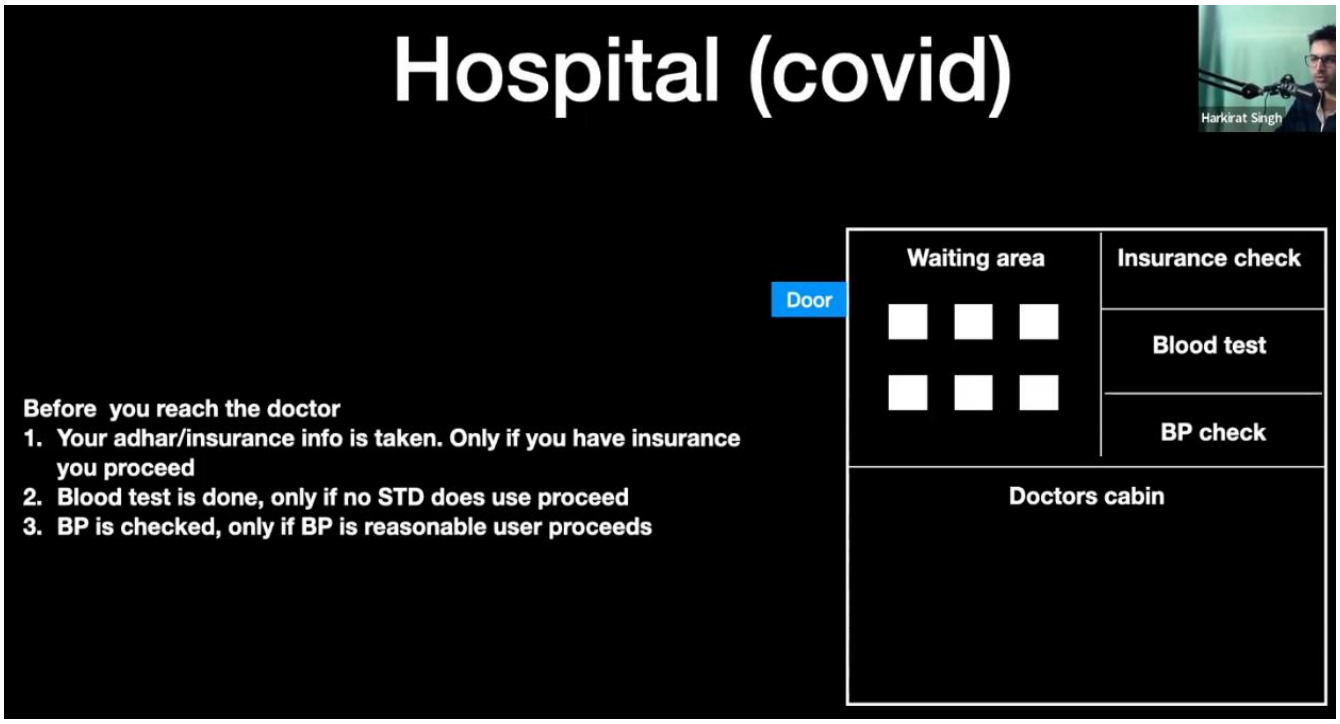
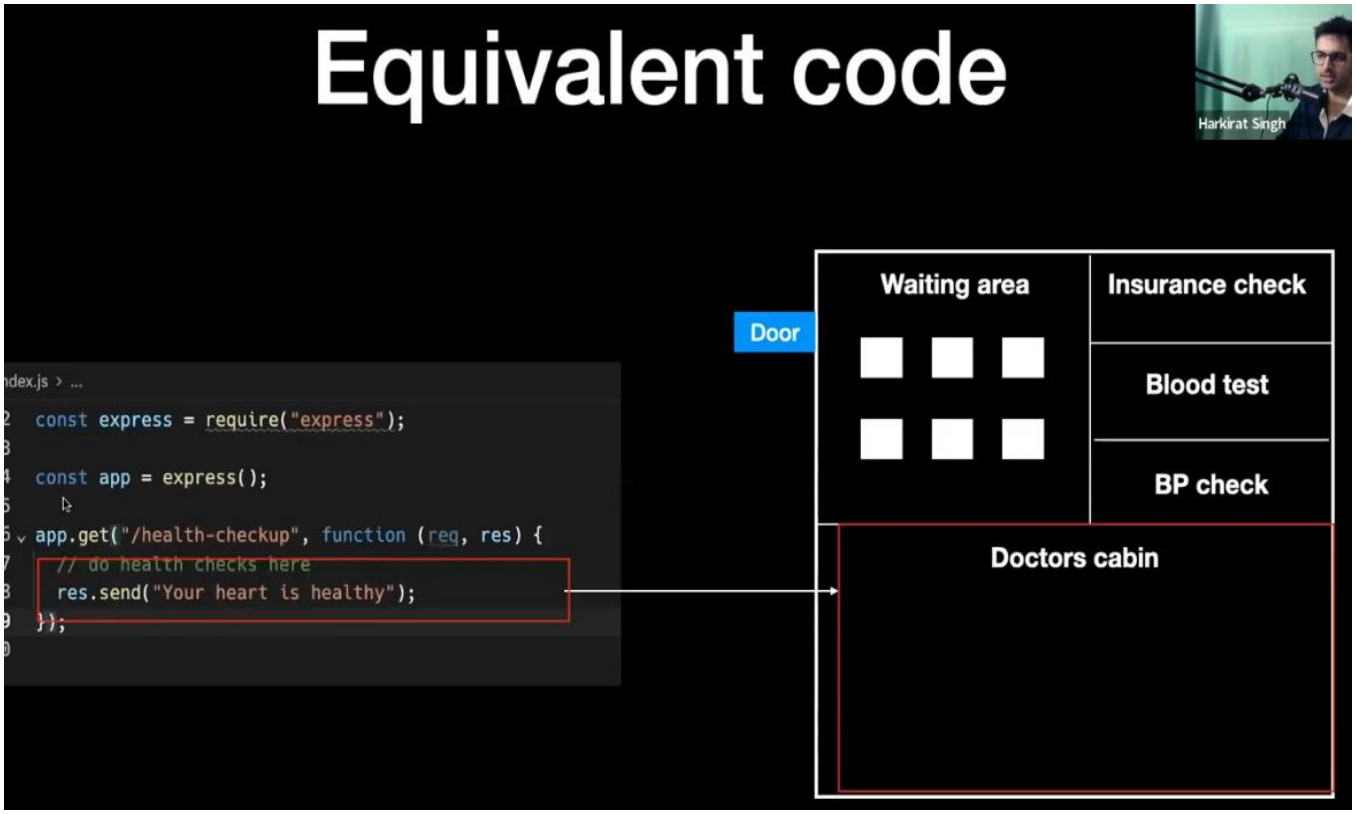
MIDDLEWARE, GLOBAL CATHES AND ZOD



