Node.js

**What is Node?**

It’s a javaScript on your backend servers, before node javaScript was mainly a frontend thing, this is not complete definition, because node offers a lot more than execution of javaScript on the server. In fact the execution is not done by javaScript. It’s done with VM like V8 or chakra node. So node is better defined as wrapper around the VM like V8.

Node JS is an open-source server side runtime environment built on Chrome's V8 JavaScript engine. It provides an event driven, non-blocking (asynchronous) I/O and cross-platform runtime environment for building highly scalable server-side application using JavaScript.

**Why Node?**

So besides being easy to write and execute with JS on the server node comes with some feature rich, built in modules. This makes it great platform for tools, not just a platform to hosting backend servers. All of those modules offer asynchronous APIs that you can use and not worry about threads. You can make Asynchronous programming in Node. Do things parallel without dealing with threads which is biggest benefit of node. And if modules or packages not enough you can build highly performing packages using c++ (add ons).

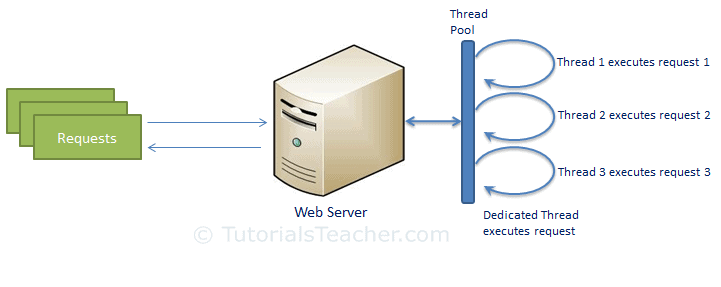
Advantages of Node.js

1. Node.js is an open-source framework under MIT license. (MIT license is a free software license originating at the Massachusetts Institute of Technology (MIT).)
2. Uses JavaScript to build entire server side application.
3. Lightweight framework that includes bare minimum modules. Other modules can be included as per the need of an application.
4. Asynchronous by default. So it performs faster than other frameworks.
5. Cross-platform framework that runs on Windows, MAC or Linux

**Node Process Model:**

## Traditional Web Server Model

In the traditional web server model, each request is handled by a dedicated thread from the thread pool. If no thread is available in the thread pool at any point of time then the request waits till the next available thread. Dedicated thread executes a particular request and does not return to thread pool until it completes the execution and returns a response.

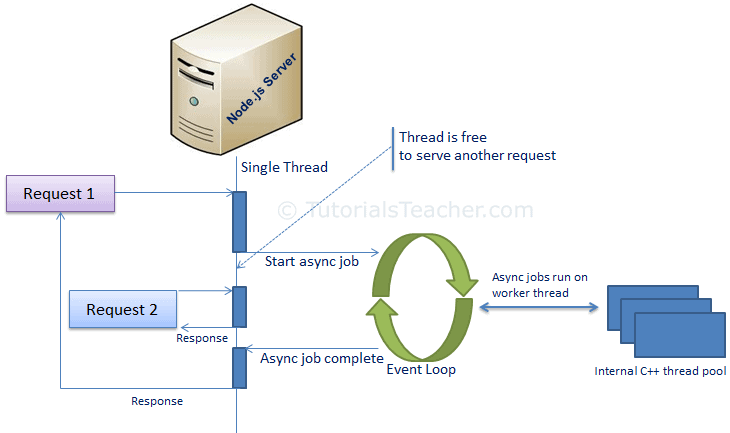


## Node.js Process Model

Node.js processes user requests differently when compared to a traditional web server model. Node.js runs in a single process and the application code runs in a single thread and thereby needs less resources than other platforms. All the user requests to your web application will be handled by a single thread and all the I/O work or long running job is performed asynchronously for a particular request. So, this single thread doesn't have to wait for the request to complete and is free to handle the next request. When asynchronous I/O work completes then it processes the request further and sends the response.

