JavaScript - Framework - Express

# Overview

Fast, unopinionated, minimalist web framework, for backend NodeJS servers.­

# Setup

## Install

Create a new directory, server.js (entry point), and package.json for the server. Then install express using yarn:

yarn add express

## Basic Server

A simple express server will only need to require the express module then serve a static string on the root route:

// server.js

const express = require('express')

const app = express()

const port = 3000

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

app.listen(port, () => console.log(`server listening at [http://localhost:${port}`)](http://localhost:$%7bport%7d%60)))

The server can then be started by running the file using node:

node server.js

## Application Settings

Application settings can be set using the app.set(<name>, <value>) method. Common settings which are set are:

* "env" - environment mode, defaults to process.env.NODE\_ENV
* "view engine" - engine for rendering view templates

## Debugging

The express debug module can be used to internally log information about route matches, middleware functionns, and application mode.

yarn add debug

Then debugging statements can be made by requiring the debug module with name of section being debugged and passing any debug queries into the debug module:

const debug = require('debug')('<debug-name>')

debug('an error')

Debugging can be viewed using the DEBUG environment variable when starting the server.

To view debug outputs for all sections, use an asterisk:

DEBUG=\* node server.js

To view debug output for a specific section, use the section name:

DEBUG=express:server node server.js

# Architecture

## MVC

Since express is unopinioned in server architecture, it comes with no MVC structure installed by default. Since MVC aids in the seperation of concerns, which allows for a more readable and maintable application, it is common to create a MVC file structure for the application using JavaScript modules.

A typical project structure could look like:

- root/

- app/

- assets/

- config/

- controllers/

- middleware/

- models/

- routes/

- views/

- spec/

- public/

- node\_modules/

- app.js

- package.json

Where routes are required into the app, with each route containing the middleware, controller, and views it needs to function.

## Webpacker

Assets are generally compiled by an asset pipeline before being served in the public folder of the application.

A common asset compiler is webpacker, which can be setup to both automatically compile changing assets in development environments, and compile and compress static assets in a production environment.

## Single Page Applications

A common alternative to performing server side rending using the express webserver on each request is creating a single page application (SPA). SPAs build using a framework such as react, will only require the serving of a single bundle of static assets on the root directory of the server. The rest of the server can be used as a JSON API accessible by the SPA running in the client's browser.

Creating a SPA and JSON API allows for further seperation of concerns, since all rendering will be performed by the client in the browser by JavaScript and the server will only have to perform database actions and business logic. This design also allows for different scaling of each section of the system, since the static assets can be served by a CDN if required.

# Routing

## Overview

Routing defines how the applications endpoints respond to client requests. Express routes are defined by call on the application object, with the most common routes being those which make up RESTful routing:

* app.all()
* app.get()
* app.post()
* app.put()
* app.delete()

The full list of routes can be seen at: <https://expressjs.com/en/4x/api.html#app.METHOD>

## Handlers

Routing methods specify a callback function (handler function) which is called when the application receives the specific HTTP request which matches the route method definition. The handler function gets passed the base objects of the 'request' and 'response' and typically will use the information provided in the request, to return a specific response to the client. For example:

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

res.send('hello world')

})

Multiple handler functions can be passed into a route with the option to send a response or call the next() callback passing control onto the next handler function in the route. Having multiple handler functions allows initial functions to act like middleware, processing the request and passing control if certain criteria are met. For example:

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

console.log(going to make a new resource')

next()

}, function (req, res, next) {

console.log('making a new resource')

})

Handlers can be in the form of a single function, mutliple functions, an array of functions, or a combination. It is common to put middleware in an array of functions then have the final handler as a separate function for readability. For example:

app.get('/example/d', [middleware1, middleware2], function (req, res) {

res.send('Hello from D!')

