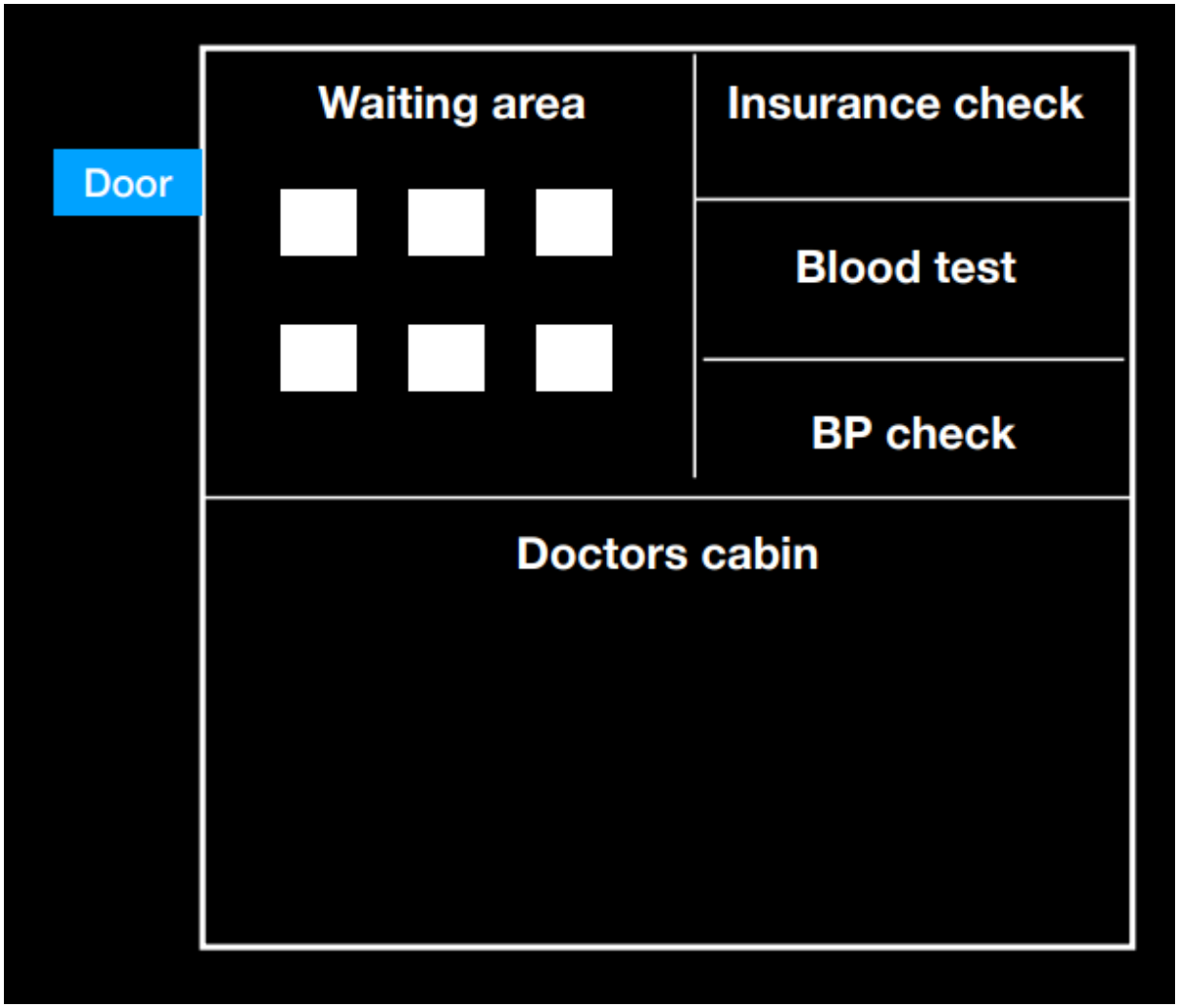
Week 3.1

Middlewares, Global Catches & Zod

In this lecture, Harkirat dives deep into M**iddlewares:** behind-the-scenes helpers that tidy up things before your main code does its thing. G**lobal catches:** safety nets for your code, they catch unexpected issues before they cause chaos. And finally, Zod: a library that ensures efficient input validation on your behalf.



