the

Express

HANDBOOK

FLAVIO COPES

Table of Contents

The Express Handbook

The Express Handbook follows the 80/20 rule: learn in 20% of the time the 80% of a topic.

I find this approach gives a well-rounded overview. This book does not try to cover everything under the sun related to Express. If you think some specific topic should be included, tell me.

You can reach me on Twitter @flaviocopes.

I hope the contents of this book will help you achieve what you want: **learn the basics Express**.

This book is written by Flavio. I **publish web development tutorials** every day on my website flaviocopes.com.

Enjoy!

Request parameters

A handy reference to all the request object properties and how to use them

Request parameters

I mentioned how the Request object holds all the HTTP request information.

These are the main properties you'll likely use:

Property	Description
.app	holds a reference to the Express app object
.baseUrl	the base path on which the app responds
.body	contains the data submitted in the request body (must be parsed and populated manually before you can access it)
.cookies	contains the cookies sent by the request (needs the `cookie-parser` middleware)
.hostname	the hostname as defined in the Host HTTP header value
.ip	the client IP
.method	the HTTP method used
.params	the route named parameters
.path	the URL path
.protocol	the request protocol
.query	an object containing all the query strings used in the request
.secure	true if the request is secure (uses HTTPS)
.signedCookies	contains the signed cookies sent by the request (needs the `cookie-parser` middleware)
.xhr	true if the request is an XMLHttpRequest

How to retrieve the GET query string parameters using Express

The query string is the part that comes after the URL path, and starts with a question mark `?`.

Example:

```
?name=flavio
```

Multiple query parameters can be added using `&`:

```
?name=flavio&age=35
```

How do you get those query string values in Express?

Express makes it very easy by populating the `Request.query` object for us:

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

app.get('/', (req, res) => {
  console.log(req.query)
})

app.listen(8080)
```

This object is filled with a property for each query parameter.

If there are no query params, it's an empty object.

This makes it easy to iterate on it using the for...in loop:

```
for (const key in req.query) {
  console.log(key, req.query[key])
}
```

This will print the query property key and the value.

You can access single properties as well:

```
req.query.name //flavio
req.query.age //35
```

How to retrieve the POST query string parameters using Express

POST query parameters are sent by HTTP clients for example by forms, or when performing a POST request sending data.

How can you access this data?

If the data was sent as JSON, using `Content-Type: application/json`, you will use the `express.json()` middleware:

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

app.use(express.json())
```

If the data was sent using `Content-Type: application/x-www-form-urlencoded`, you will need to use the `express.urlencoded()` middleware:

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

app.use(express.urlencoded({
  extended: true
}))
```

In both cases you can access the data by referencing it from `Request.body`:

```
app.post('/form', (req, res) => {
  const name = req.body.name
})
```

> Note: older Express versions required the use of the `body-parser` module to process POST data. This is no longer the case as of Express 4.16 (released in September 2017) and later versions.

Sending a response

How to send a response back to the client using Express

In the Hello World example we used the `Response.send()` method to send a simple string as a response, and to close the connection:

```
(req, res) => res.send('Hello World!')
```

If you pass in a string, it sets the `Content-Type` header to `text/html`.

if you pass in an object or an array, it sets the `application/json` `Content-Type` header, and parses that parameter into JSON.

`send()` automatically sets the `Content-Length` HTTP response header.

`send()` also automatically closes the connection.

Use end() to send an empty response

An alternative way to send the response, without any body, it's by using the `Response.end()` method:

```
res.end()
```

Set the HTTP response status

Use the `Response.status()`:

```
res.status(404).end()
```

or

```
res.status(404).send('File not found')
```

`sendStatus()` is a shortcut:

```
res.sendStatus(200)
// === res.status(200).send('OK')

res.sendStatus(403)
// === res.status(403).send('Forbidden')

res.sendStatus(404)
// === res.status(404).send('Not Found')

res.sendStatus(500)
// === res.status(500).send('Internal Server Error')
```

Sending a JSON response

How to serve JSON data using the Node.js Express library

When you listen for connections on a route in Express, the callback function will be invoked on every network call with a Request object instance and a Response object instance.

Example:

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

Here we used the `Response.send()` method, which accepts any string.

You can send JSON to the client by using `Response.json()`, a useful method.

It accepts an object or array, and converts it to JSON before sending it:

```
res.json({ username: 'Flavio' })
```

Manage Cookies

How to use the `Response.cookie()` method to manipulate your cookies

Use the `Response.cookie()` method to manipulate your cookies.

Examples:

```
res.cookie('username', 'Flavio')
```

This method accepts a third parameter, which contains various options:

```
res.cookie('username', 'Flavio', { domain: '.flaviocopes.com', path: '/administrator', se

res.cookie('username', 'Flavio', { expires: new Date(Date.now() + 900000), httpOnly: true
```

The most useful parameters you can set are:

Value	Description
`domain`	The cookie domain name
`expires`	Set the cookie expiration date. If missing, or 0, the cookie is a session cookie
`httpOnly`	Set the cookie to be accessible only by the web server. See HttpOnly
`maxAge`	Set the expiry time relative to the current time, expressed in milliseconds
`path`	The cookie path. Defaults to '/'
`secure`	Marks the cookie HTTPS only
`signed`	Set the cookie to be signed
`sameSite`	Value of `SameSite`

A cookie can be cleared with:

```
res.clearCookie('username')
```

Work with HTTP headers

Learn how to access and change HTTP headers using Express

Access HTTP headers values from a request

You can access all the HTTP headers using the `Request.headers` property:

```
app.get('/', (req, res) => {
  console.log(req.headers)
})
```

Use the `Request.header()` method to access one individual request header's value:

```
app.get('/', (req, res) => {
  req.header('User-Agent')
})
```

Change any HTTP header value for a response

You can change any HTTP header value using `Response.set()` :

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

There is a shortcut for the Content-Type header, however:

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

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

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

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

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

Redirects

How to redirect to other pages server-side

Redirects are common in Web Development. You can create a redirect using the `Response.redirect()` method:

```
res.redirect('/go-there')
```

This creates a 302 redirect.

A 301 redirect is made in this way:

```
res.redirect(301, '/go-there')
```

You can specify an absolute path (`/go-there`), an absolute url (`https://anothersite.com`), a relative path (`go-there`) or use the `..` to go back one level:

```
res.redirect('../go-there')
res.redirect('..')
```

You can also redirect back to the Referer HTTP header value (defaulting to `/` if not set) using

```
res.redirect('back')
```

Routing

Routing is the process of determining what should happen when a URL is called, or also which parts of the application should handle a specific incoming request.

Routing is the process of determining what should happen when a URL is called, or also which parts of the application should handle a specific incoming request.

In the Hello World example we used this code

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

This creates a route that maps accessing the root domain URL `/` using the HTTP GET method to the response we want to provide.

Named parameters

What if we want to listen for custom requests, maybe we want to create a service that accepts a string, and returns that uppercase, and we don't want the parameter to be sent as a query string, but part of the URL. We use named parameters:

```
app.get('/uppercase/:theValue', (req, res) => res.send(req.params.theValue.toUpperCase())
```

If we send a request to `/uppercase/test`, we'll get `TEST` in the body of the response.

You can use multiple named parameters in the same URL, and they will all be stored in `req.params`.

Use a regular expression to match a path

You can use regular expressions to match multiple paths with one statement:

```
app.get(/post/, (req, res) => { /* */ })
```

will match `/post`, `/post/first`, `/thepost`, `/posting/something`, and so on.

CORS

How to allow cross site requests by setting up CORS

A JavaScript application running in the browser can usually only access HTTP resources from the same domain (origin) that serves them.

Loading images or scripts/styles from the same origin always works. Also, loading Web Fonts using `@font-face` have the 'same-origin' policy set by default. Likewise with other, less popular things (like WebGL textures and `drawImage` resources loaded in the Canvas API).

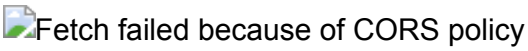
However, XHR and Fetch calls to an external, 3rd party server will fail. That is unless the 3rd party server implements a mechanism which allows the connection to be made and requested resources to be downloaded and used.

This mechanism is called CORS, **Cross-Origin Resource Sharing**.

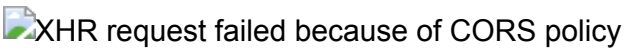
One very important thing that needs CORS is **ES Modules**, recently introduced in modern browsers.

If you don't set up a CORS policy **on the server** that allows it to serve 3rd party origins, the request will fail.

Fetch example:



XHR example:



A Cross-Origin resource fails if it's:

- to a different **domain**
- to a different **subdomain**
- to a different **port**
- to a different **protocol**

CORS is there for your security, to prevent malicious users from exploiting whatever Web Platform you happen to be using.

If you control both the server **and** the client, you know that both parties are trustworthy, and therefore have good reason to allow resource sharing.

How?

It depends on your server-side stack.

Browser support

Pretty good (basically all except IE<10):

Example with Express

If you are using Node.js and Express as a framework, use the CORS middleware package.

Here's a simple implementation of an Express Node.js server:

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

app.get('/without-cors', (req, res, next) => {
  res.json({ msg: '😞 no CORS, no party!' })
})

const server = app.listen(3000, () => {
  console.log('Listening on port %s', server.address().port)
})
```

If you hit `/without-cors` with a fetch request from a different origin, it's going to raise the CORS issue.

