[Express](http://www.expressjs.com.cn/)

* [首页](http://www.expressjs.com.cn/)
  + [入门](http://www.expressjs.com.cn/starter/installing.html)
  + [使用指南](http://www.expressjs.com.cn/guide/routing.html)
  + [API 中文手册](http://www.expressjs.com.cn/4x/api.html)
  + [进阶话题](http://www.expressjs.com.cn/advanced/developing-template-engines.html)
  + [有用的资源](http://www.expressjs.com.cn/resources/glossary.html)

Express 4.x API 中文手册

**express()**

创建一个 Express 应用。express() 是一个由 ***express*** 模块导出的入口（top-level）函数。

var express = require('express');//这里使用了require，看下面知道这里应该是导入了一个函数

var app = express();

**内置方法**

**express.static(root, [options])**

express.static 是 Express 内置的唯一一个中间件。是基于 [serve-static](https://github.com/expressjs/serve-static) 开发的，负责托管 Express 应用内的静态资源。

root 参数指的是静态资源文件所在的根目录。

options 对象是可选的，支持以下属性：

| **属性** | **描述** | **类型** | **默认值** |
| --- | --- | --- | --- |
| dotfiles | Option for serving dotfiles. Possible values are “allow”, “deny”, and “ignore” | String | “ignore” |
| etag | Enable or disable etag generation（me:是否产生etag标签，控制浏览器的缓存） | Boolean | true |
| extensions | Sets file extension fallbacks. | Boolean | false |
| index | Sends directory index file. Set false to disable directory indexing. | Mixed | “index.html” |
| lastModified | Set the Last-Modified header to the last modified date of the file on the OS. Possible values are true or false. | Boolean | true |
| maxAge | Set the max-age property of the Cache-Control header in milliseconds or a string in [ms format](https://www.npmjs.org/package/ms) | Number | 0 |
| redirect | Redirect to trailing “/” when the pathname is a directory（me：当路径名是文件目录时从定向到/-即根目录）. | Boolean | true |
| setHeaders | Function for setting HTTP headers to serve with the file. | Function |  |

关于此中间件的细节，请参考 [通过 Express 托管静态资源文件](http://www.expressjs.com.cn/starter/static-files.html)。

**Application**

The app object conventionally denotes the Express application（app对象就代表了Express 应用）. Create it by calling the top-level express() function exported by the Express module（通过导出Express模块，并调用express()方法来创建）:

var express = require('express');

var app = express();

//注册api，即路由

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

res.send('hello world');//响应

});

//监听端口

app.listen(3000);

The app object has methods for

* Routing HTTP requests; see for example, [app.METHOD](http://www.expressjs.com.cn/4x/api.html#app.METHOD) and [app.param](http://www.expressjs.com.cn/4x/api.html#app.param).
* Configuring middleware; see [app.route](http://www.expressjs.com.cn/4x/api.html#app.route).
* Rendering HTML views; see [app.render](http://www.expressjs.com.cn/4x/api.html#app.render).
* Registering a template engine; see [app.engine](http://www.expressjs.com.cn/4x/api.html#app.engine).

App对象包含的方法有：

1. 路由http请求
2. 配置中间件
3. 渲染html视图
4. 注册模板引擎
5. 和一些其他的属性设置来控制app的表现和行为

It also has settings (properties) that affect how the application behaves; for more information, see [Application settings](http://www.expressjs.com.cn/4x/api.html#app.settings.table).

**Properties**

**app.locals**

The app.locals object is a JavaScript object, and its properties are local variables（局部变量） within the application.

app.locals.title

// => 'My App'

app.locals.email

// => 'me@myapp.com'

Once set, the value of app.locals properties persist throughout the life of the application, in contrast with [res.locals](http://www.expressjs.com.cn/4x/api.html#res.locals) properties that are valid only for the lifetime of the request.

app.locals属性一旦设定则会贯穿应用的整个生命周期，与之相对的是res.locals,它的变量则只会与请求的生命周期相关

You can access local variables in templates rendered within the application（在应用的模板渲染中可以访问这些local变量）. This is useful for providing helper functions to templates, as well as app-level data. Note, however, that you cannot access local variables in middleware（在中间件中不能访问这些local变量）.

app.locals.title = 'My App';

app.locals.strftime = require('strftime');

app.locals.email = 'me@myapp.com';

**app.mountpath**

The app.mountpath property is the path pattern(s) on which a sub app（二级app） was mounted.

二级app也是express的实例，它们可能用来处理特定的路由请求

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

var express = require('express');

var app = express(); // the main app

var 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

It is similar to the [baseUrl](http://www.expressjs.com.cn/4x/api.html#req.baseUrl) property of the req object, except req.baseUrl returns the matched URL path（req.baseUrl返回匹配的URL路径而非匹配的模式）, instead of the matched pattern(s).

If a sub-app is mounted on multiple path patterns, app.mountpath returns the list of patterns it is mounted on, as shown in the following example.

var admin = express();

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

console.log(admin.mountpath); // [ '/adm\*n', '/manager' ]

res.send('Admin Homepage');

})

var secret = express();

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

console.log(secret.mountpath); // /secr\*t

res.send('Admin Secret');

});

admin.use('/secr\*t', secret); // load the 'secret' router on '/secr\*t', on the 'admin' sub app

app.use(['/adm\*n', '/manager'], admin); // load the 'admin' router on '/adm\*n' and '/manager', on the parent app

**Events**

**app.on('mount', callback(parent))**

The mount event is fired on a sub-app（mount事件在二级app上会被触发）, when it is mounted on a parent app. The parent app is passed to the callback function（父app将会被传递到这个回调函数）.

var 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);

//===========20170921

**Methods**

**app.all(path, callback [, callback ...])**

This method is like the standard [app.METHOD()](http://www.expressjs.com.cn/4x/api.html#app.METHOD) methods, except it matches all HTTP verbs.

It’s 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.

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

Or the equivalent:

app.all('\*', requireAuthentication)

app.all('\*', loadUser);

Another example is white-listed “global” functionality. The example is much like before, however it only restricts paths that start with “/api”:

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

**app.delete(path, callback [, callback ...])**

Routes HTTP DELETE requests to the specified path with the specified callback functions. For more information, see the [routing guide](http://www.expressjs.com.cn/guide/routing.html).

You can provide multiple callback functions that behave just like middleware, except these callbacks can invoke next('route') to bypass the remaining route callback(s). You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there’s no reason to proceed with the current route.

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

res.send('DELETE request to homepage');

});

**app.disable(name)**

Sets the Boolean setting name to false, where name is one of the properties from the [app settings table](http://www.expressjs.com.cn/4x/api.html#app.settings.table). Calling app.set('foo', false) for a Boolean property is the same as calling app.disable('foo').

