Using middleware

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 (`req`), the response object (`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:

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

An Express application can use the following types of middleware:

- Application-level middleware
- Router-level middleware
- Error-handling middleware
- Built-in middleware
- Third-party middleware

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 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 express = require('express')
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/:id` path.

```
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.send(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')
})
```

Middleware can also be declared in an array for reusability.

This example shows an array with a middleware sub-stack that handles GET requests to the `/user/:id` path

```
function logOriginalUrl (req, res, next) {
  console.log('Request URL:', req.originalUrl)
  next()
}

function logMethod (req, res, next) {
  console.log('Request Type:', req.method)
  next()
}

var logStuff = [logOriginalUrl, logMethod]
app.get('/user/:id', logStuff, function (req, res, next) {
  res.send('User Info')
})
```

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 express = require('express')
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 express = require('express')
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('/user/:id', 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.

Built-in middleware

Starting with version 4.x, Express no longer depends on Connect. The middleware functions that were previously included with Express are now in separate modules; see the list of middleware functions.

Express has the following built-in middleware functions:

- express.static serves static assets such as HTML files, images, and so on.
- express.json parses incoming requests with JSON payloads. **NOTE: Available with Express 4.16.0+**
- 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.