An event loop is constantly watching for the events to be raised for an asynchronous job and executing callback function when the job completes. Internally, Node.js uses [libev](http://software.schmorp.de/pkg/libev.html) for the event loop which in turn uses internal C++ thread pool to provide asynchronous I/O.

The following figure illustrates asynchronous web server model using Node.js.



Node.js process model increases the performance and scalability with a few caveats. Node.js is not fit for an application which performs CPU-intensive operations like image processing or other heavy computation work because it takes time to process a request and thereby blocks the single thread.

**Node REPL Mode:**

If you write node in bash or cmd without any script it will start REPL session (read evil print loop) it’s quickly way for test small JS.

TAP and Under Score:

TAP Completion It’s very power feature.

Single tap: It’s complete your available path.

Double tap: It’s show you all available properties.

Under Score : it’s store last valuated expression.

**The Process Object:**

The process object is a global that provides information about, and control over, the current Node.js process. As a global.

Each Node.js script runs in a process. It includes **process** object to get all the information about the current process of Node.js application.

**NPM (Node Package Manager):**

Official node package manager that enable users to do mainly 3 things:

1) Share their code with others.

2) Reuse their code across projects.

3) use code written by others.

So NPM all about code sharing and reusability, and compose ability of bigger project use smaller packages. NPM is an asset for any one work on JS project. Handle dependencies of the project.

**NPM** **command**:

It’s installed by default with node you doesn’t need install it. NPM frequently more than Node itself updated so you need to update.

To install any package you must have (package.json) and (package-lock.json) files.

1)package.json:

Adjacent file contain information about packages such version of dependency.

2)package-lock.json:

Extra information

**Semantic versioning:**

Node use this technique to update packages it’s contain 3 numbers (12.36.41)

1)first number => called major number means breaking changes happened in the release.

2)second => called minor means new feature was added to the release.

3)third => called patch number means there is bug fixes and security fixes.

~ update most new patch version

^ update most new minor verion

**NPX:**

Another binary that you get when you install node.

**Modules and Concurrency:**

Module in node mean file or folder contains code.

Node doesn’t just execute files as it wrap files with a function first:

If you access the arguments inside any function it will show you all arguments passed to this function. then if you access this key word on top of node file it will return some objects

This because node wrap the file with a function when it execute this file so module means function that receives arguments and it will return something, this arguments such require

Exports so this objects is not global they are just wrapping arguments there are customized for each module

## Node.js Module Types:

1. Core Modules
2. Local Modules
3. Third Party Modules

| Core Module | Description |
| --- | --- |
| [http](https://nodejs.org/api/http.html) | http module includes classes, methods and events to create Node.js http server. |
| [url](https://nodejs.org/api/url.html) | url module includes methods for URL resolution and parsing. |
| [querystring](https://nodejs.org/api/querystring.html) | querystring module includes methods to deal with query string. |
| [path](https://nodejs.org/api/path.html) | path module includes methods to deal with file paths. |
| [fs](https://nodejs.org/api/fs.html) | fs module includes classes, methods, and events to work with file I/O. |
| [util](https://nodejs.org/api/util.html) | util module includes utility functions useful for programmers. |

Load core module : var module = require('module\_name');

# Node.jsLocalModule

Local modules are modules created locally in your Node.js application. These modules include different functionalities of your application in separate files and folders. You can also package it and distribute it via NPM, so that Node.js community can use it. For example, if you need to connect to MongoDB and fetch data then you can create a module for it, which can be reused in your application.

## Writing Simple Module

Let's write simple logging module which logs the information, warning or error to the console.

In Node.js, module should be placed in a separate JavaScript file. So, create a Log.js file and write the following code in it.

var log = {

info: function (info) {

console.log('Info: ' + info);

},

warning:function (warning) {

console.log('Warning: ' + warning);

},

error:function (error) {

console.log('Error: ' + error);

}

};

module.exports = log

Load local module :

var myLogModule = require('./Log.js');

myLogModule.info('Node.js started');

# **Export Module in Node.js:**

The module.exports is a special object which is included in every JavaScript file in the Node.js application by default. The module is a variable that represents the current module, and exports is an object that will be exposed as a module. So, whatever you assign to module.exports will be exposed as a module.