})

## Resources

Routes can be chained together using the route function to create resource like collections:

app.route('/book')

.get(function (req, res) {

res.send('Get a random book')

})

.post(function (req, res) {

res.send('Add a book')

})

.put(function (req, res) {

res.send('Update the book')

})

## Router

The express.router instance can be used to modularise routes even further by creating a mini app with its own middleware and mouting it in a route of the main application. For example, create a router instance in a node module:

// app/routes/birds.js

var express = require('express')

var router = express.Router()

// middleware that is specific to this router

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

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

next()

})

// define the home page route

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

res.send('Birds home page')

})

// define the about route

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

res.send('About birds')

})

module.exports = router

Then require and mount the module in the main application:

var birds = require('./routes/birds')

app.use('/birds', birds)

## Paths

Route paths can be strings, string patterns, or regular expressions. Note since Express uses path-to-regexp for matching routes, in string routes the charaters ?, +, \*, and () will be subsets of their regular expression counterparts. Also, to use a $ symbol it must be escaped and enclosed: '/ data/([\$])'.

The Express route tester can be used to test string routes:

<http://forbeslindesay.github.io/express-route-tester/>

Route parameters can be used to capture information in the route and pass it into the request object to be handled. Route parameters are defined by the colon symbol followed by the parameter name, for example:

:userId -> req.params{ "userId": <value-from-route> }

app.get('/users/:userId/books/:bookId', function (req, res) {

res.send(req.params)

})

Since hypens and dots are interpreted literally, they can be useful to parse routes. Regular expressions can also be appended to route parameters using double brackets ((<regex>))for specifitiy on type, for example:

/flights/:from((\w{3}))-:to((\w{3}))

http://localhost:3000/flights/LAX-SFO

req.params: { "from": "LAX", "to": "SFO" }

## Response Object

The response object is passed into each handler function and is used to send a reponse to the client then terminate the request-response cycle. There are various different methods which can be used to return specific responses to the client:

* res.download() - promt client to download file
* res.end() - end request reponse process
* res.json() - send json response
* res.jsonp() - send json reponse with jsonp support
* res.redirect() - redirect
* res.render() - render a view template
* res.send() - send reponse of various types
* res.sendFile() - send file as octet stream
* res.sendStatus() - set reponse status code and send as string in reponse body

If a handler function does not end the request-response process by using the response object, next function, or throwing an error, the request will be left hanging.

# Middleware

## Overview

Middleware in Express is used to pre-process the request, perform some business logic, and then pass control onto more middleware and finally the route handler if certain criteria are met. Common uses of middleware are to provide authentication, file uploads, custom messages, etc.

## Loading Middleware

As mentioned in the routes section, specific middleware handler functions can be manually loaded in arrays prior to route handlers, however express also provides several methods to load global middleware to the application or specific routes.

The express.all() method can be used to handle all HTTP requests to a specific path, then pass control onto a different handler. For example:

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

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

next() // pass control to the next route

})

The express.use(<path>, <function>) method can be used to load middleware for all requests to a specific path (or the whole application). The path argumnet will match any paths that follows it path immediatley, for example a path of '/apples' will match request to both '/apples' and '/apples/images' etc. Since the path argument defaults to root ('/'), if excluded, middleware will be applied to all request to the application.

When loading middleware in the application, the order in which they are loader matters. The order in which routes and middleware are defined, is the order in which they're route matchers are tested against. Therefore, if a express.get() route to the root ('/') is defined in the server before a express.use() method with middleware for the application is defined, the request will match the get route and the middleware handlers will never be run.

Middleware loaded onto the app object, is called Application-level middleware.

## Next()

The next function, when called with no arguments, is used to pass control onto the next handler function in a route. However, the next function can also be used to skip functions when it is called with different arguments:

* next('route') - will pass control onto the next route, skipping all preceding handler functions in current route. The 'route' argument will only work if the route has been defined as an app.<method>() or router.<method>() method.
* next('router') - will pass control back out of the router instance and the app will continue searching for the next route which matches the request path.
* next('<anything-else>') - will cause express to throw and error with what has been passed as an argument.

## Built In Middleware

### .json()

The express.json([options]) function is built in middleware which parses incoming JSON payloads. When called it returns middleware which looks at requests which have the "application/json" content-type header. The type header can be modified by options.

### .urlencoded()

The express.urlencoded([options]) function is built in middleware which parses urlencoded payloads. When called it returns middleware which parses urlencoded bodies of requests which have the "application/x-www-form-urlencoded" content-type header. The type header can be modified by options.

### .static()

The express.static(<root>, [options]) function serves static files from the defined <root> directory. The files are located by combining the request url with the root directory. A common root directory is the /public folder, in which static assest are commonly stored.

