatsD()
statistics.socket.on('error', function (err) {
console.error(err.stack)
})
application.use(respTime(function (req, res, time) {
var stat = (req.method + req.url).toLowerCase()
.replace(/[:.]/g, '')
.replace(/\//g, '_')
statistics.timing(stat, time)
}))
application.get('/', function (req, res) {
res.send('hello, there!')
})
```

Chapter 15- express-namespace

With this, namespaced routing capabilities are added to Express. To install it into the system, one can execute the following command:

$ npm install express-namespace

Consider the example given below, which can be used for responding to any of the requests. Here is the example:

GET /forum/12

GET /forum/12/view

GET /forum/12/edit

GET /forum/12/thread/5

DELETE /forum/12

For the module to be used, one must use the "*app.namespace()*" and "*require('express-*

namespace’).” The module will then be made available to you.

To use it, we have to pass a callback function, and then perform a routing to the method. After each of the callbacks, the invocation will be completed, and the namespace will be stored together with its state. This is shown below:

```
application.namespace(‘/forum/:id’, function(){
application.get(‘/(view)?’, function(req, res){
res.send(‘GET forum ‘ + req.params.id);
});
application.get(‘/edit’, function(req, res){
res.send(‘GET forum ‘ + req.params.id + ‘ edit the page’);
});
application.namespace(‘/thread’, function(){
application.get(‘/:tid’, function(req, res){
res.send(‘GET forum ‘ + req.params.id + ‘ thread ‘ + req.params.tid);

});
});
application.del(‘/’, function(req, res){
```

res.send(‘DELETE the forum ‘ + req.params.id);

});

});

The current namespace can also be accessed via the “*app.currentNamespace*.” You should also know how to run the tests. However, before doing this, make sure that you have all of the sub-modules installed into the system. These can be installed by executing the following command:

$ git submodule update –init

The tests can be run by executing the following command:

$ make test

That is how the module can be used.

Chapter 16- express-expose

This is used for exposing the objects, functions, and raw js to the client side. This feature is good for sharing of settings, utils, and the current data for the user. The helpers and the local variables are also exposed to the client side.

To install this module, execute the command given below:

npm install -S express-expose

It can be used as shown below:

var **express** = require**(‘express’);**

var **expose** = require**(‘express-expose’);**

application = **expose(application);**

application.expose(…);

The above is how the module can be used in express 4.x. In express 2.x and 3.x, it can be done as follows:

var **express** = require**(‘express’);**

var **expose** = require**(‘express-expose’);**

application.expose(…);

How to express the objects

One way that objects can be exposed to the client side is by exposing the properties, and most probably the Express configuration. When the "*app.expose(object)"* is exposed, the properties will be exposed to the "*application.**" The example given below shows how this can be done:

application.set('views', __dirname + **'/views');**

application.set(' the view engine ', 'jade');

application.set('title', ' Sample ');

application.set('default language', 'en');

application.expose(application.settings);

Helper methods can also be exposed by another use case. This can be in the same way that you are currently exposing. This is shown below:

application.expose({ en: **'English', fr**: **'French' }, 'application', 'languages');**

You then need to output your default buffer to your template. For jade users, it can be done as follows:

script!= **javascript**

For EJS users, this can be done as follows:

<script><%- **javascript** %>**</script>**

Raw JavaScript strings can also be exposed as shown below:

application.expose(‘var some = “variable”;’);

Exposing Functions

A named function can easily be exposed. You only have to pass it with a default name for the buffer to replace it with a template name. An example of this is shown below:

application.expose(function **someFunction(){**

return **‘yes’;**

}, ‘foot’);

That is how it can be done.

Self-Calling Functions

An anonymous function can also be passed, and this will execute itself so as to create a wrapper function. Consider the example given below, which shows how this can be done:

application.expose(function**(){**

function **notify() {**

alert**(‘this will be executed finely:D’);**

}

notify();

});

That is how it can be done.

Request-Level Exposure

All the above which has been discussed can be applied to the request-level. Consider the example given below showing how to do this. Here is the example:

application.get(‘/’, function**(req, res){**

var **user** = **{ name**: **‘john’ };**

res.expose(user, ‘application.current.user’);

res.render(‘index’, { layout: false **});**

});

Chapter 17- connect-render

This is a helper for the template render in connect. It can be installed by executing the following command:

$ npm install connect-render

It can be used as shown below:

var **connect** = require**(‘connect’);**

var **render** = require**(‘connect-render’);**

var **application** = **connect(**

render({

root: __dirname + **‘/views’,**

layout: **‘layout.html’,**

cache: true**, // `false` for the debug**

helpers: **{**

sitename: **‘connect-render sample site’,**

starttime: new **Date().**getTime**(),**

now: function **(req, res) {**

return new **Date();**

}

}

})

);

application.use(function **(req, res) {**

res.render('index.html', { url: **req.url });**

});

application.listen(8080**);**

The API can be used as shown below:

/**

*** connect-render: A Template Render helper for the connect**

*****

*** Use case:**

*****

```
* var render = require('connect-render');
* var connect = require('connect');
*
* connect(
*   render({
*     root: __dirname + '/views',
*     cache: true, // must be set to `true` in the production env
*     layout: 'layout.html', // or false if there is no layout
*     open: "<%",
*     close: "%>", // the default ejs close tag is '%>'
*     helpers: {
*       config: config,
*       sitename: 'NodeBlog Engine',
*       _csrf: function (req, res) {
*         return req.session ? req.session._csrf : "";
*       },
*     }
*   });
* );
*
* res.render('index.html', { title: 'Index Page', items: items });
```

*****

*** // no layout**

*** res.render('blue.html', { items: items, layout: false });**

*****

***** @param **{Object} [options={}] for the render options.**

*** - {String} layout, layout name, the default one is `'layout.html'`.**

***** **Set `layout=''` or `layout=false` meaning that no layout.**

*** - {String} root, the root dir for view files.**

*** - {Boolean} cache, the cache view content, the default is `true`.**

***** **Must be set `cache = true` on production.**

*** - {String} viewExt, the view file extname, the default is `''`.**

***** @return **{Function} rendering middleware for the `connect`**

***/**

function **middleware(options) {}**

Conclusion

It can be said that ExpressJS is a Node framework. It provides developers with a framework which they can use to create web and mobile applications. It has numerous features which developers can take advantage of so as to create their applications. It works by use of its middleware. The framework also provides developers with numerous modules which they can use for the development of their applications.

Each of these modules has its own command which can be used for its installation. To use of any of its modules, you have to begin by installing it. It supports the use of sessions which are very useful in the creation of web applications. It has a variety of options which you can use for the purpose of development. You should learn how to use the different options which are available for each ExpressJS module. The serve-index module is used for serving pages having listings of a directory to a particular path.

This is very interesting in development. With the “cookie-sessions” module, one can provide guest sessions so that each of the visitors will have a session, whether authenticated or not. This is also one of the best and interesting features in Express. This Node framework is very easy to learn even for beginners. This book helps you in learning how to use it.