**Express JS**

Express is Node.js web application framework to develop web and mobile applications. It facilitates the rapid development of Node based Web applications. Following are some of the core features of Express framework −

       Allows to set up middlewares to respond to HTTP Requests.

       Defines a routing table which is used to perform different actions based on HTTP Method and URL.

       Allows to dynamically render HTML Pages based on passing arguments to templates.

* **body-parser** − This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.
* **cookie-parser** − Parse Cookie header and populate req.cookies with an object keyed by the cookie names.
* **multer** − This is a node.js middleware for handling multipart/form-data.

Cookies

An HTTP cookie is a small piece of data sent from a website and stored on the user's computer by the user's web browser while the user is browsing.

Cookies are small pieces of data sent from a website and are stored in user's web browser while user is browsing that website. Every time the user loads that website back, the browser sends that stored data back to website or server, to distinguish user's previous activit

Cookies Management

You can send cookies to a Node.js server which can handle the same using the following middleware option. Following is a simple example to print all the cookies sent by the client.

var express      = require('express')

var cookieParser = require('cookie-parser')

var app = express();

**app.use(cookieParser());**

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

   console.log("Cookies: ", req.cookies)

})

app.listen(8081)

 Cookie-parser parses Cookie header and populate **req.cookies** with an object keyed by the cookie names. To set a new cookie lets define a new route in your express app like :

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

res.cookie(cookie\_name , 'cookie\_value').send('Cookie is set');

});

To check whether cookie has been set or not, goto to browser's console and write **document.cookie**.

Browser sends back that cookie to the server, every time when it requests that website. And to get a cookie which a browser might be sending to server by attaching it to request header, we can write following code :

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

console.log("Cookies : ", req.cookies);

});

**How to Set Cookie Expiration Time?**

Cookie expire time can be set easily by :

res.cookie(name , 'value', {expire : **new** Date() + 9999});

## How to Delete Existing Cookie?

Existing cookies can be deleted very easily using **clearCookie** method, which accepts the name of the cookie which you want to delete.

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

clearCookie('cookie\_name');

res.send('Cookie deleted');

});

# Serving static files in Express

To serve static files such as images, CSS files, and JavaScript files, use the express.static built-in middleware function in Express.

The function signature is:

express.static(root, [options])

The root argument specifies the root directory from which to serve static assets. For more information on the options argument, see [express.static](https://expressjs.com/en/4x/api.html#express.static).

For example, use the following code to serve images, CSS files, and JavaScript files in a directory named public:

app.use(express.static('public'))

Now, you can load the files that are in the public directory:

http://localhost:3000/images/kitten.jpg

http://localhost:3000/css/style.css

http://localhost:3000/js/app.js

http://localhost:3000/images/bg.png

http://localhost:3000/hello.html

Express looks up the files relative to the static directory, so the name of the static directory is not part of the URL.

To use multiple static assets directories, call the express.static middleware function multiple times:

app.use(express.static('public'))

app.use(express.static('files'))

Express looks up the files in the order in which you set the static directories with the express.static middleware function.

NOTE: For best results, [use a reverse proxy](https://expressjs.com/en/advanced/best-practice-performance.html#use-a-reverse-proxy) cache to improve performance of serving static assets.

To create a virtual path prefix (where the path does not actually exist in the file system) for files that are served by the express.static function, [specify a mount path](https://expressjs.com/en/4x/api.html#app.use) for the static directory, as shown below:

app.use('/static', express.static('public'))

Now, you can load the files that are in the public directory from the /static path prefix.

http://localhost:3000/static/images/kitten.jpg

http://localhost:3000/static/css/style.css

http://localhost:3000/static/js/app.js

http://localhost:3000/static/images/bg.png

http://localhost:3000/static/hello.html

However, the path that you provide to the express.static function is relative to the directory from where you launch your node process. If you run the express app from another directory, it’s safer to use the absolute path of the directory that you want to serve:

app.use('/static', express.static(path.join(\_\_dirname, 'public')))

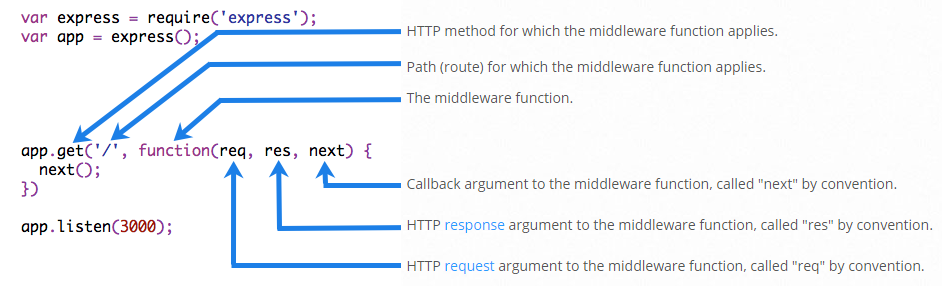
# Custom middleware

***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 function 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.



To load the middleware function, call app.use(), specifying the middleware function. For example, the following code loads the myLoggermiddleware 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.