During the time of COVID-19, patients are not allowed to go to the doctor's cabin directly before there are certain pre-checks.



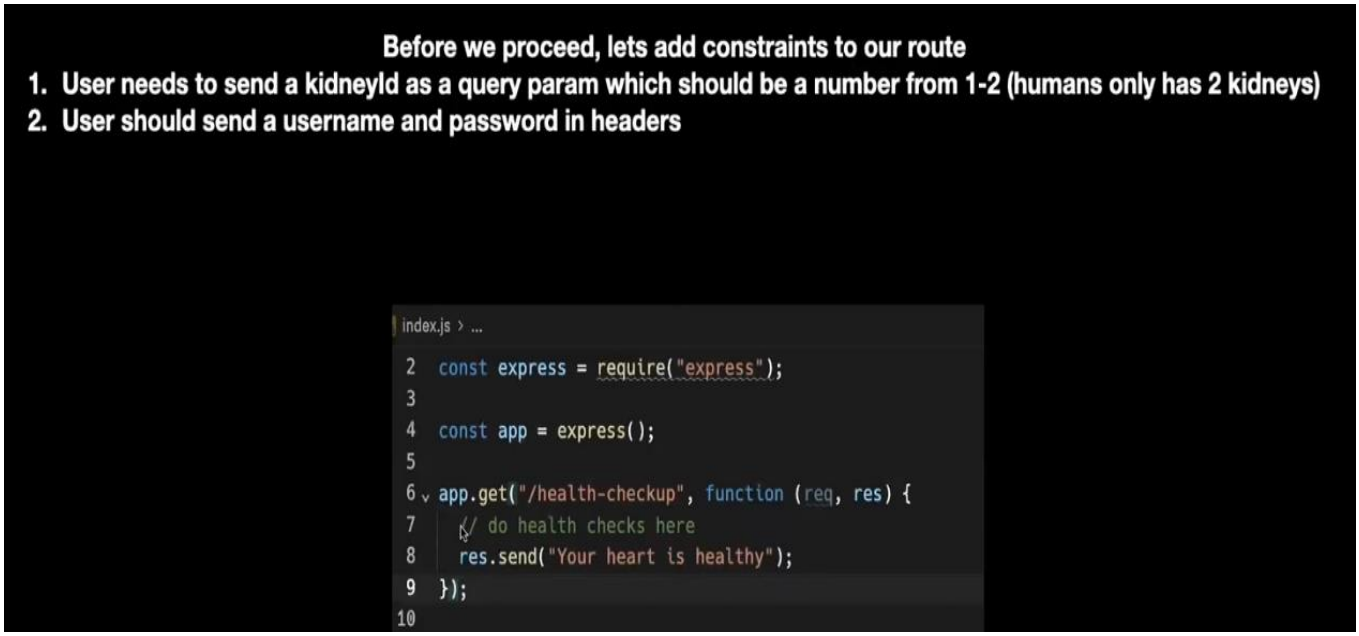
The above code is of the doctor's cabin, and we implement these pre-checks by using Middlewares.

