

# EXPRESS Docs - English



## Table of contents

• Advanced - Best practice performance	4
• Advanced - Best practice security	15
• Advanced - Developing template engines	21
• Advanced - Healthcheck graceful shutdown	22
• Advanced - Pm	26
• Advanced - Security updates	27
• En - Api	29
• Changelog - 4x	30
• Guide - Behind proxies	37
• Guide - Database integration	39
• Guide - Debugging	47
• Guide - Error handling	50
• Guide - Migrating 4	62
• Guide - Migrating 5	64
• Guide - Overriding express api	68
• Guide - Routing	70
• Guide - Using middleware	77
• Guide - Using template engines	82
• Guide - Writing middleware	84
• Resources - Community	88
• Resources - Companies using express	90
• Resources - Contributing	91
• Resources - Frameworks	96
• Resources - Glossary	97
• Resources - Learning	99
• Resources - Middleware	101
• Resources - Open source using express	103
• Resources - Template engines	104
• Resources - Utils	105
• Starter - Basic routing	106
• Starter - Examples	108
• Starter - Faq	109
• Starter - Generator	111
• Starter - Hello world	113

• Starter - Installing	114
• Starter - Static files	115

# Production best practices: performance and reliability

## Overview

This article discusses performance and reliability best practices for Express applications deployed to production.

This topic clearly falls into the "devops" world, spanning both traditional development and operations. Accordingly, the information is divided into two parts:

- Things to do in your code (the dev part):
  - [Use gzip compression](#)
  - [Don't use synchronous functions](#)
  - [Do logging correctly](#)
  - [Handle exceptions properly](#)
- Things to do in your environment / setup (the ops part):
  - [Set NODE\\_ENV to "production"](#)
  - [Ensure your app automatically restarts](#)
  - [Run your app in a cluster](#)
  - [Cache request results](#)
  - [Use a load balancer](#)
  - [Use a reverse proxy](#)

## Things to do in your code

Here are some things you can do in your code to improve your application's performance:

- [Use gzip compression](#)
- [Don't use synchronous functions](#)
- [Do logging correctly](#)
- [Handle exceptions properly](#)

## Use gzip compression

Gzip compressing can greatly decrease the size of the response body and hence increase the speed of a web app. Use the [compression](#) middleware for gzip compression in your Express app. For example:

```
const compression = require('compression')
const express = require('express')
const app = express()
app.use(compression())
```

For a high-traffic website in production, the best way to put compression in place is to implement it at a reverse proxy level (see [Use a reverse proxy](#)). In that case, you do not need to use compression middleware. For details on enabling gzip compression in Nginx, see [Module ngx\\_http\\_gzip\\_module](#) in the Nginx documentation.

## Don't use synchronous functions

Synchronous functions and methods tie up the executing process until they return. A single call to a synchronous function might return in a few microseconds or milliseconds, however in high-traffic websites, these calls add up and reduce the performance of the app. Avoid their use in production.

Although Node and many modules provide synchronous and asynchronous versions of their functions, always use the asynchronous version in production. The only time when a synchronous function can be justified is upon initial startup.

If you are using Node.js 4.0+ or io.js 2.1.0+, you can use the `--trace-sync-io` command-line flag to print a warning and a stack trace whenever your application uses a synchronous API. Of course, you wouldn't want to use this in production, but rather to ensure that your code is ready for production. See the [node command-line options documentation](#) for more information.

## Do logging correctly

In general, there are two reasons for logging from your app: For debugging and for logging app activity (essentially, everything else). Using `console.log()` or `console.error()` to print log messages to the terminal is common practice in development. But [these functions are synchronous](#) when the destination is a terminal or a file, so they are not suitable for production, unless you pipe the output to another program.

### For debugging

If you're logging for purposes of debugging, then instead of using `console.log()`, use a special debugging module like [debug](#). This module enables you to use the DEBUG environment variable to control what debug messages are sent to `console.error()`, if any. To keep your app purely asynchronous, you'd still want to pipe `console.error()` to another program. But then, you're not really going to debug in production, are you?

### For app activity

If you're logging app activity (for example, tracking traffic or API calls), instead of using `console.log()`, use a logging library like [Winston](#) or [Bunyan](#). For a detailed comparison of these two libraries, see the StrongLoop blog post [Comparing Winston and Bunyan Node.js Logging](#).

## Handle exceptions properly

Node apps crash when they encounter an uncaught exception. Not handling exceptions and taking appropriate actions will make your Express app crash and go offline. If you follow the advice in [Ensure your app automatically restarts](#) below, then your app will recover from a crash. Fortunately, Express apps typically have a short startup time. Nevertheless, you want to avoid crashing in the first place, and to do that, you need to handle exceptions properly.

To ensure you handle all exceptions, use the following techniques:

- [Use try-catch](#)
- [Use promises](#)

Before diving into these topics, you should have a basic understanding of Node/Express error handling: using error-first callbacks, and propagating errors in middleware. Node uses an "error-first callback" convention for returning errors from asynchronous functions, where the first parameter to the callback function is the error object, followed by result data in succeeding parameters. To indicate no error, pass null as the first parameter. The callback function must correspondingly follow the error-first callback convention to meaningfully handle the error. And in Express, the best practice is to use the `next()` function to propagate errors through the middleware chain.

For more on the fundamentals of error handling, see:

- [Error Handling in Node.js](#)
- [Building Robust Node Applications: Error Handling](#) (StrongLoop blog)

### What not to do

One thing you should *not* do is to listen for the `uncaughtException` event, emitted when an exception bubbles all the way back to the event loop. Adding an event listener for `uncaughtException` will change the default behavior of the process that is encountering an exception; the process will continue to run despite the exception. This might sound like a good way of preventing your app from crashing, but continuing to run the app after an uncaught exception is a dangerous practice and is not recommended, because the state of the process becomes unreliable and unpredictable.

Additionally, using `uncaughtException` is officially recognized as *crude*. So listening for `uncaughtException` is just a bad idea. This is why we recommend things like multiple processes and supervisors: crashing and restarting is often the most reliable way to recover from an error.

We also don't recommend using `domains`. It generally doesn't solve the problem and is a deprecated module.

### Use try-catch

Try-catch is a JavaScript language construct that you can use to catch exceptions in synchronous code. Use try-catch, for example, to handle JSON parsing errors as shown below.

Use a tool such as [JSHint](#) or [JSLint](#) to help you find implicit exceptions like [reference errors on undefined variables](#).

Here is an example of using try-catch to handle a potential process-crashing exception. This middleware function accepts a query field parameter named "params" that is a JSON object.

```
app.get('/search', (req, res) => {
  // Simulating async operation
  setImmediate(() => {
    const jsonStr = req.query.params
    try {
      const jsonObj = JSON.parse(jsonStr)
```

```

    res.send('Success')
  } catch (e) {
    res.status(400).send('Invalid JSON string')
  }
}
})

```

However, try-catch works only for synchronous code. Because the Node platform is primarily asynchronous (particularly in a production environment), try-catch won't catch a lot of exceptions.

### Use promises

Promises will handle any exceptions (both explicit and implicit) in asynchronous code blocks that use `then()`. Just add `.catch(next)` to the end of promise chains. For example:

```

app.get('/', (req, res, next) => {
  // do some sync stuff
  queryDb()
    .then((data) => makeCsv(data)) // handle data
    .then((csv) => { /* handle csv */ })
    .catch(next)
})

app.use((err, req, res, next) => {
  // handle error
})

```

Now all errors asynchronous and synchronous get propagated to the error middleware.

However, there are two caveats:

1. All your asynchronous code must return promises (except emitters). If a particular library does not return promises, convert the base object by using a helper function like [Bluebird.promisifyAll\(\)](#).
2. Event emitters (like streams) can still cause uncaught exceptions. So make sure you are handling the error event properly; for example:

```

const wrap = fn => (...args) => fn(...args).catch(args[2])

app.get('/', wrap(async (req, res, next) => {
  const company = await getCompanyById(req.query.id)
  const stream = getLogoStreamById(company.id)
  stream.on('error', next).pipe(res)
}))

```

The `wrap()` function is a wrapper that catches rejected promises and calls `next()` with the error as the first argument. For details, see [Asynchronous Error Handling in Express with Promises, Generators and ES7](#).

For more information about error-handling by using promises, see [Promises in Node.js with Q – An Alternative to Callbacks](#).

## Things to do in your environment / setup

Here are some things you can do in your system environment to improve your app's performance:

- [Set NODE\\_ENV to "production"](#)
- [Ensure your app automatically restarts](#)
- [Run your app in a cluster](#)
- [Cache request results](#)
- [Use a load balancer](#)
- [Use a reverse proxy](#)

## Set NODE\_ENV to "production"

The `NODE_ENV` environment variable specifies the environment in which an application is running (usually, development or production). One of the simplest things you can do to improve performance is to set `NODE_ENV` to "production."

Setting `NODE_ENV` to "production" makes Express:

- Cache view templates.
- Cache CSS files generated from CSS extensions.
- Generate less verbose error messages.

[Tests indicate](#) that just doing this can improve app performance by a factor of three!

If you need to write environment-specific code, you can check the value of `NODE_ENV` with `process.env.NODE_ENV`. Be aware that checking the value of any environment variable incurs a performance penalty, and so should be done sparingly.

In development, you typically set environment variables in your interactive shell, for example by using `export` or your `.bash_profile` file. But in general you shouldn't do that on a production server; instead, use your OS's init system (systemd or Upstart). The next section provides more details about using your init system in general, but setting `NODE_ENV` is so important for performance (and easy to do), that it's highlighted here.

With Upstart, use the `env` keyword in your job file. For example:

```
# /etc/init/env.conf
env NODE_ENV=production
```

For more information, see the [Upstart Intro, Cookbook and Best Practices](#).

With systemd, use the `Environment` directive in your unit file. For example:

```
# /etc/systemd/system/myservice.service
Environment=NODE_ENV=production
```

For more information, see [Using Environment Variables In systemd Units](#).



## Ensure your app automatically restarts

In production, you don't want your application to be offline, ever. This means you need to make sure it restarts both if the app crashes and if the server itself crashes. Although you hope that neither of those events occurs, realistically you must account for both eventualities by:

- Using a process manager to restart the app (and Node) when it crashes.
- Using the init system provided by your OS to restart the process manager when the OS crashes. It's also possible to use the init system without a process manager.

Node applications crash if they encounter an uncaught exception. The foremost thing you need to do is to ensure your app is well-tested and handles all exceptions (see [handle exceptions properly](#) for details). But as a fail-safe, put a mechanism in place to ensure that if and when your app crashes, it will automatically restart.

### Use a process manager

In development, you started your app simply from the command line with `node server.js` or something similar. But doing this in production is a recipe for disaster. If the app crashes, it will be offline until you restart it. To ensure your app restarts if it crashes, use a process manager. A process manager is a "container" for applications that facilitates deployment, provides high availability, and enables you to manage the application at runtime.

In addition to restarting your app when it crashes, a process manager can enable you to:

- Gain insights into runtime performance and resource consumption.
- Modify settings dynamically to improve performance.
- Control clustering (StrongLoop PM and pm2).

The most popular process managers for Node are as follows:

- [StrongLoop Process Manager](#)
- [PM2](#)
- [Forever](#)

For a feature-by-feature comparison of the three process managers, see <http://strong-pm.io/compare/>. For a more detailed introduction to all three, see [Process managers for Express apps](#).

Using any of these process managers will suffice to keep your application up, even if it does crash from time to time.

However, StrongLoop PM has lots of features that specifically target production deployment. You can use it and the related StrongLoop tools to:

- Build and package your app locally, then deploy it securely to your production system.
- Automatically restart your app if it crashes for any reason.
- Manage your clusters remotely.
- View CPU profiles and heap snapshots to optimize performance and diagnose memory leaks.
- View performance metrics for your application.

- Easily scale to multiple hosts with integrated control for Nginx load balancer.

As explained below, when you install StrongLoop PM as an operating system service using your init system, it will automatically restart when the system restarts. Thus, it will keep your application processes and clusters alive forever.

## Use an init system

The next layer of reliability is to ensure that your app restarts when the server restarts. Systems can still go down for a variety of reasons. To ensure that your app restarts if the server crashes, use the init system built into your OS. The two main init systems in use today are [systemd](#) and [Upstart](#).

There are two ways to use init systems with your Express app:

- Run your app in a process manager, and install the process manager as a service with the init system. The process manager will restart your app when the app crashes, and the init system will restart the process manager when the OS restarts. This is the recommended approach.
- Run your app (and Node) directly with the init system. This is somewhat simpler, but you don't get the additional advantages of using a process manager.

## Systemd

Systemd is a Linux system and service manager. Most major Linux distributions have adopted systemd as their default init system.

A systemd service configuration file is called a *unit file*, with a filename ending in `.service`. Here's an example unit file to manage a Node app directly. Replace the values enclosed in `<angle brackets>` for your system and app:

```
[Unit]
Description=<Awesome Express App>

[Service]
Type=simple
ExecStart=/usr/local/bin/node </projects/myapp/index.js>
WorkingDirectory=</projects/myapp>

User=nobody
Group=nogroup

# Environment variables:
Environment=NODE_ENV=production

# Allow many incoming connections
LimitNOFILE=infinity

# Allow core dumps for debugging
LimitCORE=infinity

StandardInput=null
StandardOutput=syslog
StandardError=syslog
Restart=always
```

```
[Install]
WantedBy=multi-user.target
```

For more information on systemd, see the [systemd reference \(man page\)](#).

### StrongLoop PM as a systemd service

You can easily install StrongLoop Process Manager as a systemd service. After you do, when the server restarts, it will automatically restart StrongLoop PM, which will then restart all the apps it is managing.

To install StrongLoop PM as a systemd service:

```
$ sudo sl-pm-install --systemd
```

Then start the service with:

```
$ sudo /usr/bin/systemctl start strong-pm
```

For more information, see [Setting up a production host \(StrongLoop documentation\)](#).

### Upstart

Upstart is a system tool available on many Linux distributions for starting tasks and services during system startup, stopping them during shutdown, and supervising them. You can configure your Express app or process manager as a service and then Upstart will automatically restart it when it crashes.

An Upstart service is defined in a job configuration file (also called a "job") with filename ending in `.conf`. The following example shows how to create a job called "myapp" for an app named "myapp" with the main file located at `/projects/myapp/index.js`.

Create a file named `myapp.conf` at `/etc/init/` with the following content (replace the bold text with values for your system and app):

```
# When to start the process
start on runlevel [2345]

# When to stop the process
stop on runlevel [016]

# Increase file descriptor limit to be able to handle more requests
limit nofile 50000 50000

# Use production mode
env NODE_ENV=production

# Run as www-data
setuid www-data
setgid www-data

# Run from inside the app dir
chdir /projects/myapp

# The process to start
exec /usr/local/bin/node /projects/myapp/index.js

# Restart the process if it is down
```

```
respawn
```

```
# Limit restart attempt to 10 times within 10 seconds
respawn limit 10 10
```

NOTE: This script requires Upstart 1.4 or newer, supported on Ubuntu 12.04-14.10.

Since the job is configured to run when the system starts, your app will be started along with the operating system, and automatically restarted if the app crashes or the system goes down.

Apart from automatically restarting the app, Upstart enables you to use these commands:

- `start myapp` – Start the app
- `restart myapp` – Restart the app
- `stop myapp` – Stop the app.

For more information on Upstart, see [Upstart Intro, Cookbook and Best Practises](#).

### StrongLoop PM as an Upstart service

You can easily install StrongLoop Process Manager as an Upstart service. After you do, when the server restarts, it will automatically restart StrongLoop PM, which will then restart all the apps it is managing.

To install StrongLoop PM as an Upstart 1.4 service:

```
$ sudo sl-pm-install
```

Then run the service with:

```
$ sudo /sbin/initctl start strong-pm
```

NOTE: On systems that don't support Upstart 1.4, the commands are slightly different. See [Setting up a production host \(StrongLoop documentation\)](#) for more information.

## Run your app in a cluster

In a multi-core system, you can increase the performance of a Node app by many times by launching a cluster of processes. A cluster runs multiple instances of the app, ideally one instance on each CPU core, thereby distributing the load and tasks among the instances.



Balancing between application instances using the cluster API

Balancing between application instances using the cluster API

**IMPORTANT:** Since the app instances run as separate processes, they do not share the same memory space. That is, objects are local to each instance of the app. Therefore, you cannot maintain state in the application code. However, you can use an in-memory datastore like [Redis](#) to store session-related data and state. This caveat applies to essentially all forms of horizontal scaling, whether clustering with multiple processes or multiple physical servers.

In clustered apps, worker processes can crash individually without affecting the rest of the processes. Apart from performance advantages, failure isolation is another reason to run a cluster of app processes. Whenever a worker process crashes, always make sure to log the event and spawn a new process using `cluster.fork()`.

### Using Node's cluster module

Clustering is made possible with Node's [cluster module](#). This enables a master process to spawn worker processes and distribute incoming connections among the workers. However, rather than using this module directly, it's far better to use one of the many tools out there that does it for you automatically; for example [node-pm](#) or [cluster-service](#).

### Using StrongLoop PM

If you deploy your application to StrongLoop Process Manager (PM), then you can take advantage of clustering *without* modifying your application code.

When StrongLoop Process Manager (PM) runs an application, it automatically runs it in a cluster with a number of workers equal to the number of CPU cores on the system. You can manually change the number of worker processes in the cluster using the `slc` command line tool without stopping the app.

For example, assuming you've deployed your app to `prod.foo.com` and StrongLoop PM is listening on port 8701 (the default), then to set the cluster size to eight using `slc`:

```
$ slc ctl -C http://prod.foo.com:8701 set-size my-app 8
```

For more information on clustering with StrongLoop PM, see [Clustering](#) in StrongLoop documentation.

### Using PM2

If you deploy your application with PM2, then you can take advantage of clustering *without* modifying your application code. You should ensure your [application is stateless](#) first, meaning no local data is stored in the process (such as sessions, websocket connections and the like).

When running an application with PM2, you can enable **cluster mode** to run it in a cluster with a number of instances of your choosing, such as the matching the number of available CPUs on the machine. You can manually change the number of processes in the cluster using the `pm2` command line tool without stopping the app.

To enable cluster mode, start your application like so:

```
# Start 4 worker processes
$ pm2 start npm --name my-app -i 4 -- start
# Auto-detect number of available CPUs and start that many worker processes
$ pm2 start npm --name my-app -i max -- start
```

This can also be configured within a PM2 process file (`ecosystem.config.js` or similar) by setting `exec_mode` to `cluster` and `instances` to the number of workers to start.

