**Writing middleware for use in Express apps**

**Overview**

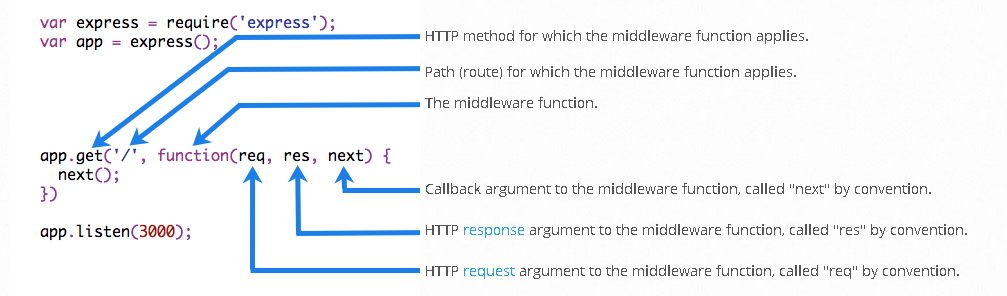
***Middleware* functions are functions that have access to the**[**request object**](https://expressjs.com/en/4x/api.html#req)**(req), the**[**response object**](https://expressjs.com/en/4x/api.html#res)**(res), and the nextfunction in the application’s request-response cycle. The next function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.**

**Middleware functions can perform the following tasks:**

* **Execute any code.**
* **Make changes to the request and the response objects.**
* **End the request-response cycle.**
* **Call the next middleware in the stack.**

**If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.**

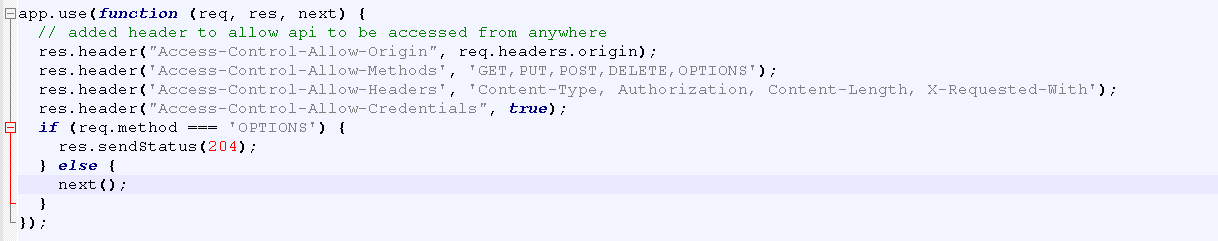
The following figure shows the elements of a middleware function call:



## Example

### Example of using a developer written middleware

This middleware function add different header to response



Here is an example of a simple “Hello World” Express application. The remainder of this article will define and add two middleware functions to the application: one called myLogger that prints a simple log message and another called requestTime that displays the timestamp of the HTTP request.

var express = require('express')

var app = express()

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

res.send('Hello World!')

})

app.listen(3000)

**Middleware function myLogger**

Here is a simple example of a middleware function called “myLogger”. This function just prints “LOGGED” when a request to the app passes through it. The middleware function is assigned to a variable named myLogger.

var myLogger = function (req, res, next) {

console.log('LOGGED')

next()

}

**Notice the call above to next(). Calling this function invokes the next middleware function in the app. The next() function is not a part of the Node.js or Express API, but is the third argument that is passed to the middleware function. The next() function could be named anything, but by convention it is always named “next”. To avoid confusion, always use this convention.**

To load the middleware function, call app.use(), specifying the middleware function. For example, the following code loads the myLogger middleware function before the route to the root path (/).

var express = require('express')

var app = express()

var myLogger = function (req, res, next) {

console.log('LOGGED')

next()

}

app.use(myLogger)

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

res.send('Hello World!')