In the real world, pre-checks are of two types:-

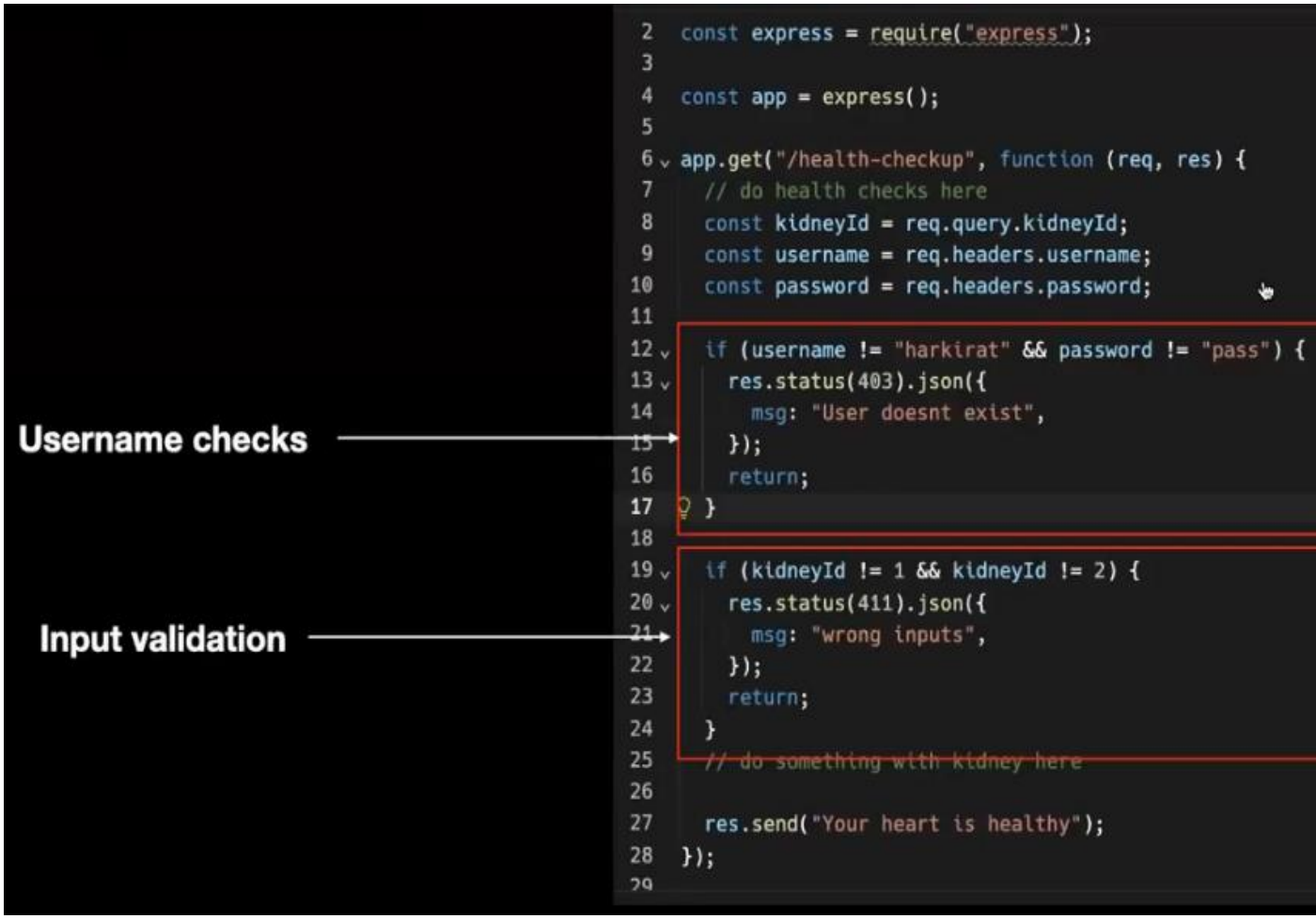
1.) Authentication.

2.) Input validation.