EX1:

module.exports = 'Hello world';

var msg = require('./Messages.js');

console.log(msg);

EX2:

exports.SimpleMessage = 'Hello world';

//or

module.exports.SimpleMessage = 'Hello world';

var msg = require('./Messages.js');

console.log(msg.SimpleMessage);

## Export Function:

module.exports = function (msg) {

console.log(msg);

};

var msg = require('./Log.js');

msg('Hello World');

## Export Function as a Class:

module.exports = function (firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

this.fullName = function () {

return this.firstName + ' ' + this.lastName;

}

}

var person = require('./Person.js');

var person1 = new person('James', 'Bond');

console.log(person1.fullName());

Load Module from the Separate Folder:

Use the full path of a module file where you have exported it using module.exports. For example, if the log module in the log.js is stored under the utility folder under the root folder of your application, then import it, as shown below.

app.js => var log = require('./utility/log.js');

In the above example, . is for the root folder, and then specify the exact path of your module file. Node.js also allows us to specify the path to the folder without specifying the file name. For example, you can specify only the utility folder without specifying log.js, as shown below.

app.js => var log = require('./utility');

In the above example, Node.js will search for a package definition file called package.json inside the utility folder. This is because Node assumes that this folder is a package and will try to look for a package definition. The package.json file should be in a module directory. The package.json under utility folder specifies the file name using the main key, as shown below.

./utility/package.json

{

"name" : "log",

"main" : "./log.js"

}

Now, Node.js will find the log.js file using the main entry in package.json and import it.

Note : If the package.json file does not exist, then it will look for index.js file as a module file by default.

**Node global object:**

Defining a variable on the top level scope node module doesn’t make global one.

The way is attach it to special (global) object. This global object is the equivalent to the window object in browser. Generally don’t use global. There is variables attached to the global object such as (setTimeOut, setInterval) you can use it directly without (global).

To add something in global scope, you need to export it using export or module.export. The same way, import modules/object using require() function to access it from the global scope.

For example, to export an object in Node.js, use exports.name = object.

**Event Loop:**

If any instruction or line of code Node will start operating system process. A node processed by default will not keep running in the background unless it has reason for. Example for make process running such timeInterval will work infinity until system was crashed or user killed it manually with CTRL+C .the reason for the background work forever is (Event Loop). It’s hidden magic that will tack care of any thing asynchronous and don’t think about threads, if you want do thing synchronous you must manage it yourself. Event Loop working with multiple queues and phases. Every node process will starts this infinite loop that called event loop , if this process has no asynchronous operation to do, the lop will exit and the OS will terminate that process, if it has asynchronous work the event loop will return the callback of the process each time it need it.

**Node Clusters:**

Why node named node? Because you want in production run single node process you run multiple , one process terminated doesn’t mean the system down. process and one is master even it’s one server, so we need to monitor those worker processes. We need server have CPU cores to manage in this case you doesn’t utilize the full power of machine, so we need cluster because it has a simple job of monitoring the actual node process and starting a new one.

**Node Asynchronous Patterns:**

Node generally use callbacks pattern for Asynchronous operations. To adapt modern languages it’s use some of new techniques for synchronization such promis.

**Event Emitters:**

There are module called event-emitter in node handle events. Streams are events you can handle the event by emit() function and on().

**Web Servers:**

Node make it very easy to work with networks and make fully and customized web server. And all built in the run time. Note that the server object that we get as a result of calling the create server method is an event emitter.

The (req, res) are the request listener function receives.

**Debugging Node:**

Just run file or module with: inspect-brk and follow it by chrome extension chrome://inspect in devices tab.

## Primitive Types:

* String
* Number
* Boolean
* Undefined
* Null
* RegExp
* Buffer : used to store binary data, while reading from a file or receiving packets over the network.