})

app.listen(3000)

**Every time the app receives a request, it prints the message “LOGGED” to the terminal.**

**The order of middleware loading is important: middleware functions that are loaded first are also executed first.**

**If myLogger is loaded after the route to the root path, the request never reaches it and the app doesn’t print “LOGGED”, because the route handler of the root path terminates the request-response cycle.**

**The middleware function myLogger simply prints a message, then passes on the request to the next middleware function in the stack by calling the next() function.**

**Middleware function requestTime**

Next, we’ll create a middleware function called “requestTime” and add a property called requestTime to the request object.

var requestTime = function (req, res, next) {

req.requestTime = Date.now()

next()

}

The app now uses the requestTime middleware function. Also, the callback function of the root path route uses the property that the middleware function adds to req (the request object).

var express = require('express')

var app = express()

var requestTime = function (req, res, next) {

req.requestTime = Date.now()

next()

}

app.use(requestTime)

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

var responseText = 'Hello World!<br>'

responseText += '<small>Requested at: ' + req.requestTime + '</small>'

res.send(responseText)

})

app.listen(3000)

When you make a request to the root of the app, the app now displays the timestamp of your request in the browser.

Because you have access to the request object, the response object, the next middleware function in the stack, and the whole Node.js API, the possibilities with middleware functions are endless.

For more information about Express middleware, see: [Using Express middleware](https://expressjs.com/en/guide/using-middleware.html).

**Configurable middleware**

If you need your middleware to be configurable, export a function which accepts an options object or other parameters, which, then returns the middleware implementation based on the input parameters.

File: my-middleware.js

module.exports = function(options) {

return function(req, res, next) {

// Implement the middleware function based on the options object

next()

