3.1- Middlewares, Global Catches, Zod, Intro to Authentication

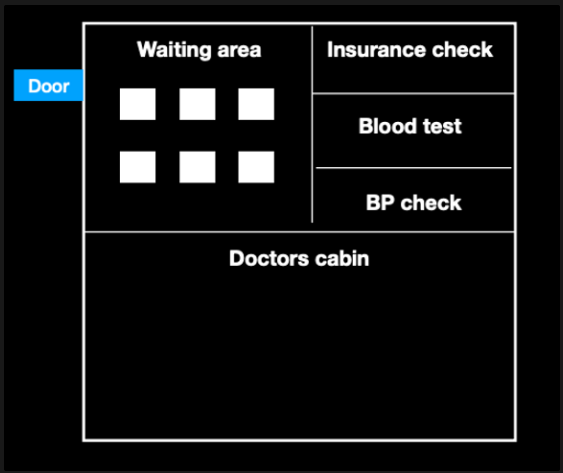
By: ID

💡 Topics Include:
1. Middlewares
2. Global Catches
3. Zod
4. Intro to Authentication

MIDDLEWARES

Consider the Scenario:

- Of a Hospital, where the doctor is single-threaded and patients wait in a callback queue.
- Add-ons include insurance checks, blood tests, and BP checks before reaching the doctor's cabin.
- Pre-checks occur before the patient meets the doctor.



In the context of JS:

- Imagine the waiting area as a single thread.
- Only one patient(in JS: request) can be served at a time.
- Insurance, Blood tests, and BP checks **are essential checks** needed before the patient/request is served.
- These can be categorised into two types:
  - **Auth checks** (does the user have enough funds to visit Doctor).
  - **Input validation check** (BP/Blood Tests).
- How to do them without harming the DRY principle? Answer: **Middlewares**.

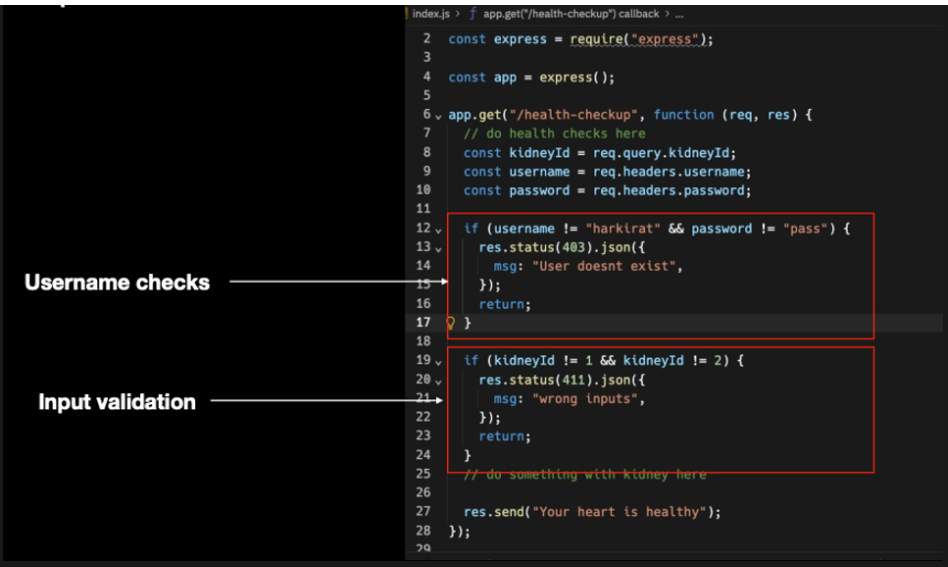
Let's go Deep:

Let's look at the following check, User needs to:

- Send kidneyId in the form of queryParam (ex: ?n=)
- Provide a correct username & password in the headers.

UGLY WAYS:

Using If/Else checks.	Using functions.



This is **acceptable for one route** but **leads to repetition in cases of multiple routes** requiring the same check (VIOLATION of DRY principle) and also is limited to only one file.

