|  |  |  |  |
|---|---|---|---|
|  | Middlewares | 51 |  |
|  | Errors | 61 |  |
|  | joi - NPM Package | 71 |  |

# Middlewares

→ It is an intermediary.

Request → Middleware → Response

→ Built-In Middlewares in Express

- Middlewares in Express are functions that come into play after the server receives the request and before the response is sent to the client.

- e.g. express.static, express.urlencoded, methodOverride, BodyParser, etc.

→ Middleware functions can perform the following tasks:

- Execute any code
- Make changes to the request and the response objects
- End the request-response cycle
- Call the next middleware functions in the stack.

## * Our 1st Middleware

→ app.use ( <path> , <callback> )

- If we do not define <path>, then this middleware will work for every path.

- visit expressjs.com/en/4x/api.html#app.use for more

e.g.
```
const express = require("express");
const app = express();

app.use( () => { console.log("Hello"); } );

app.get("/", (req,res) => { res.send("I am root"); } );

app.listen( 8080 , () => { console.log("Server is listening to port 8080"
```

output: Terminal : Server is listening to port 8080

Webpage: no response ( browser is loading website ... )

e.g.
```
app.use((req, res) => {

        console.log("I am middleware");
        res.send("middleware finished");
});

app.get("/", (req, res) => {

        res.send("I am root");
});

app.get("/random", (req, res) => {

        res.send("This is random page");
});
```

Output : URL : localhost:8080/
webpage : middleware finished

URL : localhost:8080/random
webpage : middleware finished

## * next

→ The next middleware function is commonly denoted by a variable named next.

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

```
        console.log(" I am Middleware");
        next();
    });
```

→ If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function.

→ Now, we wrote (replaced res.send with next()) next(), then output of page-53 example will be:

URL : localhost:8080/
Webpage: I am root

URL: localhost:8080/random
Webpage: This is random page

e.g.
```
app.use( (req, res, next) => {
    console.log("I am 1st middleware");
    next();
});
app.use( (req, res, next) => {
    console.log(" I am 2nd middleware");
    next();
});
app.get("/", (req, res) => {
    res.send("I am root");
});
app.get("/random", (req, res) => {
    res.send(" I am random");
});
```

Output: URL: localhost:8080/
```
Terminal: I am 1st middleware
          I am 2nd middleware
Webpage: I am root
```

URL: localhost:8080/random
```
Terminal: I am 1st middleware
          I am 2nd middleware
Webpage: I am random
```

Note: We can write some code after next() function call, but in best practices, it is not recomended. So, most developers write " return next(); ". Now, control will not go to next line.

* Utility Middleware

→ logger - log information

- useful information of req and res

e.g.
```
app.use( (req, res, next) => {

      req.responseTime = new Date(Date.now()).toString();
      console.log( req.method,
                   req.path,
                   req.responseTime,
                   req.hostname);
      next();
   });
```

→ Always, write middlewares above our get, post,... code block except error middlewares (which will be at the bottom).

→ If we write this utility middleware at the end and execution completes at get, post, .. then middleware at the end will not be executed.

→ Now, If ~~not~~ no path match with our given URL then last middleware will be in action and we use it for display error.

* app.use with path

→ Path parameter is for which the middleware function is invoked.

e.g.
```
app.use("/random", (req, res, next) => {
    console.log("I am only for random");
    next();
});
```

output : URL: localhost:8080/random
Terminal ~~Webpage~~ : I am only for random

URL : localhost:8080/random/abc
Terminal ~~Webpage~~ : I am only for random

URL: localhost:8080/random/cde
Terminal ~~Webpage~~ : I am only for random

## * API Token as Query String

```
- app.use("/api", (req, res, next) => {

      let {token} = req.query;

      if (token === "giveaccess") {   next(); }
      else {   res.send("ACCESS DENIED!"); }

  });

  app.get("/api", (req, res) => {

      res.send("data");
  });
```

- Output: URL: localhost:8080/api?token=giveaccess
  Webpage: data

  URL: localhost:8080/api?token=abcd
  Webpage: access denied!

* Middleware use for Multiple times

```
- const checkToken = (req, res, next) => {
      let { token} = req. query ;
      if ( token === "giveaccess") {return next(); }
      res. send ("ACCESS DENIED!");
  };

  app. get("/api" , checkToken , (req, res) => {
      res. send("data");
  });
```

# ERRORS

## * Default Error Handler of Express

→ Express gives a "RefenceError" by default on webpage

→ For our own custom error, we can write

```
throw new ERROR("ACCESS DENIED!");
```

- Now, In default error of Express, we can see "ReferenceError: ..." replaced with "ERROR: ACCESS DENIED!"