For example:

app.disable('trust proxy');

app.get('trust proxy');

// => false

**app.disabled(name)**

Returns true if the Boolean setting name is disabled (false), where name is one of the properties from the [app settings table](http://www.expressjs.com.cn/4x/api.html#app.settings.table).

app.disabled('trust proxy');

// => true

app.enable('trust proxy');

app.disabled('trust proxy');

// => false

**app.enable(name)**

Sets the Boolean setting name to true, where name is one of the properties from the [app settings table](http://www.expressjs.com.cn/4x/api.html#app.settings.table). Calling app.set('foo', true) for a Boolean property is the same as calling app.enable('foo').

app.enable('trust proxy');

app.get('trust proxy');

// => true

**app.enabled(name)**

Returns true if the setting name is enabled (true), where name is one of the properties from the [app settings table](http://www.expressjs.com.cn/4x/api.html#app.settings.table).

app.enabled('trust proxy');

// => false

app.enable('trust proxy');

app.enabled('trust proxy');

// => true

**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.jade” file, Express invokes the following internally, and caches the require() on subsequent calls to increase performance.

app.engine('jade', require('jade').\_\_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 seemlessly with Express.

var engines = require('consolidate');

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

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

**app.get(name)**

Returns the value of name app setting, where name is one of strings in the [app settings table](http://www.expressjs.com.cn/4x/api.html#app.settings.table). For example:

app.get('title');

// => undefined

app.set('title', 'My Site');

app.get('title');

// => "My Site"

**app.get(path, callback [, callback ...])**

Routes HTTP GET requests to the specified path with the specified callback functions. For more information, see the [routing guide](http://www.expressjs.com.cn/guide/routing.html).

You can provide multiple callback functions that behave just like middleware, except these callbacks can invoke next('route') to bypass the remaining route callback(s). You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there’s no reason to proceed with the current route.

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

res.send('GET request to homepage');

});

**app.listen(port, [hostname], [backlog], [callback])**

Binds and listens for connections on the specified host and port. This method is identical to Node’s [http.Server.listen()](http://nodejs.org/api/http.html#http_server_listen_port_hostname_backlog_callback).

var express = require('express');

var app = express();

app.listen(3000);

The app returned by express() is in fact a JavaScript Function, designed to be passed to Node’s HTTP servers as a callback to handle requests. This makes it easy to provide both HTTP and HTTPS versions of your app with the same code base, as the app does not inherit from these (it is simply a callback):

var express = require('express');

var https = require('https');

var http = require('http');

var app = express();

http.createServer(app).listen(80);

https.createServer(options, app).listen(443);

The app.listen() method is a convenience method for the following (for HTTP only):

app.listen = function() {

var server = http.createServer(this);

return server.listen.apply(server, arguments);

};

**app.METHOD(path, callback [, callback ...])**

Routes an HTTP request, where METHOD is the HTTP method of the request, such as GET, PUT, POST, and so on, in lowercase. Thus, the actual methods are app.get(), app.post(), app.put(), and so on. See below for the complete list.

For more information, see the [routing guide](http://www.expressjs.com.cn/guide/routing.html).

Express supports the following routing methods corresponding to the HTTP methods of the same names:

|  |  |  |
| --- | --- | --- |
| * checkout * connect * copy * delete * get * head * lock * merge * mkactivity | * mkcol * move * m-search * notify * options * patch * post * propfind * proppatch | * purge * put * report * search * subscribe * trace * unlock * unsubscribe |

To route methods which translate to invalid JavaScript variable names, use the bracket notation. For example, app['m-search']('/', function ....

You can provide multiple callback functions that behave just like middleware, except that these callbacks can invoke next('route') to bypass the remaining route callback(s). You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there is no reason to proceed with the current route.

The API documentation has explicit entries only for the most popular HTTP methods app.get(), app.post(), app.put(), and app.delete(). However, the other methods listed above work in exactly the same way.

There is a special routing method, app.all(), that is not derived from any HTTP method. It loads middleware at a path for all request methods.

In the following example, the handler is executed for requests to “/secret” whether using GET, POST, PUT, DELETE, or any other HTTP request method.

app.all('/secret', function (req, res, next) {

console.log('Accessing the secret section ...')

next() // pass control to the next handler

})

**app.param([name], callback)**

Add callback triggers to route parameters, where name is the name of the parameter or an array of them, and function is the callback function. The parameters of the callback function are the request object, the response object, the next middleware, and the value 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', 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.

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

console.log('CALLED ONLY ONCE');

next();

})

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

console.log('although this matches');

next();

});

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

console.log('and this matches too');

res.end();

});

On GET /user/42, the following is printed:

CALLED ONLY ONCE

although this matches

and this matches too

app.param(['id', 'page'], function (req, res, next, value) {

console.log('CALLED ONLY ONCE with', value);

next();

})

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

console.log('although this matches');

next();

});

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

console.log('and this matches too');

res.end();

});

On GET /user/42/3, the following is printed:

CALLED ONLY ONCE with 42

CALLED ONLY ONCE with 3

although this matches

and this matches too

The following section describes app.param(callback), which is deprecated as of v4.11.0.

The behavior of the app.param(name, callback) method can be altered entirely by passing only a function to app.param(). This function is a custom implementation of how app.param(name, callback) should behave - it accepts two parameters and must return a middleware.

The first parameter of this function is the name of the URL parameter that should be captured, the second parameter can be any JavaScript object which might be used for returning the middleware implementation.

The middleware returned by the function decides the behavior of what happens when a URL parameter is captured.

In this example, the app.param(name, callback) signature is modified to app.param(name, accessId). Instead of accepting a name and a callback, app.param() will now accept a name and a number.

var express = require('express');

var app = express();

// customizing the behavior of app.param()

app.param(function(param, option) {

return function (req, res, next, val) {

if (val == option) {

next();

}

else {

res.sendStatus(403);

}

}

});

// using the customized app.param()

app.param('id', 1337);

// route to trigger the capture

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

res.send('OK');

})

app.listen(3000, function () {

console.log('Ready');

})

In this example, the app.param(name, callback) signature remains the same, but instead of a middleware callback, a custom data type checking function has been defined to validate the data type of the user id.

app.param(function(param, validator) {

return function (req, res, next, val) {

if (validator(val)) {

next();

}

else {

res.sendStatus(403);

}

}

})

app.param('id', function (candidate) {

return !isNaN(parseFloat(candidate)) && isFinite(candidate);

});

The ‘.’ character can’t be used to capture a character in your capturing regexp. For example you can’t use '/user-.+/' to capture 'users-gami', use [\\s\\S]or [\\w\\W] instead (as in '/user-[\\s\\S]+/'.

Examples:

//captures '1-a\_6' but not '543-azser-sder'

router.get('/[0-9]+-[[\\w]]\*', function);

//captures '1-a\_6' and '543-az(ser"-sder' but not '5-a s'

router.get('/[0-9]+-[[\\S]]\*', function);

//captures all (equivalent to '.\*')

router.get('[[\\s\\S]]\*', function);

**app.path()**

Returns the canonical path of the app, a string.

var app = express()

, blog = express()

, 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'

The behavior of this method can become very complicated in complex cases of mounted apps: it is usually better to use [req.baseUrl](http://www.expressjs.com.cn/4x/api.html#req.baseUrl) to get the canonical path of the app.

**app.post(path, callback [, callback ...])**

Routes HTTP POST requests to the specified path with the specified callback functions. For more information, see the [routing guide](http://www.expressjs.com.cn/guide/routing.html).

