MIDDLEWARE

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.

The different types of middleware:

1. Application-level middleware
2. Router-level middleware
3. Error-handling middleware
4. Built-in middleware
5. Third-party middleware

1) 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.

```
const express = require('express')
const app = express()

app.use((req, res, next) => {
  console.log('Time:', Date.now())
  next()
})
```

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', (req, res, next) => {
  console.log('Request URL:', req.originalUrl)
  next()
}, (req, res, next) => {
  console.log('Request Type:', req.method)
  next()
})
```

2) 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().

```
const router = express.Router()
```

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

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

```
const express = require('express')
const app = express()
const router = express.Router()

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

3) Error-handling middleware

Error Handling refers to how Express catches and processes errors that occur both synchronously and asynchronously. We 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)):

Note:

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.

```
app.use((err, req, res, next) => {
  console.error(err.stack)
  res.status(500).send('Something broke!')
})
```

4) Built-in middleware

In the context of Express.js, built-in middleware refers to middleware functions that come bundled with the Express framework and can be easily integrated into your application using the app.use() method. These middleware functions provide common functionalities that are often required in web applications

i. `express.static` Middleware

This middleware is used for serving static files, such as HTML, CSS, images, and JavaScript files. It simplifies the process of serving static content by specifying a directory that contains the static files. When a request is made for a static file, Express will look for the file in the specified directory and serve it if found.

In this example, any files in the "public" directory can be accessed directly by the client by visiting the corresponding URL. For instance, if you have a file named "styles.css" in the "public" directory, it can be accessed at http://yourdomain.com/styles.css

```
// Serve static files from the 'public' directory
app.use(express.static('public'));
```

ii. `body-parser` Middleware:

While body-parser used to be a separate module, it is now part of the Express.js library. It parses incoming request bodies in a middleware and makes the parsed data available in req.body. This is particularly useful when dealing with form submissions or HTTP requests that contain data in the request body.

Here, bodyParser.json() parses incoming JSON payloads, and bodyParser.urlencoded({ extended: true }) parses incoming URL-encoded form data.

```
const bodyParser = require('body-parser');

// Parse incoming request bodies in a middleware before your handlers
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

These built-in middleware functions enhance the capabilities of your Express application by providing convenient and standardized ways to handle common tasks. They contribute to the framework's flexibility and ease of use, allowing developers to focus on building application logic rather than dealing with low-level details of HTTP request handling.

5) 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
```

```
const express = require('express')
const app = express()
const cookieParser = require('cookie-parser')

// load the cookie-parsing middleware
app.use(cookieParser())
```