Once running, the application can be scaled like so:

```
# Add 3 more workers
$ pm2 scale my-app +3
# Scale to a specific number of workers
$ pm2 scale my-app 2
```

For more information on clustering with PM2, see [Cluster Mode](#) in the PM2 documentation.

## Cache request results

Another strategy to improve the performance in production is to cache the result of requests, so that your app does not repeat the operation to serve the same request repeatedly.

Use a caching server like [Varnish](#) or [Nginx](#) (see also [Nginx Caching](#)) to greatly improve the speed and performance of your app.

## Use a load balancer

No matter how optimized an app is, a single instance can handle only a limited amount of load and traffic. One way to scale an app is to run multiple instances of it and distribute the traffic via a load balancer. Setting up a load balancer can improve your app's performance and speed, and enable it to scale more than is possible with a single instance.

A load balancer is usually a reverse proxy that orchestrates traffic to and from multiple application instances and servers. You can easily set up a load balancer for your app by using [Nginx](#) or [HAProxy](#).

With load balancing, you might have to ensure that requests that are associated with a particular session ID connect to the process that originated them. This is known as *session affinity*, or *sticky sessions*, and may be addressed by the suggestion above to use a data store such as Redis for session data (depending on your application). For a discussion, see [Using multiple nodes](#).

## Use a reverse proxy

A reverse proxy sits in front of a web app and performs supporting operations on the requests, apart from directing requests to the app. It can handle error pages, compression, caching, serving files, and load balancing among other things.

Handing over tasks that do not require knowledge of application state to a reverse proxy frees up Express to perform specialized application tasks. For this reason, it is recommended to run Express behind a reverse proxy like [Nginx](#) or [HAProxy](#) in production.

---

[Go to TOC](#)

# Production Best Practices: Security

## Overview

The term "*production*" refers to the stage in the software lifecycle when an application or API is generally available to its end-users or consumers. In contrast, in the "*development*" stage, you're still actively writing and testing code, and the application is not open to external access. The corresponding system environments are known as *production* and *development* environments, respectively.

Development and production environments are usually set up differently and have vastly different requirements. What's fine in development may not be acceptable in production. For example, in a development environment you may want verbose logging of errors for debugging, while the same behavior can become a security concern in a production environment. And in development, you don't need to worry about scalability, reliability, and performance, while those concerns become critical in production.

```
{% include note.html content="If you believe you have discovered a security vulnerability in Express, please see Security Policies and Procedures. " %}
```

Security best practices for Express applications in production include:

- [Don't use deprecated or vulnerable versions of Express](#)
- [Use TLS](#)
- [Use Helmet](#)
- [Use cookies securely](#)
- [Prevent brute-force attacks against authorization](#)
- [Ensure your dependencies are secure](#)
- [Avoid other known vulnerabilities](#)
- [Additional considerations](#)

## Don't use deprecated or vulnerable versions of Express

Express 2.x and 3.x are no longer maintained. Security and performance issues in these versions won't be fixed. Do not use them! If you haven't moved to version 4, follow the [migration guide](#).

Also ensure you are not using any of the vulnerable Express versions listed on the [Security updates page](#). If you are, update to one of the stable releases, preferably the latest.

## Use TLS

If your app deals with or transmits sensitive data, use [Transport Layer Security](#) (TLS) to secure the connection and the data. This technology encrypts data before it is sent from the client to the server, thus preventing some common (and easy) hacks. Although Ajax and POST requests might not be visibly obvious and seem "hidden" in browsers, their network traffic is vulnerable to [packet sniffing](#) and [man-in-the-middle attacks](#).

You may be familiar with Secure Socket Layer (SSL) encryption. [TLS is simply the next progression of SSL](#). In other words, if you were using SSL before, consider upgrading to TLS. In general, we recommend Nginx to handle TLS. For a good reference to configure TLS on Nginx (and other servers), see [Recommended Server Configurations \(Mozilla Wiki\)](#).

Also, a handy tool to get a free TLS certificate is [Let's Encrypt](#), a free, automated, and open certificate authority (CA) provided by the [Internet Security Research Group \(ISRG\)](#).

## Use Helmet

[Helmet](#) can help protect your app from some well-known web vulnerabilities by setting HTTP headers appropriately.

Helmet is a collection of several smaller middleware functions that set security-related HTTP response headers. Some examples include:

- `helmet.contentSecurityPolicy` which sets the `Content-Security-Policy` header. This helps prevent cross-site scripting attacks among many other things.
- `helmet.hsts` which sets the `Strict-Transport-Security` header. This helps enforce secure (HTTPS) connections to the server.
- `helmet.frameguard` which sets the `X-Frame-Options` header. This provides [clickjacking](#) protection.

Helmet includes several other middleware functions which you can read about [at its documentation website](#).

Install Helmet like any other module:

```
$ npm install --save helmet
```

Then to use it in your code:

```
// ...  
  
const helmet = require('helmet')  
app.use(helmet())  
  
// ...
```



## Reduce Fingerprinting

It can help to provide an extra layer of obsecrurity to reduce server fingerprinting. Though not a security issue itself, a method to improve the overall posture of a web server is to take measures to reduce the ability to fingerprint the software being used on the server. Server software can be fingerprinted by quirks in how they respond to specific requests.

By default, Express.js sends the `X-Powered-By` response header banner. This can be disabled using the `app.disable()` method:

```
app.disable('x-powered-by')
```

{% include note.html content="Disabling the `X-Powered-By` header does not prevent a sophisticated attacker from determining that an app is running Express. It may discourage a casual exploit, but there are other ways to determine an app is running Express. "%}

Express.js also sends it's own formatted 404 Not Found messages and own formatted error response messages. These can be changed by [adding your own not found handler](#) and [writing your own error handler](#):

```
// last app.use calls right before app.listen():

// custom 404
app.use((req, res, next) => {
  res.status(404).send("Sorry can't find that!")
})

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

## Use cookies securely

To ensure cookies don't open your app to exploits, don't use the default session cookie name and set cookie security options appropriately.

There are two main middleware cookie session modules:

- [express-session](#) that replaces `express.session` middleware built-in to Express 3.x.
- [cookie-session](#) that replaces `express.cookieSession` middleware built-in to Express 3.x.

The main difference between these two modules is how they save cookie session data. The [express-session](#) middleware stores session data on the server; it only saves the session ID in the cookie itself, not session data. By default, it uses in-memory storage and is not designed for a production environment. In production, you'll need to set up a scalable session-store; see the list of [compatible session stores](#).

In contrast, [cookie-session](#) middleware implements cookie-backed storage: it serializes the entire session to the cookie, rather than just a session key. Only use it when session data is relatively small and easily encoded as primitive values (rather than objects). Although browsers are supposed to support at least 4096

bytes per cookie, to ensure you don't exceed the limit, don't exceed a size of 4093 bytes per domain. Also, be aware that the cookie data will be visible to the client, so if there is any reason to keep it secure or obscure, then `express-session` may be a better choice.

## Don't use the default session cookie name

Using the default session cookie name can open your app to attacks. The security issue posed is similar to `X-Powered-By`: a potential attacker can use it to fingerprint the server and target attacks accordingly.

To avoid this problem, use generic cookie names; for example using `express-session` middleware:

```
const session = require('express-session')
app.set('trust proxy', 1) // trust first proxy
app.use(session({
  secret: 's3Cur3',
  name: 'sessionId'
}))
```

## Set cookie security options

Set the following cookie options to enhance security:

- `secure` - Ensures the browser only sends the cookie over HTTPS.
- `httpOnly` - Ensures the cookie is sent only over HTTP(S), not client JavaScript, helping to protect against cross-site scripting attacks.
- `domain` - indicates the domain of the cookie; use it to compare against the domain of the server in which the URL is being requested. If they match, then check the path attribute next.
- `path` - indicates the path of the cookie; use it to compare against the request path. If this and domain match, then send the cookie in the request.
- `expires` - use to set expiration date for persistent cookies.

Here is an example using `cookie-session` middleware:

```
const session = require('cookie-session')
const express = require('express')
const app = express()

const expiryDate = new Date(Date.now() + 60 * 60 * 1000) // 1 hour
app.use(session({
  name: 'session',
  keys: ['key1', 'key2'],
  cookie: {
    secure: true,
    httpOnly: true,
    domain: 'example.com',
    path: 'foo/bar',
    expires: expiryDate
  }
}))
```

## Prevent brute-force attacks against authorization

Make sure login endpoints are protected to make private data more secure.

A simple and powerful technique is to block authorization attempts using two metrics:

1. The first is number of consecutive failed attempts by the same user name and IP address.
2. The second is number of failed attempts from an IP address over some long period of time. For example, block an IP address if it makes 100 failed attempts in one day.

[rate-limiter-flexible](#) package provides tools to make this technique easy and fast. You can find [an example of brute-force protection in the documentation](#)

## Ensure your dependencies are secure

Using npm to manage your application's dependencies is powerful and convenient. But the packages that you use may contain critical security vulnerabilities that could also affect your application. The security of your app is only as strong as the "weakest link" in your dependencies.

Since npm@6, npm automatically reviews every install request. Also you can use 'npm audit' to analyze your dependency tree.

```
$ npm audit
```

If you want to stay more secure, consider [Snyk](#).

Snyk offers both a [command-line tool](#) and a [Github integration](#) that checks your application against [Snyk's open source vulnerability database](#) for any known vulnerabilities in your dependencies. Install the CLI as follows:

```
$ npm install -g snyk  
$ cd your-app
```

Use this command to test your application for vulnerabilities:

```
$ snyk test
```

Use this command to open a wizard that walks you through the process of applying updates or patches to fix the vulnerabilities that were found:

```
$ snyk wizard
```

## Avoid other known vulnerabilities

Keep an eye out for [Node Security Project](#) or [Snyk](#) advisories that may affect Express or other modules that your app uses. In general, these databases are excellent resources for knowledge and tools about Node security.

Finally, Express apps - like any other web apps - can be vulnerable to a variety of web-based attacks. Familiarize yourself with known [web vulnerabilities](#) and take precautions to avoid them.

## Additional considerations

Here are some further recommendations from the excellent [Node.js Security Checklist](#). Refer to that blog post for all the details on these recommendations:

- Always filter and sanitize user input to protect against cross-site scripting (XSS) and command injection attacks.
- Defend against SQL injection attacks by using parameterized queries or prepared statements.
- Use the open-source [sqlmap](#) tool to detect SQL injection vulnerabilities in your app.
- Use the [nmap](#) and [sslyze](#) tools to test the configuration of your SSL ciphers, keys, and renegotiation as well as the validity of your certificate.
- Use [safe-regex](#) to ensure your regular expressions are not susceptible to [regular expression denial of service](#) attacks.

---

[Go to TOC](#)

# Developing template engines for Express

Use the `app.engine(ext, callback)` method to create your own template engine. `ext` refers to the file extension, and `callback` is the template engine function, which accepts the following items as parameters: the location of the file, the options object, and the callback function.

The following code is an example of implementing a very simple template engine for rendering `.nt1` files.

```
const fs = require('fs') // this engine requires the fs module
app.engine('nt1', (filePath, options, callback) => { // define the template engine
  fs.readFile(filePath, (err, content) => {
    if (err) return callback(err)
    // this is an extremely simple template engine
    const rendered = content.toString()
      .replace('#title#', '<title>${options.title}</title>')
      .replace('#message#', '<h1>${options.message}</h1>')
    return callback(null, rendered)
  })
})
app.set('views', './views') // specify the views directory
app.set('view engine', 'nt1') // register the template engine
```

Your app will now be able to render `.nt1` files. Create a file named `index.nt1` in the `views` directory with the following content.

```
#title#
#message#
```

Then, create the following route in your app.

```
app.get('/', (req, res) => {
  res.render('index', { title: 'Hey', message: 'Hello there!' })
})
```

When you make a request to the home page, `index.nt1` will be rendered as HTML.

# Health Checks and Graceful Shutdown

## Graceful shutdown

When you deploy a new version of your application, you must replace the previous version. The [process manager](#) you're using will first send a SIGTERM signal to the application to notify it that it will be killed. Once the application gets this signal, it should stop accepting new requests, finish all the ongoing requests, clean up the resources it used, including database connections and file locks then exit.

### Example Graceful Shutdown

```
const server = app.listen(port)

process.on('SIGTERM', () => {
  debug('SIGTERM signal received: closing HTTP server')
  server.close(() => {
    debug('HTTP server closed')
  })
})
```

## Health checks

A load balancer uses health checks to determine if an application instance is healthy and can accept requests. For example, [Kubernetes has two health checks](#):

- `liveness`, that determines when to restart a container.
- `readiness`, that determines when a container is ready to start accepting traffic. When a pod is not ready, it is removed from the service load balancers.

## Third-party solutions

### Terminus

[Terminus](#) is an open-source project that adds health checks and graceful shutdown to your application to eliminate the need to write boilerplate code. You just provide the cleanup logic for graceful shutdowns and the health check logic for health checks, and terminus handles the rest.

Install terminus as follows:

```
$ npm i @godaddy/terminus --save
```

Here's a basic template that illustrates using terminus. For more information, see <https://github.com/godaddy/terminus>.

```
const http = require('http')
const express = require('express')
const { createTerminus } = require('@godaddy/terminus')
```

```

const app = express()

app.get('/', (req, res) => {
  res.send('ok')
})

const server = http.createServer(app)

function onSignal () {
  console.log('server is starting cleanup')
  // start cleanup of resource, like databases or file descriptors
}

async function onHealthCheck () {
  // checks if the system is healthy, like the db connection is live
  // resolves, if health, rejects if not
}

createTerminus(server, {
  signal: 'SIGINT',
  healthChecks: { '/healthcheck': onHealthCheck },
  onSignal
})

server.listen(3000)

```

## Lightship

[Lightship](#) is an open-source project that adds health, readiness and liveness checks to your application. Lightship is a standalone HTTP-service that runs as a separate HTTP service; this allows having health-readiness-liveness HTTP endpoints without exposing them on the public interface.

Install Lightship as follows:

```
$ npm install lightship
```

Basic template that illustrates using Lightship:

```

const http = require('http')
const express = require('express')
const {
  createLightship
} = require('lightship')

// Lightship will start a HTTP service on port 9000.
const lightship = createLightship()

const app = express()

app.get('/', (req, res) => {
  res.send('ok')
})

app.listen(3000, () => {
  lightship.signalReady()
})

// You can signal that the service is not ready using
`lightship.signalNotReady()`.

```

[Lightship documentation](#) provides examples of the corresponding [Kubernetes configuration](#) and a complete example of integration with [Express.js](#).

## http-terminator

[http-terminator](#) implements logic for gracefully terminating an express.js server.

Terminating a HTTP server in Node.js requires keeping track of all open connections and signaling them that the server is shutting down. [http-terminator](#) implements the logic for tracking all connections and their termination upon a timeout. [http-terminator](#) also ensures graceful communication of the server intention to shutdown to any clients that are currently receiving response from this server.

Install [http-terminator](#) as follows:

```
$ npm install http-terminator
```

Basic template that illustrates using [http-terminator](#):

```
const express = require('express')
const { createHttpTerminator } = require('http-terminator')

const app = express()

const server = app.listen(3000)

const httpTerminator = createHttpTerminator({ server })

app.get('/', (req, res) => {
  res.send('ok')
})

// A server will terminate after invoking `httpTerminator.terminate()`.
// Note: Timeout is used for illustration of delayed termination purposes only.
setTimeout(() => {
  httpTerminator.terminate()
}, 1000)
```

[http-terminator documentation](#) provides API documentation and comparison to other existing third-party solutions.

## express-actuator

[express-actuator](#) is a middleware to add endpoints to help you monitor and manage applications.

Install [express-actuator](#) as follows:

```
$ npm install --save express-actuator
```

Basic template that illustrates using [express-actuator](#):