## * Error Handling Middleware

```
- app.get("/random", (req, res) => {
        abcd = abcd;
  });

  app.use( (err, req, res, next) => {
        console.log("--- ERROR---");
        console.log(err);
  });
```

Output : URL = localhost:8080/random
Terminal = 
```
--- ERROR ---
ReferenceError: abcd is not defined
    at ...
```

```
- app.use( (err, req, res, next) => {
        console.log("--ERROR--");
        next();
  });
```

Output: URL = localhost:8080/random
  Terminal = --ERROR--
  Webpage = Cannot GET /random

→ In above example, next() will trigger non-error handling middleware. That's why on webpage we don't see error name (message) because default error handling middleware of express doesn't trigger.

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

Output: URL = localhost:8080/random
  Webpage = ReferenceError: abcd is not defined
      at ...
      at ...
      ⋮

→ next(err) will trigger, default error handling middleware of express or another our own error handling middleware if we made it.

## * Custom Error Class

→ Default handler generates 500-Internal Server Error with status and error message with stack trace.

→ In our own custom error, we can set status and message of our own

→ visit MDN for HTTP response status codes
  client error responses ( 400 - 499 )
  server error responses ( 500 - 599 )

In ExpressError.js

```
- class ExpressError extends Error {

      constructor (status, message) {

            super();
            this. status = status;
            this. message = message;
      }
}

module. exports = ExpressError;
```

In <u>app.js</u>

```
- const ExpressError = require("./ExpressError.js");
  const checkToken = (req, res, next) => {
      let {token} = req.query;
      if( token === "giveaccess" ) {
          return next();
      }
      throw new ExpressError (401, "ACCESS DENIED");
  };
  app.get("/api", checkToken, (req,res) => {
      res.send("data");
  });
```

<u>Output</u>: URL = localhost:8080/api?token=abcd
Webpage = ERROR : ACCESS DENIED
at ...
⋮
console = 401 Unauthorised
Terminal = ERROR: ACCESS DENIED
at ...
⋮

add after app.get in app.js

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

Output: URL = localhost:2020/api?token = abcd
Webpage = { "status": 401, "message": "ACCESS DENIED" }

* Default Status & Message

→ We can extract status & message from above "err" object and print it.

```
- app.use( (err, req, res, next) => {
     let { status = 500, message = "SOME ERROR" } = err ;
                    ↑ default value if err object doesn't have anything
     res.status(status).send(message);
  });
```

Output : URL = localhost:8080/api?token = abc
Webpage = ACCESS DENIED

* Async Errors

→ In async, Express does not call next by default. We cannot write throw new Express...

```
- app.get("/chats/:id", async (req, res, next) => {

    let { id } = req.params;
    let chat = await Chat.findById(id);

    if(!chat) {
      return next(new ExpressError(404, "Chat not found"));
    }

    res.render("show.ejs", {chat});
  });
```

* Using try-catch

→ wrap our code with try-catch

```
- app.get(" /chats/:id" , async (req, res, next)=>{
      try {
            ...

      } catch (err) {
            next(err);
      }
  });
```

* Using asyncWrap

→ In try-catch, we have to write try-catch in all code blocks. So now we make a function one time and make it callback for all other codeblocks. This one function will catch error.

```
- function asyncWrap ( fn ) {

    return function (req, res, next) {

          fn(req, res, next). catch( (err) => next(err));
    };
}

app. get("/chats/:id", asyncWrap( async (req, res, next) =>{

      ......

}));
```

## * Mongoose Errors

→ There are various errors generates in mongoose. Based on names of the error, we can set different outputs.

```
- const handleValidationError = (err) => {
      console.log("This is Validation Error");
      return err;
  };

  app.use( (err, req, res, next) => {
      console.log(err.name);
      if (err.name === "ValidationError") {
          err = handleValidationError(err);
      }
      next(err);
  });
```

# joi - NPM Package

→ We can use joi as a middleware before storing the data into MongoDB.

→ joi - npm package used as schema validatio before storing in the database.

→ npm i joi ← In Terminal

In schema.js

```
- const Joi = require("joi");

module.exports.listingSchema = Joi.object({

    listing : Joi.object({

        title: Joi.string().required(),
        description: Joi.string().required(),
        location: Joi.string().required(),
        country: Joi.string().required(),
        price : Joi.number().required().min(0),
        image: Joi.string().allow("", null)
    }).required()
});
```

## In app.js

```
- const {listingSchema}= require("./schema.js");

const validateListing =(req, res, next) => {
    let {error} = listingSchema. validate (req.body);
    if(error) {
        let errMsg = error.details. map((el) => el.message).
                                                    join(",");
        throw new ExpressError(400, errMsg);
    } else {    next();    }
};

app.post("/listings", validateListing, wrapAsync(async
    ....

app.put("/listings", validateListing, ....
    ...
```