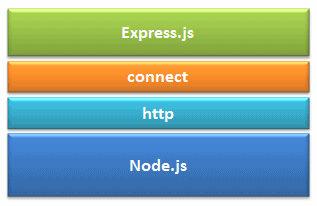
## Defaults to local:

Node's JavaScript is different from browser's JavaScript when it comes to global scope. In the browser's JavaScript, variables declared without var keyword become global. In Node.js, everything becomes local by default.

# **Express.js**

Express is a fast, unopinionated minimalist web framework for Node.js.

Express is a web application framework for Node.js. it provides various features that make web application development fast and easy which otherwise takes more time using only Node.js.



Express is based on node.js framework called (connect) that use (http) module. So any middleware which is based on (connect) will also work with express.js.

Advantages of Express.js

1. Makes Node.js web application development fast and easy.
2. Easy to configure and customize.
3. Allows you to define routes of your application based on HTTP methods and URLs.
4. Includes various middleware modules which you can use to perform additional tasks on request and response.
5. Easy to integrate with different template engines like Jade, Vash, EJS etc.
6. Allows you to define an error handling middleware.
7. Easy to serve static files and resources of your application.
8. Allows you to create REST API server.
9. Easy to connect with databases such as MongoDB, Redis, MySQL.

**API Reference :**

1. Create application : thre express function is a top-level function exported by the express modukle
2. const express = require('express')
3. const app = express()

Functions:

* 1. express.json([options]) : This is a built-in middleware function in Express. It parses incoming requests with JSON payloads and is based on [body-parser](https://expressjs.com/resources/middleware/body-parser.html).

A new body object containing the parsed data is populated on the request object after the middleware (i.e. req.body), or an empty object ({}) if there was no body to parse, the Content-Type was not matched, or an error occurred.

As req.body’s shape is based on user-controlled input, all properties and values in this object are untrusted and should be validated before trusting. For example, req.body.foo.toString() may fail in multiple ways, for example foo may not be there or may not be a string, and toString may not be a function and instead a string or other user-input.

### express.static(root, [options]) : This is a built-in middleware function in Express. It serves static files and is based on [serve-static](https://expressjs.com/resources/middleware/serve-static.html) .

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.

const options = {

dotfiles: 'ignore',

etag: false,

extensions: ['htm', 'html'],

index: false,

maxAge: '1d',

redirect: false,

setHeaders: function (res, path, stat) {

res.set('x-timestamp', Date.now())

}

}

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

* 1. Express.router([options]) : creates a new router object

const router = express.Router([options])

### express.urlencoded([options]) : It parses incoming requests with urlencoded payloads and is based on [body-parser](https://expressjs.com/resources/middleware/body-parser.html).

## Application

The app object has methods for

* Routing HTTP requests; see for example, [app.METHOD](https://expressjs.com/en/5x/api.html" \l "app.METHOD) and [app.param](https://expressjs.com/en/5x/api.html" \l "app.param).
* Configuring middleware; see [app.route](https://expressjs.com/en/5x/api.html" \l "app.route).
* Rendering HTML views; see [app.render](https://expressjs.com/en/5x/api.html" \l "app.render).
* Registering a template engine; see [app.engine](https://expressjs.com/en/5x/api.html" \l "app.engine).

It also has settings (properties) that affect how the application behaves; [app.](https://expressjs.com/en/5x/api.html" \l "app.param)set

The Express application object can be referred from the [request object](https://expressjs.com/en/5x/api.html#req) and the [response object](https://expressjs.com/en/5x/api.html#res) as req.app, and res.app, respectively.

### **Properties :**

### 1) app.locals

The app.locals object has properties that are local variables within the application.

### 2) app.mountpath : the app.mountpath property contains one or more path patterns on which a sub-app was mounted.

 sub-app is an instance of express that may be used for handling the request to a route.

const express = require('express')

const app = express() // the main app

const admin = express() // the sub app

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

console.log(admin.mountpath) // /admin

res.send('Admin Homepage')

})

app.use('/admin', admin) // mount the sub app

1. App.router : The application’s in-built instance of router. This is created lazily, on first access. You can add middleware and HTTP method routes to the router just like an application.