## 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' }))

### Middleware function requestTime

Next, we’ll create a middleware function called “requestTime” and add it as 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.

# 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](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:

* 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](https://expressjs.com/en/guide/using-middleware.html#middleware.application)
* [Router-level middleware](https://expressjs.com/en/guide/using-middleware.html#middleware.router)
* [Error-handling middleware](https://expressjs.com/en/guide/using-middleware.html#middleware.error-handling)
* [Built-in middleware](https://expressjs.com/en/guide/using-middleware.html#middleware.built-in)
* [Third-party middleware](https://expressjs.com/en/guide/using-middleware.html#middleware.third-party)

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/: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.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) {

// render a regular page

res.render('regular')

})

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

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

res.render('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())

# Using template engines with Express

A **template engine** enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page.

Some popular template engines that work with Express are [Pug](https://pugjs.org/api/getting-started.html), [Mustache](https://www.npmjs.com/package/mustache), and [EJS](https://www.npmjs.com/package/ejs). The [Express application generator](https://expressjs.com/en/starter/generator.html) uses [Jade](https://www.npmjs.com/package/jade) as its default, but it also supports several others.

See [Template Engines (Express wiki)](https://github.com/strongloop/express/wiki#template-engines) for a list of template engines you can use with Express. See also [Comparing JavaScript Templating Engines: Jade, Mustache, Dust and More](https://strongloop.com/strongblog/compare-javascript-templates-jade-mustache-dust/).

**Note**: Jade has been renamed to [Pug](https://www.npmjs.com/package/pug). You can continue to use Jade in your app, and it will work just fine. However if you want the latest updates to the template engine, you must replace Jade with Pug in your app.

To render template files, set the following [application setting properties](https://expressjs.com/en/4x/api.html#app.set), set in app.js in the default app created by the generator:

* views, the directory where the template files are located. Eg: app.set('views', './views'). This defaults to the views directory in the application root directory.
* view engine, the template engine to use. For example, to use the Pug template engine: app.set('view engine', 'pug').

Then install the corresponding template engine npm package; for example to install Pug:

$ npm install pug --save

Express-compliant template engines such as Jade and Pug export a function named \_\_express(filePath, options, callback), which is called by the res.render() function to render the template code.

Some template engines do not follow this convention. The [Consolidate.js](https://www.npmjs.org/package/consolidate) library follows this convention by mapping all of the popular Node.js template engines, and therefore works seamlessly within Express.

After the view engine is set, you don’t have to specify the engine or load the template engine module in your app; Express loads the module internally, as shown below (for the above example).

app.set('view engine', 'pug')

Create a Pug template file named index.pug in the views directory, with the following content:

html

head

title= title

body

h1= message

Then create a route to render the index.pug file. If the view engine property is not set, you must specify the extension of the view file. Otherwise, you can omit it.

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

res.render('index', { title: 'Hey', message: 'Hello there!' })

})

When you make a request to the home page, the index.pug file will be rendered as HTML.

Note: The view engine cache does not cache the contents of the template’s output, only the underlying template itself. The view is still re-rendered with every request even when the cache is on.

# Error handling

Define error-handling middleware functions in the same way as other middleware functions, except error-handling functions have four arguments instead of three: (err, req, res, next). For example:

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

console.error(err.stack)

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

})

You define error-handling middleware last, after other app.use() and routes calls; for example:

var bodyParser = require('body-parser')

var methodOverride = require('method-override')

app.use(bodyParser.urlencoded({

extended: true

}))

app.use(bodyParser.json())

app.use(methodOverride())

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

// logic

})

Responses from within a middleware function can be in any format that you prefer, such as an HTML error page, a simple message, or a JSON string.

For organizational (and higher-level framework) purposes, you can define several error-handling middleware functions, much like you would with regular middleware functions. For example, if you wanted to define an error-handler for requests made by using XHR, and those without, you might use the following commands:

var bodyParser = require('body-parser')

var methodOverride = require('method-override')

app.use(bodyParser.urlencoded({

extended: true

}))

app.use(bodyParser.json())

app.use(methodOverride())

app.use(logErrors)

app.use(clientErrorHandler)

app.use(errorHandler)

In this example, the generic logErrors might write request and error information to stderr, for example:

function logErrors (err, req, res, next) {

console.error(err.stack)

next(err)

}

Also in this example, clientErrorHandler is defined as follows; in this case, the error is explicitly passed along to the next one.

Notice that when **not** calling “next” in an error-handling function, you are responsible for writing (and ending) the response. Otherwise those requests will “hang” and will not be eligible for garbage collection.

function clientErrorHandler (err, req, res, next) {

if (req.xhr) {

res.status(500).send({ error: 'Something failed!' })

} else {

next(err)

}

}

The “catch-all” errorHandler function might be implemented as follows:

function errorHandler (err, req, res, next) {

res.status(500)

res.render('error', { error: err })

}

If you pass anything to the next() function (except the string 'route'), Express regards the current request as being in error and will skip any remaining non-error handling routing and middleware functions. If you want to handle that error in some way, you’ll have to create an error-handling route as described in the next section.