```
const express = require('express')
const actuator = require('express-actuator')

const app = express()
```



```
app.use(activator())  
app.listen(3000)
```

The [express-actuator documentation](#) provides different options for customization.

# Process managers for Express apps

When you run Express apps for production, it is helpful to use a *process manager* to:

- Restart the app automatically if it crashes.
- Gain insights into runtime performance and resource consumption.
- Modify settings dynamically to improve performance.
- Control clustering.

A process manager is somewhat like an application server: it's a "container" for applications that facilitates deployment, provides high availability, and enables you to manage the application at runtime.

The most popular process managers for Express and other Node.js applications are:

- **Forever**: A simple command-line interface tool to ensure that a script runs continuously (forever). Forever's simple interface makes it ideal for running smaller deployments of Node.js apps and scripts.
- **PM2**: A production process manager for Node.js applications that has a built-in load balancer. PM2 enables you to keep applications alive forever, reloads them without downtime, helps you to manage application logging, monitoring, and clustering.
- **StrongLoop Process Manager (Strong-PM)**: A production process manager for Node.js applications with built-in load balancing, monitoring, and multi-host deployment. Includes a CLI to build, package, and deploy Node.js applications to a local or remote system.
- **SystemD**: The default process manager on modern Linux distributions, that makes it simple to run a Node application as a service. For more information, see ["Run node.js service with systemd" by Ralph Slooten \(@axllent\)](#).

---

[Go to TOC](#)

# Security updates

Node.js vulnerabilities directly affect Express. Therefore [keep a watch on Node.js vulnerabilities] (<http://blog.nodejs.org/vulnerability/>) and make sure you are using the latest stable version of Node.js.

The list below enumerates the Express vulnerabilities that were fixed in the specified version update.

**NOTE:** If you believe you have discovered a security vulnerability in Express, please see [Security Policies and Procedures](#).

## 4.x

- 4.16.0
  - The dependency `forwarded` has been updated to address a [vulnerability](#). This may affect your application if the following APIs are used: `req.host`, `req.hostname`, `req.ip`, `req.ips`, `req.protocol`.
  - The dependency `mime` has been updated to address a [vulnerability](#), but this issue does not impact Express.
  - The dependency `send` has been updated to provide a protection against a [Node.js 8.5.0 vulnerability](#). This only impacts running Express on the specific Node.js version 8.5.0.
- 4.15.5
  - The dependency `debug` has been updated to address a [vulnerability](#), but this issue does not impact Express.
  - The dependency `fresh` has been updated to address a [vulnerability](#). This will affect your application if the following APIs are used: `express.static`, `req.fresh`, `res.json`, `res.jsonp`, `res.send`, `res.sendFile`, `res.sendFile`, `res.sendStatus`.
- 4.15.3
  - The dependency `ms` has been updated to address a [vulnerability](#). This may affect your application if untrusted string input is passed to the `maxAge` option in the following APIs: `express.static`, `res.sendFile`, and `res.sendFile`.
- 4.15.2
  - The dependency `qs` has been updated to address a [vulnerability](#), but this issue does not impact Express. Updating to 4.15.2 is a good practice, but not required to address the vulnerability.
- 4.11.1
  - Fixed root path disclosure vulnerability in `express.static`, `res.sendFile`, and `res.sendFile`.
- 4.10.7
  - Fixed open redirect vulnerability in `express.static` ([advisory](#), [CVE-2015-1164](#)).
- 4.8.8
  - Fixed directory traversal vulnerabilities in `express.static` ([advisory](#), [CVE-2014-6394](#)).
- 4.8.4
  - Node.js 0.10 can leak `fd`s in certain situations that affect `express.static` and `res.sendFile`. Malicious requests could cause `fd`s to leak and eventually lead to `EMFILE` errors and server unresponsiveness.

- 4.8.0
  - Sparse arrays that have extremely high indexes in the query string could cause the process to run out of memory and crash the server.
  - Extremely nested query string objects could cause the process to block and make the server unresponsive temporarily.

## 3.x

**\*\*Express 3.x IS NO LONGER MAINTAINED\*\***

Known and unknown security issues in 3.x have not been addressed since the last update (1 August, 2015). Using the 3.x line should not be considered secure.

- 3.19.1
  - Fixed root path disclosure vulnerability in `express.static`, `res.sendFile`, and `res.sendFile`.
- 3.19.0
  - Fixed open redirect vulnerability in `express.static` ([advisory](#), [CVE-2015-1164](#)).
- 3.16.10
  - Fixed directory traversal vulnerabilities in `express.static`.
- 3.16.6
  - Node.js 0.10 can leak `fd`s in certain situations that affect `express.static` and `res.sendFile`. Malicious requests could cause `fd`s to leak and eventually lead to `EMFILE` errors and server unresponsiveness.
- 3.16.0
  - Sparse arrays that have extremely high indexes in query string could cause the process to run out of memory and crash the server.
  - Extremely nested query string objects could cause the process to block and make the server unresponsive temporarily.
- 3.3.0 \* The 404 response of an unsupported method override attempt was susceptible to cross-site scripting attacks.

## 4.x API

{% include api../4x/express.md %}

{% include api../4x/app.md %}

{% include api../4x/req.md %}

{% include api../4x/res.md %}

{% include api../4x/router.md %}

---

[Go to TOC](#)

# Release Change Log

## 4.18.1 - Release date: 2022-04-29

The 4.18.1 patch release includes the following bug fix:

- Fix the condition where if an Express.js application is created with a very large stack of routes, and all of those routes are sync (call `next()` synchronously), then the request processing may hang.

For a complete list of changes in this release, see [History.md](#).

## 4.18.0 - Release date: 2022-04-25

The 4.18.0 minor release includes bug fixes and some new features, including:

- The `[`app.get()` method](../4x/api.html#app.get)` and the `[`app.set()` method](../4x/api.html#app.set)` now ignores properties directly on `Object.prototype` when getting a setting value.
- The `[`res.cookie()` method](../4x/api.html#res.cookie)` now accepts a "priority" option to set the Priority attribute on the Set-Cookie response header.
- The `[`res.cookie()` method](../4x/api.html#res.cookie)` now rejects an Invalid Date object provided as the "expires" option.
- The `[`res.cookie()` method](../4x/api.html#res.cookie)` now works when `null` or `undefined` is explicitly provided as the "maxAge" argument.
- Starting with this version, Express supports Node.js 18.x.
- The `[`res.download()` method](../4x/api.html#res.download)` now accepts a "root" option to match `[`res.sendFile()`](../4x/api.html#res.sendFile)`.
- The `[`res.download()` method](../4x/api.html#res.download)` can be supplied with an `options` object without providing a `filename` argument, simplifying calls when the default `filename` is desired.
- The `[`res.format()` method](../4x/api.html#res.format)` now invokes the provided "default" handler with the same arguments as the type handlers (`req`, `res`, and `next`).
- The `[`res.send()` method](../4x/api.html#res.send)` will not attempt to send a response body when the response code is set to 205.
- The default error handler will now remove certain response headers that will break the error response rendering, if they were set previously.
- The status code 425 is now represented as the standard "Too Early" instead of "Unordered Collection".

For a complete list of changes in this release, see [History.md](#).

## 4.17.3 - Release date: 2022-02-16

The 4.17.3 patch release includes one bug fix:

- Update to `[qs module](https://www.npmjs.com/package/qs)` for a fix around parsing `__proto__` properties.

For a complete list of changes in this release, see [History.md](#).

## 4.17.2 - Release date: 2021-12-16

The 4.17.2 patch release includes the following bug fixes:

- Fix handling of `undefined` in `res.jsonp` when a callback is provided.
- Fix handling of `undefined` in `res.json` and `res.jsonp` when `"json escape"` is enabled.
- Fix handling of invalid values to the `maxAge` option of `res.cookie()`.
- Update to [jshttp/proxy-addr module](https://www.npmjs.com/package/proxy-addr) to use `req.socket` over deprecated `req.connection`.
- Starting with this version, Express supports Node.js 14.x.

For a complete list of changes in this release, see [History.md](#).

## 4.17.1 - Release date: 2019-05-25

The 4.17.1 patch release includes one bug fix:

- The change to the `res.status()` API has been reverted due to causing regressions in existing Express 4 applications.

For a complete list of changes in this release, see [History.md](#).

## 4.17.0 - Release date: 2019-05-16

The 4.17.0 minor release includes bug fixes and some new features, including:

- The `express.raw()` and `express.text()` middleware have been added to provide request body parsing for more raw request payloads. This uses the [expressjs/body-parser module](https://www.npmjs.com/package/body-parser) module underneath, so apps that are currently requiring the module separately can switch to the built-in parsers.
- The `res.cookie()` API now supports the `"none"` value for the `sameSite` option.
- When the `"trust proxy"` setting is enabled, the `req.hostname` now supports multiple `X-Forwarded-For` headers in a request.
- Starting with this version, Express supports Node.js 10.x and 12.x.
- The `res.sendFile()` API now provides a more immediate and easier to understand error when a non-string is passed as the `path` argument.
- The `res.status()` API now provides a more immediate and easier to understand error when `null` or `undefined` is passed as the argument.

For a complete list of changes in this release, see [History.md](#).

## 4.16.4 - Release date: 2018-10-10

The 4.16.4 patch release includes various bug fixes:

- Fix issue where `"Request aborted"` may be logged in `res.sendFile`.

For a complete list of changes in this release, see [History.md](#).

## 4.16.3 - Release date: 2018-03-12

The 4.16.3 patch release includes various bug fixes:

- Fix issue where a plain `%` at the end of the url in the `res.location` method or the `res.redirect` method would not get encoded as `%25`.
- Fix issue where a blank `req.url` value can result in a thrown error within the default 404 handling.
- Fix the generated HTML document for `express.static` redirect responses to properly include ```.

For a complete list of changes in this release, see [History.md](#).

## 4.16.2 - Release date: 2017-10-09

The 4.16.2 patch release includes a regression bug fix:

- Fix a `TypeError` that can occur in the `res.send` method when a `Buffer` is passed to `res.send` and the `ETag` header is already set on the response.

For a complete list of changes in this release, see [History.md](#).

## 4.16.1 - Release date: 2017-09-29

The 4.16.1 patch release includes a regression bug fix:

- Update to [pillarjs/send module](https://www.npmjs.com/package/send) to fix an edge case scenario regression that affected certain users of `express.static`.

For a complete list of changes in this release, see [History.md](#).

## 4.16.0 - Release date: 2017-09-28

The 4.16.0 minor release includes security updates, bug fixes, performance enhancements, and some new features, including:

- Update to [jshttp/forwarded module](https://www.npmjs.com/package/forwarded) to address a [vulnerability](https://npmjs.com/advisories/527). This may affect your application if the following APIs are used: `req.host`, `req.hostname`, `req.ip`, `req.ips`, `req.protocol`.
- Update a dependency of the [pillarjs/send module](https://www.npmjs.com/package/send) to address a [vulnerability](https://npmjs.com/advisories/535) in the `mime` dependency. This may affect your application if untrusted string input is passed to the following APIs: `res.type()`.
- The [pillarjs/send module](https://www.npmjs.com/package/send) has implemented a protection against the Node.js 8.5.0 [vulnerability](https://nodejs.org/en/blog/vulnerability/september-2017-path-validation/). Using any prior version of Express.js with Node.js 8.5.0 (that specific Node.js version) will make the following APIs vulnerable: `express.static`, `res.sendFile`, and `res.sendFile`.



- Starting with this version, Express supports Node.js 8.x.
- The new setting `"json escape"` can be enabled to escape characters in `res.json()`, `res.jsonp()` and `res.send()` responses that can trigger clients to sniff the response as HTML instead of honoring the `Content-Type`. This can help protect an Express app from a class of persistent XSS-based attacks.
- The `res.download()` method (`../4x/api.html#res.download`) now accepts an optional `options` object.
- The `express.json()` and `express.urlencoded()` middleware have been added to provide request body parsing support out-of-the-box. This uses the `[expressjs/body-parser module]` (<https://www.npmjs.com/package/body-parser>) module underneath, so apps that are currently requiring the module separately can switch to the built-in parsers.
- The `express.static()` middleware (`../4x/api.html#express.static`) and `res.sendFile()` method (`../4x/api.html#res.sendFile`) now support setting the `immutable` directive on the `Cache-Control` header. Setting this header with an appropriate `maxAge` will prevent supporting web browsers from sending any request to the server when the file is still in their cache.
- The `[pillarjs/send module]` (<https://www.npmjs.com/package/send>) has an updated list of MIME types to better set the `Content-Type` of more files. There are 70 new types for file extensions.

For a complete list of changes in this release, see [History.md](#).

## 4.15.5 - Release date: 2017-09-24

The 4.15.5 patch release includes security updates, some minor performance enhancements, and a bug fix:

- Update to `[debug module]` (<https://www.npmjs.com/package/debug>) to address a [vulnerability] (<https://snyk.io/vuln/npm:debug:20170905>), but this issue does not impact Express.
- Update to `[jshttp/fresh module]` (<https://www.npmjs.com/package/fresh>) to address a [vulnerability] (<https://npmjs.com/advisories/526>). This will affect your application if the following APIs are used: `express.static`, `req.fresh`, `res.json`, `res.jsonp`, `res.send`, `res.sendfile`, `res.sendFile`, `res.sendStatus`.
- Update to `[jshttp/fresh module]` (<https://www.npmjs.com/package/fresh>) fixes handling of modified headers with invalid dates and makes parsing conditional headers (like `If-None-Match`) faster.

For a complete list of changes in this release, see [History.md](#).

## 4.15.4 - Release date: 2017-08-06

The 4.15.4 patch release includes some minor bug fixes:

- Fix array being set for `"trust proxy"` value being manipulated in certain conditions.

For a complete list of changes in this release, see [History.md](#).

## 4.15.3 - Release date: 2017-05-16

The 4.15.3 patch release includes a security update and some minor bug fixes:

- Update a dependency of the [pillarjs/send module](https://www.npmjs.com/package/send) to address a [vulnerability](https://snyk.io/vuln/npm:ms:20170412). This may affect your application if untrusted string input is passed to the `maxAge` option in the following APIs: `express.static`, `res.sendFile`, and `res.sendFile`.
- Fix error when `res.set` cannot add charset to `Content-Type`.
- Fix missing `` in HTML document.

For a complete list of changes in this release, see [History.md](#).

## 4.15.2 - Release date: 2017-03-06

The 4.15.2 patch release includes a minor bug fix:

- Fix regression parsing keys starting with `[` in the extended (default) query parser.

For a complete list of changes in this release, see [History.md](#).

## 4.15.1 - Release date: 2017-03-05

The 4.15.1 patch release includes a minor bug fix:

- Fix compatibility issue when using the datejs 1.x library where the [ `express.static()` middleware ](../4x/api.html#express.static) and [ `res.sendFile()` method ](../4x/api.html#res.sendFile) would incorrectly respond with 412 Precondition Failed.

For a complete list of changes in this release, see [History.md](#).

## 4.15.0 - Release date: 2017-03-01

The 4.15.0 minor release includes bug fixes, performance improvements, and other minor feature additions, including:

- Starting with this version, Express supports Node.js 7.x.
- The [ `express.static()` middleware ](../4x/api.html#express.static) and [ `res.sendFile()` method ](../4x/api.html#res.sendFile) now support the `If-Match` and `If-Unmodified-Since` request headers.
- Update to [jshttp/etag module](https://www.npmjs.com/package/etag) to generate the default ETags for responses which work when Node.js has [FIPS-compliant crypto enabled](https://nodejs.org/dist/latest/docs/api/cli.html#cli\_enable\_fips).
- Various auto-generated HTML responses like the default not found and error handlers will respond with complete HTML 5 documents and additional security headers.

For a complete list of changes in this release, see [History.md](#).

## 4.14.1 - Release date: 2017-01-28

The 4.14.1 patch release includes bug fixes and performance improvements, including:

- Update to [pillarjs/finalhandler module](https://www.npmjs.com/package/finalhandler) fixes an exception when Express handles an `Error` object which has a `headers` property that is not an object.

For a complete list of changes in this release, see [History.md](#).

## 4.14.0 - Release date: 2016-06-16

The 4.14.0 minor release includes bug fixes, security update, performance improvements, and other minor feature additions, including:

- Starting with this version, Express supports Node.js 6.x.
- Update to [jshttp/negotiator module](https://www.npmjs.com/package/negotiator) fixes a [regular expression denial of service vulnerability](https://npmjs.com/advisories/106).
- The [ `res.sendFile()` method ](../4x/api.html#res.sendFile) now accepts two new options: `acceptRanges` and `cacheControl`.
  - `acceptRanges` (default is `true`), enables or disables accepting ranged requests. When disabled, the response does not send the `Accept-Ranges` header and ignores the contents of the `Range` request header.
  - `cacheControl`, (default is `true`), enables or disables the `Cache-Control` response header. Disabling it will ignore the `maxAge` option.
  - `res.sendFile` has also been updated to handle `Range` header and redirections better.
- The [ `res.location()` method ](../4x/api.html#res.location) and [ `res.redirect()` method ](../4x/api.html#res.redirect) will now URL-encode the URL string, if it is not already encoded.
- The performance of the [ `res.json()` method ](../4x/api.html#res.json) and [ `res.jsonp()` method ](../4x/api.html#res.jsonp) have been improved in the common cases.
- The [jshttp/cookie module](https://www.npmjs.com/package/cookie) (in addition to a number of other improvements) has been updated and now the [ `res.cookie()` method ](../4x/api.html#res.cookie) supports the `sameSite` option to let you specify the [SameSite cookie attribute](https://tools.ietf.org/html/draft-west-first-party-cookies-07). NOTE: This attribute has not yet been fully standardized, may change in the future, and many clients may ignore it.

The possible value for the `sameSite` option are:

- `true`, which sets the `SameSite` attribute to `Strict` for strict same site enforcement.
- `false`, which does not set the `SameSite` attribute.
- `'lax'`, which sets the `SameSite` attribute to `Lax` for lax same site enforcement.
- `'strict'`, which sets the `SameSite` attribute to `Strict` for strict same site enforcement.
- Absolute path checking on Windows, which was incorrect for some cases, has been fixed.
- IP address resolution with proxies has been greatly improved.
- The [ `req.range()` method ](../4x/api.html#req.range) options object now supports a `combine` option (`false` by default), which when `true`, combines overlapping and adjacent ranges and returns them as if they were specified that way in the header.

For a complete list of changes in this release, see [History.md](#).

---

[Go to TOC](#)

# Express behind proxies

When running an Express app behind a reverse proxy, some of the Express APIs may return different values than expected. In order to adjust for this, the `trust proxy` application setting may be used to expose information provided by the reverse proxy in the Express APIs. The most common issue is express APIs that expose the client's IP address may instead show an internal IP address of the reverse proxy.

When configuring the `'trust proxy'` setting, it is important to understand the exact setup of the reverse proxy. Since this setting will trust values provided in the request, it is important that the combination of the setting in Express matches how the reverse proxy operates.

The application setting `trust proxy` may be set to one of the values listed in the following table.

Type	Value
Boolean	<p>If <code>'true'</code>, the client's IP address is understood as the left-most entry in the <code>'X-Forwarded-For'</code> header.</p> <p>If <code>false</code>, the app is understood as directly facing the client and the client's IP address is derived from <code>req.socket.remoteAddress</code>. This is the default setting.</p> <p>When setting to <code>'true'</code>, it is important to ensure that the last reverse proxy trusted is removing/overwriting all of the following HTTP headers: <code>'X-Forwarded-For'</code>, <code>'X-Forwarded-Host'</code>, and <code>'X-Forwarded-Proto'</code> otherwise it may be possible for the client to provide any value.</p>
IP address-es	<p>An IP address, subnet, or an array of IP addresses and subnets to trust as being a reverse proxy. The following list shows the pre-configured subnet names:</p> <ul style="list-style-type: none"> <li>• loopback - <code>127.0.0.1/8</code>, <code>::1/128</code></li> <li>• linklocal - <code>169.254.0.0/16</code>, <code>fe80::/10</code></li> <li>• uniquelocal - <code>10.0.0.0/8</code>, <code>172.16.0.0/12</code>, <code>192.168.0.0/16</code>, <code>fc00::/7</code></li> </ul> <p>You can set IP addresses in any of the following ways:</p> <pre>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</pre> <p>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. This works by checking if <code>req.socket.remoteAddress</code> is trusted. If so, then each address in <code>X-Forwarded-For</code> is checked from right to left until the first non-trusted address.</p>

Type	Value
Number	<p>Use the address that is at most <code>n</code> number of hops away from the Express application. <code>req.socket.remoteAddress</code> is the first hop, and the rest are looked for in the <code>X-Forwarded-For</code> header from right to left. A value of <code>0</code> means that the first untrusted address would be <code>req.socket.remoteAddress</code>, i.e. there is no reverse proxy.</p> <p>When using this setting, it is important to ensure there are not multiple, different-length paths to the Express application such that the client can be less than the configured number of hops away, otherwise it may be possible for the client to provide any value.</p>
Function	<p>Custom trust implementation.</p> <pre>app.set('trust proxy', (ip) =&gt; {   if (ip === '127.0.0.1'    ip === '123.123.123.123') return true //     trusted IPs   else return false })</pre>

Enabling `trust proxy` will have the following impact:

- The value of `[req.hostname](../api.html#req.hostname)` is derived from the value set in the `X-Forwarded-Host` header, which can be set by the client or by the proxy.
- `X-Forwarded-Proto` can be set by the reverse proxy to tell the app whether it is `https` or `http` or even an invalid name. This value is reflected by `[req.protocol](../api.html#req.protocol)`.
- The `[req.ip](../api.html#req.ip)` and `[req.ips](../api.html#req.ips)` values are populated based on the socket address and `X-Forwarded-For` header, starting at the first untrusted address.

The `trust proxy` setting is implemented using the [proxy-addr](#) package. For more information, see its documentation.

---

[Go to TOC](#)

# Database integration

Adding the capability to connect databases to Express apps is just a matter of loading an appropriate Node.js driver for the database in your app. This document briefly explains how to add and use some of the most popular Node.js modules for database systems in your Express app:

- [Cassandra](#)
- [Couchbase](#)
- [CouchDB](#)
- [LevelDB](#)
- [MySQL](#)
- [MongoDB](#)
- [Neo4j](#)
- [Oracle](#)
- [PostgreSQL](#)
- [Redis](#)
- [SQL Server](#)
- [SQLite](#)
- [Elasticsearch](#)

These database drivers are among many that are available. For other options, search on the [npm] (<https://www.npmjs.com/>) site.

## Cassandra

**Module:** [cassandra-driver](#)

### Installation