You can provide multiple callback functions that behave just like middleware, except that these callbacks can invoke next('route') to bypass the remaining route callback(s). You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there’s no reason to proceed with the current route.

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

res.send('POST request to homepage');

});

**app.put(path, callback [, callback ...])**

Routes HTTP PUT requests to the specified path with the specified callback functions. For more information, see the [routing guide](http://www.expressjs.com.cn/guide/routing.html).

You can provide multiple callback functions that behave just like middleware, except that these callbacks can invoke next('route') to bypass the remaining route callback(s). You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there’s no reason to proceed with the current route.

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

res.send('PUT request to homepage');

});

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

Returns the rendered HTML of a view via the callback function. It accepts an optional parameter that is an object containing local variables for the view. It is like [res.render()](http://www.expressjs.com.cn/4x/api.html#res.render), except it cannot send the rendered view to the client on its own.

Think of app.render() as a utility function for generating rendered view strings. Internally res.render() uses app.render() to render views.

The local variable cache is reserved for enabling view cache. Set it to true, if you want to cache view during development; view caching is enabled in production by default.

app.render('email', function(err, html){

// ...

});

app.render('email', { name: 'Tobi' }, function(err, html){

// ...

});

**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).

var 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...

})

**app.set(name, value)**

Assigns setting name to value, where name is one of the properties from the [app settings table](http://www.expressjs.com.cn/4x/api.html#app.settings.table).

Calling app.set('foo', true) for a Boolean property is the same as calling app.enable('foo'). Similarly, calling app.set('foo', false) for a Boolean property is the same as calling app.disable('foo').

Retrieve the value of a setting with [app.get()](http://www.expressjs.com.cn/4x/api.html#app.get).

app.set('title', 'My Site');

app.get('title'); // "My Site"

**Application Settings**

If name is one of the application settings, it affects the behavior of the application. The following table lists application settings.

| **Property** | **Type** | **Value** | **Default** |
| --- | --- | --- | --- |
| case sensitive routing | Boolean | Enable case sensitivity. | Disabled. Treats "/Foo" and "/foo" as the same. |
| env | String | Environment mode. | process.env.NODE\_ENV(NODE\_ENVenvironment variable) or “development”. |
| etag | Varied | Set the ETag response header. For possible values, see the [etag options table](http://www.expressjs.com.cn/4x/api.html#etag.options.table).  [More about the HTTP ETag header](http://en.wikipedia.org/wiki/HTTP_ETag). |  |
| jsonp callback name | String | Specifies the default JSONP callback name. | ?callback= |
| json replacer | String | JSON replacer callback. | null |
| json spaces | Number | When set, sends prettified JSON string indented with the specified amount of spaces. | Disabled. |
| query parser | String | The query parser to use, either “simple” or “extended”. The simple query parser is based on Node’s native query parser, [querystring](http://nodejs.org/api/querystring.html). The extended query parser is based on [qs](https://www.npmjs.org/package/qs). | "extended" |
| strict routing | Boolean | Enable strict routing. | Disabled. Treats "/foo" and "/foo/" as the same by the router. |
| subdomain offset | Number | The number of dot-separated parts of the host to remove to access subdomain. | 2 |
| trust proxy | Varied | Indicates the app is behind a front-facing proxy, and to use the X-Forwarded-\*headers to determine the connection and the IP address of the client. NOTE: X-Forwarded-\* headers are easily spoofed and the detected IP addresses are unreliable.  trust proxy is disabled by default. When enabled, Express attempts to determine the IP address of the client connected through the front-facing proxy, or series of proxies. The req.ips property, then, contains an array of IP addresses the client is connected through. To enable it, use the values described in the [trust proxy options table](http://www.expressjs.com.cn/4x/api.html#trust.proxy.options.table).  The trust proxy setting is implemented using the [proxy-addr](https://www.npmjs.org/package/proxy-addr) package. For more information, see its documentation. | Disabled. |
| views | String or Array | A directory or an array of directories for the application's views. If an array, the views are looked up in the order they occur in the array. | process.cwd() + '/views' |
| view cache | Boolean | Enables view template compilation caching. | true in production. |
| view engine | String | The default engine extension to use when omitted. |  |
| x-powered-by | Boolean | Enables the "X-Powered-By: Express" HTTP header. | true |

**Options for `trust proxy` setting**

| **Type** | **Value** |
| --- | --- |
| Boolean | If true, the client’s IP address is understood as the left-most entry in the X-Forwarded-\* header.  If false, the app is understood as directly facing the Internet and the client’s IP address is derived from req.connection.remoteAddress. This is the default setting. |
| IP addresses | An IP address, subnet, or an array of IP addresses, and subnets to trust. The following is the list of pre-configured subnet names.   * loopback - 127.0.0.1/8, ::1/128 * linklocal - 169.254.0.0/16, fe80::/10 * uniquelocal - 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16, fc00::/7   Set IP addresses in any of the following ways:  app.set('trust proxy', 'loopback') // specify a single subnet  app.set('trust proxy', 'loopback, 123.123.123.123') // specify a subnet and an address  app.set('trust proxy', 'loopback, linklocal, uniquelocal') // specify multiple subnets as CSV  app.set('trust proxy', ['loopback', 'linklocal', 'uniquelocal']) // specify multiple subnets as an array  When specified, the IP addresses or the subnets are excluded from the address determination process, and the untrusted IP address nearest to the application server is determined as the client’s IP address. |
| Number | Trust the nth hop from the front-facing proxy server as the client. |
| Function | Custom trust implementation. Use this only if you know what you are doing.  app.set('trust proxy', function (ip) {  if (ip === '127.0.0.1' || ip === '123.123.123.123') return true; // trusted IPs  else return false;  }) |

**Options for `etag` setting**

| **Type** | **Value** |
| --- | --- |
| Boolean | true enables weak ETag. This is the default setting. false disables ETag altogether. |
| String | If "strong", enables strong ETag. If "weak", enables weak ETag. |
| Function | Custom ETag function implementation. Use this only if you know what you are doing.  app.set('etag', function (body, encoding) {  return generateHash(body, encoding); // consider the function is defined  }) |

**app.use([path,] function [, function...])**

Mounts the [middleware](http://www.expressjs.com.cn/guide/using-middleware.html) function(s) at the path. If path is not specified, it defaults to “/”.

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

req.originalUrl in a middleware is a combination of req.baseUrl and req.path, as shown in the following example.

app.use('/admin', function(req, res, next) {

// GET 'http://www.example.com/admin/new'

console.log(req.originalUrl); // '/admin/new'

console.log(req.baseUrl); // '/admin'

console.log(req.path); // '/new'

next();

});

Mounting a middleware at a path will cause the middleware function to be executed whenever the base of the requested path matches the path.

Since path defaults to “/”, middleware mounted without a path will be executed for every request to the app.

// this middleware will be executed for every request to the app

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

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

next();

})

Middleware functions are executed sequentially, therefore the order of middleware inclusion is important.

