**Middlewares**

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.

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 next middleware function in the application’s request-response cycle. The next middleware function is commonly denoted by a variable named next.

Middleware functions can perform the following tasks:

* Middleware simplifies software development
* It controls connections between application components
* No doubt, it is a cost-effective tool for managing multi-cloud resources.
* It plays a crucial role in load balancing, concurrent processing, and transaction management.
* With middleware, you can make secured access to back-end resources such as storage, databases, etc.
* It supports authentication functionalities on a greater scale
* You can enhance rendering performance for client-side significantly
* By using middleware, you can easily set HTTP headers

There are many types of node.JS middleware, such as application-level, router-level, built-in, error-handling, and third-party middleware. Middleware is nothing but a function that has access to response objects, request objects, and the next middleware function. It exists in between the request and response cycles of Node.JS execution. Generally, the request object is referred to by the variable ‘req', whereas the response object is referred to by the variable ‘res’. Note that middleware can process ‘req’ objects before the server sends a response.



With middleware Node.JS, we can do a multitude of things. To begin with, we can run any codes with middleware functions. Also, we can make changes in response and request objects. We can end the request and response cycle in Node.JS execution. Middleware eases connectivity, supports secured connections and data transfer, and manages traffic across distributed systems. With middleware, you don’t need to make custom integrations every time.

Know that there are many types of middleware, such as message-oriented middleware, RPC middleware, database middleware, ORB middleware, and so on.

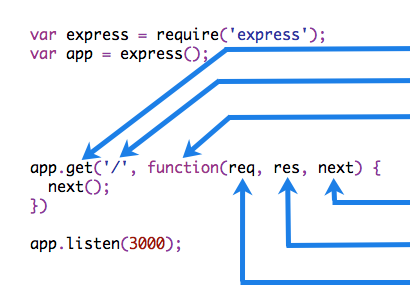
What is the Next ( ) Function?

The next ( ) function plays a vital role in applications' request and response cycle. It is a middleware function that runs the next middleware function once it is invoked. In other words, the Next function is invoked if the current middleware function doesn’t end the request and response cycle. It is essential to note that no middleware function should be hanging in the queue. Starting with Express 5, middleware functions that return a Promise will call next(value) when they reject or throw an error. next will be called with either the rejected value or the thrown Error.



The image below is the middleware function where you can find its different components.

|  |
| --- |
|  |





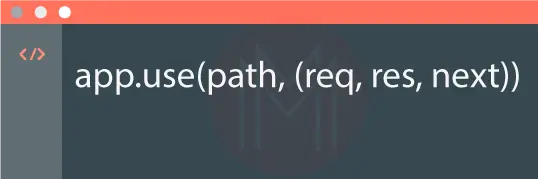
### What is Express Middleware?

It is nothing but a routing as well as a middleware framework with which you can handle different routing of web pages. It works in-between request and response cycles. It can handle any error since it comes with default error-handling middleware functions. They are the functions compiled while making requests to the express server.

Every HTTPS request, as well as the response, can be easily accessed by the middleware. Even the middleware can terminate HTTPS requests independently. Or, it can transfer the request to another middleware function using the next function.

With the help of express middleware, you can build mobile as well as web applications. So, you can build single-page, hybrid, and multipage web applications seamlessly.

The syntax for the express.js middleware function is given as follows:



### Types of Node.JS Middleware

There are five types of middleware as follows



|  |
| --- |
| const express = require('express')  const app = express()  const myLogger = function (req, res, next) {  console.log('LOGGED')  next()  }  app.use(myLogger)  app.get('/', (req, res) => {  res.send('Hello World!')  })  app.listen(3000)  **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.  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 |

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

const express = require('express')

const app = express()

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

  req.requestTime = Date.now()

  next()

}

app.use(requestTime)

app.get('/', (req, res) => {

  let responseText = 'Hello World!<br>'

  responseText += `<small>Requested at: ${req.requestTime}</small>`

  res.send(responseText)

})

app.listen(3000)

Finally, we’ll create a middleware function that validates incoming cookies and sends a **400 response if cookies are invalid**. Here, we use the [**cookie-parser**](https://expressjs.com/resources/middleware/cookie-parser.html) middleware to parse incoming cookies off the req object and pass them to our **cookieValidator** function. **The validateCookies** middleware returns a Promise that upon rejection will automatically trigger our error handler.