```
$ npm install cassandra-driver
```

### Example

```
const cassandra = require('cassandra-driver')
const client = new cassandra.Client({ contactPoints: ['localhost'] })

client.execute('select key from system.local', (err, result) => {
  if (err) throw err
  console.log(result.rows[0])
})
```

## Couchbase

**Module:** [couchnode](#)

### Installation

```
$ npm install couchbase
```

## Example

```
const couchbase = require('couchbase')
const bucket = (new
couchbase.Cluster('http://localhost:8091')).openBucket('bucketName')

// add a document to a bucket
bucket.insert('document-key', { name: 'Matt', shoeSize: 13 }, (err, result) => {
  if (err) {
    console.log(err)
  } else {
    console.log(result)
  }
})

// get all documents with shoe size 13
const n1ql = 'SELECT d.* FROM `bucketName` d WHERE shoeSize = $1'
const query = N1qlQuery.fromString(n1ql)
bucket.query(query, [13], (err, result) => {
  if (err) {
    console.log(err)
  } else {
    console.log(result)
  }
})
```

## CouchDB

Module: [nano](#)

## Installation

```
$ npm install nano
```

## Example

```
const nano = require('nano')('http://localhost:5984')
nano.db.create('books')
const books = nano.db.use('books')

// Insert a book document in the books database
books.insert({ name: 'The Art of war' }, null, (err, body) => {
  if (err) {
    console.log(err)
  } else {
    console.log(body)
  }
})

// Get a list of all books
books.list((err, body) => {
  if (err) {
    console.log(err)
  } else {
    console.log(body.rows)
  }
})
```



## LevelDB

Module: `levelup`

### Installation

```
$ npm install level levelup leveledown
```

### Example

```
const levelup = require('levelup')
const db = levelup('./mydb')

db.put('name', 'LevelUP', (err) => {
  if (err) return console.log('0oops!', err)

  db.get('name', (err, value) => {
    if (err) return console.log('0oops!', err)

    console.log(`name=${value}`)
  })
})
```

## MySQL

Module: `mysql`

### Installation

```
$ npm install mysql
```

### Example

```
const mysql = require('mysql')
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'dbuser',
  password: 's3kre337',
  database: 'my_db'
})

connection.connect()

connection.query('SELECT 1 + 1 AS solution', (err, rows, fields) => {
  if (err) throw err

  console.log('The solution is: ', rows[0].solution)
})

connection.end()
```

## MongoDB

Module: `mongodb`

## Installation

```
$ npm install mongodb
```

### Example (v2.\*)

```
const MongoClient = require('mongodb').MongoClient

MongoClient.connect('mongodb://localhost:27017/animals', (err, db) => {
  if (err) throw err

  db.collection('mammals').find().toArray((err, result) => {
    if (err) throw err

    console.log(result)
  })
})
```

### Example (v3.\*)

```
const MongoClient = require('mongodb').MongoClient

MongoClient.connect('mongodb://localhost:27017/animals', (err, client) => {
  if (err) throw err

  const db = client.db('animals')

  db.collection('mammals').find().toArray((err, result) => {
    if (err) throw err

    console.log(result)
  })
})
```

If you want an object model driver for MongoDB, look at [Mongoose](#).

## Neo4j

**Module:** [neo4j-driver](#)

### Installation

```
$ npm install neo4j-driver
```

### Example

```
const neo4j = require('neo4j-driver')
const driver = neo4j.driver('neo4j://localhost:7687', neo4j.auth.basic('neo4j', 'letmein'))

const session = driver.session()

session.readTransaction((tx) => {
  return tx.run('MATCH (n) RETURN count(n) AS count')
  .then((res) => {
    console.log(res.records[0].get('count'))
  })
})
```

```

    })
    .catch((error) => {
      console.log(error)
    })
  })
})

```

## Oracle

**Module:** `oracledb`

### Installation

NOTE: [See installation prerequisites.](#)

```
$ npm install oracledb
```

### Example

```

const oracledb = require('oracledb')
const config = {
  user: '<your db user>',
  password: '<your db password>',
  connectString: 'localhost:1521/orcl'
}

async function getEmployee (empId) {
  let conn

  try {
    conn = await oracledb.getConnection(config)

    const result = await conn.execute(
      'select * from employees where employee_id = :id',
      [empId]
    )

    console.log(result.rows[0])
  } catch (err) {
    console.log('Ouch!', err)
  } finally {
    if (conn) { // conn assignment worked, need to close
      await conn.close()
    }
  }
}

getEmployee(101)

```

## PostgreSQL

**Module:** `pg-promise`

### Installation

```
$ npm install pg-promise
```

## Example

```
const pgp = require('pg-promise')(/* options */)
const db = pgp('postgres://username:password@host:port/database')

db.one('SELECT $1 AS value', 123)
  .then((data) => {
    console.log('DATA:', data.value)
  })
  .catch((error) => {
    console.log('ERROR:', error)
  })
```

## Redis

Module: [redis](#)

### Installation

```
$ npm install redis
```

## Example

```
const redis = require('redis')
const client = redis.createClient()

client.on('error', (err) => {
  console.log(`Error ${err}`)
})

client.set('string key', 'string val', redis.print)
client.hset('hash key', 'hashtest 1', 'some value', redis.print)
client.hset(['hash key', 'hashtest 2', 'some other value'], redis.print)

client.hkeys('hash key', (err, replies) => {
  console.log(`${replies.length} replies:`)

  replies.forEach((reply, i) => {
    console.log(`  ${i}: ${reply}`)
  })

  client.quit()
})
```

## SQL Server

Module: [tedious](#)

### Installation

```
$ npm install tedious
```

## Example

```
const Connection = require('tedious').Connection
const Request = require('tedious').Request

const config = {
  server: 'localhost',
  authentication: {
    type: 'default',
    options: {
      userName: 'your_username', // update me
      password: 'your_password' // update me
    }
  }
}

const connection = new Connection(config)

connection.on('connect', (err) => {
  if (err) {
    console.log(err)
  } else {
    executeStatement()
  }
})

function executeStatement () {
  request = new Request("select 123, 'hello world'", (err, rowCount) => {
    if (err) {
      console.log(err)
    } else {
      console.log(`${rowCount} rows`)
    }
    connection.close()
  })

  request.on('row', (columns) => {
    columns.forEach((column) => {
      if (column.value === null) {
        console.log('NULL')
      } else {
        console.log(column.value)
      }
    })
  })
})

connection.execSql(request)
}
```

## SQLite

Module: [sqlite3](#)

### Installation

```
$ npm install sqlite3
```

## Example

```
const sqlite3 = require('sqlite3').verbose()
const db = new sqlite3.Database(':memory:')

db.serialize(() => {
  db.run('CREATE TABLE lorem (info TEXT)')
  const stmt = db.prepare('INSERT INTO lorem VALUES (?)')

  for (let i = 0; i < 10; i++) {
    stmt.run(`Ipsum ${i}`)
  }

  stmt.finalize()

  db.each('SELECT rowid AS id, info FROM lorem', (err, row) => {
    console.log(`${row.id}: ${row.info}`)
  })
})

db.close()
```

## Elasticsearch

Module: [elasticsearch](#)

### Installation

```
$ npm install elasticsearch
```

### Example

```
const elasticsearch = require('elasticsearch')
const client = elasticsearch.Client({
  host: 'localhost:9200'
})

client.search({
  index: 'books',
  type: 'book',
  body: {
    query: {
      multi_match: {
        query: 'express js',
        fields: ['title', 'description']
      }
    }
  }
}).then((response) => {
  const hits = response.hits.hits
}, (error) => {
  console.trace(error.message)
})
```

---

[Go to TOC](#)

# Debugging Express

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

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

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

```
$ DEBUG=express:* node index.js
```

On Windows, use the corresponding command.

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

Running this command on the default app generated by the [express generator](#) prints the following output:

```
$ DEBUG=express:* node ./bin/www
express:router:route new / +0ms
express:router:layer new / +1ms
express:router:route get / +1ms
express:router:layer new / +0ms
express:router:route new / +1ms
express:router:layer new / +0ms
express:router:route get / +0ms
express:router:layer new / +0ms
express:application compile etag weak +1ms
express:application compile query parser extended +0ms
express:application compile trust proxy false +0ms
express:application booting in development mode +1ms
express:router use / query +0ms
express:router:layer new / +0ms
express:router use / expressInit +0ms
express:router:layer new / +0ms
express:router use / favicon +1ms
express:router:layer new / +0ms
express:router use / logger +0ms
express:router:layer new / +0ms
express:router use / jsonParser +0ms
express:router:layer new / +1ms
express:router use / urlencodedParser +0ms
express:router:layer new / +0ms
express:router use / cookieParser +0ms
express:router:layer new / +0ms
express:router use / stylus +90ms
express:router:layer new / +0ms
express:router use / serveStatic +0ms
express:router:layer new / +0ms
express:router use / router +0ms
express:router:layer new / +1ms
express:router use /users router +0ms
```

```

express:router:layer new /users +0ms
express:router use / <anonymous> +0ms
express:router:layer new / +0ms
express:router use / <anonymous> +0ms
express:router:layer new / +0ms
express:router use / <anonymous> +0ms
express:router:layer new / +0ms

```

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

```

express:router dispatching GET / +4h
express:router query : / +2ms
express:router expressInit : / +0ms
express:router favicon : / +0ms
express:router logger : / +1ms
express:router jsonParser : / +0ms
express:router urlencodedParser : / +1ms
express:router cookieParser : / +0ms
express:router stylus : / +0ms
express:router serveStatic : / +2ms
express:router router : / +2ms
express:router dispatching GET / +1ms
express:view lookup "index.pug" +338ms
express:view stat "/projects/example/views/index.pug" +0ms
express:view render "/projects/example/views/index.pug" +1ms

```

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

## Applications generated by `express`

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

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

```
$ DEBUG=sample-app:* node ./bin/www
```

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

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

## Advanced options

When running through Node.js, you can set a few environment variables that will change the behavior of the debug logging:

Name	Purpose
<code>DEBUG</code>	Enables/disables specific debugging namespaces.



Name	Purpose
<code>DEBUG_COLORS</code>	Whether or not to use colors in the debug output.
<code>DEBUG_DEPTH</code>	Object inspection depth.
<code>DEBUG_FD</code>	File descriptor to write debug output to.
<code>DEBUG_SHOW_HIDDEN</code>	Shows hidden properties on inspected objects.

**Note:** The environment variables beginning with `DEBUG_` end up being converted into an Options object that gets used with `%o` / `%O` formatters. See the Node.js documentation for `util.inspect()` for the complete list.

## Resources

For more information about `debug`, see the [debug](#).

# Error Handling

*Error Handling* refers to how Express catches and processes errors that occur both synchronously and asynchronously. Express comes with a default error handler so you don't need to write your own to get started.

## Catching Errors

It's important to ensure that Express catches all errors that occur while running route handlers and middleware.

Errors that occur in synchronous code inside route handlers and middleware require no extra work. If synchronous code throws an error, then Express will catch and process it. For example:

```
app.get('/', (req, res) => {
  throw new Error('BROKEN') // Express will catch this on its own.
})
```

For errors returned from asynchronous functions invoked by route handlers and middleware, you must pass them to the `next()` function, where Express will catch and process them. For example:

```
app.get('/', (req, res, next) => {
  fs.readFile('/file-does-not-exist', (err, data) => {
    if (err) {
      next(err) // Pass errors to Express.
    } else {
      res.send(data)
    }
  })
})
```

Starting with Express 5, route handlers and middleware that return a Promise will call `next(value)` automatically when they reject or throw an error. For example:

```
app.get('/user/:id', async (req, res, next) => {
  const user = await getUserById(req.params.id)
  res.send(user)
})
```

If `getUserById` throws an error or rejects, `next` will be called with either the thrown error or the rejected value. If no rejected value is provided, `next` will be called with a default Error object provided by the Express router.

If you pass anything to the `next()` function (except the string `'route'`), Express regards the current request as being an error and will skip any remaining non-error handling routing and middleware functions.

If the callback in a sequence provides no data, only errors, you can simplify this code as follows:

```
app.get('/', [
  function (req, res, next) {
    fs.writeFile('/inaccessible-path', 'data', next)
  }
])
```

```

    },
    function (req, res) {
      res.send('OK')
    }
  ])
}

```

In the above example `next` is provided as the callback for `fs.writeFile`, which is called with or without errors. If there is no error the second handler is executed, otherwise Express catches and processes the error.

You must catch errors that occur in asynchronous code invoked by route handlers or middleware and pass them to Express for processing. For example:

```

app.get('/', (req, res, next) => {
  setTimeout(() => {
    try {
      throw new Error('BROKEN')
    } catch (err) {
      next(err)
    }
  }, 100)
})

```

The above example uses a `try...catch` block to catch errors in the asynchronous code and pass them to Express. If the `try...catch` block were omitted, Express would not catch the error since it is not part of the synchronous handler code.

Use promises to avoid the overhead of the `try...catch` block or when using functions that return promises. For example:

```

app.get('/', (req, res, next) => {
  Promise.resolve().then(() => {
    throw new Error('BROKEN')
  }).catch(next) // Errors will be passed to Express.
})

```

Since promises automatically catch both synchronous errors and rejected promises, you can simply provide `next` as the final catch handler and Express will catch errors, because the catch handler is given the error as the first argument.

You could also use a chain of handlers to rely on synchronous error catching, by reducing the asynchronous code to something trivial. For example:

```

app.get('/', [
  function (req, res, next) {
    fs.readFile('/maybe-valid-file', 'utf-8', (err, data) => {
      res.locals.data = data
      next(err)
    })
  },
  function (req, res) {
    res.locals.data = res.locals.data.split(',')[1]
    res.send(res.locals.data)
  }
])

```

The above example has a couple of trivial statements from the `readFile` call. If `readFile` causes an error, then it passes the error to Express, otherwise you quickly return to the world of synchronous error handling in the next handler in the chain. Then, the example above tries to process the data. If this fails then the synchronous error handler will catch it. If you had done this processing inside the `readFile` callback then the application might exit and the Express error handlers would not run.

Whichever method you use, if you want Express error handlers to be called in and the application to survive, you must ensure that Express receives the error.

## The default error handler

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

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

Set the environment variable ``NODE_ENV`` to ``production``, to run the app in production mode.

When an error is written, the following information is added to the response:

- The `res.statusCode` is set from `err.status` (or `err.statusCode`). If this value is outside the 4xx or 5xx range, it will be set to 500.
- The `res.statusMessage` is set according to the status code.
- The body will be the HTML of the status code message when in production environment, otherwise will be `err.stack`.
- Any headers specified in an `err.headers` object.

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

So when you add a custom error handler, you must delegate to the default Express error handler, when the headers have already been sent to the client:

```
function errorHandler (err, req, res, next) {
  if (res.headersSent) {
    return next(err)
  }
  res.status(500)
  res.render('error', { error: err })
}
```

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

## Writing error handlers

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

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

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

```
const bodyParser = require('body-parser')
const methodOverride = require('method-override')

app.use(bodyParser.urlencoded({
  extended: true
}))
app.use(bodyParser.json())
app.use(methodOverride())
app.use((err, req, res, next) => {
  // logic
})
```

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

For organizational (and higher-level framework) purposes, you can define several error-handling middleware functions, much as you would with regular middleware functions. For example, to define an error-handler for requests made by using `XHR` and those without:

```
const bodyParser = require('body-parser')
const methodOverride = require('method-override')

app.use(bodyParser.urlencoded({
  extended: true
}))
app.use(bodyParser.json())
app.use(methodOverride())
app.use(logErrors)
app.use(clientErrorHandler)
app.use(errorHandler)
```

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

```
function logErrors (err, req, res, next) {
  console.error(err.stack)
  next(err)
}
```

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

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

```
function clientErrorHandler (err, req, res, next) {
  if (req.xhr) {
    res.status(500).send({ error: 'Something failed!' })
  } else {
    next(err)
  }
}
```

Implement the "catch-all" `errorHandler` function as follows (for example):

```
function errorHandler (err, req, res, next) {
  res.status(500)
  res.render('error', { error: err })
}
```

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