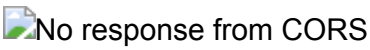
All you need to do to make things work smoothly, is to require the `cors` package linked above, and pass it in as a middleware function to an endpoint request handler:

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

app.get('/with-cors', cors(), (req, res, next) => {
  res.json({ msg: 'WHOAH with CORS it works! 🔝 🎉' })
})

/* the rest of the app */
```

I made a simple Glitch example, here's its code: https://glitch.com/edit/#!/flavio-cors-client.

This is the Node.js Express server: https://glitch.com/edit/#!/flaviocopes-cors-example-express

Note how the request that fails because the server does not handle the CORS headers correctly, is still received. As you can see in the Network panel, where you can see a message the server sent:


No response from CORS

Allow only specific origins

This example has a problem however: ANY request will be accepted by the server as cross-origin.

As you can see in the Network panel, the request that passed has a response header `access-control-allow-origin: *`:


The CORS response header

You need to configure the server to only allow one origin to serve, and block all the others.

Using the same `cors` Node library, here's how you would do it:

```
const cors = require('cors')

const corsOptions = {
  origin: 'https://yourdomain.com',
}

app.get('/products/:id', cors(corsOptions), (req, res, next) => {
  //...
})
```

You can serve more as well:

```
const whitelist = ['http://example1.com', 'http://example2.com']
const corsOptions = {
  origin: function (origin, callback) {
    if (whitelist.indexOf(origin) !== -1) {
      callback(null, true)
    } else {
      callback(new Error('Not allowed by CORS'))
    }
  },
}
```

Preflight

There are some requests that are handled in a "simple" way. All `GET` requests belong to this group.

Also *some* `POST` and `HEAD` requests do as well.

`POST` requests are also in this group, if they satisfy the requirement of using a Content-Type of

- `application/x-www-form-urlencoded`
- `multipart/form-data`
- `text/plain`

All other requests must run through a pre-approval phase, called preflight. The browser does this to determine if it has the permission to perform an action, by issuing an `OPTIONS` request.

A preflight request contains a few headers that the server will use to check permissions (irrelevant fields omitted):

```
OPTIONS /the/resource/you/request
Access-Control-Request-Method: POST
Access-Control-Request-Headers: origin, x-requested-with, accept
Origin: https://your-origin.com
```

The server will respond with something like this (irrelevant fields omitted):

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://your-origin.com
Access-Control-Allow-Methods: POST, GET, OPTIONS, DELETE
```

We checked for POST, but the server tells us we can also issue other HTTP request types for that particular resource.

Following the Node.js Express example above, the server must also handle the OPTIONS request:

```
var express = require('express')
var cors = require('cors')
var app = express()

//allow OPTIONS on just one resource
app.options('/the/resource/you/request', cors())

//allow OPTIONS on all resources
app.options('*', cors())
```

Templating

Express is capable of handling server-side template engines. Template engines allow us to add data to a view, and generate HTML dynamically.

Express is capable of handling server-side template engines.

Template engines allow us to add data to a view, and generate HTML dynamically.

Express uses Jade as the default. Jade is the old version of Pug, specifically Pug 1.0.

> The name was changed from Jade to Pug due to a trademark issue in 2016, when the project released version 2. You can still use Jade, aka Pug 1.0, but going forward, it's best to use Pug 2.0

Although the last version of Jade is 3 years old (at the time of writing, summer 2018), it's still the default in Express for backward compatibility reasons.

In any new project, you should use Pug or another engine of your choice. The official site of Pug is https://pugjs.org/.

You can use many different template engines, including Pug, Handlebars, Mustache, EJS and more.

Using Pug

To use Pug we must first install it:

```
npm install pug
```

and when initializing the Express app, we need to set it:

```
const express = require('express')
const app = express()
app.set('view engine', 'pug')
```

We can now start writing our templates in `.pug` files.

Create an about view:

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

and the template in `views/about.pug`:

```
p Hello from Flavio
```

This template will create a `p` tag with the content `Hello from Flavio`.

You can interpolate a variable using

```
app.get('/about', (req, res) => {
  res.render('about', { name: 'Flavio' })
})
```

```
p Hello from #{name}
```

This is a very short introduction to Pug, in the context of using it with Express. Look at the Pug guide for more information on how to use Pug.

If you are used to template engines that use HTML and interpolate variables, like Handlebars (described next), you might run into issues, especially when you need to convert existing HTML to Pug. This online converter from HTML to Jade (which is very similar, but a little different than Pug) will be a great help: https://jsonformatter.org/html-to-jade

> Also see the differences between Jade and Pug

Using Handlebars

Let's try and use Handlebars instead of Pug.

You can install it using `npm install hbs`.

Put an `about.hbs` template file in the `views/` folder:

```
Hello from {{name}}
```

and then use this Express configuration to serve it on `/about`:

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

app.set('view engine', 'hbs')
app.set('views', path.join(__dirname, 'views'))

app.get('/about', (req, res) => {
  res.render('about', { name: 'Flavio' })
})

app.listen(3000, () => console.log('Server ready'))
```

You can also **render a React application server-side**, using the `express-react-views` package.

Start with `npm install express-react-views react react-dom`.

Now instead of requiring `hbs` we require `express-react-views` and use that as the engine, using `jsx` files:

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

app.set('view engine', 'jsx')
app.engine('jsx', require('express-react-views').createEngine())

app.get('/about', (req, res) => {
  res.render('about', { name: 'Flavio' })
})

app.listen(3000, () => console.log('Server ready'))
```

Just put an `about.jsx` file in `views/`, and calling `/about` should present you an "Hello from Flavio" string:

```
const React = require('react')

class HelloMessage extends React.Component {
  render() {
    return <div>Hello from {this.props.name}</div>
  }
}

module.exports = HelloMessage
```

Middleware

A piece of middleware is a function that hooks into the routing process, performing an arbitrary operation at some point in the chain (depending on what we want it to do).