// this middleware will not allow the request to go beyond it

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

res.send('Hello World');

})

// requests will never reach this route

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

res.send('Welcome');

})

path can be a string representing a path, a path pattern, a regular expression to match paths, or an array of combinations thereof.

The middleware in the below are simple examples.

| **Type** | **Example** |
| --- | --- |
| Path | // will match paths starting with /abcd  app.use('/abcd', function (req, res, next) {  next();  }) |
| Path Pattern | // will match paths starting with /abcd and /abd  app.use('/abc?d', function (req, res, next) {  next();  })  // will match paths starting with /abcd, /abbcd, /abbbbbcd and so on  app.use('/ab+cd', function (req, res, next) {  next();  })  // will match paths starting with /abcd, /abxcd, /abFOOcd, /abbArcd and so on  app.use('/ab\\*cd', function (req, res, next) {  next();  })  // will match paths starting with /ad and /abcd  app.use('/a(bc)?d', function (req, res, next) {  next();  }) |
| Regular Expression | // will match paths starting with /abc and /xyz  app.use(/\/abc|\/xyz/, function (req, res, next) {  next();  }) |
| Array | // will match paths starting with /abcd, /xyza, /lmn, and /pqr  app.use(['/abcd', '/xyza', /\/lmn|\/pqr/], function (req, res, next) {  next();  }) |

function can be a middleware function, a series of middleware functions, an array of middleware functions, or a combination of all of them. Since [router](http://www.expressjs.com.cn/4x/api.html#router) and [app](http://www.expressjs.com.cn/4x/api.html#application) implement the middleware interface, you can use them as you would any other middleware function.

| **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.  var router = express.Router();  router.get('/', function (req, res, next) {  next();  })  app.use(router);  An Express app is valid middleware.  var 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.  var r1 = express.Router();  r1.get('/', function (req, res, next) {  next();  })  var 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.  var r1 = express.Router();  r1.get('/', function (req, res, next) {  next();  })  var 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(); }  var r1 = express.Router();  r1.get('/', function (req, res, next) { next(); });  var r2 = express.Router();  r2.get('/', function (req, res, next) { next(); });  var subApp = express();  subApp.get('/', function (req, res, next) { next(); });  app.use(mw1, [mw2, r1, r2], subApp); |

Following are some examples of using the [express.static](http://www.expressjs.com.cn/guide/using-middleware.html#middleware.built-in) middleware in an Express app.

Serve static content for the app from the “public” directory in the application directory:

// GET /style.css etc

app.use(express.static(\_\_dirname + '/public'));

Mount the middleware at “/static” to serve static content only when their request path is prefixed with “/static”:

// GET /static/style.css etc.

app.use('/static', express.static(\_\_dirname + '/public'));

Disable logging for static content requests by loading the logger middleware after the static middleware:

app.use(express.static(\_\_dirname + '/public'));

app.use(logger());

Serve static files from multiple directories, but give precedence to “./public” over the others:

app.use(express.static(\_\_dirname + '/public'));

app.use(express.static(\_\_dirname + '/files'));

app.use(express.static(\_\_dirname + '/uploads'));

**Request**

The req object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on. In this documentation and by convention, the object is always referred to as req (and the HTTP response is res) but its actual name is determined by the parameters to the callback function in which you’re working.

For example:

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

res.send('user ' + req.params.id);

});

But you could just as well have:

app.get('/user/:id', function(request, response){

response.send('user ' + request.params.id);

});

**Properties**

In Express 4, req.files is no longer available on the req object by default. To access uploaded files on the req.files object, use a multipart-handling middleware like [busboy](https://www.npmjs.com/package/busboy), [multer](https://www.npmjs.com/package/multer), [formidable](https://www.npmjs.com/package/formidable), [multiparty](https://www.npmjs.com/package/multiparty), [connect-multiparty](https://www.npmjs.com/package/connect-multiparty), or [pez](https://www.npmjs.com/package/pez).

**req.app**

This property holds a reference to the instance of the express application that is using the middleware.

If you follow the pattern in which you create a module that just exports a middleware in order for requiring it in your main file, then the middleware can access the express instance via req.app

For example:

//index.js

app.get("/viewdirectory", require("./mymiddleware.js"))

//mymiddleware.js

module.exports = function (req, res) {

res.send("The views directory is " + req.app.get("views"));

});

**req.baseUrl**

The URL path on which a router instance was mounted. For example:

var greet = express.Router();

greet.get('/jp', function (req, res) {

console.log(req.baseUrl); // /greet

res.send('Konichiwa!');

});

app.use('/greet', greet); // load the router on '/greet'

Even if you use a path pattern or a set of path patterns to load the router, the baseUrl property returns the matched string, not the pattern(s). In the following example, the greet router is loaded on two path patterns.

app.use(['/gre+t', '/hel{2}o'], greet); // load the router on '/gre+t' and '/hel{2}o'

When a request is made to /greet/jp, req.baseUrl is “/greet”. When a request is made to /hello/jp, req.baseUrl is “/hello”.

req.baseUrl is similar to the [mountpath](http://www.expressjs.com.cn/4x/api.html#app.mountpath) property of the app object, except app.mountpath returns the matched path pattern(s).

**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).

This example shows how to use body-parsing middleware to populate req.body.

var app = require('express')();

var bodyParser = require('body-parser');

var multer = require('multer');

app.use(bodyParser.json()); // for parsing application/json

app.use(bodyParser.urlencoded({ extended: true })); // for parsing application/x-www-form-urlencoded

app.use(multer()); // for parsing multipart/form-data

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

console.log(req.body);

res.json(req.body);

})

**req.cookies**

When using [cookie-parser](https://www.npmjs.com/package/cookie-parser) middleware, this property is an object that contains cookies sent by the request. If the request contains no cookies, it defaults to {}.

// Cookie: name=tj

req.cookies.name

// => "tj"

For more information, issues, or concerns, see [cookie-parser](https://github.com/expressjs/cookie-parser).

**req.fresh**

Indicates whether the request is “fresh.” It is the opposite of req.stale.

It is true if the cache-control request header doesn’t have a no-cache directive and any of the following is true:

* The if-modified-since request header is specified and last-modified request header is equal to or eariler than the modified response header.
* The if-none-match request header is \*.
* The if-none-match request header, after being parsed into its directives, does not match the etag response header.

req.fresh

// => true

For more information, issues, or concerns, see [fresh](https://github.com/jshttp/fresh).

**req.hostname**

Contains the hostname from the “Host” HTTP header.

// Host: "example.com:3000"

req.hostname

// => "example.com"

**req.ip**

The remote IP address of the request.

If the trust proxy is setting enabled, it is the upstream address; see [Express behind proxies](http://www.expressjs.com.cn/guide/behind-proxies.html) for more information.

req.ip

// => "127.0.0.1"

**req.ips**

When the trust proxy setting is true, this property contains an array of IP addresses specified in the “X-Forwarded-For” request header. Otherwise, it contains an empty array.

For example, if “X-Forwarded-For” is “client, proxy1, proxy2”, req.ips would be ["client", "proxy1", "proxy2"], where “proxy2” is the furthest downstream.