```
app.get('/a_route_behind_paywall',
  (req, res, next) => {
    if (!req.user.hasPaid) {
      // continue handling this request
      next('route')
    } else {
      next()
    }
  }, (req, res, next) => {
    PaidContent.find((err, doc) => {
      if (err) return next(err)
      res.json(doc)
    })
  })
```

In this example, the `getPaidContent` handler will be skipped but any remaining handlers in `app` for `/a_route_behind_paywall` would continue to be executed.

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

# Moving to Express 4

## Overview

Express 4 is a breaking change from Express 3. That means an existing Express 3 app will *not* work if you update the Express version in its dependencies.

This article covers:

- [Changes in Express 4.](#)
- [An example](#) of migrating an Express 3 app to Express 4.
- [Upgrading to the Express 4 app generator.](#)

## Changes in Express 4

There are several significant changes in Express 4:

- [Changes to Express core and middleware system.](#) The dependencies on Connect and built-in middleware were removed, so you must add middleware yourself.
- [Changes to the routing system.](#)
- [Various other changes.](#)

See also:

- [New features in 4.x.](#)
- [Migrating from 3.x to 4.x.](#)

## Changes to Express core and middleware system

Express 4 no longer depends on Connect, and removes all built-in middleware from its core, except for the `express.static` function. This means that Express is now an independent routing and middleware web framework, and Express versioning and releases are not affected by middleware updates.

Without built-in middleware, you must explicitly add all the middleware that is required to run your app. Simply follow these steps:

1. Install the module: `npm install --save <module-name>`
2. In your app, require the module: `require('module-name')`
3. Use the module according to its documentation: `app.use( ... )`

The following table lists Express 3 middleware and their counterparts in Express 4.

Express 3	Express 4
<code>express.bodyParser</code>	<code>body-parser</code> + <code>multer</code>
<code>express.compress</code>	<code>compression</code>

<code>express.cookieSession</code>	<a href="#">cookie-session</a>
<code>express.cookieParser</code>	<a href="#">cookie-parser</a>
<code>express.logger</code>	<a href="#">morgan</a>
<code>express.session</code>	<a href="#">express-session</a>
<code>express.favicon</code>	<a href="#">serve-favicon</a>
<code>express.responseTime</code>	<a href="#">response-time</a>
<code>express.errorHandler</code>	<a href="#">errorhandler</a>
<code>express.methodOverride</code>	<a href="#">method-override</a>
<code>express.timeout</code>	<a href="#">connect-timeout</a>
<code>express.vhost</code>	<a href="#">vhost</a>
<code>express.csrf</code>	<a href="#">csurf</a>
<code>express.directory</code>	<a href="#">serve-index</a>
<code>express.static</code>	<a href="#">serve-static</a>

Here is the [complete list](#) of Express 4 middleware.

In most cases, you can simply replace the old version 3 middleware with its Express 4 counterpart. For details, see the module documentation in GitHub.

#### `app.use` accepts parameters

In version 4 you can use a variable parameter to define the path where middleware functions are loaded, then read the value of the parameter from the route handler. For example:

```
app.use('/book/:id', (req, res, next) => {
  console.log('ID:', req.params.id)
  next()
})
```

## The routing system

Apps now implicitly load routing middleware, so you no longer have to worry about the order in which middleware is loaded with respect to the `router` middleware.

The way you define routes is unchanged, but the routing system has two new features to help organize your routes:

```
{: .doclist }
```

- A new method, `app.route()`, to create chainable route handlers for a route path.
- A new class, `express.Router`, to create modular mountable route handlers.



**app.route() method**

The new `app.route()` method enables you to create chainable route handlers for a route path. Because the path is specified in a single location, creating modular routes is helpful, as is reducing redundancy and typos. For more information about routes, see [Router\(\) documentation](#).

Here is an example of chained route handlers that are defined by using the `app.route()` function.

```
app.route('/book')
  .get((req, res) => {
    res.send('Get a random book')
  })
  .post((req, res) => {
    res.send('Add a book')
  })
  .put((req, res) => {
    res.send('Update the book')
  })
```

**express.Router class**

The other feature that helps to organize routes is a new class, `express.Router`, that you can use to create modular mountable route handlers. A `Router` instance is a complete middleware and routing system; for this reason it is often referred to as a "mini-app".

The following example creates a router as a module, loads middleware in it, defines some routes, and mounts it on a path on the main app.

For example, create a router file named `birds.js` in the app directory, with the following content:

```
var express = require('express')
var router = express.Router()

// middleware specific to this router
router.use((req, res, next) => {
  console.log('Time: ', Date.now())
  next()
})
// define the home page route
router.get('/', (req, res) => {
  res.send('Birds home page')
})
// define the about route
router.get('/about', (req, res) => {
  res.send('About birds')
})

module.exports = router
```

Then, load the router module in the app:

```
var birds = require('./birds')
// ...
app.use('/birds', birds)
```

The app will now be able to handle requests to the `/birds` and `/birds/about` paths, and will call the `timeLog` middleware that is specific to the route.

## Other changes

The following table lists other small but important changes in Express 4:

Object	Description
Node.js	Express 4 requires Node.js 0.10.x or later and has dropped support for Node.js 0.8.x.
<code>http.createServer()</code>	The <code>http</code> module is no longer needed, unless you need to directly work with it (socket.io/SPDY/HTTPS). The app can be started by using the <code>app.listen()</code> function.
<code>app.configure()</code>	The <code>app.configure()</code> function has been removed. Use the <code>process.env.NODE_ENV</code> or <code>app.get('env')</code> function to detect the environment and configure the app accordingly.
<code>json spaces</code>	The <code>json spaces</code> application property is disabled by default in Express 4.
<code>req.accepted()</code>	Use <code>req.accepts()</code> , <code>req.acceptsEncodings()</code> , <code>req.acceptsCharsets()</code> , and <code>req.acceptsLanguages()</code> .
<code>res.location()</code>	No longer resolves relative URLs.
<code>req.params</code>	Was an array; now an object.
<code>res.locals</code>	Was a function; now an object.
<code>res.headerSent</code>	Changed to <code>res.headersSent</code> .
<code>app.route</code>	Now available as <code>app.mountpath</code> .
<code>res.on('header')</code>	Removed.
<code>res.charset</code>	Removed.
<code>res.setHeader('Set-Cookie', val)</code>	Functionality is now limited to setting the basic cookie value. Use <code>res.cookie()</code> for added functionality.

## Example app migration

Here is an example of migrating an Express 3 application to Express 4. The files of interest are `app.js` and `package.json`.

### Version 3 app

`app.js`

Consider an Express v.3 application with the following `app.js` file:

```

var express = require('express')
var routes = require('./routes')
var user = require('./routes/user')
var http = require('http')
var path = require('path')

var app = express()

// all environments
app.set('port', process.env.PORT || 3000)
app.set('views', path.join(__dirname, 'views'))
app.set('view engine', 'pug')
app.use(express.favicon())
app.use(express.logger('dev'))
app.use(express.methodOverride())
app.use(express.session({ secret: 'your secret here' }))
app.use(express.bodyParser())
app.use(app.router)
app.use(express.static(path.join(__dirname, 'public')))

// development only
if (app.get('env') === 'development') {
  app.use(express.errorHandler())
}

app.get('/', routes.index)
app.get('/users', user.list)

http.createServer(app).listen(app.get('port'), () => {
  console.log('Express server listening on port ' + app.get('port'))
})

```

#### package.json

The accompanying version 3 `package.json` file might look something like this:

```

{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.12.0",
    "pug": "*"
  }
}

```

## Process

Begin the migration process by installing the required middleware for the Express 4 app and updating Express and Pug to their respective latest version with the following command:

```
$ npm install serve-favicon morgan method-override express-session body-parser
multer errorhandler express@latest pug@latest --save
```

Make the following changes to `app.js`:

1. The built-in Express middleware functions `express.favicon`, `express.logger`, `express.methodOverride`, `express.session`, `express.bodyParser` and `express.errorHandler` are no longer available on the `express` object. You must install their alternatives manually and load them in the app.
2. You no longer need to load the `app.router` function. It is not a valid Express 4 app object, so remove the `app.use(app.router);` code.
3. Make sure that the middleware functions are loaded in the correct order - load `errorHandler` after loading the app routes.

## Version 4 app

### package.json

Running the above `npm` command will update `package.json` as follows:

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "body-parser": "^1.5.2",
    "errorhandler": "^1.1.1",
    "express": "^4.8.0",
    "express-session": "^1.7.2",
    "pug": "^2.0.0",
    "method-override": "^2.1.2",
    "morgan": "^1.2.2",
    "multer": "^0.1.3",
    "serve-favicon": "^2.0.1"
  }
}
```

### app.js

Then, remove invalid code, load the required middleware, and make other changes as necessary. The `app.js` file will look like this:

```
var http = require('http')
var express = require('express')
var routes = require('./routes')
var user = require('./routes/user')
var path = require('path')

var favicon = require('serve-favicon')
var logger = require('morgan')
var methodOverride = require('method-override')
var session = require('express-session')
var bodyParser = require('body-parser')
var multer = require('multer')
var errorHandler = require('errorhandler')

var app = express()
```

```
// all environments
app.set('port', process.env.PORT || 3000)
app.set('views', path.join(__dirname, 'views'))
app.set('view engine', 'pug')
app.use(favicon(path.join(__dirname, '/public/favicon.ico')))
app.use(logger('dev'))
app.use(methodOverride())
app.use(session({
  resave: true,
  saveUninitialized: true,
  secret: 'uwotm8'
}))
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({ extended: true }))
app.use(multer())
app.use(express.static(path.join(__dirname, 'public')))

app.get('/', routes.index)
app.get('/users', user.list)

// error handling middleware should be loaded after the loading the routes
if (app.get('env') === 'development') {
  app.use(errorHandler())
}

var server = http.createServer(app)
server.listen(app.get('port'), () => {
  console.log('Express server listening on port ' + app.get('port'))
})
```

Unless you need to work directly with the `http` module (socket.io/SPDY/HTTPS), loading it is not required, and the app can be simply started this way:

```
app.listen(app.get('port'), () => {
  console.log('Express server listening on port ' + app.get('port'))
})
```

## Run the app

The migration process is complete, and the app is now an Express 4 app. To confirm, start the app by using the following command:

```
$ node .
```

Load <http://localhost:3000> and see the home page being rendered by Express 4.

## Upgrading to the Express 4 app generator

The command-line tool to generate an Express app is still `express`, but to upgrade to the new version, you must uninstall the Express 3 app generator and then install the new `express-generator`.

### Installing

If you already have the Express 3 app generator installed on your system, you must uninstall it:

```
$ npm uninstall -g express
```

Depending on how your file and directory privileges are configured, you might need to run this command with `sudo`.

Now install the new generator:

```
$ npm install -g express-generator
```

Depending on how your file and directory privileges are configured, you might need to run this command with `sudo`.

Now the `express` command on your system is updated to the Express 4 generator.

## Changes to the app generator

Command options and use largely remain the same, with the following exceptions:

`{: .doclist }`

- Removed the `--sessions` option.
- Removed the `--jshtml` option.
- Added the `--hogan` option to support [Hogan.js](#).

## Example

Execute the following command to create an Express 4 app:

```
$ express app4
```

If you look at the contents of the `app4/app.js` file, you will notice that all the middleware functions (except `express.static`) that are required for the app are loaded as independent modules, and the `router` middleware is no longer explicitly loaded in the app.

You will also notice that the `app.js` file is now a Node.js module, in contrast to the standalone app that was generated by the old generator.

After installing the dependencies, start the app by using the following command:

```
$ npm start
```

If you look at the npm start script in the `package.json` file, you will notice that the actual command that starts the app is `node ./bin/www`, which used to be `node app.js` in Express 3.

Because the `app.js` file that was generated by the Express 4 generator is now a Node.js module, it can no longer be started independently as an app (unless you modify the code). The module must be loaded in a Node.js file and started via the Node.js file. The Node.js file is `./bin/www` in this case.

Neither the `bin` directory nor the extensionless `www` file is mandatory for creating an Express app or starting the app. They are just suggestions made by the generator, so feel free to modify them to suit your needs.

To get rid of the `www` directory and keep things the "Express 3 way", delete the line that says `module.exports = app;` at the end of the `app.js` file, then paste the following code in its place:

```
app.set('port', process.env.PORT || 3000)

var server = app.listen(app.get('port'), () => {
  debug('Express server listening on port ' + server.address().port)
})
```

Ensure that you load the `debug` module at the top of the `app.js` file by using the following code:

```
var debug = require('debug')('app4')
```

Next, change `"start": "node ./bin/www"` in the `package.json` file to `"start": "node app.js"`.

You have now moved the functionality of `./bin/www` back to `app.js`. This change is not recommended, but the exercise helps you to understand how the `./bin/www` file works, and why the `app.js` file no longer starts on its own.

# Moving to Express 5

## Overview

Express 5.0 is still in the beta release stage, but here is a preview of the changes that will be in the release and how to migrate your Express 4 app to Express 5.

To install the latest beta and to preview Express 5, enter the following command in your application root directory:

```
$ npm install "express@>=5.0.0-beta.1" --save
```

You can then run your automated tests to see what fails, and fix problems according to the updates listed below. After addressing test failures, run your app to see what errors occur. You'll find out right away if the app uses any methods or properties that are not supported.

## Changes in Express 5

### Removed methods and properties

- [app.del\(\)](#)
- [app.param\(fn\)](#)
- [Pluralized method names](#)
- [Leading colon in name argument to app.param\(name, fn\)](#)
- [req.param\(name\)](#)
- [res.json\(obj, status\)](#)
- [res.jsonp\(obj, status\)](#)
- [res.send\(body, status\)](#)
- [res.send\(status\)](#)
- [res.sendFile\(\)](#)

### Changed

- [Path route matching syntax](#)
- [Rejected promises handled from middleware and handlers](#)
- [app.router](#)
- [req.host](#)
- [req.query](#)

### Improvements

- [res.render\(\)](#)



## Removed methods and properties

If you use any of these methods or properties in your app, it will crash. So, you'll need to change your app after you update to version 5.

### **app.del()**

Express 5 no longer supports the `app.del()` function. If you use this function an error is thrown. For registering HTTP DELETE routes, use the `app.delete()` function instead.

Initially `del` was used instead of `delete`, because `delete` is a reserved keyword in JavaScript. However, as of ECMAScript 6, `delete` and other reserved keywords can legally be used as property names.

### **app.param(fn)**

The `app.param(fn)` signature was used for modifying the behavior of the `app.param(name, fn)` function. It has been deprecated since v4.11.0, and Express 5 no longer supports it at all.

### **Pluralized method names**

The following method names have been pluralized. In Express 4, using the old methods resulted in a deprecation warning. Express 5 no longer supports them at all:

`req.acceptsCharset()` is replaced by `req.acceptsCharsets()`.

`req.acceptsEncoding()` is replaced by `req.acceptsEncodings()`.

`req.acceptsLanguage()` is replaced by `req.acceptsLanguages()`.

### **Leading colon (:) in the name for app.param(name, fn)**

A leading colon character (:) in the name for the `app.param(name, fn)` function is a remnant of Express 3, and for the sake of backwards compatibility, Express 4 supported it with a deprecation notice. Express 5 will silently ignore it and use the name parameter without prefixing it with a colon.

This should not affect your code if you follow the Express 4 documentation of [app.param](#), as it makes no mention of the leading colon.

### **req.param(name)**

This potentially confusing and dangerous method of retrieving form data has been removed. You will now need to specifically look for the submitted parameter name in the `req.params`, `req.body`, or `req.query` object.

### **res.json(obj, status)**

Express 5 no longer supports the signature `res.json(obj, status)`. Instead, set the status and then chain it to the `res.json()` method like this: `res.status(status).json(obj)`.

**res.jsonp(obj, status)**

Express 5 no longer supports the signature `res.jsonp(obj, status)`. Instead, set the status and then chain it to the `res.jsonp()` method like this: `res.status(status).jsonp(obj)`.

**res.send(body, status)**

Express 5 no longer supports the signature `res.send(obj, status)`. Instead, set the status and then chain it to the `res.send()` method like this: `res.status(status).send(obj)`.

**res.send(status)**

Express 5 no longer supports the signature `res.send(status)`, where `status` is a number. Instead, use the `res.sendStatus(statusCode)` function, which sets the HTTP response header status code and sends the text version of the code: "Not Found", "Internal Server Error", and so on. If you need to send a number by using the `res.send()` function, quote the number to convert it to a string, so that Express does not interpret it as an attempt to use the unsupported old signature.

**res.sendfile()**

The `res.sendfile()` function has been replaced by a camel-cased version `res.sendFile()` in Express 5.

## Changed

**Path route matching syntax**

Path route matching syntax is when a string is supplied as the first parameter to the `app.all()`, `app.use()`, `app.METHOD()`, `router.all()`, `router.METHOD()`, and `router.use()` APIs. The following changes have been made to how the path string is matched to an incoming request:

- Add new `?`, `*`, and `+` parameter modifiers.
- Matching group expressions are only RegEx syntax.
  - `(*)` is no longer valid and must be written as `(.*)`, for example.
- Named matching groups no longer available by position in `req.params`.
  - `/:foo(.*)` only captures as `req.params.foo` and not available as `req.params[0]`.
- Regular expressions can only be used in a matching group.
  - `/\\d+` is no longer valid and must be written as `/(\d+)`.
- Special `*` path segment behavior removed.
  - `/foo/*/bar` will match a literal `*` as the middle segment.

**Rejected promises handled from middleware and handlers**

Request middleware and handlers that return rejected promises are now handled by forwarding the rejected value as an `Error` to the error handling middleware. This means that using `async` functions as middleware and handlers are easier than ever. When an error is thrown in an `async` function or a rejected promise is `await`ed inside an `async` function, those errors will be passed to the error handler as if calling `next(err)`.

Details of how Express handles errors is covered in the [error handling documentation](#).

### **app.router**

The `app.router` object, which was removed in Express 4, has made a comeback in Express 5. In the new version, this object is just a reference to the base Express router, unlike in Express 3, where an app had to explicitly load it.

### **req.host**

In Express 4, the `req.host` function incorrectly stripped off the port number if it was present. In Express 5 the port number is maintained.

### **req.query**

The `req.query` property is no longer a writable property and is instead a getter. The default query parser has been changed from "extended" to "simple".

## **Improvements**

### **res.render()**

This method now enforces asynchronous behavior for all view engines, avoiding bugs caused by view engines that had a synchronous implementation and that violated the recommended interface.

# Overriding the Express API

The Express API consists of various methods and properties on the request and response objects. These are inherited by prototype. There are two extension points for the Express API:

1. The global prototypes at `express.request` and `express.response`.
2. App-specific prototypes at `app.request` and `app.response`.

Altering the global prototypes will affect all loaded Express apps in the same process. If desired, alterations can be made app-specific by only altering the app-specific prototypes after creating a new app.

## Methods

You can override the signature and behavior of existing methods with your own, by assigning a custom function.