If a file is not found, the next() function is called allowing for stacking of static functions and fallbacks. A common example would be searching for an image file, if non is found, a fallback middleware is placed to serve a 'image-not-found' image instead.

# Views

## Overview

When using Express to perform server side HTML rendering, views can be rendered from static templates using different template engines. Common engines used with express are Pug, Mustache, and EJS.

## Using Pug

The Pug templating engine can be used by first installing pug using the package manager:

yarn add pug

Then setting the view settings:

app.set('views', 'app/views')

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

Create a view template file:

// app/views/index.pug

html

head

title= title

body

h1= message

Template can then be rendered using the res.render method, referencing the view template to use and passing locals:

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

res.render('index', { title: 'Hey', message: 'Hello there!' })

})

# Errors

## Overview

Since the express server is lightweight, if an error is thrown and not caught, the server will crash and not restart, blocking any future client requests. Therefore, all errors must be caught and handle appropriately.

## Error Object

The error object (often abreviated to err) has four properties:

* stack - stack trace (do not return on production)
* message - error message
* name - error name
* errors - array of errors provided

## Next()

Express automatically handles errors which get thrown in synchronous code inside route handlers and middleware. However, errors returned from asynchronous functions invoked by handlers and middleware must be passed into the next() function for express to process and handle them.

If anything, but the string 'route' or 'router', gets passed into the next function, it will skip any remaining non-error handling handlers and proceed to error handling. Therefore, if an external function call is designed to return nothing unless there is an error, it can be passed into the next function to throw any errors it encounters.

A common design pattern is to use promises to catch errors and pass them into the next function using the catch handler (error is given as first argument).

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

Promise.resolve().then(function () {

throw new Error('BROKEN')

}).catch(next) // Errors will be passed to Express.

})

Note: if the handler function is not designed to call next after completion, the request-response cycle must be ended in the handler definition, otherwise the request will hang.

## Error Handling Middleware

Express comes with built-in error handling, which by default is placed at the end of the middleware function stack. When an error is thrown:

* res.statusCode is set to 500 if not already in the 400 to 500 range
* res.statusMessage is set
* body is set to status code
* headers set to any specified in err.headers object

Custom error handling middleware can be added to routes to handle errors which may occur within their handling functions. Error handling functions take four arguments, with the addition being the error, and function in a similar way to normal handler functions, for example:

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

if (res.headersSent) { // delegate to default handler is res being sent

return next(err)

}

console.error(err.stack)

res.status(500).send('Something broke!')

})

# Databases

## Overview

Since Express does not come with a built in database driver there are two common ways to interact with databases. Firstly, there are modules available for direct interaction with popular databases, where the specific driver is downloaded for the database being used. Secondly, an ORM can be used to abstract interaction, sometimes allowing for better maintability and seperation of concerns.

<https://expressjs.com/en/guide/database-integration.html>

## MySQL

Add driver using package manager:

yarn add mysql

Configure db:

var mysql = require('mysql')

var connection = mysql.createConnection({

host: 'localhost',

user: 'dbuser',

password: 's3kreee7',

database: 'my\_db'

})

Perform db actions:

connection.connect()

connection.query('SELECT 1 + 1 AS solution', function (err, rows, fields) {

if (err) throw err

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

})

connection.end()

## PostgresSQL

Add driver using package manager:

yarn add pg-promise

Configure db:

var pgp = require('pg-promise')(/\* options \*/)

var db = pgp('postgres://username:password@host:port/database')

Perform db actions:

db.one('SELECT $1 AS value', 123)

.then(function (data) {

console.log('DATA:', data.value)

})

.catch(function (error) {

console.log('ERROR:', error)

})

## ORMs

The two most common ORMs which are used with express are TypeORM and Sequelize. Read separate notes on configuration.

# Production

## Overview

To run Express in a production environment, set the node environment to NODE\_ENV=production and precompile any static assets using the asset pipeline in the project. A production node server will run on 0.0.0.0 and not return stack traces on thrown errors to the client.