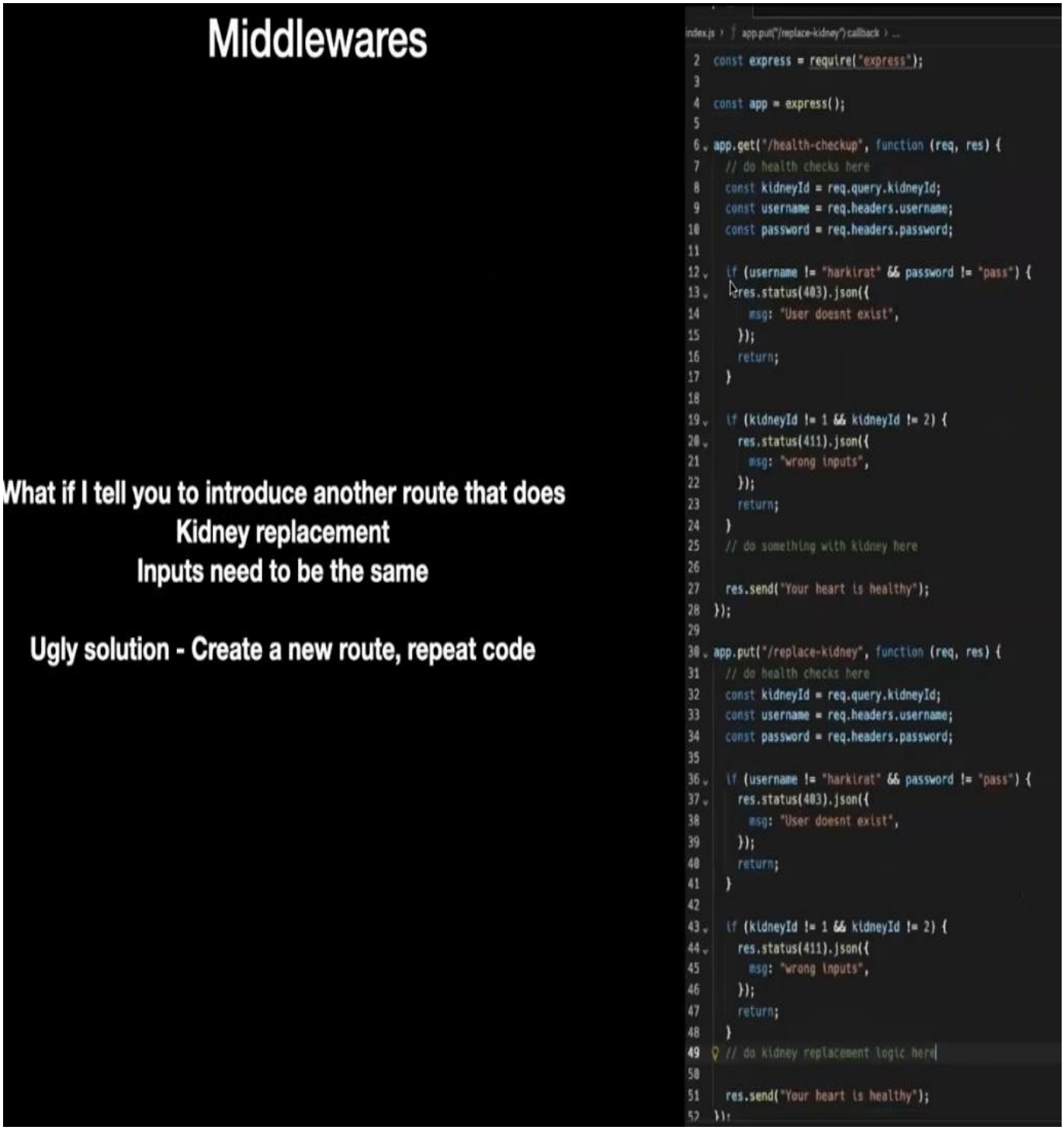
Authentication:-



One of the ugly ways is this way:-



This code returns early, after if and else, if there is no return then the code will send the response. The early returns happen only in the case of wrong inputs.



Another route for kidney replacement has been introduced and the same pre-checks are in

demand, so the above code is written for this situation.

Rather than the same pre-check codes repetitively, we can create a function for that like this:-

```
function usernameValidator(username, password) {
  if (username != "harkirat" && password != "pass") {
    return false;
  }
  return true
}

function kidneyValidator(kidneyId) {
  if (kidneyId != 1 && kidneyId != 2) {
    return false;
  }
  return true;
}

app.get("/health-checkup", function (req, res) {
  // do health checks here
  const kidneyId = req.query.kidneyId;

  if (!usernameValidator(req.query.username, req.query.password)) {
    res.status(403).json({
      msg: "User doesnt exist",
    });
    return;
  }

  if (!kidneyValidator(kidneyId)) {
    res.status(411).json({
      msg: "wrong inputs",
    });
    return;
  }
  // do something with kidney here

  res.send("Your heart is healthy");
});

app.put("/replace-kidney", function (req, res) {
  // do health checks here
  const kidneyId = req.query.kidneyId;
  const username = req.headers.username;
  const password = req.headers.password;

  if (!usernameValidator(req.query.username, req.query.password)) {
    res.status(403).json({
      msg: "User doesnt exist",
    });
    return;
  }

  if (!kidneyValidator(kidneyId)) {
    res.status(411).json({
      msg: "wrong inputs",
    });
    return;
  }
  // do kidney replacement logic here

  res.send("Your heart is healthy");
```

In red boxes, we have defined our validators as a function.

