THE VALLEY OF CODE

EXPRESS HANDBOOK

FLAVIO COPES

Table of Contents

Preface

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

In particular, the goal is to get you up to speed quickly with Express.

This book is written by Flavio. I **publish programming tutorials** on my blog flaviocopes.com and The Valley Of Code.

You can reach me on Twitter @flaviocopes.

Enjoy!

The Express Handbook

Introduction to Express

Express is a Web Framework built upon Node.js.

Node.js is an amazing tool for building networking services and applications.

Express builds on top of its features to provide easy to use functionality that satisfies the needs of the Web Server use-case. It's Open Source, free, easy to extend and very performant.

There are also lots and lots of pre-built packages you can just drop in and use to do all kinds of things.

Installation

You can install Express into any project with npm.

If you're in an empty folder, first create a new Node.js project with

```
npm init -y
```

then run

```
npm install express
```

to install Express into the project.

The first "Hello, World" example

The first example we're going to create is a simple Express Web Server.

Copy this code:

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

app.get('/', (req, res) => res.send('Hello World!'))
app.listen(3000, () => console.log('Server ready'))
```

Save this to an `index.js` file in your project root folder, and start the server using

```
node index.js
```

You can open the browser to port 3000 on localhost and you should see the `Hello World!` message.

Those 4 lines of code do a lot behind the scenes.

First, we import the `express` package to the `express` value.

We instantiate an application by calling the `express()` method.

Once we have the application object, we tell it to listen for GET requests on the `/` path, using the `get()` method.

There is a method for every HTTP **verb**: `get()`, `post()`, `put()`, `delete()`, `patch()`:

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

Those methods accept a callback function - which is called when a request is started - and we need to handle it.

We pass in an arrow function:

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

Express sends us two objects in this callback, which we called `req` and `res`, they represent the Request and the Response objects.

Both are standards and you can read more on them here:

- https://developer.mozilla.org/en-US/docs/Web/API/Request
- https://developer.mozilla.org/en-US/docs/Web/API/Response

Request is the HTTP request. It gives us all the request information, including the request parameters, the headers, the body of the request, and more.

Response is the HTTP response object that we'll send to the client.

What we do in this callback is to send the 'Hello World!' string to the client, using the `Response.send()` method.

This method sets that string as the body, and it closes the connection.

The last line of the example actually starts the server, and tells it to listen on port `3000`. We pass in a callback that is called when the server is ready to accept new requests.

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

Send a response to the client

In the Hello World example we used the `send()` method of the Response object 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.

After this, `send()` closes the connection.

`send()` automatically sets the `Content-Length` HTTP response header, unlike `end()` which requires you to do that.

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 `status()` method on the response object:

```
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')
```

Send a JSON response

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

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: '/adr

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

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

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:
```

Handling redirects

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.

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
```

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.

Templates

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.

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}
```

Look at the Pug guide for more information on how to use Pug.

This online converter from HTML to Pug will be a great help: https://html-to-pug.com/

Middleware

A 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 Assets with 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 to the client

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

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

Let's see 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

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` .

Handling file uploads in forms

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)
    })
    //...
})
```

Conclusion

Thanks a lot for reading this book.

For more, head over to The Valley Of Code.

Send any feedback, errata or opinions at hello@thevalleyofcode.com