A piece of middleware is a function that hooks into the routing process, performing an arbitrary operation at some point in the chain (depending on what we want it to do).

It's commonly used to edit the request or response objects, or terminate the request before it reaches the route handler code.

Middleware is added to the execution stack like so:

```
app.use((req, res, next) => { /* */ })
```

This is similar to defining a route, but in addition to the Request and Response objects instances, we also have a reference to the *next* middleware function, which we assign to the variable `next`.

We always call `next()` at the end of our middleware function, in order to pass the execution to the next handler. That is unless we want to prematurely end the response and send it back to the client.

You typically use pre-made middleware, in the form of `npm` packages. A big list of the available ones can be found here.

One example is `cookie-parser`, which is used to parse cookies into the `req.cookies` object. You can install it using `npm install cookie-parser` and you use it thusly:

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

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

app.use(cookieParser())
app.listen(3000, () => console.log('Server ready'))
```

We can also set a middleware function to run for specific routes only (not for all), by using it as the second parameter of the route definition:

```
const myMiddleware = (req, res, next) => {
  /* ... */
  next()
}

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

If you need to store data that's generated in a middleware to pass it down to subsequent middleware functions, or to the request handler, you can use the `Request.locals` object. It will attach that data to the current request:

```
req.locals.name = 'Flavio'
```

Serving static files

How to serve static assets directly from a folder in Express

It's common to have images, CSS and more in a `public` subfolder, and expose them to the root level:

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

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

/* ... */

app.listen(3000, () => console.log('Server ready'))
```

If you have an `index.html` file in `public/`, that will be served if you now hit the root domain URL (`http://localhost:3000`)

Send files

Express provides a handy method to transfer a file as attachment: `Response.download()`

Express provides a handy method to transfer a file as attachment: `Response.download()`.

Once a user hits a route that sends a file using this method, browsers will prompt the user for download.

The `Response.download()` method allows you to send a file attached to the request, and the browser instead of showing it in the page, it will save it to disk.

```
app.get('/', (req, res) => res.download('./file.pdf'))
```

In the context of an app:

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

app.get('/', (req, res) => res.download('./file.pdf'))
app.listen(3000, () => console.log('Server ready'))
```

You can set the file to be sent with a custom filename:

```
res.download('./file.pdf', 'user-facing-filename.pdf')
```

This method provides a callback function which you can use to execute code once the file has been sent:

```
res.download('./file.pdf', 'user-facing-filename.pdf', (err) => {
  if (err) {
    //handle error
    return
  } else {
    //do something
  }
})
```

Sessions

How to use sessions to identify users across requests

By default Express requests are sequential and no request can be linked to each other. There is no way to know if this request comes from a client that already performed a request previously.

Users cannot be identified unless using some kind of mechanism that makes it possible.

That's what sessions are.

When implemented, every user of your API or website will be assigned a unique session, and this allows you to store the user state.

We'll use the `express-session` module, which is maintained by the Express team.

You can install it using

```
npm install express-session
```

and once you're done, you can instantiate it in your application with

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

This is a middleware, so you *install* it in Express using

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

const app = express()
app.use(session({
  'secret': '343ji43j4n3jn4jk3n'
}))
```

After this is done, all the requests to the app routes are now using sessions.

`secret` is the only required parameter, but there are many more you can use. It should be a randomly unique string for your application.

The session is attached to the request, so you can access it using `req.session` here:

```
app.get('/', (req, res, next) => {
  // req.session
}
```

This object can be used to get data out of the session, and also to set data:

```
req.session.name = 'Flavio'
console.log(req.session.name) // 'Flavio'
```

This data is serialized as JSON when stored, so you are safe to use nested objects.

You can use sessions to communicate data to middleware that's executed later, or to retrieve it later on, on subsequent requests.

Where is the session data stored? It depends on how you set up the `express-session` module.

It can store session data in

- **memory**, not meant for production
- a **database** like MySQL or Mongo
- a **memory cache** like Redis or Memcached

> There is a big list of 3rd packages that implement a wide variety of different compatible caching stores in https://github.com/expressjs/session

All solutions store the session id in a cookie, and keep the data server-side. The client will receive the session id in a cookie, and will send it along with every HTTP request.

We'll reference that server-side to associate the session id with the data stored locally.

Memory is the default, it requires no special setup on your part, it's the simplest thing but it's meant only for development purposes.

The best choice is a memory cache like Redis, for which you need to setup its own infrastructure.

Another popular package to manage sessions in Express is `cookie-session`, which has a big difference: it stores data client-side in the cookie. I do not recommend doing that because storing data in cookies means that it's stored client-side, and sent back and forth in every single request made by the user. It's also limited in size, as it can only store 4 kilobytes of data. Cookies also need to be secured, but by default they are not, since secure Cookies are possible on HTTPS sites and you need to configure them if you have proxies.

Validating input

Learn how to validate any data coming in as input in your Express endpoints

Say you have a POST endpoint that accepts the name, email and age parameters:

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

app.use(express.json())

app.post('/form', (req, res) => {
  const name  = req.body.name
  const email = req.body.email
  const age   = req.body.age
})
```

How do you perform server-side validation on those results to make sure:

- name is a string of at least 3 characters?
- email is a real email?
- age is a number, between 0 and 110?