For more information on the trust proxy setting, see [app.set](http://www.expressjs.com.cn/4x/api.html#app.set).

**req.originalUrl**

req.url is not a native Express property, it is inherited from Node’s [http module](https://nodejs.org/api/http.html#http_message_url).

This property is much like req.url; however, it retains the original request URL, allowing you to rewrite req.url freely for internal routing purposes. For example, the “mounting” feature of [app.use()](http://www.expressjs.com.cn/4x/api.html#app.use) will rewrite req.url to strip the mount point.

// GET /search?q=something

req.originalUrl

// => "/search?q=something"

**req.params**

An object containing properties mapped to the named route “parameters”. For example, if you have the route /user/:name, then the “name” property is available as req.params.name. This object defaults to {}.

// GET /user/tj

req.params.name

// => "tj"

When you use a regular expression for the route definition, capture groups are provided in the array using req.params[n], where n is the nthcapture group. This rule is applied to unnamed wild card matches with string routes such as /file/\*:

// GET /file/javascripts/jquery.js

req.params[0]

// => "javascripts/jquery.js"

**req.path**

Contains the path part of the request URL.

// example.com/users?sort=desc

req.path

// => "/users"

When called from a middleware, the mount point is not included in req.path. Look up [app.use()](http://www.expressjs.com.cn/4x/api.html#app.use) for more details.

**req.protocol**

The request protocol string, “http” or “https” when requested with TLS. When the “trust proxy” [setting](http://www.expressjs.com.cn/4x/api.html#trust.proxy.options.table) trusts the socket address, the value of the “X-Forwarded-Proto” header (“http” or “https”) field will be trusted and used if present.

req.protocol

// => "http"

**req.query**

An object containing a property for each query string parameter in the route. If there is no query string, it is the empty object, {}.

// GET /search?q=tobi+ferret

req.query.q

// => "tobi ferret"

// GET /shoes?order=desc&shoe[color]=blue&shoe[type]=converse

req.query.order

// => "desc"

req.query.shoe.color

// => "blue"

req.query.shoe.type

// => "converse"

**req.route**

The currently-matched route, a string. For example:

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

console.log(req.route);

res.send('GET');

})

Example output from the previous snippet:

{ path: '/user/:id?',

stack:

[ { handle: [Function: userIdHandler],

name: 'userIdHandler',

params: undefined,

path: undefined,

keys: [],

regexp: /^\/?$/i,

method: 'get' } ],

methods: { get: true } }

**req.secure**

A Boolean that is true if a TLS connection is established. Equivalent to:

'https' == req.protocol;

**req.signedCookies**

When using [cookie-parser](https://www.npmjs.com/package/cookie-parser) middleware, this property contains signed cookies sent by the request, unsigned and ready for use. Signed cookies reside in a different object to show developer intent; otherwise, a malicious attack could be placed on req.cookie values (which are easy to spoof). Note that signing a cookie does not make it “hidden” or encrypted; but simply prevents tampering (because the secret used to sign is private). If no signed cookies are sent, the property defaults to {}.

// Cookie: user=tobi.CP7AWaXDfAKIRfH49dQzKJx7sKzzSoPq7/AcBBRVwlI3

req.signedCookies.user

// => "tobi"

For more information, issues, or concerns, see [cookie-parser](https://github.com/expressjs/cookie-parser).

**req.stale**

Indicates whether the request is “stale,” and is the opposite of req.fresh. For more information, see [req.fresh](http://www.expressjs.com.cn/4x/api.html#req.fresh).

req.stale

// => true

**req.subdomains**

An array of subdomains in the domain name of the request.

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

req.subdomains

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

**req.xhr**

A Boolean value that is true if the request’s “X-Requested-With” header field is “XMLHttpRequest”, indicating that the request was issued by a client library such as jQuery.

req.xhr

// => true

**Methods**

**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 undefined (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');

// => undefined

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

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

// => "json"

For more information, or if you have issues or concerns, see [accepts](https://github.com/expressjs/accepts).

**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.

For more information, or if you have issues or concerns, see [accepts](https://github.com/expressjs/accepts).

**req.acceptsEncodings(encoding [, ...])**

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.

For more information, or if you have issues or concerns, see [accepts](https://github.com/expressjs/accepts).

**req.acceptsLanguages(lang [, ...])**

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.

For more information, or if you have issues or concerns, see [accepts](https://github.com/expressjs/accepts).

**req.get(field)**

Returns the specified HTTP request header field (case-insensitive match). The Referrer and Referer fields are interchangeable.

req.get('Content-Type');

// => "text/plain"

req.get('content-type');

// => "text/plain"

req.get('Something');

// => undefined

Aliased as req.header(field).

**req.is(type)**

Returns true if the incoming request’s “Content-Type” HTTP header field matches the MIME type specified by the type parameter. Returns falseotherwise.

// With Content-Type: text/html; charset=utf-8

req.is('html');

req.is('text/html');

req.is('text/\*');

// => true

// When Content-Type is application/json

req.is('json');

req.is('application/json');

req.is('application/\*');

// => true

req.is('html');

// => false

For more information, or if you have issues or concerns, see [type-is](https://github.com/expressjs/type-is).

**req.param(name [, defaultValue])**

Deprecated. Use either req.params, req.body or req.query, as applicable.

Return the value of param name when present.

// ?name=tobi

req.param('name')

// => "tobi"

// POST name=tobi

req.param('name')

// => "tobi"

// /user/tobi for /user/:name

req.param('name')

// => "tobi"

Lookup is performed in the following order:

* req.params
* req.body
* req.query

Optionally, you can specify defaultValue to set a default value if the parameter is not found in any of the request objects.

Direct access to req.body, req.params, and req.query should be favoured for clarity - unless you truly accept input from each object.

Body-parsing middleware must be loaded for req.param() to work predictably. Refer [req.body](http://www.expressjs.com.cn/4x/api.html#req.body) for details.

**Response**

The res object represents the HTTP response that an Express app sends when it gets an HTTP request.

In this documentation and by convention, the object is always referred to as res (and the HTTP request is req) but its actual name is determined by the parameters to the callback function in which you’re working.

For example:

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

res.send('user ' + req.params.id);

});

But you could just as well have:

app.get('/user/:id', function(request, response){

response.send('user ' + request.params.id);

});

**Properties**

**res.app**

This property holds a reference to the instance of the express application that is using the middleware.

res.app is identical to the [req.app](http://www.expressjs.com.cn/4x/api.html#req.app) property in the request object.

**res.headersSent**

Boolean property that indicates if the app sent HTTP headers for the response.

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

console.log(res.headersSent); // false

res.send('OK');

console.log(res.headersSent); // true

})

**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](http://www.expressjs.com.cn/4x/api.html#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**

**res.append(field [, value])**

res.append() is supported by Express v4.11.0+

Appends the specified value to the HTTP response header field. If the header is not already set, it creates the header with the specified value. The value parameter can be a string or an array.