## Proxies

To run an express server behind a reverse proxy, set the application setting "trust proxy" to one of the follow values:

* boolean
  + true - client IP is most left entry in X-Forwarded-\* header
  + false - app is directly facing internet, client IP is req.connection.remoteAddress
* IP address - nearest untrusted IP is client IP
  + loopback - localhost, allows 127.0.0.1/8
  + linklocal - allows lan address ranges specified e.g. 169.254.0.0/16
  + uniquelocal - allows specific ip address ranges
* Number - nth hop from front-facing proxy is client
* Function - custom

<https://expressjs.com/en/guide/behind-proxies.html>

## Helmet

Helmet is a collection of middleware functions which set HTTP headers on responses to help protect the application of known vulnerabilities. Install using package manager:

yarn add helmet

Apply middleware collection to application:

var helmet = require('helmet')

app.use(helmet())

## Cookies

To ensure cookies don't allow a attacker to know the application software and use targeted attacks against it, the default session cookie name should be changed. A common cookie middleware is express-session, which replaces the express.session middleware built into Express. Express-session can be added using a package manager:

yarn add express-session

Add express-session to the application:

var session = require('cookie-session')

var 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

}

}))

Since express-session uses in memory storage for saving session details by default, in production a session store must be used, of which the most common is redis:

<https://www.npmjs.com/package/connect-redis>

## Rate Limiting

Generally, rate limiting is applied to servers to stop dos attacks and brute force attacks against authentication methods. While server based rate limiting utilities such as fail2ban can be useful, they can also be rudimentary in how they limit requests. Therefore, it is also common to add application level rate limiting, since it has the advantage of more context making it more specific and effective for that application.

A commonly used Express rate limiting module is rate-limiter-flexibile:

<https://github.com/animir/node-rate-limiter-flexible>

## CSFR

<https://github.com/expressjs/csurf#readme>

## Graceful Shutdown

When a new version of an application is deployed, the old must be shutdown. Since the application may still be receiving and processing requests, it is important to have a process which gracefully shutdowns the server upon the SIGTERM signal from the process manager.

Graceful shutdown should finish all requests and close any database connections which are open at the time, while closing the server. Noting that the close function needs to be called on the instance of the application listening, an example graceful shutdown process is:

const server = app.listen(port, () => {

console.log(`Starting server in ${app.get("env")} mode`);

console.log("Server is listening on port " + port);

});

process.on('SIGTERM', () => {

console.log('SIGTERM signal received: closing HTTP server')

server.close(() => {

console.log('HTTP server closed')

})

})

# Testing

## How to test

The canonical way to test ExpressJS API only backends is using the supertest plugin to test API routes, however, in some cases where there are no external services this is a full integration test. Therefore, application design should keep strong seperation between concerns using modules, to allow for effective unit testing of each module.

## Split the App and Server

By splitting the application, where express loads the middleware and routes, and the server, where express actually starts the server and handles signals, the application can be exported and tested seperatley to running the server. Therefore, create two files, the app.js and server.js.

## Unit testing

Test application modules using the Jest CLI testing suite. General modules in express, such as middleware and controllers, accept request and response arguments and provide behaviour based on those. Therefore, to create unit test, mocks of the request and response must be generated and asserted against.

Create request mock:

// spec/\_\_mocks\_\_/requestMock.js

## API Integration Testing

Add Jest, Supertest, and Superagent to perform API only integration testing.

yarn add jest babel-cli babel-preset-env supertest superagent --dev

Add jest config to package.json:

"jest": {

"testRegex": ".\*\_spec.js",

"roots": [

"<rootDir>/spec"

],

"moduleFileExtensions": [

"js"

],

"moduleDirectories": [

"node\_modules",

"<rootDir>/app"

],

"moduleNameMapper": {

"\\.(jpg|jpeg|png|gif|eot|otf|webp|svg|ttf|woff|woff2|mp4|webm|wav|mp3|m4a|aac|oga)$": "<rootDir>/spec/\_\_mocks\_\_/fileMock.js",

"\\.(css|less)$": "<rootDir>/spec/\_\_mocks\_\_/styleMock.js"

},

"coveragePathIgnorePatterns": [

"/node\_modules/",

"/spec/",

"/config/"

],

"transform": {

"^.+\\.js$": "babel-jest"

}

}

Create a spec, returning a promise:

// spec/homepage\_spec.js

const request = require("supertest");

const app = require("app");

describe("Test the root path", () => {

test("It should response the GET method", () => {

return request(app)

.get("/")

.then(response => {

expect(response.statusCode).toBe(200);

});

});

});