**cookieValidator.js**

async function cookieValidator (cookies) {

    try {

      await externallyValidateCookie(cookies.testCookie)

    } catch {

      throw new Error('Invalid cookies')

    }

  }

**App.js**

const express = require('express')

const cookieParser = require('cookie-parser')

const cookieValidator = require('./cookieValidator')

const app = express()

async function validateCookies (req, res, next) {

  await cookieValidator(req.cookies)

  next()

}

app.use(cookieParser())

app.use(validateCookies)

// error handler

app.use((err, req, res, next) => {

  res.status(400).send(err.message)

})

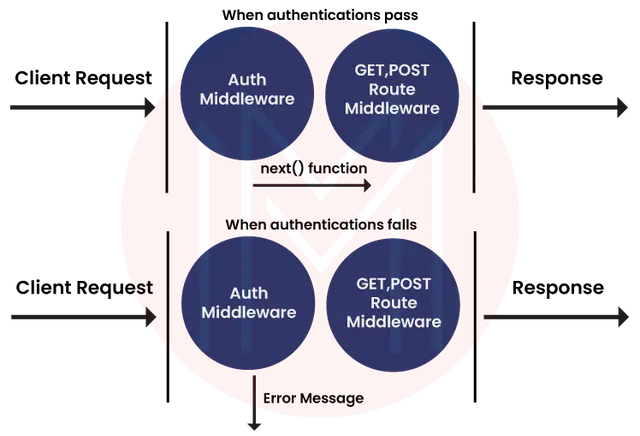
app.listen(3000)

#### **Application-level Middleware**

Know that every GET and POST call needs authentication. So, if you need to authenticate GET and POST calls, you can develop authentication middleware. This is because application-level middleware focuses on authenticating users. Once the authentication request is received, the authentication middleware invokes authentication code logic.

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.

If the authentication is successful, the rest of the route is progressed with the help of the next function. On the other hand, if the authentication is unsuccessful, there is no progress in the route, and the middleware releases error messages.



EXAMPLE 1:- const express = require('express')

const app = express()

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

    console.log("new req rec"+Date());

});

app.get('/users', (req, res, next) => {

    res.send('USER')

  })

app.listen(3000)

Now run the file & in console u will see the current date but in browser the window is still loading so we need to include next() so that it can execute the next step:

**EXAMPLE 2:-const express = require('express')**

**const app = express()**

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

**console.log("new req rec"+Date());**

**next();**

**});**

**app.get('/users', (req, res, next) => {**

**res.send('USER')**

**})**

**app.listen(3000)**

**EXAMPLE 3:-** const express = require('express')

const app = express()

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

    console.log("new req rec"+Date());

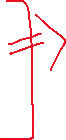
    next();

});

app.get('/users', (req, res, next) => {

    res.send('USER');

    next();



  })

app.get('/users', (req, res, next) => {

    res.send('end user')

  })

app.listen(3000)

#### **Router-level Middleware**

This middleware works in the same way application-level middleware works. By using the express. router ( ) function, this middleware supports creating and managing instances. Moreover, you can load router-level middleware with the help of the router.use ( ) function as well as router.METHOD ( ) function.

#### **Built-in Middleware**

There are three built-in middleware functions. Know what they are from below:

* **Static:** They are functions that act as static assets to applications. HTML files and images are a few examples of static assets.
* **JSON:** This function processes incoming requests along with the JSON payloads.
* **Express.URL-encoded:** This function processes incoming requests along with URL-encoded payloads.

**EXAMPLES:**-

const express = require('express')

const app = express()

const router = express.Router()

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

router.use((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', (req, res, next) => {

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

  next()

}, (req, res, next) => {

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

  next()

})

// mount the router on the app

app.use('/', router)

app.listen(3000)

**RUN the file:-** node app.js

<URL:-> localhost:3000/users/5

**OUTPUT:-** Time: 1721640495910

Request URL: /user/5

Request Type: GET

**EXAMPLE 2:-**

const express = require('express')

const app = express()

const router = express.Router()

router.use((req, res, next) => {

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

    next()

  })

  router.get('/user/:id', (req, res) => {

    res.send('hello, user!')

  })

  // use the router and 401 anything falling through

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

    res.sendStatus(401)

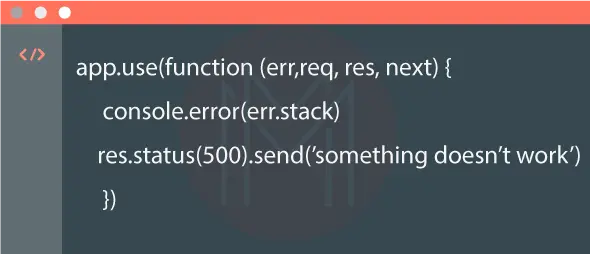
  })

app.listen(3000)

#### **Error-Handling Middleware**

Express.JS has default error-handling middleware functions. That's why it can handle any error. 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.

The functions have four arguments instead of three.



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+**