Intro To Express

Development

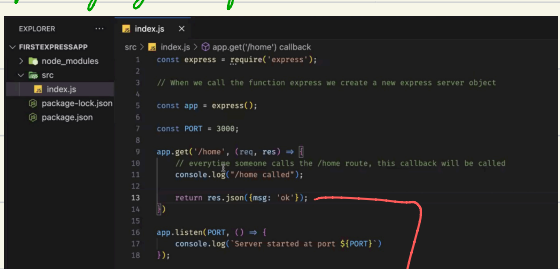
- customer facing apps (abstractions can be used)
- developer productivity tools (you need to create your own abstractions) might work on low level stuff as well.

Express JS doesn't impose any guidelines or rules on the user. (not opinionated)

for logging etc you need to install your own libraries.

To start an express project.

1) just create an npm project. (create package.json file) → metadata for project.
2) install express. (npm install express)
3) create src folder → index.js inside.



JSON — because structure is similar to a JS object.

When you call res.json and pass a JS object. Express converts it to JSON

All of this is in the application layer. Since we're using http.

Fastify uses plugins.

Express uses middleware (sequence or chain of functions).

middlewares are just like functions except-.

middlewares have access to next middleware as well.

middlewares are generally chained.

Design pattern – chain of responsibility

middleware → req obj, res obj, next middleware

```
// express middleware
function m1(req, res, next) {
    console.log('Inside middleware m1');
    next();
}

function m2(req, res, next) {
    console.log('Inside middleware m2');
    next();
}

app.get('/home', m1, m2, (req, res) => {
    // everytime someone calls the /home route, this callback will be called
    console.log("/home called");

    return res.json({msg: 'ok'}); // here we are passing a js object
})
```

Register these MW with app request.

→ defining order of middleware.

cb1 then cb2 then cb3.

if you do not call next() then your response will be halted.

You can see the tab loading ← because it keeps on waiting for the response

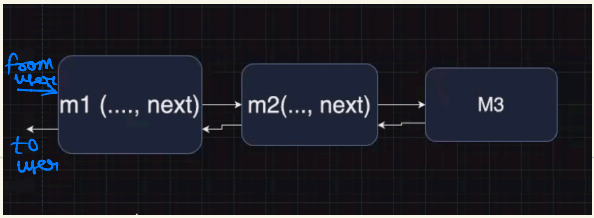
validation layer can be formed using these middleware.

You can return a response object from the middleware as well.

middlewares can be passed as an array as well. (in app.get)

if you modify req object then next middleware will get the updated value.

end of chain is always app.get final function.



chain.
next is not a return statement.

normal function calling stack.
You can try consoling after next().

There are preexisting middlewares as well.

```
// how can the client send custom data to the server
/**
 * 1. URL Params -- /products/7
 * 2. Query Params -- ?key1=value1&key2=value2&key3=value3
 * 3. Request Body
 *
 */
```

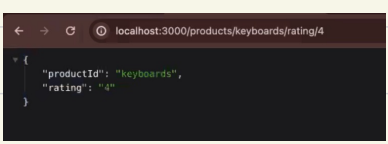
→ dynamic routes. /products/:id
static | variable or dynamic

req.params.id

you can access it on the server using req.query & you'll get a query object.

id variable captures that part of the url.

```
app.get('/products/:product_id/rating/:rate', (req, res) => {
    // :id part is variable and products is static
    // :id part is your url params and overall these kind of routes are called as dynamm
    console.log(req.params);
    const pid = req.params.product_id;
    return res.json({productId: pid, rating: req.params.rate});
})
```



whatever user sends can be captured in the variables ex :id

Design doc — logics

TSD — technical specification documents (specifications of agreements between different teams)

https://github.com/MobiowinMobiTech/windchimes/blob/master/Unified%20Payment%20Interface%20-%20API%20Technology%20Specifications%20V_1.2.3.pdf

Sensitive info isn't sent through url params or query params.

For that we use request body → part of http request object.

it can contain data in different formats.

→ JSON, text, url encoded, form data etc.

Express uses middleware for every incoming request. (for conversion)

app.use

whatever you use inside app.use is used before every request.

body JSON → Express does conversion (body-parser middleware) → proper JS compatible object.

app.use() has to be before other middlewares

```
app.use(bodyParser.json());
app.use(bodyParser.text());
```

app.use(bodyParser.json())

→ adds middleware to every route

before using body parser you cannot access request body as it is not parsed yet. Express doesn't know type of body you can accept.

---

urlencoded data → %20 %5D etc

in urls if you put in special characters the browser can't read it.

URL encoding converts non-ASCII characters into a format that can be transmitted over the Internet. URL encoding **replaces non-ASCII characters with a "%" followed by hexadecimal digits**. URLs cannot contain spaces. URL encoding normally replaces a space with a plus (+) sign, or %20.