Note: calling res.set() after res.append() will reset the previously-set header value.

res.append('Link', ['<http://localhost/>', '<http://localhost:3000/>']);

res.append('Set-Cookie', 'foo=bar; Path=/; HttpOnly');

res.append('Warning', '199 Miscellaneous warning');

**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

**res.cookie(name, value [, options])**

Sets cookie name to value. The value parameter may be a string or object converted to JSON.

The options parameter is an object that can have the following properties.

| **Property** | **Type** | **Description** |
| --- | --- | --- |
| domain | String | Domain name for the cookie. Defaults to the domain name of the app. |
| expires | Date | Expiry date of the cookie in GMT. If not specified or set to 0, creates a session cookie. |
| httpOnly | Boolean | Flags the cookie to be accessible only by the web server. |
| maxAge | String | Convenient option for setting the expiry time relative to the current time in milliseconds. |
| path | String | Path for the cookie. Defaults to “/”. |
| secure | Boolean | Marks the cookie to be used with HTTPS only. |
| signed | Boolean | Indicates if the cookie should be signed. |

All res.cookie() does is set the HTTP Set-Cookie header with the options provided. Any option not specified defaults to the value stated in [RFC 6265](http://tools.ietf.org/html/rfc6265).

For example:

res.cookie('name', 'tobi', { domain: '.example.com', path: '/admin', secure: true });

res.cookie('rememberme', '1', { expires: new Date(Date.now() + 900000), httpOnly: true });

The maxAge option is a convenience option for setting “expires” relative to the current time in milliseconds. The following is equivalent to the second example above.

res.cookie('rememberme', '1', { maxAge: 900000, httpOnly: true })

You can pass an object as the value parameter; it is then serialized as JSON and parsed by bodyParser() middleware.

res.cookie('cart', { items: [1,2,3] });

res.cookie('cart', { items: [1,2,3] }, { maxAge: 900000 });

When using [cookie-parser](https://www.npmjs.com/package/cookie-parser) middleware, this method also supports signed cookies. Simply include the signed option set to true. Then res.cookie()will use the secret passed to cookieParser(secret) to sign the value.

res.cookie('name', 'tobi', { signed: true });

Later you may access this value through the [req.signedCookie](http://www.expressjs.com.cn/4x/api.html#req.signedCookies) object.

**res.clearCookie(name [, options])**

Clears the cookie specified by name. For details about the options object, see [res.cookie()](http://www.expressjs.com.cn/4x/api.html#res.cookie).

res.cookie('name', 'tobi', { path: '/admin' });

res.clearCookie('name', { path: '/admin' });

**res.download(path [, filename] [, fn])**

Transfers the file at path as an “attachment”. Typically, browsers will prompt the user for download. By default, the Content-Disposition header “filename=” parameter is path (this typically appears in the brower dialog). Override this default with the filename parameter.

When an error ocurrs or transfer is complete, the method calls the optional callback function fn. This method uses [res.sendFile()](http://www.expressjs.com.cn/4x/api.html#res.sendFile) to transfer the file.

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.

}

});

**res.end([data] [, encoding])**

Ends the response process. This method actually comes from Node core, specifically the [response.end() method of http.ServerResponse](https://nodejs.org/api/http.html#http_response_end_data_encoding_callback).

Use to quickly end the response without any data. If you need to respond with data, instead use methods such as [res.send()](http://www.expressjs.com.cn/4x/api.html#res.send) and [res.json()](http://www.expressjs.com.cn/4x/api.html#res.json).

res.end();

res.status(404).end();

**res.format(object)**

Performs content-negotiation on the Accept HTTP header on the request object, when present. It uses [req.accepts()](http://www.expressjs.com.cn/4x/api.html#req.accepts) to select a handler for the request, based on the acceptable types ordered by their quality values. If the header is not specified, the first callback is invoked. When no match is found, the server responds with 406 “Not Acceptable”, or invokes the default callback.

The Content-Type response header is set when a callback is selected. However, you may alter this within the callback using methods such as res.set() or res.type().

The following example would respond with { "message": "hey" } when the Accept header field is set to “application/json” or “\*/json” (however if it is “\*/\*”, then the response will be “hey”).

res.format({

'text/plain': function(){

res.send('hey');

},

'text/html': function(){

res.send('<p>hey</p>');

},

'application/json': function(){

res.send({ message: 'hey' });

},

'default': function() {

// log the request and respond with 406

res.status(406).send('Not Acceptable');

}

});

In addition to canonicalized MIME types, you may also use extension names mapped to these types for a slightly less verbose implementation:

res.format({

text: function(){

res.send('hey');

},

html: function(){

res.send('<p>hey</p>');

},

json: function(){

res.send({ message: 'hey' });

}

});

**res.get(field)**

Returns the HTTP response header specified by field. The match is case-insensitive.

res.get('Content-Type');

// => "text/plain"

**res.json([body])**

Sends a JSON response. This method is identical to res.send() with an object or array as the parameter. However, you can use it to convert other values to JSON, such as null, and undefined. (although these are technically not valid JSON).

res.json(null)

res.json({ user: 'tobi' })

res.status(500).json({ error: 'message' })

**res.jsonp([body])**

Sends a JSON response with JSONP support. This method is identical to res.json(), except that it opts-in to JSONP callback support.

res.jsonp(null)

// => null

res.jsonp({ user: 'tobi' })

// => { "user": "tobi" }

res.status(500).jsonp({ error: 'message' })

// => { "error": "message" }

By default, the JSONP callback name is simply callback. Override this with the [jsonp callback name](http://www.expressjs.com.cn/4x/api.html#app.settings.table) setting.

The following are some examples of JSONP responses using the same code:

// ?callback=foo

res.jsonp({ user: 'tobi' })

// => foo({ "user": "tobi" })

app.set('jsonp callback name', 'cb');

// ?cb=foo

res.status(500).jsonp({ error: 'message' })

// => foo({ "error": "message" })

**res.links(links)**

Joins the links provided as properties of the parameter to populate the response’s Link HTTP header field.

res.links({

next: 'http://api.example.com/users?page=2',

last: 'http://api.example.com/users?page=5'

});

yields:

Link: <http://api.example.com/users?page=2>; rel="next",

<http://api.example.com/users?page=5>; rel="last"

**res.location(path)**

Sets the response Location HTTP header to the specified path parameter.

res.location('/foo/bar');

res.location('http://example.com');

res.location('back');

A path value of “back” has a special meaning, it refers to the URL specified in the Referer header of the request. If the Referer header was not specified, it refers to “/”.

Express passes the specified URL string as-is to the browser in the Location header, without any validation or manipulation, except in case of back.

Browsers take the responsibility of deriving the intended URL from the current URL or the referring URL, and the URL specified in the Location header; and redirect the user accordingly.

**res.redirect([status,] path)**

Redirects to the URL dervied from the specified path, with specified [HTTP status code](http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html) status. If you don’t specify status, the status code defaults to “302 “Found”.

res.redirect('/foo/bar');

res.redirect('http://example.com');

res.redirect(301, 'http://example.com');

res.redirect('../login');

Redirects can be a fully-qualified URL for redirecting to a different site:

res.redirect('http://google.com');

Redirects can be relative to the root of the host name. For example, if the application is on http://example.com/admin/post/new, the following would redirect to the URL http://example.com/admin:

res.redirect('/admin');

Redirects can be relative to the current URL. For example, from http://example.com/blog/admin/ (notice the trailing slash), the following would redirect to the URL http://example.com/blog/admin/post/new.

res.redirect('post/new');

Redirecting to post/new from http://example.com/blog/admin (no trailing slash), will redirect to http://example.com/blog/post/new.