Following is an example of overriding the behavior of `res.sendStatus`.

```
app.response.sendStatus = function (statusCode, type, message) {
  // code is intentionally kept simple for demonstration purpose
  return this.contentType(type)
    .status(statusCode)
    .send(message)
}
```

The above implementation completely changes the original signature of `res.sendStatus`. It now accepts a status code, encoding type, and the message to be sent to the client.

The overridden method may now be used this way:

```
res.sendStatus(404, 'application/json', '{"error":"resource not found"}')
```

## Properties

Properties in the Express API are either:

1. Assigned properties (ex: `req.baseUrl`, `req.originalUrl`)
2. Defined as getters (ex: `req.secure`, `req.ip`)

Since properties under category 1 are dynamically assigned on the `request` and `response` objects in the context of the current request-response cycle, their behavior cannot be overridden.

Properties under category 2 can be overwritten using the Express API extensions API.

The following code rewrites how the value of `req.ip` is to be derived. Now, it simply returns the value of the `Client-IP` request header.

```
Object.defineProperty(app.request, 'ip', {
  configurable: true,
  enumerable: true,
  get () { return this.get('Client-IP') }
})
```

## Prototype

In order to provide the Express.js API, the request/response objects passed to Express.js (via `app(req, res)`, for example) need to inherit from the same prototype chain. By default this is `http.IncomingRequest.prototype` for the request and `http.ServerResponse.prototype` for the response.

Unless necessary, it is recommended that this be done only at the application level, rather than globally. Also, take care that the prototype that is being used matches the functionality as closely as possible to the default prototypes.

```
// Use FakeRequest and FakeResponse in place of http.IncomingRequest and
// http.ServerResponse
// for the given app reference
Object.setPrototypeOf(Object.getPrototypeOf(app.request), FakeRequest.prototype)
Object.setPrototypeOf(Object.getPrototypeOf(app.response), FakeResponse.prototype)
```

# Routing

*Routing* refers to how an application's endpoints (URIs) respond to client requests. For an introduction to routing, see [Basic routing](#).

You define routing using methods of the Express `app` object that correspond to HTTP methods; for example, `app.get()` to handle GET requests and `app.post` to handle POST requests. For a full list, see [app.METHOD](#). You can also use `app.all()` to handle all HTTP methods and `app.use()` to specify middleware as the callback function (See [Using middleware](#) for details).

These routing methods specify a callback function (sometimes called "handler functions") called when the application receives a request to the specified route (endpoint) and HTTP method. In other words, the application "listens" for requests that match the specified route(s) and method(s), and when it detects a match, it calls the specified callback function.

In fact, the routing methods can have more than one callback function as arguments. With multiple callback functions, it is important to provide `next` as an argument to the callback function and then call `next()` within the body of the function to hand off control to the next callback.

The following code is an example of a very basic route.

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

// respond with "hello world" when a GET request is made to the homepage
app.get('/', (req, res) => {
  res.send('hello world')
})
```

## Route methods

A route method is derived from one of the HTTP methods, and is attached to an instance of the `express` class.

The following code is an example of routes that are defined for the GET and the POST methods to the root of the app.

```
// GET method route
app.get('/', (req, res) => {
  res.send('GET request to the homepage')
})

// POST method route
app.post('/', (req, res) => {
  res.send('POST request to the homepage')
})
```

Express supports methods that correspond to all HTTP request methods: `get`, `post`, and so on. For a full list, see [app.METHOD](#).

There is a special routing method, `app.all()`, used to load middleware functions at a path for *all* HTTP request methods. For example, the following handler is executed for requests to the route `/secret` whether using GET, POST, PUT, DELETE, or any other HTTP request method supported in the [http module](#).

```
app.all('/secret', (req, res, next) => {
  console.log('Accessing the secret section ...')
  next() // pass control to the next handler
})
```

## Route paths

Route paths, in combination with a request method, define the endpoints at which requests can be made. Route paths can be strings, string patterns, or regular expressions.

The characters `?`, `+`, `*`, and `()` are subsets of their regular expression counterparts. The hyphen (`-`) and the dot (`.`) are interpreted literally by string-based paths.

If you need to use the dollar character (`$`) in a path string, enclose it escaped within `([` and `])`. For example, the path string for requests at `/data/$book`, would be `/data/([\$])book`.

Express uses `[path-to-regexp]` (<https://www.npmjs.com/package/path-to-regexp>) for matching the route paths; see the `path-to-regexp` documentation for all the possibilities in defining route paths. `[Express Route Tester]` (<http://forbeslindesay.github.io/express-route-tester/>) is a handy tool for testing basic Express routes, although it does not support pattern matching.

Query strings are not part of the route path.

Here are some examples of route paths based on strings.

This route path will match requests to the root route, `/`.

```
app.get('/', (req, res) => {
  res.send('root')
})
```

This route path will match requests to `/about`.

```
app.get('/about', (req, res) => {
  res.send('about')
})
```

This route path will match requests to `/random.text`.

```
app.get('/random.text', (req, res) => {
  res.send('random.text')
})
```

Here are some examples of route paths based on string patterns.

This route path will match `acd` and `abcd`.

```
app.get('/ab?cd', (req, res) => {
  res.send('ab?cd')
})
```

This route path will match `abcd`, `abxcd`, `abbbcd`, and so on.

```
app.get('/ab+cd', (req, res) => {
  res.send('ab+cd')
})
```

This route path will match `abcd`, `abxcd`, `abRANDOMcd`, `ab123cd`, and so on.

```
app.get('/ab*cd', (req, res) => {
  res.send('ab*cd')
})
```

This route path will match `/abe` and `/abcde`.

```
app.get('/ab(cd)?e', (req, res) => {
  res.send('ab(cd)?e')
})
```

Examples of route paths based on regular expressions:

This route path will match anything with an "a" in it.

```
app.get(/a/, (req, res) => {
  res.send('/a/')
})
```

This route path will match `butterfly` and `dragonfly`, but not `butterflyman`, `dragonflyman`, and so on.

```
app.get(/.*fly$/, (req, res) => {
  res.send('/.*fly$/')
```

## Route parameters

Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the `req.params` object, with the name of the route parameter specified in the path as their respective keys.

```
Route path: /users/:userId/books/:bookId
Request URL: http://localhost:3000/users/34/books/8989
req.params: { "userId": "34", "bookId": "8989" }
```

To define routes with route parameters, simply specify the route parameters in the path of the route as shown below.

```
app.get('/users/:userId/books/:bookId', (req, res) => {
  res.send(req.params)
})
```



The name of route parameters must be made up of "word characters" ([A-Za-z0-9\_]).

Since the hyphen ( - ) and the dot ( . ) are interpreted literally, they can be used along with route parameters for useful purposes.

```
Route path: /flights/:from-:to
Request URL: http://localhost:3000/flights/LAX-SFO
req.params: { "from": "LAX", "to": "SFO" }
```

```
Route path: /plantae/:genus.:species
Request URL: http://localhost:3000/plantae/Prunus.persica
req.params: { "genus": "Prunus", "species": "persica" }
```

To have more control over the exact string that can be matched by a route parameter, you can append a regular expression in parentheses ( ( ) ):

```
Route path: /user/:userId(\d+)
Request URL: http://localhost:3000/user/42
req.params: { "userId": "42" }
```

Because the regular expression is usually part of a literal string, be sure to escape any \ characters with an additional backslash, for example \\d+.

In Express 4.x, the \* character in regular expressions is not interpreted in the usual way. As a workaround, use {0,} instead of \*. This will likely be fixed in Express 5.

## Route handlers

You can provide multiple callback functions that behave like [middleware](#) to handle a request. The only exception is that these callbacks might invoke `next('route')` to bypass the remaining route callbacks. 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.

Route handlers can be in the form of a function, an array of functions, or combinations of both, as shown in the following examples.

A single callback function can handle a route. For example:

```
app.get('/example/a', (req, res) => {
  res.send('Hello from A!')
})
```

More than one callback function can handle a route (make sure you specify the `next` object). For example:

```
app.get('/example/b', (req, res, next) => {
  console.log('the response will be sent by the next function ...')
  next()
}, (req, res) => {
  res.send('Hello from B!')
})
```

An array of callback functions can handle a route. For example:

```
const cb0 = function (req, res, next) {
  console.log('CB0')
  next()
}

const cb1 = function (req, res, next) {
  console.log('CB1')
  next()
}

const cb2 = function (req, res) {
  res.send('Hello from C!')
}

app.get('/example/c', [cb0, cb1, cb2])
```

A combination of independent functions and arrays of functions can handle a route. For example:

```
const cb0 = function (req, res, next) {
  console.log('CB0')
  next()
}

const cb1 = function (req, res, next) {
  console.log('CB1')
  next()
}

app.get('/example/d', [cb0, cb1], (req, res, next) => {
  console.log('the response will be sent by the next function ...')
  next()
}, (req, res) => {
  res.send('Hello from D!')
})
```

## Response methods

The methods on the response object (`res`) in the following table can send a response to the client, and terminate the request-response cycle. If none of these methods are called from a route handler, the client request will be left hanging.

Method	Description
<code>res.download()</code>	Prompt a file to be downloaded.
<code>res.end()</code>	End the response process.
<code>res.json()</code>	Send a JSON response.
<code>res.jsonp()</code>	Send a JSON response with JSONP support.
<code>res.redirect()</code>	Redirect a request.
<code>res.render()</code>	Render a view template.
<code>res.send()</code>	Send a response of various types.
<code>res.sendFile()</code>	Send a file as an octet stream.

Method	Description
<code>res.sendStatus()</code>	Set the response status code and send its string representation as the response body.

## app.route()

You can create chainable route handlers for a route path by using `app.route()`. Because the path is specified at a single location, creating modular routes is helpful, as is reducing redundancy and typos. For more information about routes, see: [Router\(\) documentation](#).

Here is an example of chained route handlers that are defined by using `app.route()`.

```
app.route('/book')
  .get((req, res) => {
    res.send('Get a random book')
  })
  .post((req, res) => {
    res.send('Add a book')
  })
  .put((req, res) => {
    res.send('Update the book')
  })
```

## express.Router

Use the `express.Router` class to create modular, mountable route handlers. A `Router` instance is a complete middleware and routing system; for this reason, it is often referred to as a "mini-app".

The following example creates a router as a module, loads a middleware function in it, defines some routes, and mounts the router module on a path in the main app.

Create a router file named `birds.js` in the app directory, with the following content:

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

// middleware that is specific to this router
router.use((req, res, next) => {
  console.log('Time: ', Date.now())
  next()
})

// define the home page route
router.get('/', (req, res) => {
  res.send('Birds home page')
})

// define the about route
router.get('/about', (req, res) => {
  res.send('About birds')
})

module.exports = router
```

Then, load the router module in the app:

```
const birds = require('./birds')  
  
// ...  
  
app.use('/birds', birds)
```

The app will now be able to handle requests to `/birds` and `/birds/about`, as well as call the `timeLog` middleware function that is specific to the route.

# Using middleware

Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.

*Middleware* functions are functions that have access to the [request object](#) (`req`), the [response object](#) (`res`), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named `next`.

Middleware functions can perform the following tasks:

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

If the current middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function. Otherwise, the request will be left hanging.

An Express application can use the following types of middleware:

- [Application-level middleware](#)
- [Router-level middleware](#)
- [Error-handling middleware](#)
- [Built-in middleware](#)
- [Third-party middleware](#)

You can load application-level and router-level middleware with an optional mount path. You can also load a series of middleware functions together, which creates a sub-stack of the middleware system at a mount point.

## Application-level middleware

Bind application-level middleware to an instance of the [app object](#) by using the `app.use()` and `app.METHOD()` functions, where `METHOD` is the HTTP method of the request that the middleware function handles (such as GET, PUT, or POST) in lowercase.

This example shows a middleware function with no mount path. The function is executed every time the app receives a request.

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

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

This example shows a middleware function mounted on the `/user/:id` path. The function is executed for any type of HTTP request on the `/user/:id` path.

```
app.use('/user/:id', (req, res, next) => {
  console.log('Request Type:', req.method)
  next()
})
```

This example shows a route and its handler function (middleware system). The function handles GET requests to the `/user/:id` path.

```
app.get('/user/:id', (req, res, next) => {
  res.send('USER')
})
```

Here is an example of loading a series of middleware functions at a mount point, with a mount path. It illustrates a middleware sub-stack that prints request info for any type of HTTP request to the `/user/:id` path.

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

Route handlers enable you to define multiple routes for a path. The example below defines two routes for GET requests to the `/user/:id` path. The second route will not cause any problems, but it will never get called because the first route ends the request-response cycle.

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

```
app.get('/user/:id', (req, res, next) => {
  console.log('ID:', req.params.id)
  next()
}, (req, res, next) => {
  res.send('User Info')
})

// handler for the /user/:id path, which prints the user ID
app.get('/user/:id', (req, res, next) => {
  res.send(req.params.id)
})
```

To skip the rest of the middleware functions from a router middleware stack, call `next('route')` to pass control to the next route. **NOTE:** `next('route')` will work only in middleware functions that were loaded by using the `app.METHOD()` or `router.METHOD()` functions.

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

```
app.get('/user/:id', (req, res, next) => {
  // if the user ID is 0, skip to the next route
  if (req.params.id === '0') next('route')
  // otherwise pass the control to the next middleware function in this stack
})
```

```

    else next()
  }, (req, res, next) => {
    // send a regular response
    res.send('regular')
  })

  // handler for the /user/:id path, which sends a special response
  app.get('/user/:id', (req, res, next) => {
    res.send('special')
  })

```

Middleware can also be declared in an array for reusability.

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

```

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

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

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

```

## Router-level middleware

Router-level middleware works in the same way as application-level middleware, except it is bound to an instance of `express.Router()`.

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

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

The following example code replicates the middleware system that is shown above for application-level middleware, by using router-level middleware:

```

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

// a middleware function with no mount path. This code is executed for every
// request to the router
router.use((req, res, next) => {
  console.log('Time:', Date.now())
  next()
})

// a middleware sub-stack shows request info for any type of HTTP request to the
// /user/:id path
router.use('/user/:id', (req, res, next) => {

```

```

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

  // a middleware sub-stack that handles GET requests to the /user/:id path
  router.get('/user/:id', (req, res, next) => {
    // if the user ID is 0, skip to the next router
    if (req.params.id === '0') next('route')
    // otherwise pass control to the next middleware function in this stack
    else next()
  }, (req, res, next) => {
    // render a regular page
    res.render('regular')
  })

  // handler for the /user/:id path, which renders a special page
  router.get('/user/:id', (req, res, next) => {
    console.log(req.params.id)
    res.render('special')
  })

  // mount the router on the app
  app.use('/', router)

```

To skip the rest of the router's middleware functions, call `next('router')` to pass control back out of the router instance.

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

```

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

// predicate the router with a check and bail out when needed
router.use((req, res, next) => {
  if (!req.headers['x-auth']) return next('router')
  next()
})

router.get('/user/:id', (req, res) => {
  res.send('hello, user!')
})

// use the router and 401 anything falling through
app.use('/admin', router, (req, res) => {
  res.sendStatus(401)
})

```

## Error-handling middleware

Error-handling middleware always takes `_four_` arguments. You must provide four arguments to identify it as an error-handling middleware function. Even if you don't need to use the ``next`` object, you must specify it to maintain the signature. Otherwise, the ``next`` object will be interpreted as regular middleware and will fail to handle errors.



Define error-handling middleware functions in the same way as other middleware functions, except with four arguments instead of three, specifically with the signature `(err, req, res, next)`:

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

For details about error-handling middleware, see: [Error handling](#).

## Built-in middleware

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

Express has the following built-in middleware functions:

- [express.static](#) serves static assets such as HTML files, images, and so on.
- [express.json](#) parses incoming requests with JSON payloads. **NOTE: Available with Express 4.16.0+**
- [express.urlencoded](#) parses incoming requests with URL-encoded payloads. **NOTE: Available with Express 4.16.0+**

## Third-party middleware

Use third-party middleware to add functionality to Express apps.

Install the Node.js module for the required functionality, then load it in your app at the application level or at the router level.

The following example illustrates installing and loading the cookie-parsing middleware function `cookie-parser`.

```
$ npm install cookie-parser
```

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

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

For a partial list of third-party middleware functions that are commonly used with Express, see: [Third-party middleware](#).

# Using template engines with Express

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

Some popular template engines that work with Express are [Pug](#), [Mustache](#), and [EJS](#). The [Express application generator](#) uses [Jade](#) as its default, but it also supports several others.

See [Template Engines \(Express wiki\)](#) for a list of template engines you can use with Express. See also [Comparing JavaScript Templating Engines: Jade, Mustache, Dust and More](#).

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

To render template files, set the following [application setting properties](#), set in `app.js` in the default app created by the generator:

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

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

```
$ npm install pug --save
```

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

Some template engines do not follow this convention. The [Consolidate.js](#) library follows this convention by mapping all of the popular Node.js template engines, and therefore works seamlessly within Express.

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

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

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

```
html
  head
    title= title
  body
    h1= message
```

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

```
app.get('/', (req, res) => {  
  res.render('index', { title: 'Hey', message: 'Hello there!' })  
})
```

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

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

To learn more about how template engines work in Express, see: ["Developing template engines for Express"](#).

# Writing middleware for use in Express apps

## Overview


*Middleware* functions are functions that have access to the `request object` (`req`), the `response object` (`res`), and the `next` function in the application's request-response cycle. The `next` function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.

Middleware functions can perform the following tasks:

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

If the current middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function. Otherwise, the request will be left hanging.

The following figure shows the elements of a middleware function call:

	HTTP method for which the middleware function applies.
	Path (route) for which the middleware function applies.
	The middleware function.
	Callback argument to the middleware function, called "next" by convention.
	HTTP <code>response</code> argument to the middleware function, called "res" by convention.
	HTTP <code>request</code> argument to the middleware function, called "req" by convention.

Starting with Express 5, middleware functions that return a Promise will call `next(value)` when they reject or throw an error. `next` will be called with either the rejected value or the thrown Error.

## Example

Here is an example of a simple "Hello World" Express application. The remainder of this article will define and add three middleware functions to the application: one called `myLogger` that prints a simple log message, one called `requestTime` that displays the timestamp of the HTTP request, and one called `validate-Cookies` that validates incoming cookies.

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

app.get('/', (req, res) => {
  res.send('Hello World!')
```

```
})
app.listen(3000)
```

## Middleware function myLogger

Here is a simple example of a middleware function called "myLogger". This function just prints "LOGGED" when a request to the app passes through it. The middleware function is assigned to a variable named `myLogger`.