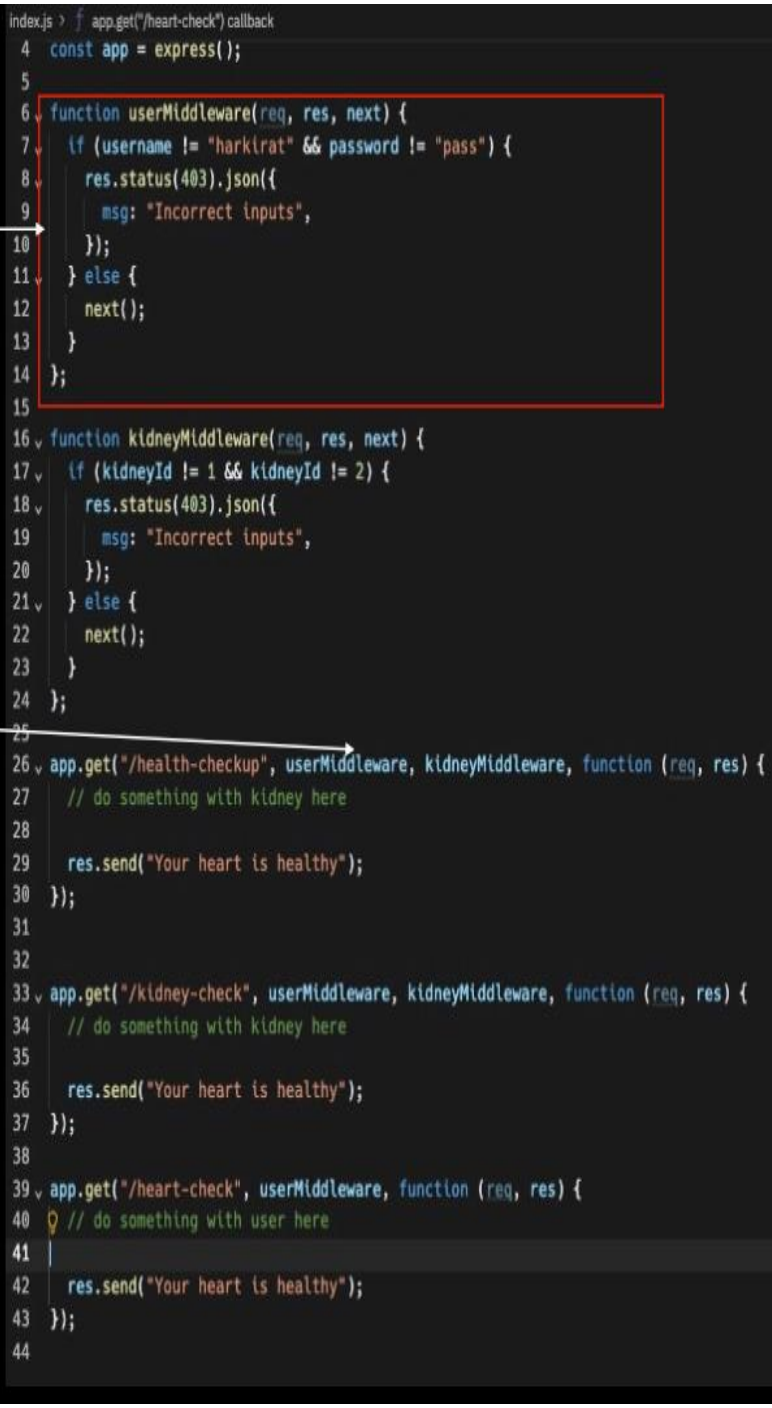
We have optimized our code slightly, but we can further optimize this completely by the use of middleware like this:-

Defining middleware (just another fn)

Using the middleware

you to introduce another route that does
Kidney replacement
Inputs need to be the same

Best solution - middleware



As we see there are two more callback functions in app.get() method, but the rule is we can give numerous callback functions.

Each callback function is called serially.

next():-

In the context of middleware in web development, the `next()` function is commonly used to pass control to the next middleware function in the stack. Middleware functions 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. They can perform tasks such as logging, authentication, parsing request data, etc.

When a middleware function is executed, it can choose to either terminate the request-response cycle by sending a response to the client or pass control to the next middleware function in the stack. The `next()` function is a callback function provided by Express.js, a popular web application framework for Node.js, to pass control to the next middleware function. When `next()` is called within a middleware function, it hands over control to the next middleware function in the chain.