### Events

The mount event is fired on a sub-app, when it is mounted on a parent app. The parent app is passed to the callback function.

const admin = express()

admin.on('mount', function (parent) {

console.log('Admin Mounted')

console.log(parent) // refers to the parent app

})

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

res.send('Admin Homepage')

})

app.use('/admin', admin)

### Methods:

1. App.all(path, callback[,callback])

The app.all() method is useful for mapping “global” logic for specific path prefixes or arbitrary matches. For example, if you put the following at the top of all other route definitions, it requires that all routes from that point on require authentication, and automatically load a user. Keep in mind that these callbacks do not have to act as end-points: loadUser can perform a task, then call next() to continue matching subsequent routes.

1. app.all('\*', requireAuthentication, loadUser)

Or the equivalent:

1. app.all('\*', requireAuthentication)
2. app.all('\*', loadUser)

app.all('/api/\*', requireAuthentication)

2) app.delete(path, callback)

3) app.disable(name) app.disable(‘proxy’) app.get(‘proxy’) //false

4) app.disabled(name) : check if name false or not

5) app.enable(name)

6) app.enabled(name)

7)app.engine(ext, callback) :

Registers the given template engine callback as ext.

By default, Express will require() the engine based on the file extension. For example, if you try to render a “foo.pug” file, Express invokes the following internally, and caches the require() on subsequent calls to increase performance.

app.engine('pug', require('pug').\_\_express)

Use this method for engines that do not provide .\_\_express out of the box, or if you wish to “map” a different extension to the template engine.

For example, to map the EJS template engine to “.html” files:

app.engine('html', require('ejs').renderFile)

In this case, EJS provides a .renderFile() method with the same signature that Express expects: (path, options, callback), though note that it aliases this method as ejs.\_\_express internally so if you’re using “.ejs” extensions you don’t need to do anything.

Some template engines do not follow this convention. The [consolidate.js](https://github.com/tj/consolidate.js) library maps Node template engines to follow this convention, so they work seamlessly with Express.

const engines = require('consolidate')

app.engine('haml', engines.haml)

app.engine('html', engines.hogan)

8) app.get(name)

### 9) app.get(path, callback [, callback ...]) //for routes

### 10) app.listen([port[, host[, backlog]]][, callback]

11) app.METHOD(path, callback) : method is one of http methods

12) app.param(name, callback)

Add callback triggers to [route parameters](https://expressjs.com/en/guide/routing.html#route-parameters), where name is the name of the parameter or an array of them, and callback is the callback function. The parameters of the callback function are the request object, the response object, the next middleware, the value of the parameter and the name of the parameter, in that order.

If name is an array, the callback trigger is registered for each parameter declared in it, in the order in which they are declared. Furthermore, for each declared parameter except the last one, a call to next inside the callback will call the callback for the next declared parameter. For the last parameter, a call to next will call the next middleware in place for the route currently being processed, just like it would if name were just a string.

For example, when :user is present in a route path, you may map user loading logic to automatically provide req.user to the route, or perform validations on the parameter input.

app.param('user', f

app.param('user', function (req, res, next, id) {

// try to get the user details from the User model and attach it to the request object

User.find(id, function (err, user) {

if (err) {

next(err)

} else if (user) {

req.user = user

next()

} else {

next(new Error('failed to load user'))

}

})

})

Param callback functions are local to the router on which they are defined. They are not inherited by mounted apps or routers. Hence, param callbacks defined on app will be triggered only by route parameters defined on app routes.

All param callbacks will be called before any handler of any route in which the param occurs, and they will each be called only once in a request-response cycle, even if the parameter is matched in multiple routes, as shown in the following examples.

### 13) app.path()

Returns the canonical path of the app, a string.

const app = express()

const blog = express()

const blogAdmin = express()

app.use('/blog', blog)

blog.use('/admin', blogAdmin)

console.log(app.path()) // ''

console.log(blog.path()) // '/blog'

console.log(blogAdmin.path()) // '/blog/admin'

14) app.render(view, [locals], callback)

15) app.route(path) : Returns an instance of a single route, which you can then use to handle HTTP verbs with optional middleware. Use app.route() to avoid duplicate route names (and thus typo errors).

const app = express()

app.route('/events')

.all(function (req, res, next) {

// runs for all HTTP verbs first

// think of it as route specific middleware!