CORRECT WAY - USING MIDDLEWARES

- Middleware allows the **creation of functions to store common code**
- And, these can be applied across multiple routes.

Middleware Usage

- Middlewares are functions that have **access to the request object** (`req`), the **response object** (`res`), and the **next middleware function** in the application's request-response cycle.
- We can **call as many callback functions in the request** method.
- But to **reach from one callback to the next one**, we need `next()` .
- In the end callback, we don't need `next()` , and we return `res.json(..)` . The best usage is explained in the example code.

```
//app.js const app = express(); //middleware 1 function userMiddleware(req, res, next) { if(username != 'john' && password == 'pass') { res.status(400).json({ msg: 'Incorrect inputs!' }); }else{ next(); } } //middleware 2 function kidneyMiddleware(req, res, next) { if(kidneyId != 1 && kidneyId != 2) { res.json({ msg: 'Incorrect inputs' }); }else{ next(); } } //using multiple middlewares app.get('/heart-checkup', userMiddleware, kidneyMiddleware, function (req, res) { res.send('Your heart is healthy!'); }); app.get('/kidney-check', userMiddleware, kidneyMiddleware, function (req, res) { res.send('Your kidney is healthy!'); }); //using only one middleware app.get('/health-checkup', userMiddleware, function (req, res) { res.send('Your health is fine!'); });
```

- `app.use()` This method is used to **mount middlewares globally** in the application. They are **executed for every incoming request**. Examples include:
  1. `app.use(calculateRequests)` : An example middleware that performs some calculations or operations for each request.
  2. `app.use(express.json());` : middleware to parse the body of the request.

GLOBAL CATCHES MIDDLEWARES

A middleware that **handles exceptions globally** and **provides a standardized response in case of errors**. Example:

```
//.. //at the end of code //.. app.use(function (err, req, res, next) { res.json({ msg: 'Sorry, something is up with our server', }); });
```

INPUT VALIDATION LIBRARY - ZOD

Zod **simplifies validation by defining a schema** for your data. For instance:

It includes **two steps**:

1. Defining **Zod Schema:** Outlines the structure your data should adhere to. This includes specifying data types, lengths, and other constraints.

   For eg: If you need **data containing array of numbers**, you can create a schema as follows:

   ```
   const zod = require('zod'); const schema = zod.array(zod.number());
   ```

2. **Response Parsing:** The `safeParse` method is used to **parse the request object against the defined schema**, providing a safe and structured way to handle input validation.

   ```
   const response = schema.safeParse(obj);
   ```

A complete example of using Zod: Simple User Registration and Input Validation

```
const express = require('express'); const app = express(); //Zod schema for validating email, password, country, and an array of kidneys. const schema = zod.object({ email: zod.string(), password: zod.string(), country: z.literal('IN').or(z.literal('US')), kidneys: z.array(z.number()) }); app.post('/register', (req, res) => { const userData = req.body; // Validate user data against the Zod schema const validationResult = userSchema.safeParse(userData); if (validationResult.success) { // If validation is successful, process the user data const { username, email, age } = validationResult.data; res.json({ success: true, message: 'User registered successfully', user: { username, email, age } }); } else { // If validation fails, send an error response res.status(400).json({ success: false, errors: validationResult.error.errors }); } })
```

More examples and detailed documentation on type validation in the Zod documentation.

INTRODUCTION TO AUTHENTICATION

Addressing the Problem

- **Anyone can send requests** to your backend.
- They can simply use tools like Postman and send a request.
- The challenge is ensuring **authorized user can only access specific resources**.

DUMB WAY

- Ask users to **send their username and password** in all requests as headers.
- This method is **not secure** as it **exposes sensitive information** with every request.

BETTER WAY

1. Give the **user a token upon signup** or signin. This token serves as a digital key that grants access.
2. Instruct the user to **include this token in the headers** of all future requests.
3. When the user **logs out**, prompt them to **forget or revoke the token,** ensuring it can no longer be used for authentication.

> Library to get comfortable with - jsonwebtokens