The best way to handle validation on any kind of input coming from outside in Express is by using the `express-validator` package:

```
npm install express-validator
```

You require the `check` and `validationResult` objects from the package:

```
const { check, validationResult } = require('express-validator');
```

We pass an array of `check()` calls as the second argument of the `post()` call. Every `check()` call accepts the parameter name as argument. Then we call `validationResult()` to verify there were no validation errors. If there are any, we tell them to the client:

```
app.post('/form', [
  check('name').isLength({ min: 3 }),
  check('email').isEmail(),
  check('age').isNumeric()
], (req, res) => {
  const errors = validationResult(req)
  if (!errors.isEmpty()) {
    return res.status(422).json({ errors: errors.array() })
  }

  const name  = req.body.name
  const email = req.body.email
  const age   = req.body.age
})
```

Notice I used

- `isLength()`
- `isEmail()`
- `isNumeric()`

There are many more of these methods, all coming from validator.js, including:

- `contains()`, check if value contains the specified value
- `equals()`, check if value equals the specified value
- `isAlpha()`
- `isAlphanumeric()`
- `isAscii()`
- `isBase64()`
- `isBoolean()`
- `isCurrency()`
- `isDecimal()`
- `isEmpty()`
- `isFQDN()`, is a fully qualified domain name?
- `isFloat()`
- `isHash()`
- `isHexColor()`
- `isIP()`
- `isIn()`, check if the value is in an array of allowed values
- `isInt()`
- `isJSON()`
- `isLatLong()`
- `isLength()`
- `isLowercase()`

- `isMobilePhone()`
- `isNumeric()`
- `isPostalCode()`
- `isURL()`
- `isUppercase()`
- `isWhitelisted()`, checks the input against a whitelist of allowed characters

You can validate the input against a regular expression using `matches()`.

Dates can be checked using

- `isAfter()`, check if the entered date is after the one you pass
- `isBefore()`, check if the entered date is before the one you pass
- `isISO8601()`
- `isRFC3339()`

For exact details on how to use those validators, refer to https://github.com/chriso/validator.js#validators.

All those checks can be combined by piping them:

```
check('name')
  .isAlpha()
  .isLength({ min: 10 })
```

If there is any error, the server automatically sends a response to communicate the error. For example if the email is not valid, this is what will be returned:

```
{
  "errors": [{
    "location": "body",
    "msg": "Invalid value",
    "param": "email"
  }]
}
```

This default error can be overridden for each check you perform, using `withMessage()`:

```
check('name')
  .isAlpha()
  .withMessage('Must be only alphabetical chars')
  .isLength({ min: 10 })
  .withMessage('Must be at least 10 chars long')
```

What if you want to write your own special, custom validator? You can use the `custom` validator.

In the callback function you can reject the validation either by throwing an exception, or by returning a rejected promise:

```
app.post('/form', [
  check('name').isLength({ min: 3 }),
  check('email').custom(email => {
    if (alreadyHaveEmail(email)) {
      throw new Error('Email already registered')
    }
  }),
  check('age').isNumeric()
], (req, res) => {
  const name  = req.body.name
  const email = req.body.email
  const age   = req.body.age
})
```

The custom validator:

```
check('email').custom(email => {
  if (alreadyHaveEmail(email)) {
    throw new Error('Email already registered')
  }
})
```

can be rewritten as

```
check('email').custom(email => {
  if (alreadyHaveEmail(email)) {
    return Promise.reject('Email already registered')
  }
})
```

Sanitizing input

You've seen how to validate input that comes from the outside world to your Express app.

There's one thing you quickly learn when you run a public-facing server: never trust the input.

Even if you sanitize and make sure that people can't enter weird things using client-side code, you'll still be subject to people using tools (even just the browser devtools) to POST directly to your endpoints.

Or bots trying every possible combination of exploit known to humans.

What you need to do is sanitizing your input.

The `express-validator` package you already use to validate input can also conveniently used to perform sanitization.