})

.get(function (req, res, next) {

res.json({})

})

.post(function (req, res, next) {

// maybe add a new event...

})

### 16) app.set(name, value)

17) app.use()

This used for Application settings note that sub-apps

* Not inherit the value of settings that have a default value. You must set the value in the sub-app.
* Inherit the value of settings with no default value; these are explicitly noted in the table below.

18) app.use(path, callback)

Mounts the specified [middleware](https://expressjs.com/guide/using-middleware.html) function or functions at the specified path: the middleware function is executed when the base of the requested path matches path.

A route will match any path that follows its path immediately with a “/”. For example: app.use('/apple', ...) will match “/apple”, “/apple/images”, “/apple/images/news”, and so on.

Since path defaults to “/”, middleware mounted without a path will be executed for every request to the app.  
For example, this middleware function will be executed for ***every*** request to the app:

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

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

next()

})

#### Middleware callback function examples

The following table provides some simple examples of middleware functions that can be used as the callback argument to app.use(), app.METHOD(), and app.all(). Even though the examples are for app.use(), they are also valid for app.use(), app.METHOD(), and app.all().

| **Usage** | **Example** |
| --- | --- |
| Single Middleware | You can define and mount a middleware function locally.  app.use(function (req, res, next) {  next();  });  A router is valid middleware.  const router = express.Router();  router.get('/', function (req, res, next) {  next();  });  app.use(router);  An Express app is valid middleware.  const subApp = express();  subApp.get('/', function (req, res, next) {  next();  });  app.use(subApp); |
| Series of Middleware | You can specify more than one middleware function at the same mount path.  const r1 = express.Router();  r1.get('/', function (req, res, next) {  next();  });  const r2 = express.Router();  r2.get('/', function (req, res, next) {  next();  });  app.use(r1, r2); |
| Array | Use an array to group middleware logically. If you pass an array of middleware as the first or only middleware parameters, then you must specify the mount path.  const r1 = express.Router();  r1.get('/', function (req, res, next) {  next();  });  const r2 = express.Router();  r2.get('/', function (req, res, next) {  next();  });  app.use('/', [r1, r2]); |
| Combination | You can combine all the above ways of mounting middleware.  function mw1(req, res, next) { next(); }  function mw2(req, res, next) { next(); }  const r1 = express.Router();  r1.get('/', function (req, res, next) { next(); });  const r2 = express.Router();  r2.get('/', function (req, res, next) { next(); });  const subApp = express();  subApp.get('/', function (req, res, next) { next(); });  app.use(mw1, [mw2, r1, r2], subApp); |

## Request

### Properties

1. Req.app
2. Req.baseUrl
3. Req.body : Contains key-value pairs of data submitted in the request body. By default, it is undefined, and is populated when you use body-parsing middleware such as [body-parser](https://www.npmjs.org/package/body-parser) and [multer](https://www.npmjs.org/package/multer).
4. const app = require('express')()
5. const bodyParser = require('body-parser')
6. const multer = require('multer') // v1.0.5
7. const upload = multer() // for parsing multipart/form-data
8. app.use(bodyParser.json()) // for parsing application/json
9. app.use(bodyParser.urlencoded({ extended: true })) // for parsing application/x-www-form-urlencoded
10. app.post('/profile', upload.array(), function (req, res, next) {
11. console.log(req.body)
12. res.json(req.body)
13. })
14. req.cookies : used with cookie-parser middleware
15. // GET 'http://www.example.com/admin/new?sort=desc'
16. app.use('/admin', function (req, res, next) {
17. console.dir(req.originalUrl) // '/admin/new?sort=desc'
18. console.dir(req.baseUrl) // '/admin'
19. console.dir(req.path) // '/new'
20. next()
21. })

6) req.params :: /user/:name req.params.name

7) req.path : contain the path part of request url

8) req.protocol : http or https or what

9) req.query

10) req.route

11) req.secure : return true if protocol is https

12) req.stale reverse to req.fresh

13) req.subdomains : An array of subdomains in the domain name of the request.