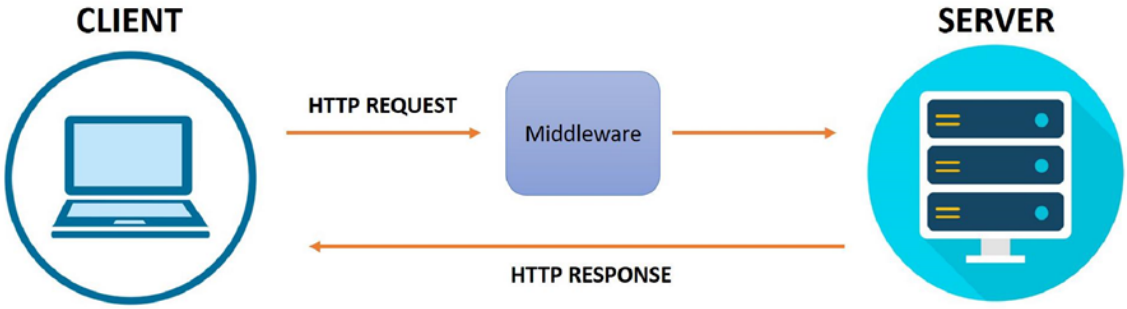
Understanding Middlewares:

Imagine a Busy Hospital:

Think of a hospital where there's a doctor, patients waiting in line, and a few helpful assistants making sure everything runs smoothly.

1. **Doctor's Cabin (Application Logic) :**
    - The doctor is like the main brain of our hospital – ready to help patients with their problems.

2. **Waiting Room (Callback Queue) :**
    - The waiting room is where patients hang out before seeing the doctor. Each patient has a unique situation.
3. Intermediates **(Middlewares) :**
    - Before a patient sees the doctor, there are some helpers doing important tasks.
    - One helper checks if patients have the right paperwork . This is like ensuring everyone is who they say they are (Authentication)
    - Another helper does quick health checks – like making sure patients' blood pressure is okay. This is similar to checking if the information coming to the doctor is healthy and makes sense (Input Validation)

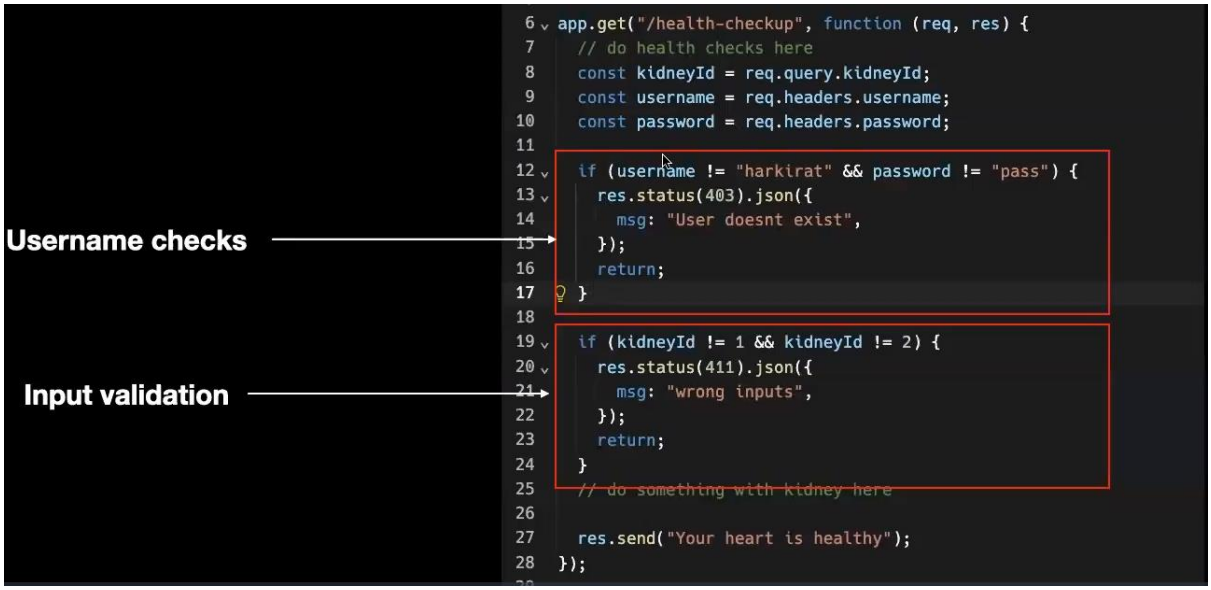


Middlewares in JS Context & Problem Statement:

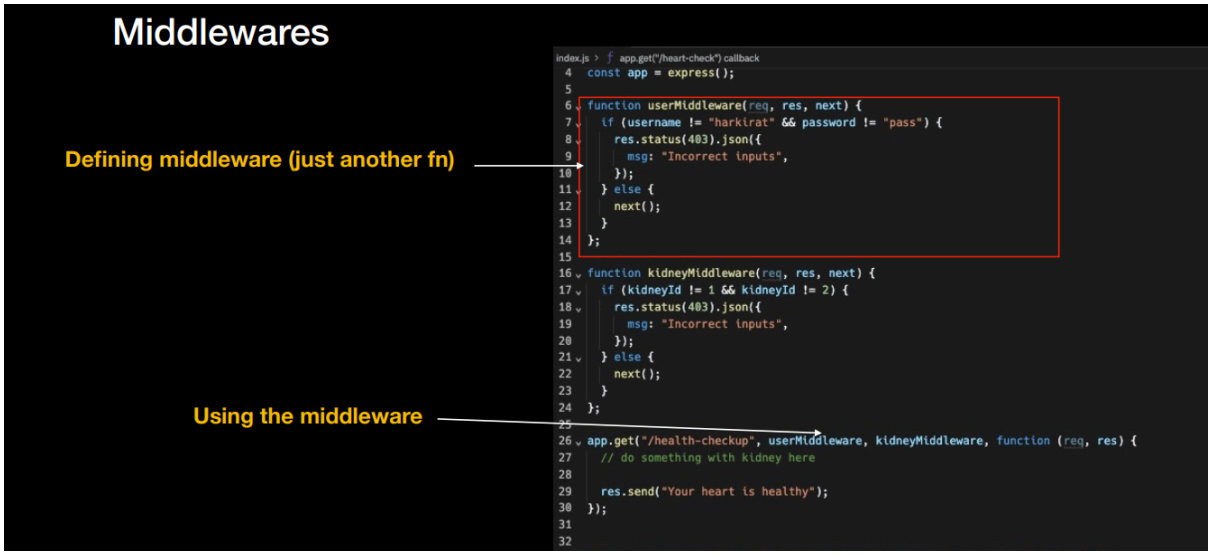
Earlier we used to organize all our prechecks followed by the application logic all in one route.

Middlewares emerged as a solution to enhance code organization by extracting prechecks from the core application logic. The motivation behind their introduction lies in our commitment to the "Don't Repeat Yourself" (DRY) principle.

By isolating these preliminary checks into distinct functions or code blocks known as middlewares, we achieve a more modular and maintainable codebase. This separation not only streamlines the primary application logic but also promotes code reuse, making it easier to manage, understand, and scale our software architecture.



Solution: Middlewares



Furthermore, with middleware, we can easily include as many precheck functions as needed. This means we have the freedom to add various checks or operations to our application without making the main code complex. It's like having building blocks that we can mix and match to create a customized process for our application, making it more adaptable and easier to manage. Here userMiddleware and kidneyMiddleware

Some Associated Concepts:

1. next() Keyword:

In middleware functions in Express, next is a callback function that is used to pass control to the next middleware function in the stack. When you call next(), it tells Express to move to the next

middleware in line. If next() is not called within a middleware function, the request-response cycle stops, and the client receives no response.

Example:

```
app.use((req, res, next) => {
  console.log('This middleware runs first.');
  next(); // Move to the next middleware
});


app.use((req, res) => {
  console.log('This middleware runs second.');
  res.send('Response sent from the second middleware.');
});
```

2. Difference between res.send and res.json:

- res.send: Sends a response of various types (string, Buffer, object, etc.). Express tries to guess the content type based on the data provided.

```
res.send('Hello, World!'); // Sends a plain text response
```

- res.json: Sends a JSON response. It automatically sets the Content-Type header to application/json.

```
res.json({ message: 'Hello, JSON!' }); // Sends a JSON response
```

3. Importance of app.use(express.json()):

app.use(express.json()) is middleware that parses incoming JSON payloads in the request body. It is crucial when dealing with JSON data sent in the request body, typically in POST or PUT requests. Without this middleware, you might receive the JSON data as a raw string, and you'd need to manually parse it.

