Saathi

Date __/__/____

④ Section IV : Express.js →

With Node.js alone, we need to write a lot of code to deal with basic things like extracting body of an incoming request.
Typically, we want to focus only on our business logic ie we don't want to care about standard tasks like handling incoming requests / routing.
∴ We use Express.js

A Node.js framework (can be installed as a third party package) that helps in outsourcing some nitty-gritty work.
↳ lot of utility functions that'll help we write clean code.

What is Express ?

Server logic is complex
|
We want to focus on the business logic, not on nitty-gritty details.
|
∴ Use a framework for all heavy lifting.
|
express.js (most popular)

Framework: Helper functions, tools & rules that help us build our application.

Vanilla Node.js
Adonis.js
Koa
Sails.js
⋮

```
const express = require('express');
const app = express(); → express package seems to export a initialise a new object function in the end, thus express '()'. where express stores & manages lot of things for us b.t.s
```

app also happens to be a valid request handler.
```
http.createServer(app);
```

Middleware => an incoming request is automatically funnelled through a bunch of functions by Express.js ie possibility of looping in multiple functions which the request will go through until we send a response.

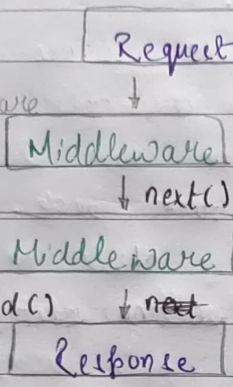
Saathi

Date __/__/____

Adding Middleware →

Allows to split code into multiple blocks/pieces instead of one function (Pluggable)

easily add other 3rd party packages that also provide middleware fns that can be plugged into Express



(req, res, next) => {...}

(req, res, next) => {...}

allows us to add new middleware function.

app.use((req, res, next) => {...});

↳ function called for every incoming request.

takes in array of request handlers.

next is a function passed by express.js — this function has use method (function (takes in function as arguement)) to be executed to allow request to travel on next middleware

We do not see 'in the second middleware' as we need to call next() to allow request to travel on the next middleware in line (goes from 1 → 3 through file).

* no next called → just dies
thus, we must send a response if next is not called.

To see implementation behind the scenes, lib → response.js

← res.send(body of type any); // with express

'<h1>Hello from express!</h1>' → under response header content type set to text/html

file/..

const server = http.createServer(app) ← app.listen(3000);
server.listen(~~of~~ 3000);

Routing in Express →

use() has many versions (4 overloads)

app.use([path], callback, [callback ...])
optional/arguement [ ]
↳ multiple callbacks.
to filter out certain requests.

app.use('/', (req, res, next) => {...}); function works for all '/'s /a, /b, /add, /add-product → because '/' does not mean full path has to be such but it has to start with '/'.

Date __/__/____ Saathi

∴ We add '/' route at the last (so request goes through file from top to bottom).

Parsing Incoming Request →

↳ POST

∴ /add-product → we send a form back.

res.redirect("/");

↳ easier than manually setting status code & location header

console.log(req.body); by default request doesn't try to parse the incoming request body.

To do this, we need to register a "parser".

↳ done by adding another middleware.

Typically done before route-handling middlewares as the parsing of body must be done no matter where request ends up

app.use(bodyParser.urlencoded()); ↳ we can pass options to configure it.

↳ registers a middleware function just like [(req, res, next) => {...}]

↳ in the end calls next() so that req reaches our middleware

before calling next, function to be executed

does entire body parsing as done in core module.

Note:

body-parser parses only bodies sent through a form.

For other kinds (files, json...), we use different parsers.

We get back a JS object with a key-value pair { title: 'Book' } makes extracting value easier.

Now, /product would also listen to GET request, but we want it only to listen to POST request??

Limiting Middleware Execution →

app.use(..) → app.get(..)

additional form of filtering (same syntax as app.use() but fires only for incoming GET request.

Typically we want to split routing code over multiple files.
ie Export logic in different files, and import it in 'app.js'

Date __/__/____ Saathi

Express.js gives us a pretty nice way to outsource routing into other files.

New folder → routes (diff paths & HTTP methods here)

/add-product & /product → admin.js (handles creation of products)
/ → shop.js (user shows)

'Router' feature of express.

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

mini-express app tied to / pluggable into other express apps that can be exported here.

router can be used to define use/get/post.. functions.

```
router.use("/", ...)
module.exports = router;
```

↳ router is exported & has two routes registered.

Nice thing about router is that it is a valid middleware function.

ie `app.use(adminRoutes);` — just the router object | automatically considers our routes in admin.js file when filing request through this middleware here

Order of imports does not matter.

But order of app.use matters. If inside shop.js, we change app.get('/', ...) to app.use('/', ...), we see "Hello from Express" in /add-product also.

Take care of order (as if we change get/post/.. → use, then code will not work)

---

404 Page →

We can take advantage of way Express use middleware and funnels request through them (T → B)

↳ finds somewhere that handles route → goes in there.

↳ no fitting middleware found → send 404 error page

done by simply catching all middleware at the bottom.

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

(require.main.filename) → figuring out the entry point for current application.

Date __/__/____

Saathi

Serving HTML files →

res.send() → res.sendFile('./views/shop.html');
path must be absolute — No such file or directory
'/' refers to root folder on our OS, not to project folder.

We use a feature provided by Node to point to file. [core module - path]

we send file by creating a path using join()
detects OS we're running & create path accordingly. [returns a constructed path (consisting of different sections)

1st segment → Node.js global variable (__dirname). Holds absolute path on OS to this project.
2nd segment → folder name
3rd segment → file.
__dirname, 'views', 'shop.html'.

__dirname points to routes folder.
thus we need to go back → so we add '../'.

Helper function for Navigation →

To get parent directory (and not always use '..'), we can create a helper function.

We create a new folder "util" → path.js file.
Name does not matter even though it clashes with global module (as import is different).

module.exports = [path.dirname (process.mainModule.filename)]
(deprecated) [global Node variable]
variable to construct path to parent directory. returns directory name of path. refers to main module that started the app (ie app.js generally) [(...) for which file we need directory name] Which file this module was run up

path.dirname(require.main.filename)
Neat way of constructing path to root directory.

Saathi

Date __/__/____

Serving files statically →

Serving external css files:
subfolder → 'public' (convention)
contains files exposed to everyone

We generally cannot access the file system in Express
localhost /views/shop.html → Page Not found.

But, we want to make an exception ie some requests can access file system ie shop.html —links to→ css file.

<link ... href="/css/main.css"> all style is gone.
as we cannot access the main.css file.
→ not handled by Express router or other middleware but directly forwarded to file system.

app.use(express.static(...));
We register a new middleware — express.js comes with — serves static files.
(...)
↳ path to the folder we want to serve statically.
folder we want to grant "read" access to.

Now
href="/public/css/main.css" ✗.
or href="/css/main.css" ✓
We act as if we're already in public folder.

Express takes any request that tries to find some file, it automatically forwards it to public folder.