If you found the above behavior confusing, think of path segments as directories (with trailing slashes) and files, it will start to make sense.

Path-relative redirects are also possible. If you were on http://example.com/admin/post/new, the following would redirect to http//example.com/admin/post:

res.redirect('..');

A back redirection redirects the request back to the [referer](http://en.wikipedia.org/wiki/HTTP_referer), defaulting to / when the referer is missing.

res.redirect('back');

**res.render(view [, locals] [, callback])**

Renders a view and sends the rendered HTML string to the client. Optional parameters:

* locals, an object whose properties define local variables for the view.
* callback, a callback function. If provided, the method returns both the possible error and rendered string, but does not perform an automated response. When an error occurs, the method invokes next(err) internally.

The local variable cache enables view caching. Set it to true, to cache the view during development; view caching is enabled in production by default.

// 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) {

// ...

});

**res.send([body])**

Sends the HTTP response.

The body parameter can be a Buffer object, a String, an object, or an Array. For example:

res.send(new Buffer('whoop'));

res.send({ some: 'json' });

res.send('<p>some html</p>');

res.status(404).send('Sorry, we cannot find that!');

res.status(500).send({ error: 'something blew up' });

This method performs many useful tasks for simple non-streaming responses: For example, it automatically assigns the Content-Length HTTP response header field (unless previously defined) and provides automatic HEAD and HTTP cache freshness support.

When the parameter is a Buffer object, the method sets the Content-Type response header field to “application/octet-stream”, unless previously defined as shown below:

res.set('Content-Type', 'text/html');

res.send(new Buffer('<p>some html</p>'));

When the parameter is a String, the method sets the Content-Type to “text/html”:

res.send('<p>some html</p>');

When the parameter is an Array or Object, Express responds with the JSON representation:

res.send({ user: 'tobi' });

res.send([1,2,3]);

**res.sendFile(path [, options] [, fn])**

res.sendFile() is supported from Express v4.8.0 onwards

Transfers the file at the given path. Sets the Content-Type response HTTP header field based on the filename’s extension. Unless the root option is set in the options object, path must be an absolute path of the file.

The details of the options object is listed in the following table.

| **Property** | **Description** | **Default** | **Availability** |
| --- | --- | --- | --- |
| maxAge | Sets the max-age property of the Cache-Control header in milliseconds or a string in [ms format](https://www.npmjs.org/package/ms) | 0 |  |
| root | Root directory for relative filenames. |  |  |
| lastModified | Sets the Last-Modified header to the last modified date of the file on the OS. Set false to disable it. | Enabled | 4.9.0+ |
| headers | Object containing HTTP headers to serve with the file. |  |  |
| dotfiles | Option for serving dotfiles. Possible values are “allow”, “deny”, “ignore”. | “ignore” |  |

The method invokes the callback function fn(err) when the transfer is complete or when an error occurs. If the callback function is specified and an error occurs, the callback function must explicitly handle the response process either by ending the request-response cycle, or by passing control to the next route.

Here is an example of using res.sendFile with all its arguments.

app.get('/file/:name', function (req, res, next) {

var options = {

root: \_\_dirname + '/public/',

dotfiles: 'deny',

headers: {

'x-timestamp': Date.now(),

'x-sent': true

}

};

var fileName = req.params.name;

res.sendFile(fileName, options, function (err) {

if (err) {

console.log(err);

res.status(err.status).end();

}

else {

console.log('Sent:', fileName);

}

});

})

res.sendFile provides fine-grained support for file serving as illustrated in the following example:

app.get('/user/:uid/photos/:file', function(req, res){

var uid = req.params.uid

, file = req.params.file;

req.user.mayViewFilesFrom(uid, function(yes){

if (yes) {

res.sendFile('/uploads/' + uid + '/' + file);

} else {

res.status(403).send('Sorry! you cant see that.');

}

});

});

For more information, or if you have issues or concerns, see [send](https://github.com/pillarjs/send).

**res.sendStatus(statusCode)**

Set the response HTTP status code to statusCode and send its string representation as the response body.

res.sendStatus(200); // equivalent to res.status(200).send('OK')

res.sendStatus(403); // equivalent to res.status(403).send('Forbidden')

res.sendStatus(404); // equivalent to res.status(404).send('Not Found')

res.sendStatus(500); // equivalent to res.status(500).send('Internal Server Error')

If an unsupported status code is specified, the HTTP status is still set to statusCode and the string version of the code is sent as the response body.

res.sendStatus(2000); // equivalent to res.status(2000).send('2000')

[More about HTTP Status Codes](http://en.wikipedia.org/wiki/List_of_HTTP_status_codes)

**res.set(field [, value])**

Sets the response’s HTTP header field to value. To set multiple fields at once, pass an object as the parameter.

res.set('Content-Type', 'text/plain');

res.set({

'Content-Type': 'text/plain',

'Content-Length': '123',

'ETag': '12345'

})

Aliased as res.header(field [, value]).

**res.status(code)**

Use this method to set the HTTP status for the response. It is a chainable alias of Node’s [response.statusCode](http://nodejs.org/api/http.html#http_response_statuscode).

res.status(403).end();

res.status(400).send('Bad Request');

res.status(404).sendFile('/absolute/path/to/404.png');

**res.type(type)**

Sets the Content-Type HTTP header to the MIME type as determined by [mime.lookup()](https://github.com/broofa/node-mime#mimelookuppath) for the specified type. If type contains the “/” character, then it sets the Content-Type to type.

res.type('.html'); // => 'text/html'

res.type('html'); // => 'text/html'

res.type('json'); // => 'application/json'

res.type('application/json'); // => 'application/json'

res.type('png'); // => image/png:

**res.vary(field)**

Adds the field to the Vary response header, if it is not there already.

res.vary('User-Agent').render('docs');

**Router**

A router object is an isolated instance of middleware and routes. You can think of it as a “mini-application,” capable only of performing middleware and routing functions. Every Express application has a built-in app router.

A router behaves like middleware itself, so you can use it as an argument to [app.use()](http://www.expressjs.com.cn/4x/api.html#app.use) or as the argument to another router’s [use()](http://www.expressjs.com.cn/4x/api.html#router.use) method.

The top-level express object has a Router() function that creates a new router object.

**Router([options])**

Create a new router as follows:

var router = express.Router([options]);

The optional options parameter specifies the behavior of the router.

| **Property** | **Description** | **Default** | **Availability** |
| --- | --- | --- | --- |
| caseSensitive | Enable case sensitivity. | Disabled by default, treating “/Foo” and “/foo” as the same. |  |
| mergeParams | Preserve the req.params values from the parent router. If the parent and the child have conflicting param names, the child’s value take precedence. | false | 4.5.0+ |
| strict | Enable strict routing. | Disabled by default, “/foo” and “/foo/” are treated the same by the router. |  |

You can add middleware and HTTP method routes (such as get, put, post, and so on) to router just like an application.