Say you have a POST endpoint that accepts the name, email and age parameters:

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

app.use(express.json())

app.post('/form', (req, res) => {
  const name  = req.body.name
  const email = req.body.email
  const age   = req.body.age
})
```

You might validate it using:

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

app.use(express.json())

app.post('/form', [
  check('name').isLength({ min: 3 }),
  check('email').isEmail(),
  check('age').isNumeric()
], (req, res) => {
  const name  = req.body.name
  const email = req.body.email
  const age   = req.body.age
})
```

You can add sanitization by piping the sanitization methods after the validation ones:

```
app.post('/form', [
  check('name').isLength({ min: 3 }).trim().escape(),
  check('email').isEmail().normalizeEmail(),
  check('age').isNumeric().trim().escape()
], (req, res) => {
  //...
})
```

Here I used the methods:

- `trim()` trims characters (whitespace by default) at the beginning and at the end of a string
- `escape()` replaces `<`, `>`, `&`, `'`, `"` and `/` with their corresponding HTML entities
- `normalizeEmail()` canonicalizes an email address. Accepts several options to lowercase email addresses or subaddresses (e.g. `flavio+newsletters@gmail.com`)

Other sanitization methods:

- `blacklist()` remove characters that appear in the blacklist
- `whitelist()` remove characters that do not appear in the whitelist
- `unescape()` replaces HTML encoded entities with `<`, `>`, `&`, `'`, `"` and `/`
- `ltrim()` like trim(), but only trims characters at the start of the string
- `rtrim()` like trim(), but only trims characters at the end of the string
- `stripLow()` remove ASCII control characters, which are normally invisible

Force conversion to a format:

- `toBoolean()` convert the input string to a boolean. Everything except for '0', 'false' and '' returns true. In strict mode only '1' and 'true' return true
- `toDate()` convert the input string to a date, or null if the input is not a date
- `toFloat()` convert the input string to a float, or NaN if the input is not a float
- `toInt()` convert the input string to an integer, or NaN if the input is not an integer

Like with custom validators, you can create a custom sanitizer.

In the callback function you just return the sanitized value:

```
const sanitizeValue = value => {
  //sanitize...
}

app.post('/form', [
  check('value').customSanitizer(value => {
    return sanitizeValue(value)
  }),
], (req, res) => {
  const value  = req.body.value
})
```

Handling forms

How to process forms using Express

This is an example of an HTML form:

```
<form method="POST" action="/submit-form">
  <input type="text" name="username" />
  <input type="submit" />
</form>
```

When the user presses the submit button, the browser will automatically make a `POST` request to the `/submit-form` URL on the same origin of the page. The browser sends the data contained, encoded as `application/x-www-form-urlencoded`. In this particular example, the form data contains the `username` input field value.

Forms can also send data using the `GET` method, but the vast majority of the forms you'll build will use `POST`.

The form data will be sent in the POST request body.

To extract it, you will need to use the `express.urlencoded()` middleware:

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

app.use(express.urlencoded({
  extended: true
}))
```

Now, you need to create a `POST` endpoint on the `/submit-form` route, and any data will be available on `Request.body`:

```
app.post('/submit-form', (req, res) => {
  const username = req.body.username
  //...
  res.end()
})
```

Don't forget to validate the data before using it, using `express-validator`.

File uploads in forms

How to manage storing and handling files uploaded via forms, in Express

This is an example of an HTML form that allows a user to upload a file:

```
<form method="POST" action="/submit-form" enctype="multipart/form-data">
  <input type="file" name="document" />
  <input type="submit" />
</form>
```

> Don't forget to add `enctype="multipart/form-data"` to the form, or files won't be uploaded

When the user press the submit button, the browser will automatically make a `POST` request to the `/submit-form` URL on the same origin of the page. The browser sends the data contained, not encoded as as a normal form `application/x-www-form-urlencoded` , but as `multipart/form-data` .

Server-side, handling multipart data can be tricky and error prone, so we are going to use a utility library called **formidable**. Here's the GitHub repo, it has over 4000 stars and is well-maintained.

You can install it using:

```
npm install formidable
```

Then include it in your Node.js file:

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

Now, in the `POST` endpoint on the `/submit-form` route, we instantiate a new Formidable form using `formidable.IncomingForm()` :

```
app.post('/submit-form', (req, res) => {
  new formidable.IncomingForm()
})
```

After doing so, we need to be able to parse the form. We can do so synchronously by providing a callback, which means all files are processed, and once formidable is done, it makes them available:

```
app.post('/submit-form', (req, res) => {
  new formidable.IncomingForm().parse(req, (err, fields, files) => {
    if (err) {
      console.error('Error', err)
      throw err
    }
    console.log('Fields', fields)
    console.log('Files', files)
    for (const file of Object.entries(files)) {
      console.log(file)
    }
  })
})
```

Or, you can use events instead of a callback. For example, to be notified when each file is parsed, or other events such as completion of file processing, receiving a non-file field, or if an error occurred:

```
app.post('/submit-form', (req, res) => {
  new formidable.IncomingForm().parse(req)
    .on('field', (name, field) => {
      console.log('Field', name, field)
    })
    .on('file', (name, file) => {
      console.log('Uploaded file', name, file)
    })
    .on('aborted', () => {
      console.error('Request aborted by the user')
    })
    .on('error', (err) => {
      console.error('Error', err)
      throw err
    })
    .on('end', () => {
      res.end()
    })
})
```

Whichever way you choose, you'll get one or more Formidable.File objects, which give you information about the file uploaded. These are some of the methods you can call:

- `file.size`, the file size in bytes
- `file.path`, the path the file is written to
- `file.name`, the name of the file

- `file.type` , the MIME type of the file

The path defaults to the temporary folder and can be modified if you listen for the `fileBegin` event:

```
app.post('/submit-form', (req, res) => {
  new formidable.IncomingForm().parse(req)
    .on('fileBegin', (name, file) => {
        file.path = __dirname + '/uploads/' + file.name
    })
    .on('file', (name, file) => {
      console.log('Uploaded file', name, file)
    })
    //...
})
```

An Express HTTPS server with a self-signed certificate

How to create a self-signed HTTPS certificate for Node.js to test apps locally

To be able to serve a site on HTTPS from localhost you need to create a self-signed certificate.

A self-signed certificate is sufficent to establish a secure, HTTPS connection for development purposes. Although browsers will complain that the certificate is self-signed (and as such is not trusted).

To create the certificate you must have **OpenSSL** installed on your system.

You may have it installed already, just try typing `openssl` in your terminal.

If not, on a Mac you can install it using `brew install openssl` (if you use Homebrew). Otherwise, search on Google "how to install openssl on ".

Once OpenSSL is installed, run this command:

```
openssl req -nodes -new -x509 -keyout server.key -out server.cert
```

You will be prompted to answer a few questions. The first is the country name:

```
Generating a 1024 bit RSA private key
..........++++++
.........++++++
writing new private key to 'server.key'
-----
You are about to be asked to enter information that will be incorporated into your certif
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
```

Then your state or province:

```
State or Province Name (full name) [Some-State]:
```

Your city:

```
Locality Name (eg, city) []:
```

...and your organization name:

```
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
```

You can leave all of these empty.

Just remember to set this to `localhost`:

```
Common Name (e.g. server FQDN or YOUR name) []: localhost
```

...and to add your email address:

```
Email Address []:
```

That's it! Now you have 2 files in the folder where you ran the original command:

- `server.cert` is the self-signed certificate file
- `server.key` is the private key of the certificate

Both files will be needed to establish the HTTPS connection, and depending on how you are going to setup your server, the process to use them will vary.

Those files need to be put in a place reachable by the application, and then you'll need to configure the server to use them.

This is an example using the `https` core module and Express:

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

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

https.createServer({}, app).listen(3000, () => {
  console.log('Listening...')
})
```

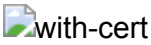
Without adding the certificate, if I connect to `https://localhost:3000` this is what the browser will show:

 without-cert

With the certificate in place:

```
const fs = require('fs')

//...

https.createServer({
  key: fs.readFileSync('server.key'),
  cert: fs.readFileSync('server.cert')
}, app).listen(3000, () => {
  console.log('Listening...')
})
```

Chrome will tell us that the certificate is invalid (since it's self-signed), and will ask us to confirm before continuing (however, the HTTPS connection will still work):


with-cert

Setup Let's Encrypt for Express

How to set up HTTPS using the popular free solution Let's Encrypt

If you run a Node.js application on your own VPS, you'll need a solution for obtaining SSL certificates.

Today, the standard for doing this is to use Let's Encrypt and Certbot, a tool from EFF, aka Electronic Frontier Foundation, the leading nonprofit organization focused on privacy, free speech, and in-general civil liberties in the digital world.

These are the steps we'll follow:

- Install Certbot
- Generate the SSL certificate using Certbot
- Allow Express to serve static files
- Confirm the domain
- Obtain the certificate
- Setup the renewal

Install Certbot

These instructions assume you are using Ubuntu, Debian or any other Linux distribution that uses `apt-get` to manage packages:

```
sudo add-apt-repository ppa:certbot/certbot
sudo apt-get update
sudo apt-get install certbot
```

You can also install Certbot on a Mac, for testing purposes (requires Homebrew):

```
brew install certbot
```

However, you will need to link that to a real domain name in order for it to be useful.

Generate the SSL certificate using Certbot

Now that Certbot is installed, you can invoke it to generate the certificate. You must run this as root:

```
certbot certonly --manual
```

...or call sudo from a non-root user:

```
sudo certbot certonly --manual
```

This is the process in detail:

The installer will ask you to provide the domain of your website.