```
const myLogger = function (req, res, next) {
  console.log('LOGGED')
  next()
}
```

Notice the call above to `next()`. Calling this function invokes the next middleware function in the app. The `next()` function is not a part of the Node.js or Express API, but is the third argument that is passed to the middleware function. The `next()` function could be named anything, but by convention it is always named "next". To avoid confusion, always use this convention.

To load the middleware function, call `app.use()`, specifying the middleware function. For example, the following code loads the `myLogger` middleware function before the route to the root path (/).

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

const myLogger = function (req, res, next) {
  console.log('LOGGED')
  next()
}

app.use(myLogger)

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(3000)
```

Every time the app receives a request, it prints the message "LOGGED" to the terminal.

The order of middleware loading is important: middleware functions that are loaded first are also executed first.

If `myLogger` is loaded after the route to the root path, the request never reaches it and the app doesn't print "LOGGED", because the route handler of the root path terminates the request-response cycle.

The middleware function `myLogger` simply prints a message, then passes on the request to the next middleware function in the stack by calling the `next()` function.

## Middleware function requestTime

Next, we'll create a middleware function called "requestTime" and add a property called `requestTime` to the request object.

```
const requestTime = function (req, res, next) {
  req.requestTime = Date.now()
  next()
}
```

The app now uses the `requestTime` middleware function. Also, the callback function of the root path route uses the property that the middleware function adds to `req` (the request object).

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

const requestTime = function (req, res, next) {
  req.requestTime = Date.now()
  next()
}

app.use(requestTime)

app.get('/', (req, res) => {
  let responseText = 'Hello World!<br>'
  responseText += `<small>Requested at: ${req.requestTime}</small>`
  res.send(responseText)
})

app.listen(3000)
```

When you make a request to the root of the app, the app now displays the timestamp of your request in the browser.

## Middleware function validateCookies

Finally, we'll create a middleware function that validates incoming cookies and sends a 400 response if cookies are invalid.

Here's an example function that validates cookies with an external async service.

```
async function cookieValidator (cookies) {
  try {
    await externallyValidateCookie(cookies.testCookie)
  } catch {
    throw new Error('Invalid cookies')
  }
}
```

Here we use the `cookie-parser` middleware to parse incoming cookies off the `req` object and pass them to our `cookieValidator` function. The `validateCookies` middleware returns a Promise that upon rejection will automatically trigger our error handler.

```

const express = require('express')
const cookieParser = require('cookie-parser')
const cookieValidator = require('./cookieValidator')

const app = express()

async function validateCookies (req, res, next) {
  await cookieValidator(req.cookies)
  next()
}

app.use(cookieParser())
app.use(validateCookies)

// error handler
app.use((err, req, res, next) => {
  res.status(400).send(err.message)
})

app.listen(3000)

```

Note how `next()` is called after `await cookieValidator(req.cookies)`. This ensures that if `cookieValidator` resolves, the next middleware in the stack will get called. If you pass anything to the `next()` function (except the string `'route'` or `'router'`), Express regards the current request as being an error and will skip any remaining non-error handling routing and middleware functions.

Because you have access to the request object, the response object, the next middleware function in the stack, and the whole Node.js API, the possibilities with middleware functions are endless.

For more information about Express middleware, see: [Using Express middleware](#).

## Configurable middleware

If you need your middleware to be configurable, export a function which accepts an options object or other parameters, which, then returns the middleware implementation based on the input parameters.

File: `my-middleware.js`

```

module.exports = function (options) {
  return function (req, res, next) {
    // Implement the middleware function based on the options object
    next()
  }
}

```

The middleware can now be used as shown below.

```

const mw = require('./my-middleware.js')
app.use(mw({ option1: '1', option2: '2' }))

```

Refer to [cookie-session](#) and [compression](#) for examples of configurable middleware.

---

[Go to TOC](#)

# Community

## Technical committee

The Express technical committee meets online every two weeks (as needed) to discuss development and maintenance of Express, and other issues relevant to the Express project. Each meeting is typically announced in an [expressjs/discussions issue](#) with a link to join or view the meeting, which is open to all observers.

The meetings are recorded; for a list of the recordings, see the [Express.js YouTube channel](#).

Members of the Express technical committee are:

### Active:

- [@blakeembrey](#) - Blake Embrey
- [@crandmck](#) - Rand McKinney
- [@dougwilson](#) - Douglas Wilson
- [@LinusU](#) - Linus Unnebäck
- [@wesleytodd](#) - Wes Todd

### Inactive:

- [@hacksparrow](#) - Hage Yaapa
- [@jonathanong](#) - jongleberry
- [@niftylettuce](#) - niftylettuce
- [@troygoode](#) - Troy Goode

## Express is made of many modules

Our vibrant community has created a large variety of extensions, [middleware modules](#) and [higher-level frameworks](#).

Additionally, the Express community maintains modules in these two GitHub orgs:

- [jshttp](#) modules providing useful utility functions; see [Utility modules](#).
- [pillarjs](#): low-level modules that Express uses internally.

To keep up with what is going on in the whole community, check out the [ExpressJS StatusBoard](#).

## Gitter

The [expressjs/express chatroom](#) is great place for developers interested in the everyday discussions related to Express.



## Issues

If you've come across what you think is a bug, or just want to make a feature request open a ticket in the [issue queue](#).

## Examples

View dozens of Express application [examples](#) in the repository covering everything from API design and authentication to template engine integration.

## Mailing List

Join over 2000 Express users or browse over 5000 discussions in the [Google Group](#).

## IRC channel

Hundreds of developers idle in #express on freenode every day. If you have questions about the framework, jump in for quick feedback.

# Companies using Express in production



---

[Go to TOC](#)

# Contributing to Express

Express and the other projects in the [expressjs organization on GitHub](#) are projects of the [Node.js Foundation](#). These projects are governed under the general policies and guidelines of the Node.js Foundation along with the additional guidelines below.

- [Technical committee](#)
- [Community contributing guide](#)
- [Collaborator's guide](#)
- [Security policies and procedures](#)

## Technical committee

The Express technical committee consists of active project members, and guides development and maintenance of the Express project. For more information, see [Express Community - Technical committee](#).

## Community contributing guide

The goal of this document is to create a contribution process that:

- Encourages new contributions.
- Encourages contributors to remain involved.
- Avoids unnecessary processes and bureaucracy whenever possible.
- Creates a transparent decision making process that makes it clear how contributors can be involved in decision making.

## Vocabulary

- A **Contributor** is any individual creating or commenting on an issue or pull request.
- A **Committer** is a subset of contributors who have been given write access to the repository.
- A **TC (Technical Committee)** is a group of committers representing the required technical expertise to resolve rare disputes.
- A **Triager** is a subset of contributors who have been given triage access to the repository.

## Logging Issues

Log an issue for any question or problem you might have. When in doubt, log an issue, and any additional policies about what to include will be provided in the responses. The only exception is security disclosures which should be sent privately.

Committers may direct you to another repository, ask for additional clarifications, and add appropriate metadata before the issue is addressed.

Please be courteous and respectful. Every participant is expected to follow the project's Code of Conduct.

## Contributions

Any change to resources in this repository must be through pull requests. This applies to all changes to documentation, code, binary files, etc. Even long term committers and TC members must use pull requests.

No pull request can be merged without being reviewed.

For non-trivial contributions, pull requests should sit for at least 36 hours to ensure that contributors in other timezones have time to review. Consideration should also be given to weekends and other holiday periods to ensure active committers all have reasonable time to become involved in the discussion and review process if they wish.

The default for each contribution is that it is accepted once no committer has an objection. During a review, committers may also request that a specific contributor who is most versed in a particular area gives a "LGTM" before the PR can be merged. There is no additional "sign off" process for contributions to land. Once all issues brought by committers are addressed it can be landed by any committer.

In the case of an objection being raised in a pull request by another committer, all involved committers should seek to arrive at a consensus by way of addressing concerns being expressed by discussion, compromise on the proposed change, or withdrawal of the proposed change.

If a contribution is controversial and committers cannot agree about how to get it to land or if it should land then it should be escalated to the TC. TC members should regularly discuss pending contributions in order to find a resolution. It is expected that only a small minority of issues be brought to the TC for resolution and that discussion and compromise among committers be the default resolution mechanism.

## Becoming a Triager

Anyone can become a triager! Read more about the process of being a triager in [the triage process document](#).

Open an issue in [expressjs/express](#) repo to request the triage role. State that you have read and agree to the [Code of Conduct](#) and details of the role.

Here is an example issue content you can copy and paste:

```
Title: Request triager role for <your GitHub username>

I have read and understood the project's Code of Conduct.
I also have read and understood the process and best practices around Express triaging.

I request for a triager role for the following GitHub organizations:

jshttp
pillarjs
express
```

Once you have opened your issue, a member of the TC will add you to the [triage](#) team in the organizations requested. They will then close the issue.

Happy triaging!

## Becoming a Committer

All contributors who land a non-trivial contribution should be on-boarded in a timely manner, and added as a committer, and be given write access to the repository.

Committers are expected to follow this policy and continue to send pull requests, go through proper review, and have other committers merge their pull requests.

## TC Process

The TC uses a "consensus seeking" process for issues that are escalated to the TC. The group tries to find a resolution that has no open objections among TC members. If a consensus cannot be reached that has no objections then a majority wins vote is called. It is also expected that the majority of decisions made by the TC are via a consensus seeking process and that voting is only used as a last-resort.

Resolution may involve returning the issue to committers with suggestions on how to move forward towards a consensus. It is not expected that a meeting of the TC will resolve all issues on its agenda during that meeting and may prefer to continue the discussion happening among the committers.

Members can be added to the TC at any time. Any committer can nominate another committer to the TC and the TC uses its standard consensus seeking process to evaluate whether or not to add this new member. Members who do not participate consistently at the level of a majority of the other members are expected to resign.

## Collaborator's guide

### Website Issues

Open issues for the `expressjs.com` website in <https://github.com/expressjs/expressjs.com>.

### PRs and Code contributions

- Tests must pass.
- Follow the [JavaScript Standard Style](#) and `npm run lint`.
- If you fix a bug, add a test.

### Branches

Use the `master` branch for bug fixes or minor work that is intended for the current release stream.

Use the correspondingly named branch, e.g. `5.0`, for anything intended for a future release of Express.

### Steps for contributing

1. [Create an issue](#) for the bug you want to fix or the feature that you want to add.
2. Create your own [fork](#) on GitHub, then checkout your fork.

3. Write your code in your local copy. It's good practice to create a branch for each new issue you work on, although not compulsory.
4. To run the test suite, first install the dependencies by running `npm install`, then run `npm test`.
5. Ensure your code is linted by running `npm run lint` -- fix any issue you see listed.
6. If the tests pass, you can commit your changes to your fork and then create a pull request from there. Make sure to reference your issue from the pull request comments by including the issue number e.g. `#123`.

## Issues which are questions

We will typically close any vague issues or questions that are specific to some app you are writing. Please double check the docs and other references before being trigger happy with posting a question issue.

Things that will help get your question issue looked at:

- Full and runnable JS code.
- Clear description of the problem or unexpected behavior.
- Clear description of the expected result.
- Steps you have taken to debug it yourself.

If you post a question and do not outline the above items or make it easy for us to understand and reproduce your issue, it will be closed.

## Security Policies and Procedures

This document outlines security procedures and general policies for the Express project.

- [Reporting a Bug](#)
- [Disclosure Policy](#)
- [Comments on this Policy](#)

### Reporting a Bug

The Express team and community take all security bugs in Express seriously. Thank you for improving the security of Express. We appreciate your efforts and responsible disclosure and will make every effort to acknowledge your contributions.

Report security bugs by emailing the lead maintainer in the Readme.md file.

To ensure the timely response to your report, please ensure that the entirety of the report is contained within the email body and not solely behind a web link or an attachment.

The lead maintainer will acknowledge your email within 48 hours, and will send a more detailed response within 48 hours indicating the next steps in handling your report. After the initial reply to your report, the security team will endeavor to keep you informed of the progress towards a fix and full announcement, and may ask for additional information or guidance.

Report security bugs in third-party modules to the person or team maintaining the module.

## Disclosure Policy

When the security team receives a security bug report, they will assign it to a primary handler. This person will coordinate the fix and release process, involving the following steps:

- Confirm the problem and determine the affected versions.
- Audit code to find any potential similar problems.
- Prepare fixes for all releases still under maintenance. These fixes will be released as fast as possible to npm.

## Comments on this Policy

If you have suggestions on how this process could be improved please submit a pull request.

# Frameworks built on Express

Several popular Node.js frameworks are built on Express:

- **Feathers**: Build prototypes in minutes and production ready real-time apps in days.
- **ItemsAPI**: Search backend for web and mobile applications built on Express and Elasticsearch.
- **KeystoneJS**: Website and API Application Framework / CMS with an auto-generated React.js Admin UI.
- **Poet**: Lightweight Markdown Blog Engine with instant pagination, tag and category views.
- **Kraken**: Secure and scalable layer that extends Express by providing structure and convention.
- **LoopBack**: Highly-extensible, open-source Node.js framework for quickly creating dynamic end-to-end REST APIs.
- **Sails**: MVC framework for Node.js for building practical, production-ready apps.
- **Hydra-Express**: Hydra-Express is a light-weight library which facilitates building Node.js Microservices using ExpressJS.
- **Blueprint**: a SOLID framework for building APIs and backend services
- **Loomotive**: Powerful MVC web framework for Node.js from the maker of Passport.js
- **graphql-yoga**: Fully-featured, yet simple and lightweight GraphQL server
- **Express Gateway**: Fully-featured and extensible API Gateway using Express as foundation
- **Dinoloop**: Rest API Application Framework powered by typescript with dependency injection
- **Kites**: Template-based Web Application Framework
- **FoalTS**: Elegant and all-inclusive Node.Js web framework based on TypeScript.
- **NestJs**: A progressive Node.js framework for building efficient, scalable, and enterprise-grade server-side applications on top of TypeScript & JavaScript (ES6, ES7, ES8)
- **Expressive Tea**: A Small framework for building modulable, clean, fast and descriptive server-side applications with Typescript and Express out of the box.

---

[Go to TOC](#)



# Glossary

## application

In general, one or more programs that are designed to carry out operations for a specific purpose. In the context of Express, a program that uses the Express API running on the Node.js platform. Might also refer to an [app object](#).

## API

Application programming interface. Spell out the abbreviation when it is first used.

## Express

A fast, un-opinionated, minimalist web framework for Node.js applications. In general, "Express" is preferred to "Express.js," though the latter is acceptable.

## libuv

A multi-platform support library which focuses on asynchronous I/O, primarily developed for use by Node.js.

## middleware

A function that is invoked by the Express routing layer before the final request handler, and thus sits in the middle between a raw request and the final intended route. A few fine points of terminology around middleware:

- `var foo = require('middleware')` is called *requiring* or *using* a Node.js module. Then the statement `var mw = foo()` typically returns the middleware.
- `app.use(mw)` is called *adding the middleware to the global processing stack*.
- `app.get('/foo', mw, function (req, res) { ... })` is called *adding the middleware to the "GET /foo" processing stack*.

## Node.js

A software platform that is used to build scalable network applications. Node.js uses JavaScript as its scripting language, and achieves high throughput via non-blocking I/O and a single-threaded event loop. See [nodejs.org](https://nodejs.org). **Usage note:** Initially, "Node.js," thereafter "Node".

## open-source, open source

When used as an adjective, hyphenate; for example: "This is open-source software." See [Open-source software on Wikipedia](#). Note: Although it is common not to hyphenate this term, we are using the standard English rules for hyphenating a compound adjective.

## request

An HTTP request. A client submits an HTTP request message to a server, which returns a response. The request must use one of several [request methods](#) such as GET, POST, and so on.

## response

An HTTP response. A server returns an HTTP response message to the client. The response contains completion status information about the request and might also contain requested content in its message body.

## route

Part of a URL that identifies a resource. For example, in `http://foo.com/products/id`, `"/products/id"` is the route.

## router

See [router](#) in the API reference.

# Additional learning

## Books

Here are a few of the many books on Express:

- [Express.js in Action](#), Manning Publications, April 2016.
- [Getting MEAN with Mongo, Express, Angular, and Node](#), Manning Publications, early 2015.
- [Getting MEAN with Mongo, Express, Angular, and Node, Second Edition](#), Manning Publications, April 2017.
- [Pro Express.js: Master Express.js: The Node.js Framework For Your Web Development](#), Apress, December 2014.
- [Mastering Web Application Development with Express](#), Packt Publishing, September 2014.
- [Web Development with Node and Express](#), O'Reilly Media, July 2014.
- [Node.js in Action](#), Manning Publications, October 2013.
- [Express Web Application Development](#), Packt Publishing, June 2013.
- [express - Middleware für node.js](#), texxtoor, September 2015. In deutscher Sprache / in German language
- [JADE - die Template Engine für node.js](#), texxtoor, September 2015. In deutscher Sprache / in German language
- [Node Web Development, 2nd edition](#), Packt Publishing, August 2013
- [Builder Book: Build a Full Stack JavaScript Web App from Scratch](#), self-published, February 2018.
- [MERN Quick Start Guide](#), Packt Publishing, May 2018
- [Functional Design Patterns for Express.js](#), self-published, June 2019.
- [SaaS Boilerplate Book: Build a Production-Ready SaaS Web App from Scratch](#), self-published, August 2020.

## Add your book here!

[Edit the Markdown file](#) and add a link to your book, then submit a pull request (GitHub login required). Follow the format of the above listings.

## Blogs

- [StrongLoop Blog: Express category](#)
- [Hage Yaapa's Blog: Express category](#)
- [Codeforgeek Blog: Express category](#)
- [Baboon Blog: Express category](#) (Persian language)
- [Techforgeek Blog: Express category](#)
- [RoseHosting.com Blog: Express tag](#)
- [ThisHosting.Rocks: Express tag](#)
- [Code with Hugo blog: Express tag](#)
- [Dev.to blog: Express category](#)

- [LinuxStans Blog: Express tag](#)
- [ButterCMS blog: Express category](#)
- [Traveling Coderman Blog: Architecture Series](#)

## Add your blog here!

[Edit the Markdown file](#) and add a link to your blog, then submit a pull request (GitHub login required). Follow the format of the above listings.

## The DEV community

[DEV's express tag](#) is a place to share Express projects, articles and tutorials as well as start discussions and ask for feedback on Express-related topics. Developers of all skill-levels are welcome to take part.

## Video tutorials

- [Learning ExpressJS: Express category](#)
- [Learn Express.js in 14 days](#) - Practice Projects included

---

[Go to TOC](#)

# Express middleware

The Express middleware modules listed here are maintained by the [Expressjs team](#).

Middleware module	Description	Replaces built-in function (Express 3)
<a href="#">body-parser</a>	Parse HTTP request body. See also: <a href="#">body</a> , <a href="#">co-body</a> , and <a href="#">raw-body</a> .	<code>express.bodyParser</code>
<a href="#">compression</a>	Compress HTTP responses.	<code>express.compress</code>
<a href="#">connect-rid</a>	Generate unique request ID.	NA
<a href="#">cookie-parser</a>	Parse cookie header and populate <code>req.cookies</code> . See also <a href="#">cookies</a> and <a href="#">keygrip</a> .	<code>express.cookieParser</code>
<a href="#">cookie-session</a>	Establish cookie-based sessions.	<code>express.cookieSession</code>
<a href="#">cors</a>	Enable cross-origin resource sharing (CORS) with various options.	NA
<a href="#">errorhandler</a>	Development error-handling/debugging.	<code>express.errorHandler</code>
<a href="#">method-override</a>	Override HTTP methods using header.	<code>express.methodOverride</code>
<a href="#">morgan</a>	HTTP request logger.	<code>express.logger</code>
<a href="#">multer</a>	Handle multi-part form data.	<code>express.bodyParser</code>
<a href="#">response-time</a>	Record HTTP response time.	<code>express.responseTime</code>
<a href="#">serve-favicon</a>	Serve a favicon.	<code>express.favicon</code>
<a href="#">serve-index</a>	Serve directory listing for a given path.	<code>express.directory</code>
<a href="#">serve-static</a>	Serve static files.	<code>express.static</code>
<a href="#">session</a>	Establish server-based sessions (development only).	<code>express.session</code>
<a href="#">timeout</a>	Set a timeout period for HTTP request processing.	<code>express.timeout</code>
<a href="#">vhost</a>	Create virtual domains.	<code>express.vhost</code>

## Additional middleware modules

These are some additional popular middleware modules.

Middleware module	Description
<a href="#">cls-rtracer</a>	Middleware for CLS-based request id generation. An out-of-the-box solution for adding request ids into your logs.
<a href="#">connect-image-optimus</a>	Optimize image serving. Switches images to <code>.webp</code> or <code>.jxr</code> , if possible.

Middleware module	Description
<a href="#">express-debug</a>	Development tool that adds information about template variables (locals), current session, and so on.
<a href="#">express-partial-response</a>	Filters out parts of JSON responses based on the <code>fields</code> query-string; by using Google API's Partial Response.
<a href="#">express-simple-cdn</a>	Use a CDN for static assets, with multiple host support.
<a href="#">express-slash</a>	Handles routes with and without trailing slashes.
<a href="#">express-stormpath</a>	User storage, authentication, authorization, SSO, and data security.
<a href="#">express-uncapitalize</a>	Redirects HTTP requests containing uppercase to a canonical lowercase form.
<a href="#">helmet</a>	Helps secure your apps by setting various HTTP headers.
<a href="#">join-io</a>	Joins files on the fly to reduce the requests count.
<a href="#">passport</a>	Authentication using "strategies" such as OAuth, OpenID and many others. See <a href="http://passportjs.org/">http://passportjs.org/</a> for more information.
<a href="#">static-expiry</a>	Fingerprint URLs or caching headers for static assets.
<a href="#">view-helpers</a>	Common helper methods for views.
<a href="#">sriracha-admin</a>	Dynamically generate an admin site for Mongoose.

For more middleware modules, see [http-framework](#).

# Open source projects using Express

Some open-source projects that use Express:

- **Builder Book**: Open source web app to publish documentation or books. Built with React, Material-UI, Next, Express, Mongoose, MongoDB.
- **SaaS Boilerplate**: Open source web app to build your own SaaS product. Built with React, Material-UI, Next, MobX, Express, Mongoose, MongoDB, Typescript.
- **BitMidi**: Open source web app powered by Express. BitMidi is a historical archive of MIDI files from the early web era. It uses the latest modern web technology including WebAssembly and Web Audio to bring MIDI back to life. ([source code](#))

# Template engines

{% include warning.html content="The packages listed below may be outdated, no longer maintained or even broken. Listing here does not constitute an endorsement or recommendation from the Expressjs project team. Use at your own risk." %}

These template engines work "out-of-the-box" with Express:

- **Pug**: Haml-inspired template engine (formerly Jade).
- **Haml.js**: Haml implementation.
- **EJS**: Embedded JavaScript template engine.
- **hbs**: Adapter for Handlebars.js, an extension of Mustache.js template engine.
- **Squirrelly**: Blazing-fast template engine that supports partials, helpers, custom tags, filters, and caching. Not white-space sensitive, works with any language.
- **Eta**: Super-fast lightweight embedded JS template engine. Supports custom delimiters, async, white-space control, partials, caching, plugins.
- **combyne.js**: A template engine that hopefully works the way you'd expect.
- **Nunjucks**: Inspired by jinja/twig.
- **marko**: A fast and lightweight HTML-based templating engine that compiles templates to CommonJS modules and supports streaming, async rendering and custom tags. (Renders directly to the HTTP response stream).
- **whiskers**: Small, fast, mustachioed.
- **Blade**: HTML Template Compiler, inspired by Jade & Haml.
- **Haml-Coffee**: Haml templates where you can write inline CoffeeScript.
- **express-hbs**: Handlebars with layouts, partials and blocks for express 3 from Barc.
- **express-handlebars**: A Handlebars view engine for Express which doesn't suck.
- **express-views-dom**: A DOM view engine for Express.
- **rivets-server**: Render Rivets.js templates on the server.
- **LiquidJS**: A simple, expressive and safe template engine.
- **express-tl**: A template-literal engine implementation for Express.
- **Twing**: First-class Twig engine for Node.js.
- **Sprightly**: A very light-weight JS template engine (45 lines of code), that consists of all the bare-bones features that you want to see in a template engine.
- **html-express-js**: A small template engine for those that want to just serve static or dynamic HTML pages using native JavaScript.

The [Consolidate.js](#) library unifies the APIs of these template engines to a single Express-compatible API.

## Add your template engine here!

[Edit the Markdown file](#) and add a link to your project, then submit a pull request (GitHub login required). Follow the format of the above listings.

---

[Go to TOC](#)



## Express utility functions

The [pillarjs](#) GitHub organization contains a number of modules for utility functions that may be generally useful.

Utility modules	Description
<a href="#">cookies</a>	Get and set HTTP(S) cookies that can be signed to prevent tampering, using Keygrip. Can be used with the Node.js HTTP library or as Express middleware.
<a href="#">csrf</a>	Contains the logic behind CSRF token creation and verification. Use this module to create custom CSRF middleware.
<a href="#">finalhandler</a>	Function to invoke as the final step to respond to HTTP request.
<a href="#">parseurl</a>	Parse a URL with caching.
<a href="#">path-match</a>	Thin wrapper around <a href="#">path-to-regexp</a> to make extracting parameter names easier.
<a href="#">path-to-regexp</a>	Turn an Express-style path string such as <code>`/user/:name`</code> into a regular expression.
<a href="#">resolve-path</a>	Resolves a relative path against a root path with validation.
<a href="#">router</a>	Simple middleware-style router.
<a href="#">routington</a>	Trie-based URL router for defining and matching URLs.
<a href="#">send</a>	Library for streaming files as a HTTP response, with support for partial responses (ranges), conditional-GET negotiation, and granular events.
<a href="#">templation</a>	View system similar to <code>res.render()</code> inspired by <a href="#">co-views</a> and <a href="#">consolidate.js</a> .

For additional low-level HTTP-related modules, see [jshttp](#) .

# Basic routing

*Routing* refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

Each route can have one or more handler functions, which are executed when the route is matched.

Route definition takes the following structure:

```
app.METHOD(PATH, HANDLER)
```

Where:

- `app` is an instance of `express`.
- `METHOD` is an [HTTP request method](#), in lowercase.
- `PATH` is a path on the server.
- `HANDLER` is the function executed when the route is matched.

This tutorial assumes that an instance of `express` named `app` is created and the server is running. If you are not familiar with creating an app and starting it, see the [\[Hello world example\]\(../starter/hello-world.html\)](#).

The following examples illustrate defining simple routes.

Respond with `Hello World!` on the homepage:

```
app.get('/', (req, res) => {
  res.send('Hello World!')
})
```

Respond to POST request on the root route (`/`), the application's home page:

```
app.post('/', (req, res) => {
  res.send('Got a POST request')
})
```

Respond to a PUT request to the `/user` route:

```
app.put('/user', (req, res) => {
  res.send('Got a PUT request at /user')
})
```

Respond to a DELETE request to the `/user` route:

```
app.delete('/user', (req, res) => {
  res.send('Got a DELETE request at /user')
})
```

For more details about routing, see the [routing guide](#).

**Previous: Express application generator in Express**

**Next: Serving static files**

```
{% capture examples %}{% include readmes/express-master/examples.md %}{% endcapture %}
replace: "){(.", "{(https://github.com/expressjs/express/tree/master/examples"
      {
        examples
      }
```

## Additional examples

These are some additional examples with more extensive integrations.

- [prisma-express-graphql](#) - GraphQL API with `express-graphql` using [Prisma](#) as an ORM
- [prisma-fullstack](#) - Fullstack app with Next.js using [Prisma](#) as an ORM
- [prisma-rest-api-js](#) - REST API with Express in JavaScript using [Prisma](#) as an ORM
- [prisma-rest-api-ts](#) - REST API with Express in TypeScript using [Prisma](#) as an ORM

**Previous: Static Files**      **Next: FAQ**

---

[Go to TOC](#)

# FAQ

## How should I structure my application?

There is no definitive answer to this question. The answer depends on the scale of your application and the team that is involved. To be as flexible as possible, Express makes no assumptions in terms of structure.

Routes and other application-specific logic can live in as many files as you wish, in any directory structure you prefer. View the following examples for inspiration:

- [Route listings](#)
- [Route map](#)
- [MVC style controllers](#)

Also, there are third-party extensions for Express, which simplify some of these patterns:

- [Resourceful routing](#)

## How do I define models?

Express has no notion of a database. This concept is left up to third-party Node modules, allowing you to interface with nearly any database.

See [LoopBack](#) for an Express-based framework that is centered around models.

## How can I authenticate users?

Authentication is another opinionated area that Express does not venture into. You may use any authentication scheme you wish. For a simple username / password scheme, see [this example](#).

## Which template engines does Express support?

Express supports any template engine that conforms with the `(path, locals, callback)` signature. To normalize template engine interfaces and caching, see the [consolidate.js](#) project for support. Unlisted template engines might still support the Express signature.

For more information, see [Using template engines with Express](#).

## How do I handle 404 responses?

In Express, 404 responses are not the result of an error, so the error-handler middleware will not capture them. This behavior is because a 404 response simply indicates the absence of additional work to do; in other words, Express has executed all middleware functions and routes, and found that none of them responded. All you need to do is add a middleware function at the very bottom of the stack (below all other functions) to handle a 404 response:

```
app.use((req, res, next) => {  
  res.status(404).send("Sorry can't find that!")  
})
```

Add routes dynamically at runtime on an instance of `express.Router()` so the routes are not superseded by a middleware function.

## How do I setup an error handler?

You define error-handling middleware in the same way as other middleware, except with four arguments instead of three; specifically with the signature `(err, req, res, next) :`

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

For more information, see [Error handling](#).

## How do I render plain HTML?

You don't! There's no need to "render" HTML with the `res.render()` function. If you have a specific file, use the `res.sendFile()` function. If you are serving many assets from a directory, use the `express.static()` middleware function.

### Previous: [More examples](#)

# Express application generator

Use the application generator tool, `express-generator`, to quickly create an application skeleton.

You can run the application generator with the `npx` command (available in Node.js 8.2.0).

```
$ npx express-generator
```

For earlier Node versions, install the application generator as a global npm package and then launch it:

```
$ npm install -g express-generator
$ express
```

Display the command options with the `-h` option:

```
$ express -h

Usage: express [options] [dir]

Options:
  -h, --help            output usage information
  --version             output the version number
  -e, --ejs             add ejs engine support
  --hbs                add handlebars engine support
  --pug                add pug engine support
  -H, --hogan           add hogan.js engine support
  --no-view            generate without view engine
  -v, --view <engine> add view <engine> support (ejs|hbs|hjs|jade|pug|twig|vash)
                      (defaults to jade)
  -c, --css <engine>  add stylesheet <engine> support (less|stylus|compass|sass)
                      (defaults to plain css)
  --git                add .gitignore
  -f, --force          force on non-empty directory
```

For example, the following creates an Express app named *myapp*. The app will be created in a folder named *myapp* in the current working directory and the view engine will be set to [Pug](#):

```
$ express --view=pug myapp

create : myapp
create : myapp/package.json
create : myapp/app.js
create : myapp/public
create : myapp/public/javascripts
create : myapp/public/images
create : myapp/routes
create : myapp/routes/index.js
create : myapp/routes/users.js
create : myapp/public/stylesheet
create : myapp/public/stylesheet/style.css
create : myapp/views
create : myapp/views/index.pug
create : myapp/views/layout.pug
```

```
create : myapp/views/error.pug
create : myapp/bin
create : myapp/bin/www
```

Then install dependencies:

```
$ cd myapp
$ npm install
```

On MacOS or Linux, run the app with this command:

```
$ DEBUG=myapp:* npm start
```

On Windows Command Prompt, use this command:

```
> set DEBUG=myapp:* & npm start
```

On Windows PowerShell, use this command:

```
PS> $env:DEBUG='myapp:*'; npm start
```

Then load <http://localhost:3000/> in your browser to access the app.

The generated app has the following directory structure:

```
.
├── app.js
├── bin
│   └── www
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   ├── stylesheets
│   └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.pug
    ├── index.pug
    └── layout.pug
```

7 directories, 9 files

The app structure created by the generator is just one of many ways to structure Express apps. Feel free to use this structure or modify it to best suit your needs.

**Previous: Hello World**    **Next: Basic routing**

---

[Go to TOC](#)



# Hello world example

Embedded below is essentially the simplest Express app you can create. It is a single file app — not what you'd get if you use the [Express generator](../starter/generator.html), which creates the scaffolding for a full app with numerous JavaScript files, Jade templates, and sub-directories for various purposes.

```
const express = require('express')
const app = express()
const port = 3000
```

```
app.get('/', (req, res) => {
  res.send('Hello World!')
})
```

```
app.listen(port, () => { console.log(`Example app listening on port ${port}`) })
```

This app starts a server and listens on port 3000 for connections. The app responds with "Hello World!" for requests to the root URL (`/`) or *route*. For every other path, it will respond with a **404 Not Found**.

The example above is actually a working server: Go ahead and click on the URL shown. You'll get a response, with real-time logs on the page, and any changes you make will be reflected in real time. This is powered by [RunKit](#), which provides an interactive JavaScript playground connected to a complete Node environment that runs in your web browser. Below are instructions for running the same app on your local machine.

RunKit is a third-party service not affiliated with the Express project.

## Running Locally

First create a directory named `myapp`, change to it and run `npm init`. Then install `express` as a dependency, as per the [installation guide](#).

In the `myapp` directory, create a file named `app.js` and copy in the code from the example above.

The ``req`` (request) and ``res`` (response) are the exact same objects that Node provides, so you can invoke ``req.pipe()``, ``req.on('data', callback)``, and anything else you would do without Express involved.

Run the app with the following command:

```
$ node app.js
```

Then, load `http://localhost:3000/` in a browser to see the output.

[Previous: Installing](#)   [Next: Express Generator](#)

---

[Go to TOC](#)

# Installing

Assuming you've already installed [Node.js](#), create a directory to hold your application, and make that your working directory.

```
$ mkdir myapp  
$ cd myapp
```

Use the `npm init` command to create a `package.json` file for your application. For more information on how `package.json` works, see [Specifics of npm's package.json handling](#).

```
$ npm init
```

This command prompts you for a number of things, such as the name and version of your application. For now, you can simply hit RETURN to accept the defaults for most of them, with the following exception:

```
entry point: (index.js)
```

Enter `app.js`, or whatever you want the name of the main file to be. If you want it to be `index.js`, hit RETURN to accept the suggested default file name.

Now install Express in the `myapp` directory and save it in the dependencies list. For example:

```
$ npm install express
```

To install Express temporarily and not add it to the dependencies list:

```
$ npm install express --no-save
```

By default with version npm 5.0+ `npm install` adds the module to the `dependencies` list in the `package.json` file; with earlier versions of npm, you must specify the `--save` option explicitly. Then, afterwards, running `npm install` in the app directory will automatically install modules in the dependencies list.

## Next: Hello World

---

[Go to TOC](#)

# Serving static files in Express

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

The function signature is:

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

The `root` argument specifies the root directory from which to serve static assets. For more information on the `options` argument, see [express.static](#).

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

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

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

```
http://localhost:3000/images/kitten.jpg
http://localhost:3000/css/style.css
http://localhost:3000/js/app.js
http://localhost:3000/images/bg.png
http://localhost:3000/hello.html
```

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

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

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

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

NOTE: For best results, [\[use a reverse proxy\]\(../{{page.lang}}/advanced/best-practice-performance.html#use-a-reverse-proxy\)](#) cache to improve performance of serving static assets.

To create a virtual path prefix (where the path does not actually exist in the file system) for files that are served by the `express.static` function, [specify a mount path](#) for the static directory, as shown below:

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

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

```
http://localhost:3000/static/images/kitten.jpg
http://localhost:3000/static/css/style.css
http://localhost:3000/static/js/app.js
http://localhost:3000/static/images/bg.png
http://localhost:3000/static/hello.html
```

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

```
const path = require('path')
app.use('/static', express.static(path.join(__dirname, 'public')))
```

For more details about the `serve-static` function and its options, see [serve-static](#).

**Previous: Basic Routing**    **Next: More examples**

# Colophon

This book is created by using the following sources:

- Express - English
- GitHub source: [expressjs/expressjs.com](https://github.com/expressjs/expressjs.com)
- Created: 2022-12-03
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/>
- By: @shinokada
- Viewer: <https://read-html-download-pdf.vercel.app/>
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>
- Viewer repo: <https://github.com/shinokada/read-html-download-pdf>