}

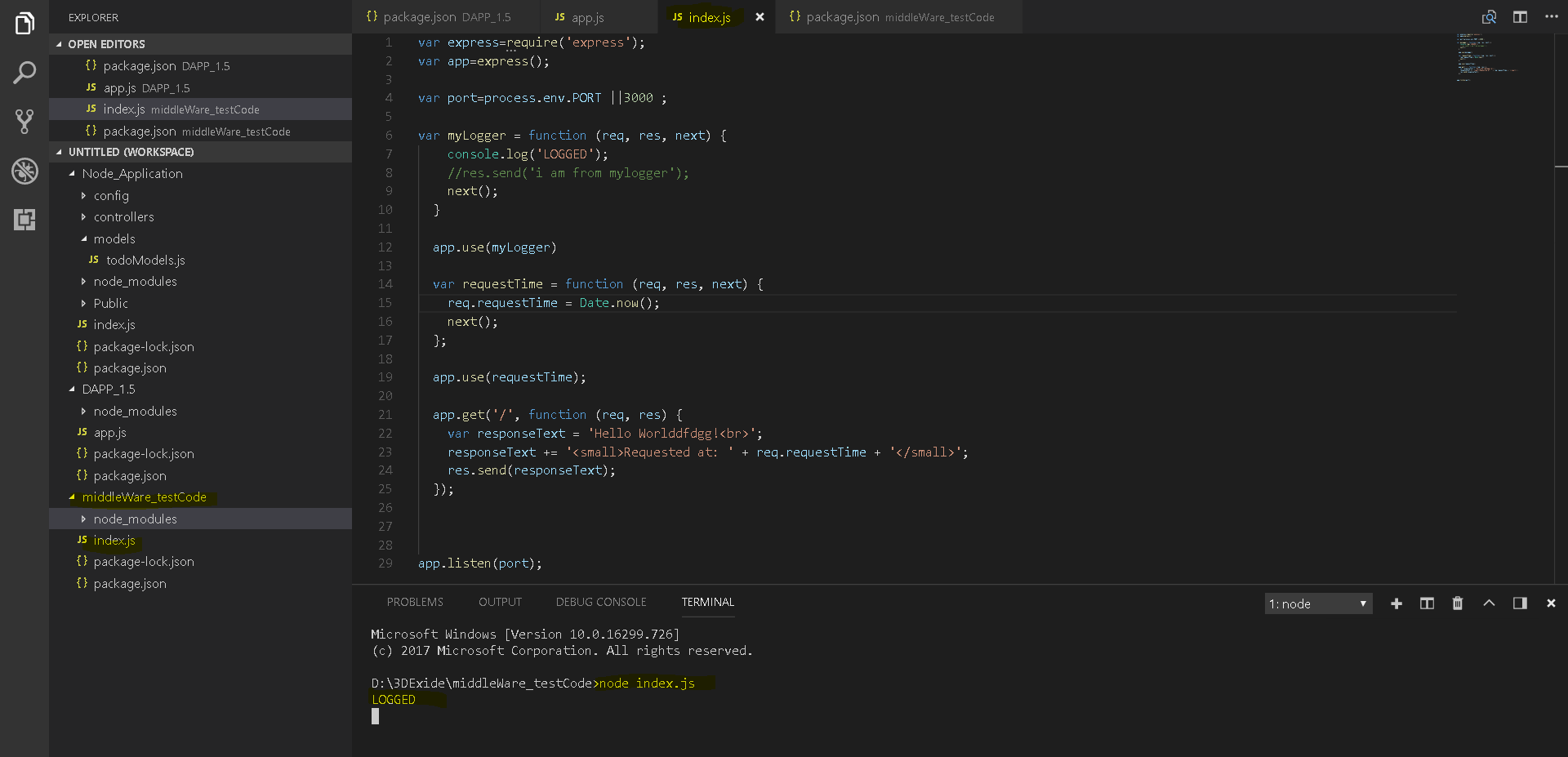
}

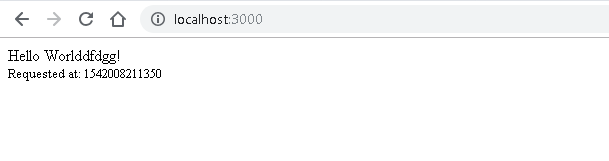
The middleware can now be used as shown below.

var mw = require('./my-middleware.js')

app.use(mw({ option1: '1', option2: '2' }))

**Example of showing the sequence of execution of middlewares by using next()**





# Using Middleware

<https://expressjs.com/en/guide/using-middleware.html>

Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.

You can load application-level and router-level middleware with an optional mount path. You can also load a series of middleware functions together, which creates a sub-stack of the middleware system at a mount point.

## Application-level middleware

Bind application-level middleware to an instance of the [app object](https://expressjs.com/en/4x/api.html#app) by using the app.use() and app.METHOD() functions, where METHOD is the HTTP method of the request that the middleware function handles (such as GET, PUT, or POST) in lowercase.

This example shows a middleware function with no mount path. The function is executed every time the app receives a request.

var app = express()

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

console.log('Time:', Date.now())

next()

})

This example shows a middleware function mounted on the /user/:id path. The function is executed for any type of HTTP request on the /user/:id path.

app.use('/user/:id', function (req, res, next) {

console.log('Request Type:', req.method)

next()

})

This example shows a route and its handler function (middleware system). The function handles GET requests to the /user/:idpath.

app.get('/user/:id', function (req, res, next) {

res.send('USER')

})

Here is an example of loading a series of middleware functions at a mount point, with a mount path. It illustrates a middleware sub-stack that prints request info for any type of HTTP request to the /user/:id path.

app.use('/user/:id', function (req, res, next) {

console.log('Request URL:', req.originalUrl)

next()

}, function (req, res, next) {

console.log('Request Type:', req.method)

next()

})

Route handlers enable you to define multiple routes for a path. The example below defines two routes for GET requests to the /user/:id path. The second route will not cause any problems, but it will never get called because the first route ends the request-response cycle.

This example shows a middleware sub-stack that handles GET requests to the /user/:id path.

app.get('/user/:id', function (req, res, next) {

console.log('ID:', req.params.id)

next()

}, function (req, res, next) {

res.send('User Info')

})

// handler for the /user/:id path, which prints the user ID

app.get('/user/:id', function (req, res, next) {

res.end(req.params.id)

})

To skip the rest of the middleware functions from a router middleware stack, call next('route') to pass control to the next route. **NOTE**: next('route') will work only in middleware functions that were loaded by using the app.METHOD() or router.METHOD() functions.