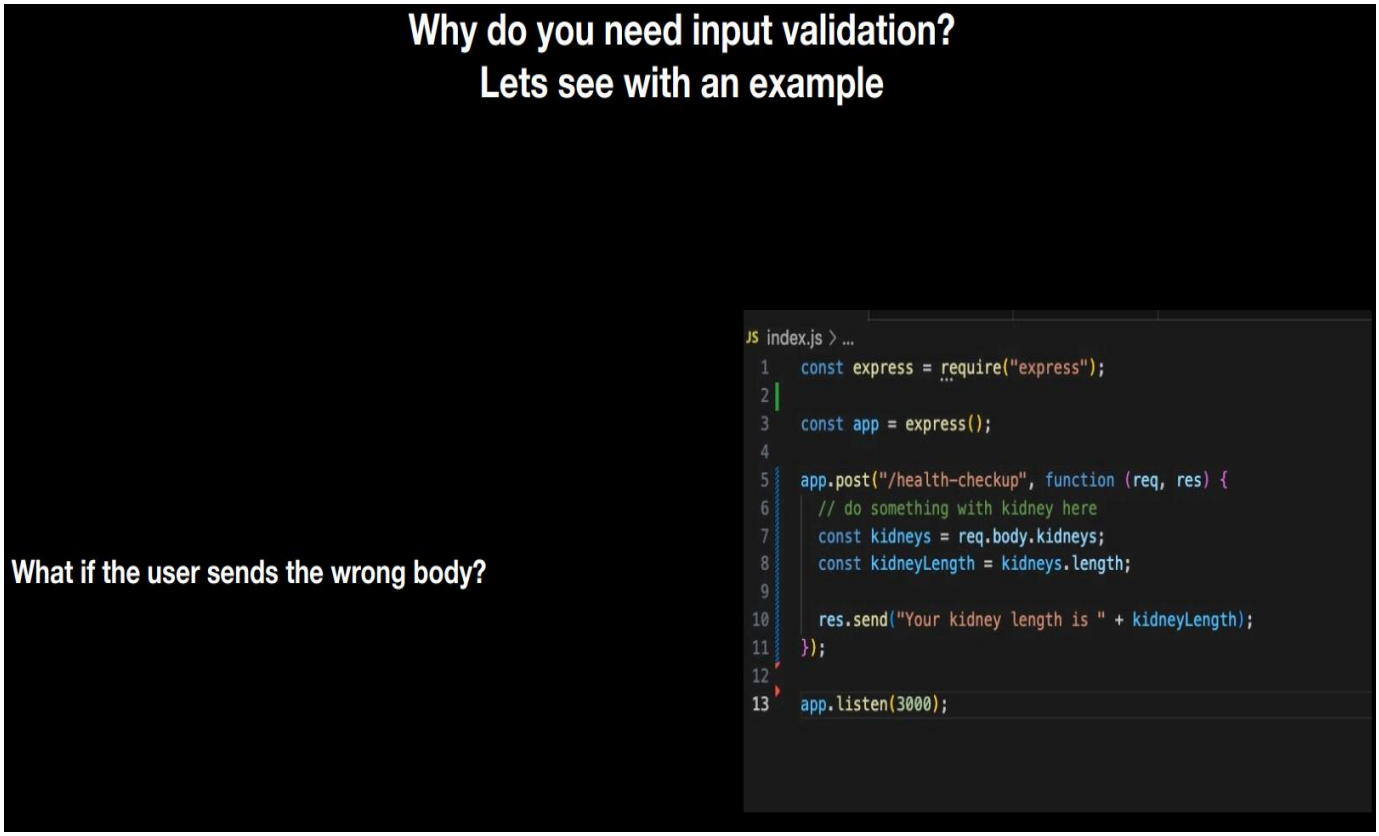
Last thing in middleware - app.use

```
const app = express();
app.use(express.json());
```

app.use(abcd), after doing this, the abcd middleware will be called for every route/method whatever route we create.