// Host: "tobi.ferrets.example.com"

console.dir(req.subdomains)

// => ["ferrets", "tobi"]

### Methods

1. Req.accepts(types)

Checks if the specified content types are acceptable, based on the request’s Accept HTTP header field. The method returns the best match, or if none of the specified content types is acceptable, returns false (in which case, the application should respond with 406 "Not Acceptable").

The type value may be a single MIME type string (such as “application/json”), an extension name such as “json”, a comma-delimited list, or an array. For a list or array, the method returns the **best** match (if any).

// Accept: text/html

req.accepts('html')

// => "html"

// Accept: text/\*, application/json

req.accepts('html')

// => "html"

req.accepts('text/html')

// => "text/html"

req.accepts(['json', 'text'])

// => "json"

req.accepts('application/json')

// => "application/json"

// Accept: text/\*, application/json

req.accepts('image/png')

req.accepts('png')

// => false

// Accept: text/\*;q=.5, application/json

req.accepts(['html', 'json'])

// => "json"

1. Req.acceptsCharsets(charset[,,,,]) :

Returns the first accepted charset of the specified character sets, based on the request’s Accept-Charset HTTP header field. If none of the specified charsets is accepted, returns false.

1. Req.acceptsEncodings(encodings[]) :

Returns the first accepted encoding of the specified encodings, based on the request’s Accept-Encoding HTTP header field. If none of the specified encodings is accepted, returns false.

1. Req.acceptLanguages(languages[]):

Returns the first accepted language of the specified languages, based on the request’s Accept-Language HTTP header field. If none of the specified languages is accepted, returns false.

1. Req.get(field) : return specified http request header field
2. Req.is(type) : return the matching content-type if the incoming http header field match the MIME type fields

## Response

### Properties

1. Res.app : identical to req.app
2. Res.headersSent : Boolean property that indicates if the app sent HTTP headers for the response
3. Res.locals : An object that contains response local variables scoped to the request, and therefore available only to the view(s) rendered during that request / response cycle (if any). Otherwise, this property is identical to [app.locals](https://expressjs.com/en/5x/api.html" \l "app.locals).

This property is useful for exposing request-level information such as the request path name, authenticated user, user settings, and so on.

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

res.locals.user = req.user

res.locals.authenticated = !req.user.anonymous

next()

})

### Methods

1. Res.append(field, [value])
2. Res.attachment([filename])

Sets the HTTP response Content-Disposition header field to “attachment”. If a filename is given, then it sets the Content-Type based on the extension name via res.type(), and sets the Content-Disposition “filename=” parameter.

res.attachment()

// Content-Disposition: attachment

res.attachment('path/to/logo.png')

// Content-Disposition: attachment; filename="logo.png"

// Content-Type: image/png

1. Res.cookie(name, value, options) : set cookie name to value
2. Res.clearCookie(name, options) : must options identicals
3. Res.download(path, [filename], [options], fn)

res.download('/report-12345.pdf')

res.download('/report-12345.pdf', 'report.pdf')

res.download('/report-12345.pdf', 'report.pdf', function (err) {

if (err) {

// Handle error, but keep in mind the response may be partially-sent

// so check res.headersSent

} else {

// decrement a download credit, etc.

}

})

1. Res.end() : end reponse process
2. Res.formate(object)
3. Res.get(field) : the field is case sensitive
4. Res.json(object) : send json response
5. Res.location(path) : set the location path
6. Res.redirect(path)
7. Res.render(view, [locals], [callback])

// send the rendered view to the client

res.render('index')

// if a callback is specified, the rendered HTML string has to be sent explicitly

res.render('index', function (err, html) {

res.send(html)

})

// pass a local variable to the view

res.render('user', { name: 'Tobi' }, function (err, html) {

// ...