This example shows a middleware sub-stack that handles GET requests to the /user/:id path.

app.get('/user/:id', function (req, res, next) {

// if the user ID is 0, skip to the next route

if (req.params.id === '0') next('route')

// otherwise pass the control to the next middleware function in this stack

else next()

}, function (req, res, next) {

// send a regular response

res.send('regular')

})

// handler for the /user/:id path, which sends a special response

app.get('/user/:id', function (req, res, next) {

res.send('special')

})

## Router-level middleware

Router-level middleware works in the same way as application-level middleware, except it is bound to an instance of express.Router().

var router = express.Router()

Load router-level middleware by using the router.use() and router.METHOD() functions.

The following example code replicates the middleware system that is shown above for application-level middleware, by using router-level middleware:

var app = express()

var router = express.Router()

// a middleware function with no mount path. This code is executed for every request to the router

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

console.log('Time:', Date.now())

next()

})

// a middleware sub-stack shows request info for any type of HTTP request to the /user/:id path

router.use('/user/:id', function (req, res, next) {

console.log('Request URL:', req.originalUrl)

next()

}, function (req, res, next) {

console.log('Request Type:', req.method)

next()

})

// a middleware sub-stack that handles GET requests to the /user/:id path

router.get('/user/:id', function (req, res, next) {

// if the user ID is 0, skip to the next router

if (req.params.id === '0') next('route')

// otherwise pass control to the next middleware function in this stack

else next()

}, function (req, res, next) {

// render a regular page

res.render('regular')

})

// handler for the /user/:id path, which renders a special page

router.get('/user/:id', function (req, res, next) {

console.log(req.params.id)

res.render('special')

})

// mount the router on the app

app.use('/', router)

To skip the rest of the router’s middleware functions, call next('router') to pass control back out of the router instance.

This example shows a middleware sub-stack that handles GET requests to the /user/:id path.

var app = express()

var router = express.Router()

// predicate the router with a check and bail out when needed

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

if (!req.headers['x-auth']) return next('router')

next()

})

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

res.send('hello, user!')

})

// use the router and 401 anything falling through

app.use('/admin', router, function (req, res) {

res.sendStatus(401)

})

## Error-handling middleware

**Error-handling middleware always takes four arguments. You must provide four arguments to identify it as an error-handling middleware function. Even if you don’t need to use the next object, you must specify it to maintain the signature. Otherwise, the next object will be interpreted as regular middleware and will fail to handle errors.**

Define error-handling middleware functions in the same way as other middleware functions, except with four arguments instead of three, specifically with the signature (err, req, res, next)):

app.use(function (err, req, res, next) {

console.error(err.stack)

res.status(500).send('Something broke!')

})

For details about error-handling middleware, see: [Error handling](https://expressjs.com/en/guide/error-handling.html).

## Built-in middleware

Starting with version 4.x, Express no longer depends on [Connect](https://github.com/senchalabs/connect). The middleware functions that were previously included with Express are now in separate modules; see [the list of middleware functions](https://github.com/senchalabs/connect#middleware).

Express has the following built-in middleware functions:

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

## Third-party middleware

Use third-party middleware to add functionality to Express apps.

Install the Node.js module for the required functionality, then load it in your app at the application level or at the router level.

The following example illustrates installing and loading the cookie-parsing middleware function cookie-parser.

$ npm install cookie-parser

var express = require('express')

var app = express()

var cookieParser = require('cookie-parser')

// load the cookie-parsing middleware

app.use(cookieParser())

For a partial list of third-party middleware functions that are commonly used with Express, see: [Third-party middleware](https://expressjs.com/en/resources/middleware.html).

## Note –

**The idea is simply that via npm you instal some JavaScript code that deals with a standard Express request and an Express response. That's Express middleware.**

It's in the middle between those two, and that doesn't mean that you can't then have your own middleware or do something after the fact, because you can just keep going. The middleware can pass the request and responses to each other down a chain and just keep on doing things until we're actually ready to send the response.