Middlewares and Error Handlers.

Routehandlers and Middlewares are two most important ~~top~~ topic

from scratch

~~Const Server = require("Server")~~

```
Const express = require ("express")   ← importing express
const app = express();   ← assigning express to app
Port = 4000;   ← giving port number or assigning port no.
app.listen (port, () => { Console.log ("server running")
});
```

listen ↑ function; port ↑ Port no; () => {...} ↑ Callback function once it connect or start listening to the port.

```
app.use ("/user", () => { //route handler });
```

"/user" ↑ route; () => {...} ↑ route handler function.

If we use app.use
↑ it can handle any type of method wether it is (get/post/patch/put/delete) ← request

```
app.use ("/user", (req, res) => {
    res.send ("route handler 1");
}
```

(req, res) ↑ parameters; res.send ↑ send method to send respond.

What if we didn't send any ~~or request~~ response
← what will return if route handler is empty →

Ans: It will keep sending the request and it will go to infinite loop (it will keep sending, sending, sending, sending, sending the request) [after some time it will get time out]

don't do this

Because we are not sending any response from the server.

one route can also have multiple route handlers?

We can also handle multiple route handler with one route

```
app.get ("/user", (req, res, next) {
  //res.send ("get data")   ← if we send res here it will stop the execution here.
  console.log (" tested 1")
  next(); ← we have to use next() to use second handler.
},
  (req, res) => {
  res.send ("get 2nd data")
  console.log (" tested 2")
});
```

comment out so it can go next

next() fn is given by express js
if call it will go to the next handler

→ If we try to send the multiple ~~send request~~ response it will throw error.
Suppose if it is not commented so o/p will be.

report man = get data.

console.log = tested 1
error → | cannot send the header after they are sent to the client.
tested 2.

once we send a data to client we cannot replay or send data to client because

A TCP connection is made between a client and server, it opens up the connection it sends the data back and it closes the connection and the connection is lost.

once our client get the response it closes the connection but here our server is trying to send more data but connection is lost

as soon we call `next()` it will put next handler into callstack as next function.

we can add as many as we want to add route handler.

to go to next handler we use next();

and if we have 3 route handler and given last next() to next with no res. send it will throw error.

```
app.get("/usr", (req, res, next) => { console.log("1"); next(); },
(req, res, next) { console.log("2"); next(); },
(req, res, next) { console.log("3"); next(); }
```

↑

No respond send it will throw error because next() function express think it must have next handler or if there is no next then it will go to infinite loop.

There are lots of corner case we will do it on code ∈ (use git hup)

we can also give the array of route handler. all

app.get ("/route", [rH1, rH2, rH3]

or we can just pass 2 to array

app.get ("/route", [rH1, rH2], rH3

or

app.get ("/route", rH1, [rH2, rH3]

all will work same and give the same output.

There is another way also.

we dont pass route handler below.
we do like this

```
app.get ("/user", (req, res, next) => { console.log ("H1");
                                        next();
});

app.get ("/user", (req, res, next) => {
    console.log ("H2")
    res.send ();
});
```

independent route handler.

it will work exactly same way

Middle wares.

(VI) Route handler is just the function that actually handling the route, which actually sending the response back.

Function that has been put in the middle, this function are known as middle ware.

Suppose request comes in

//get /user ⇒

whenever a request will comes in "user" express will go and check for wether we have route is matching somewhere, then it will go through the middlewares and then it will handle the request.

~~it~~ if it get request it will go throug all the middleware till it get (res.send())
or till it get response back, once it get response back it will close collection
and
where it get response back that will called as ~~request response handler~~. request handler.

when a request comes to a express is server the job of express server is to go one by one one by one and goest from to top to bottom all the app. function and tryes to send response back if it does not get the request back it get hangs back. ← that's how express ~~back~~ works

What is different between (use, all)

app.use / app.all

Date
Page 64

```
app.use("/user", (req, res, next) => {
    console.log("Handling 1");
    next();
})
(req, res, next) => {
    console.log("Handling 2");
    next(); }
(req, res, next) => {
    console.log("handling 3")
    next(); }
(req, res, next) => {
    console.log("handling 4")
    res.send("got data");
})
```

All are middleware

(1) way

This is request handler because it is actually sending the data.

Second way of defining route handlers (Separate route handler.)

```
app.use("/Profile", (req, res, next) => {
    console.log("1");
    next();
});
```

(2) way

← middleware

```
app.use("/Profile", (req, res, next) => {
    console.log("2")
    res.send("success");
});
```

← request handler.

both will work same

Why do we even need middlewares, what is the use of middlewares

Suppose we have a route of /admin and we are handling the data.

```
            /admin/getAlldata
app.get ("~~/admin~~", (req, res) => {
        // logic of fetching data
// check if req is authorized, if it ~~auto~~ authorise.
    res.send ("All data Sent");
  });
```

now we have to protect that.

we will use tokens.

```
app.use ("/admin/getAllData", (req, res) => {
    res.send ("All data Sent")
  }
```

→ now we will use middle ware, — auth part

```
app.use ("/admin" (req, res, next) => {
        const token = "xyz"
        const authtoken = token === xyz
    if (! authtoken) {
        res.status(401).send("not authorized")
    } else {
    res.send ("send All data"}
        };
```

we can use it in different file

See github for more.

Error Handeling.

Thumb rule - Always try to use (try catch block)

but if you forgot or somthing else occured anything your code get broken.

then we can use wild card to manage unexpected errors.

Lets see how.

app.use("/", (error, res, req, next) => {
↑ means wild card.

res.status(500).send("error occured")

});

↑ use this last of the code.