})

1. Res.send([body])
2. Res.sendFile(path,[options],[callbacks])
3. Res.sendStatus(statusCode) == res.status().send()
4. Res.set(field, value) //set header field value
5. Res.status(code)
6. Res.type(type) //set response header type

## Router

### Methods

1. Router.all(path, callbacks[])

This method is just like the router.METHOD() methods, except that it matches all HTTP methods (verbs).

This method is extremely useful for mapping “global” logic for specific path prefixes or arbitrary matches. For example, if you placed the following route at the top of all other route definitions, it would require that all routes from that point on would require authentication, and automatically load a user. Keep in mind that these callbacks do not have to act as end points; loadUser can perform a task, then call next() to continue matching subsequent routes.

router.all('\*', requireAuthentication, loadUser)

Or the equivalent:

router.all('\*', requireAuthentication)

router.all('\*', loadUser)

Another example of this is white-listed “global” functionality. Here the example is much like before, but it only restricts paths prefixed with “/api”:

router.all('/api/\*', requireAuthentication)

1. Router.METHOD(path, callback) : such get, post, put, patch
2. Router.params(name, callback) : Adds callback triggers to route parameters, where name is the name of the parameter and callback is the callback function. Although name is technically optional, using this method without it is deprecated starting with Express v4.11.0 (see below).

The parameters of the callback function are:

* req, the request object.
* res, the response object.
* next, indicating the next middleware function.
* The value of the name parameter.
* The name of the parameter.
* router.param('user', function (req, res, next, id) {
* // try to get the user details from the User model and attach it to the request object
* User.find(id, function (err, user) {
* if (err) {
* next(err)
* } else if (user) {
* req.user = user
* next()
* } else {
* next(new Error('failed to load user'))
* }
* })
* })

1. Router.route(path) : handle all verbs

const router = express.Router()

router.param('user\_id', function (req, res, next, id) {

// sample user, would actually fetch from DB, etc...

req.user = {

id: id,

name: 'TJ'

}

next()

})

router.route('/users/:user\_id')

.all(function (req, res, next) {

// runs for all HTTP verbs first

// think of it as route specific middleware!

next()

})

.get(function (req, res, next) {

res.json(req.user)

})

.put(function (req, res, next) {

// just an example of maybe updating the user

req.user.name = req.params.name

// save user ... etc

res.json(req.user)

})

.post(function (req, res, next) {

next(new Error('not implemented'))

})

.delete(function (req, res, next) {

next(new Error('not implemented'))

})

1. Router.use(path, function) : Uses the specified middleware function or functions, with optional mount path path, that defaults to “/”.

const express = require('express')

const app = express()

const router = express.Router()

// simple logger for this router's requests

// all requests to this router will first hit this middleware

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

console.log('%s %s %s', req.method, req.url, req.path)

next()

})

// this will only be invoked if the path starts with /bar from the mount point

router.use('/bar', function (req, res, next) {

// ... maybe some additional /bar logging ...

next()

})

// always invoked

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

res.send('Hello World')

})

app.use('/foo', router)

app.listen(3000)

const logger = require('morgan')

router.use(logger())

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

router.use(function (req, res) {

res.send('Hello')

})

**NOTE**: Although these middleware functions are added via a particular router, **when** they run is defined by the path they are attached to (not the router). Therefore, middleware added via one router may run for other routers if its routes match. For example, this code shows two different routers mounted on the same path:

const authRouter = express.Router()

const openRouter = express.Router()

authRouter.use(require('./authenticate').basic(usersdb))

authRouter.get('/:user\_id/edit', function (req, res, next) {

// ... Edit user UI ...

})

openRouter.get('/', function (req, res, next) {

// ... List users ...

})

openRouter.get('/:user\_id', function (req, res, next) {

// ... View user ...

})

app.use('/users', authRouter)

app.use('/users', openRouter)

Even though the authentication middleware was added via the authRouter it will run on the routes defined by the openRouter as well since both routers were mounted on /users. To avoid this behavior, use different paths for each router.