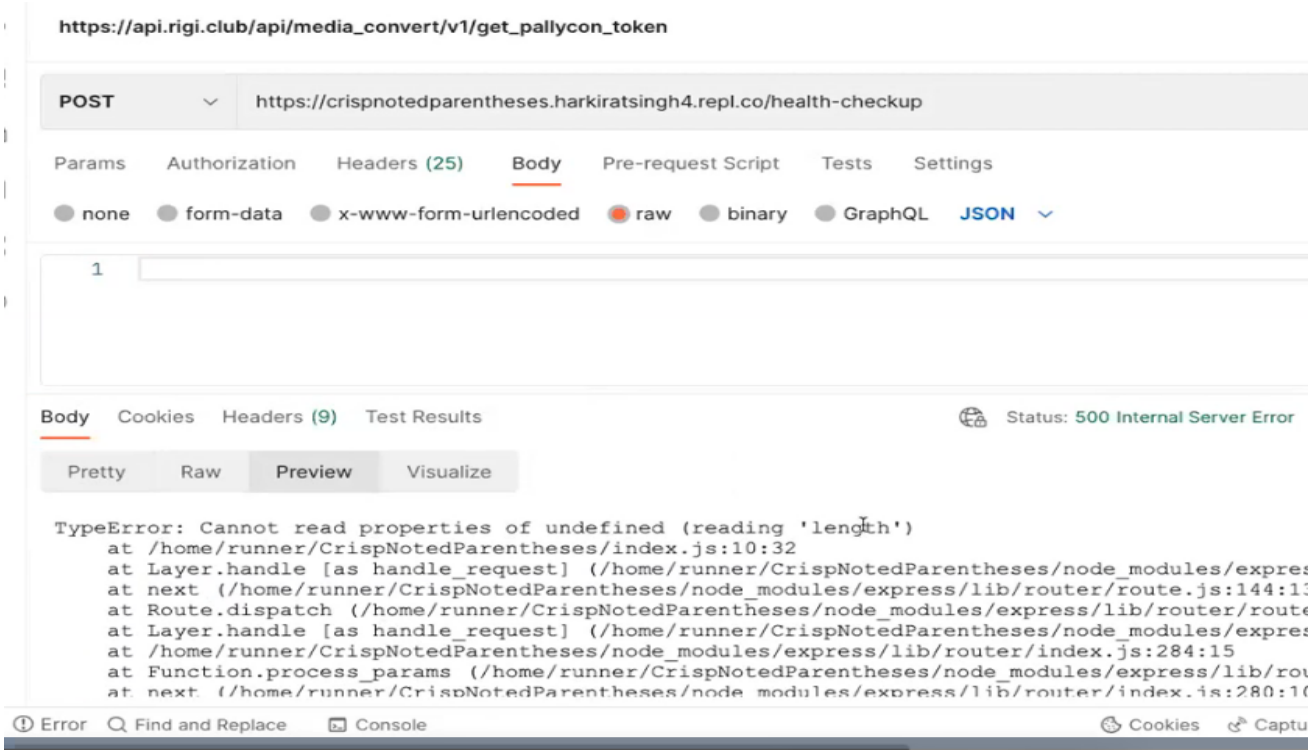
We use app.use(express.json()); so that we can post data through postman and it gets into our database (it means req.body won't get extracted in the Post method if we do not write the above middleware [body was sent through the postman]).

Input Validation:-



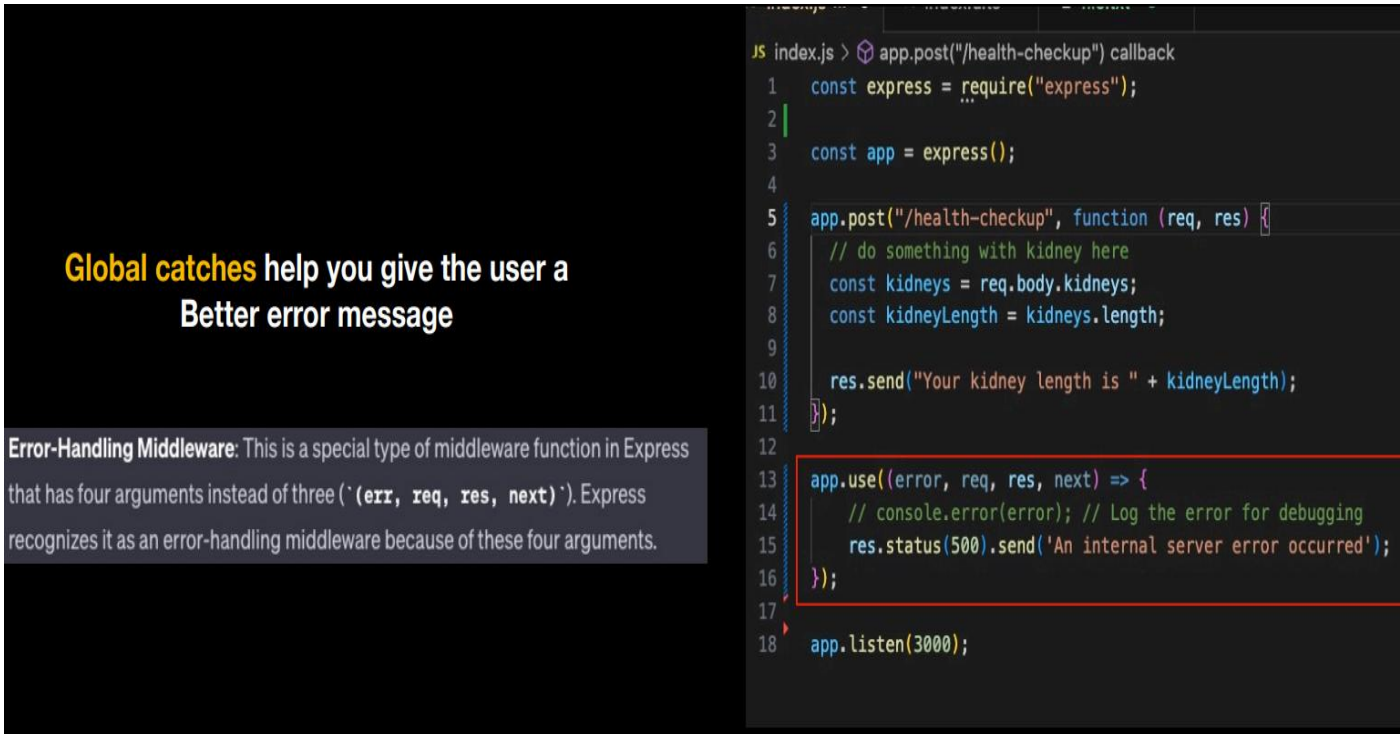
Here, kidneys are expected to be provided by an array, but what if we provide a string then it will show some error like this:-