// invoked for any requests passed to this router

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

// .. some logic here .. like any other middleware

next();

});

// will handle any request that ends in /events

// depends on where the router is "use()'d"

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

// ..

});

You can then use a router for a particular root URL in this way separating your routes into files or even mini-apps.

// only requests to /calendar/\* will be sent to our "router"

app.use('/calendar', router);

**Methods**

**router.all(path, [callback, ...] callback)**

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

**router.METHOD(path, [callback, ...] callback)**

The router.METHOD() methods provide the routing functionality in Express, where METHOD is one of the HTTP methods, such as GET, PUT, POST, and so on, in lowercase. Thus, the actual methods are router.get(), router.post(), router.put(), and so on.

You can provide multiple callbacks, and all are treated equally, and behave just like middleware, except that these callbacks may invoke next('route') to bypass the remaining route callback(s). You can use this mechanism to perform pre-conditions on a route then pass control to subsequent routes when there is no reason to proceed with the route matched.

The following snippet illustrates the most simple route definition possible. Express translates the path strings to regular expressions, used internally to match incoming requests. Query strings are ***not*** considered when performing these matches, for example “GET /” would match the following route, as would “GET /?name=tobi”.

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

res.send('hello world');

});

You can also use regular expressions—useful if you have very specific constraints, for example the following would match “GET /commits/71dbb9c” as well as “GET /commits/71dbb9c..4c084f9”.

router.get(/^\/commits\/(\w+)(?:\.\.(\w+))?$/, function(req, res){

var from = req.params[0];

var to = req.params[1] || 'HEAD';

res.send('commit range ' + from + '..' + to);

});

**router.param([name,] callback)**

Add callback triggers to route parameters, where name is the name of the parameter or an array of them, and function is the callback function. The parameters of the callback function are the request object, the response object, the next middleware, and the value 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.

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'));

}

});

});

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 router will be triggered only by route parameters defined on router routes.

A param callback will be called only once in a request-response cycle, even if the parameter is matched in multiple routes, as shown in the following examples.

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

console.log('CALLED ONLY ONCE');

next();

})

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

console.log('although this matches');

next();

});

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

console.log('and this matches too');

res.end();

});

On GET /user/42, the following is printed:

CALLED ONLY ONCE

although this matches

and this matches too

router.param(['id', 'page'], function (req, res, next, value) {

console.log('CALLED ONLY ONCE with', value);

next();

})

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

console.log('although this matches');

next();

});

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

console.log('and this matches too');

res.end();

});

On GET /user/42/3, the following is printed:

CALLED ONLY ONCE with 42

CALLED ONLY ONCE with 3

although this matches

and this matches too

The following section describes router.param(callback), which is deprecated as of v4.11.0.

The behavior of the router.param(name, callback) method can be altered entirely by passing only a function to router.param(). This function is a custom implementation of how router.param(name, callback) should behave - it accepts two parameters and must return a middleware.

The first parameter of this function is the name of the URL parameter that should be captured, the second parameter can be any JavaScript object which might be used for returning the middleware implementation.

The middleware returned by the function decides the behavior of what happens when a URL parameter is captured.

In this example, the router.param(name, callback) signature is modified to router.param(name, accessId). Instead of accepting a name and a callback, router.param() will now accept a name and a number.

var express = require('express');

var app = express();

var router = express.Router();

// customizing the behavior of router.param()

router.param(function(param, option) {

return function (req, res, next, val) {

if (val == option) {

next();

}

else {

res.sendStatus(403);

}

}

});

// using the customized router.param()

router.param('id', 1337);

// route to trigger the capture

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

res.send('OK');

})

app.use(router);

app.listen(3000, function () {

console.log('Ready');

})

In this example, the router.param(name, callback) signature remains the same, but instead of a middleware callback, a custom data type checking function has been defined to validate the data type of the user id.

router.param(function(param, validator) {

return function (req, res, next, val) {

if (validator(val)) {

next();

}

else {

res.sendStatus(403);

}

}

})

router.param('id', function (candidate) {

return !isNaN(parseFloat(candidate)) && isFinite(candidate);

});

**router.route(path)**

Returns an instance of a single route which you can then use to handle HTTP verbs with optional middleware. Use router.route() to avoid duplicate route naming and thus typo errors.

Building on the router.param() example above, the following code shows how to use router.route() to specify various HTTP method handlers.

var 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'));

})

This approach re-uses the single ‘/users/:user\_id’ path and add handlers for various HTTP methods.

**router.use([path], [function, ...] function)**

Uses the given middleware function, with optional mount path path, that defaults to “/”.

This method is similar to [app.use()](http://www.expressjs.com.cn/4x/api.html#app.use). A simple example and usecase is described below. See [app.use()](http://www.expressjs.com.cn/4x/api.html#app.use) for more information.

Middleware is like a plumbing pipe, requests start at the first middleware you define and work their way “down” the middleware stack processing for each path they match.

var express = require('express');

var app = express();

var 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);

The “mount” path is stripped and is ***not*** visible to the middleware function. The main effect of this feature is that mounted middleware may operate without code changes regardless of its “prefix” pathname.

The order in which you define middleware with router.use() is very important. They are invoked sequentially, thus the order defines middleware precedence. For example, usually a logger is the very first middleware you would use, so every request is logged.

var logger = require('morgan');

router.use(logger());

router.use(express.static(\_\_dirname + '/public'));

router.use(function(req, res){

res.send('Hello');

});

Now suppose you wanted to ignore logging requests for static files, but to continue logging routes and middleware defined after logger(). You would simply move static() above:

router.use(express.static(\_\_dirname + '/public'));

router.use(logger());

router.use(function(req, res){

res.send('Hello');

});

Another concrete example is serving files from multiple directories, giving precedence to “./public” over the others:

app.use(express.static(\_\_dirname + '/public'));

app.use(express.static(\_\_dirname + '/files'));

app.use(express.static(\_\_dirname + '/uploads'));

The router.use() method also supports named parameters so that your mount points for other routers can benefit from preloading using named parameters.

[http://www.expressjs.com.cn/images/arrow.png](http://www.expressjs.com.cn/4x/api.html)

[Github Repo](https://github.com/strongloop/express)

Express 项目由 [StrongLoop](http://strongloop.com/) 赞助。

[Edit this page on GitHub](https://github.com/strongloop/expressjs.com/tree/gh-pages/_includes/api/en/4x)

Copyright © StrongLoop, Inc., and other expressjs.com contributors.

[Bootstrap中文网](http://www.bootcss.com/) [Grunt中文网](http://www.gruntjs.net/) [Gulp中文网](http://www.gulpjs.com.cn/) [BootCDN](http://www.bootcdn.cn/) [Node.js 中文文档](http://www.nodeapp.cn/)

[冀ICP备15008021号](http://www.miibeian.gov.cn/)|© Express 中文网 2015