If you have a route handler with multiple callback functions you can use the route parameter to skip to the next route handler. For example:

app.get('/a\_route\_behind\_paywall',

function checkIfPaidSubscriber (req, res, next) {

if (!req.user.hasPaid) {

// continue handling this request

next('route')

}

else{

next();

}

}, function getPaidContent (req, res, next) {

PaidContent.find(function (err, doc) {

if (err) return next(err)

res.json(doc)

})

})

In this example, the getPaidContent handler will be skipped but any remaining handlers in app for /a\_route\_behind\_paywall would continue to be executed.

Calls to next() and next(err) indicate that the current handler is complete and in what state. next(err) will skip all remaining handlers in the chain except for those that are set up to handle errors as described above.

## The Default Error Handler

Express comes with a built-in error handler, which takes care of any errors that might be encountered in the app. This default error-handling middleware function is added at the end of the middleware function stack.

If you pass an error to next() and you do not handle it in an error handler, it will be handled by the built-in error handler; the error will be written to the client with the stack trace. The stack trace is not included in the production environment.

Set the environment variable NODE\_ENV to production, to run the app in production mode.

If you call next() with an error after you have started writing the response (for example, if you encounter an error while streaming the response to the client) the Express default error handler closes the connection and fails the request.

So when you add a custom error handler, you will want to delegate to the default error handling mechanisms in Express, when the headers have already been sent to the client:

function errorHandler (err, req, res, next) {

if (res.headersSent) {

return next(err)

}

res.status(500)

res.render('error', { error: err })

}

Note that the default error handler can get triggered if you call next() with an error in your code more than once, even if custom error handling middleware is in place.

# Debugging Express

Express uses the [debug](https://www.npmjs.com/package/debug) module internally to log information about route matches, middleware functions that are in use, application mode, and the flow of the request-response cycle.

debug is like an augmented version of console.log, but unlike console.log, you don’t have to comment out debug logs in production code. Logging is turned off by default and can be conditionally turned on by using the DEBUG environment variable.

To see all the internal logs used in Express, set the DEBUG environment variable to express:\* when launching your app.

$ DEBUG=express:\* node index.js

On Windows, use the corresponding command.

> set DEBUG=express:\* & node index.js

Running this command on the default app generated by the [express generator](https://expressjs.com/en/starter/generator.html) prints the following output:

$ DEBUG=express:\* node ./bin/www

express:router:route new / +0ms

express:router:layer new / +1ms

express:router:route get / +1ms

express:router:layer new / +0ms

express:router:route new / +1ms

express:router:layer new / +0ms

express:router:route get / +0ms

express:router:layer new / +0ms

express:application compile etag weak +1ms

express:application compile query parser extended +0ms

express:application compile trust proxy false +0ms

express:application booting in development mode +1ms

express:router use / query +0ms

express:router:layer new / +0ms

express:router use / expressInit +0ms

express:router:layer new / +0ms

express:router use / favicon +1ms

express:router:layer new / +0ms

express:router use / logger +0ms

express:router:layer new / +0ms

express:router use / jsonParser +0ms

express:router:layer new / +1ms

express:router use / urlencodedParser +0ms

express:router:layer new / +0ms

express:router use / cookieParser +0ms

express:router:layer new / +0ms

express:router use / stylus +90ms

express:router:layer new / +0ms

express:router use / serveStatic +0ms

express:router:layer new / +0ms

express:router use / router +0ms

express:router:layer new / +1ms

express:router use /users router +0ms

express:router:layer new /users +0ms

express:router use / &lt;anonymous&gt; +0ms

express:router:layer new / +0ms

express:router use / &lt;anonymous&gt; +0ms

express:router:layer new / +0ms

express:router use / &lt;anonymous&gt; +0ms

express:router:layer new / +0ms

When a request is then made to the app, you will see the logs specified in the Express code:

express:router dispatching GET / +4h

express:router query : / +2ms

express:router expressInit : / +0ms

express:router favicon : / +0ms

express:router logger : / +1ms

express:router jsonParser : / +0ms

express:router urlencodedParser : / +1ms

express:router cookieParser : / +0ms

express:router stylus : / +0ms

express:router serveStatic : / +2ms

express:router router : / +2ms

express:router dispatching GET / +1ms

express:view lookup "index.pug" +338ms

express:view stat "/projects/example/views/index.pug" +0ms

express:view render "/projects/example/views/index.pug" +1ms

To see the logs only from the router implementation set the value of DEBUG to express:router. Likewise, to see logs only from the application implementation set the value of DEBUG to express:application, and so on.

## Applications generated by express

An application generated by the express command also uses the debug module and its debug namespace is scoped to the name of the application.

For example, if you generated the app with $ express sample-app, you can enable the debug statements with the following command:

$ DEBUG=sample-app node ./bin/www

You can specify more than one debug namespace by assigning a comma-separated list of names:

$ DEBUG=http,mail,express:\* node index.js

For more information about debug, see the [debug](https://www.npmjs.com/package/debug).