Example:

```
const express = require('express');

const app = express();


app.use(express.json()); // Middleware to parse JSON in the request body


app.post('/api/data', (req, res) => {
  const jsonData = req.body; // Now req.body contains the parsed JSON data
  // Process the data...
```

```
  res.json({ success: true });
});
```

4. Middleware and req.body:

- req.query and req.headers don't require middleware because they represent the query parameters and headers of the incoming request, respectively. Express automatically parses them.
- req.body requires middleware like express.json() to parse the request body, especially when the body contains JSON data. Other middleware, like express.urlencoded(), is used for parsing form data in the request body.

Middleware helps in processing the request at different stages and is essential for tasks like parsing, logging, authentication, and more in a modular and organized way.

3 Ways of Sending Inputs to a Response:

1. Query Parameter:

- **What it is:** Like giving specific instructions in the web address.
- **Example:** In **www.example.com/search?topic=animals**, the query parameter is **topic** with the value **animals**.
- **Use Case:** Good for simple stuff you want everyone to see, like search terms in a URL.

2. Body:

- **What it is:** Imagine it as the hidden part of a request, carrying more detailed information.
- **Example:** When you fill out a form on a website, the details you enter (name, email) go in the body of the request.
- **Use Case:** Great for sending lots of information, especially when you're submitting something like a form.

3. Headers:

- **What it is:** Extra information attached to the request, kind of like details about a letter.
- **Example:** Headers could include things like your identity or the type of data you're sending.
- **Use Case:** Perfect for passing along special information that doesn't fit neatly in the URL or body, like who you are or how to handle the data.

Bottom Line:

- **Query Parameters:** Simple instructions visible in the web address.
- **Body:** Hidden part of the request for more detailed info, great for forms.
- **Headers:** Extra details about the request, useful for special information.

Global Catches:

It essentially help us the developers give a better error message to the user.

Global Catch or Error-Handling Middleware is a special type of middleware function in Express that has four arguments instead of three ((err, req, res, next)). Express recognizes it as an error-handling middleware because of these four arguments.

```
// Error Handling Middleware
const errorHandler = (err, req, res, next) => {
  console.error('Error:', err);

  // Customize the error response based on your requirements
  res.status(500).json({ error: 'Something went wrong!' });
};
```

Importance of Global Error Handling:

1. **Centralized Handling:**
    - Global catch blocks allow you to centrally manage and handle errors that occur anywhere in your application. Instead of handling errors at each specific location, you can capture and process them in a centralized location.
2. **Consistent Error Handling:**
    - Using a global catch mechanism ensures a consistent approach to error handling throughout the application. You can define how errors are logged, reported, or displayed in one place, making it easier to maintain a uniform user experience.
3. **Fallback Mechanism:**
    - Global catches often serve as a fallback mechanism. If an unexpected error occurs and is not handled locally, the global catch can capture it, preventing the application from crashing and providing an opportunity to log the error for further analysis.

Input Validation:

Input validation is a crucial aspect of securing your application. It helps ensure that the data received by your server is in the expected format and meets certain criteria. Take for instance a login schema, now instead of passing a username and password in the body, the user can pass in any gibberish and may try to crash the server. Thus, it is our responsibility to ensure that our application logic handles all these input vulnerabilities. Let's explore two approaches to input validation: the naive way with multiple if-else statements and using the **zod** library for schema validation.

1. Naive Way - Multiple If-Else Statements:

In the naive approach, you manually check each input parameter to ensure it meets your criteria. Here's an example using Express.js:

```
const express = require('express');

const app = express();


app.use(express.json());


app.post('/login', (req, res) => {
  const { username, password } = req.body;


  if (!username || typeof username !== 'string' || username.length < 3 ||
      !password || typeof password !== 'string' || password.length < 6) {
    return res.status(400).json({ error: 'Invalid input.' });
  }


  // Proceed with authentication logic
  // ...


  res.json({ success: true });
});


const PORT = 3000;

app.listen(PORT, () => console.log(`Server is running on http://localhost:${PORT}`));
```

In this example, we manually check the username and password fields for their existence, data type, and minimum length. This approach can become cumbersome as the number of input parameters increases, and it may lead to code duplication.