...it then asks for your email:

```
→ sudo certbot certonly --manual
Password: XXXXXXXXXXXXXXXXXX
Saving debug log to /var/log/letsencrypt/letsencrypt.log
Plugins selected: Authenticator manual, Installer None
Enter email address (used for urgent renewal and security notices) (Enter 'c' to
cancel): your@email.com
```

...and to accept the ToS:

```
Please read the Terms of Service at
https://letsencrypt.org/documents/LE-SA-v1.2-November-15-2017.pdf. You must
agree in order to register with the ACME server at
https://acme-v02.api.letsencrypt.org/directory

(A)gree/(C)ancel: A
```

...and for permission to share your email address:

```
Would you be willing to share your email address with the Electronic Frontier
Foundation, a founding partner of the Let's Encrypt project and the non-profit
organization that develops Certbot? We'd like to send you email about our work
encrypting the web, EFF news, campaigns, and ways to support digital freedom.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(Y)es/(N)o: Y
```

...finally, we can enter the domain where we want to use the SSL certificate:

```
Please enter in your domain name(s) (comma and/or space separated)  (Enter 'c'
to cancel): copesflavio.com
```

...the installer asks if it's ok to log your IP address:

```
Obtaining a new certificate
Performing the following challenges:
http-01 challenge for copesflavio.com

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
NOTE: The IP of this machine will be publicly logged as having requested this
certificate. If you're running certbot in manual mode on a machine that is not
your server, please ensure you're okay with that.

Are you OK with your IP being logged?
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(Y)es/(N)o: y
```

...and finally we get to the verification phase!

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Create a file containing just this data:

TS_oZ2-ji23jrio3j2irj3iroj_U51u1o0x7rrDY2E.1DzOo_voCOsrpddP_2kpoek2opeko2pke-UAPb21sW1c

And make it available on your web server at this URL:

http://copesflavio.com/.well-known/acme-challenge/TS_oZ2-ji23jrio3j2irj3iroj_U51u1o0x7rrD
```

Now, let's leave Certbot alone for a couple of minutes.

We need to verify we own the domain, by creating a file named `TS_oZ2-ji23jrio3j2irj3iroj_U51u1o0x7rrDY2E` in the `.well-known/acme-challenge/` folder. Pay attention! The weird string I just pasted will change every single time you go through this process.

You'll need to create the folder and the file, since they do not exist by default.

In this file you need to put the content that Certbot printed:

```
TS_oZ2-ji23jrio3j2irj3iroj_U51u1o0x7rrDY2E.1DzOo_voCOsrpddP_2kpoek2opeko2pke-UAPb21sW1c
```

As for the filename - this string is unique each time you run Certbot.

Allow Express to serve static files

In order to serve that file from Express, you need to enable serving static files. You can create a `static` folder, and add there the `.well-known` subfolder, then configure Express like this:

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

//...

app.use(express.static(__dirname + '/static', { dotfiles: 'allow' }))

//...
```

The `dotfiles` option is mandatory otherwise `.well-known`, which is a dotfile (as it starts with a dot), won't be made visible. This is a security measure, because dotfiles can contain sensitive information and they are better-off preserved by default.

Confirm the domain

Now run the application and make sure the file is reachable from the public internet. Go back to Certbot, which is still running, and press ENTER to go on with the script.

Obtain the certificate

That's it! If all went well, Certbot created the certificate and the private key, and made them available in a folder on your computer (and it will tell you which folder, of course).

Now, simply copy/paste the paths into your application to start using them to serve your requests:

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

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

https
  .createServer(
    {
      key: fs.readFileSync('/etc/letsencrypt/path/to/key.pem'),
      cert: fs.readFileSync('/etc/letsencrypt/path/to/cert.pem'),
      ca: fs.readFileSync('/etc/letsencrypt/path/to/chain.pem'),
    },
    app
  )
  .listen(443, () => {
    console.log('Listening...')
  })
```

Note that I made this server listen on port 443, so it needs to be run with root permissions.

Also, the server is exclusively running in HTTPS, because I used `https.createServer()`. You can also deploy an HTTP server alongside this, by running:

```
http.createServer(app).listen(80, () => {
  console.log('Listening...')
})

https
  .createServer(
    {
      key: fs.readFileSync('/etc/letsencrypt/path/to/key.pem'),
      cert: fs.readFileSync('/etc/letsencrypt/path/to/cert.pem'),
      ca: fs.readFileSync('/etc/letsencrypt/path/to/chain.pem'),
    },
    app
  )
  .listen(443, () => {
    console.log('Listening...')
  })
```

Setup the renewal

The SSL certificate is only going to be valid for 90 days, so you need to set up an automated system for renewing it.

How? Using a cron job.

A cron job is a way to run tasks at a specified interval of time. It can be every week, every minute, every month, and so on.

In our case, we'll run the renewal script twice per day, as recommended in the Certbot documentation.

First, find out the absolute path of `certbot` on your system. I use `type certbot` on macOS to get it, and in my case it's in `/usr/local/bin/certbot`.

Here's the script we need to run:

```
certbot renew
```

This is the cron job entry:

```
0 */12 * * * root /usr/local/bin/certbot renew >/dev/null 2>&1
```

The above says 'run it every 12 hours, every day: at 00:00 and at 12:00'.

> Tip: I generated this line using https://crontab-generator.org/

Add your newly-created script to the system's crontab using this command:

```
env EDITOR=pico crontab -e
```

This opens the `pico` editor (feel free to substitute with whichever editor you prefer). Simply enter the new script, save, and the cron job is installed.

Once this is done, you can see the list of active cron jobs by running:

```
crontab -l
```