2. Using zod Library for Schema Validation:

zod is a TypeScript-first schema declaration and validation library. It provides a concise way to define schemas and validate input data. Here's an example using zod for the same login scenario:

```
const express = require('express');
```

```
const { z } = require('zod');
const app = express();

app.use(express.json());

const loginSchema = z.object({
  username: z.string().min(3),
  password: z.string().min(6),
});

app.post('/login', (req, res) => {
  const { username, password } = req.body;

  try {
    loginSchema.parse({ username, password });
    // Proceed with authentication logic
    // ...
    res.json({ success: true });
  } catch (error) {
    res.status(400).json({ error: 'Invalid input.', details: error.errors });
  }
});

const PORT = 3000;
app.listen(PORT, () => console.log(`Server is running on http://localhost:${PORT}`));
```

In this example, we define a loginSchema using zod that specifies the expected structure and constraints for the input data. The parse method is then used to validate the input against the schema. If the input is invalid, zod throws an error, and we can handle it appropriately. This approach is more concise and less error-prone compared to the manual if-else checks.

Zod:

Zod is a TypeScript-first schema declaration and validation library. It provides a simple and expressive way to define the structure and constraints of your data, allowing you to easily validate and parse input against those specifications. Here's a brief explanation of Zod and its syntax:

Zod Syntax Overview:

1. **Basic Types:**
    - Zod provides basic types such as string, number, boolean, null, undefined, etc.

```
const schema = z.string();
```

2. **Object Schema:**
    - You can define the structure of an object using the object method and specify the shape of its properties.

```
const userSchema = z.object({
  username: z.string(),
  age: z.number(),
});
```

3. **Nested Schemas:**
    - You can nest schemas within each other to create more complex structures.

```
const addressSchema = z.object({
  street: z.string(),
  city: z.string(),
});


const userSchema = z.object({
  username: z.string(),
  address: addressSchema,
});
```

4. **Array Schema:**
    - You can define the schema for arrays using the array method.

```
const numbersSchema = z.array(z.number());
```

5. **Union and Intersection Types:**
    - Zod supports union and intersection types for more flexibility.

```
const numberOrStringSchema = z.union([z.number(), z.string()]);
```

```
const combinedSchema = z.intersection([userSchema, addressSchema]);
```

6. **Optional and Nullable:**
    - You can make properties optional or nullable using optional and nullable methods.

```
const userSchema = z.object({
  username: z.string(),
  age: z.optional(z.number()),
  email: z.nullable(z.string()),
});
```

7. **Custom Validators:**
    - Zod allows you to define custom validation logic using the refine method.

```
const positiveNumberSchema = z.number().refine((num) => num > 0, {
  message: 'Number must be positive',
});
```

8. **Parsing and Validation:**
    - To validate and parse data, use the parse method. If the data is invalid, it throws an error with details about the validation failure.

```
try {
  const userData = userSchema.parse({
    username: 'john_doe',
    age: 25,
    address: {
      street: '123 Main St',
      city: 'Exampleville',
    },
  });
  console.log('Parsed data:', userData);
} catch (error) {
  console.error('Validation error:', error.errors);
}
```

Why Zod:

- **TypeScript-First Approach:** Zod is designed with TypeScript in mind, providing strong type-checking and autocompletion for your schemas.
- **Concise and Expressive Syntax:** Zod's syntax is concise and expressive, making it easy to define complex data structures with minimal code.
- **Validation and Parsing:** Zod not only validates data but also automatically parses it into the expected TypeScript types.
- **Rich Set of Features:** Zod includes a variety of features, such as custom validation, optional and nullable types, union and intersection types, making it a powerful tool for data validation in your applications.

Overall, Zod simplifies the process of declaring and validating data structures, reducing the likelihood of runtime errors and improving the overall robustness of your code.