GLOBAL CATCHES:-



Now, we introduce global catches, the work of the global catch is to catch an error that happens in any route and throw the custom error, created by the backend developer.

That's why global catches are written in the last of all the routes.



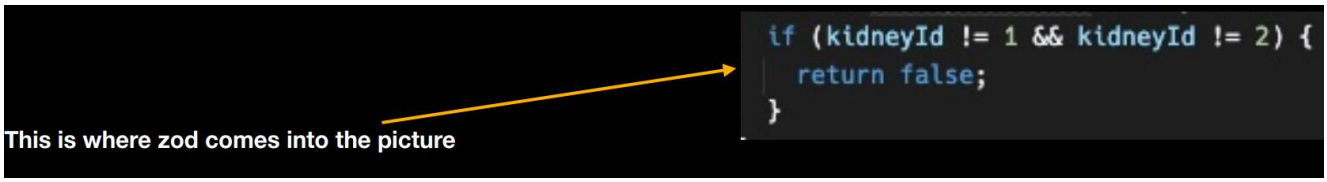
These are also termed error handling middleware.

ZOD:-

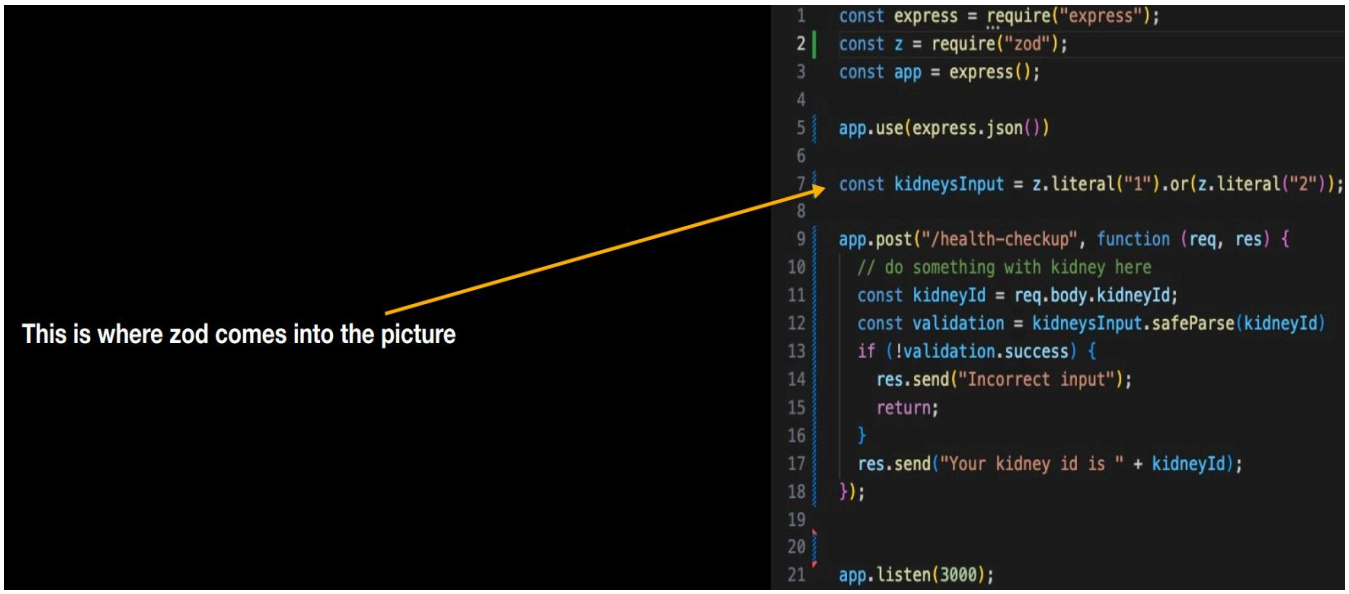
Zod (Z Object Definition) is a TypeScript-first schema declaration and validation library. It's commonly used in backend development for several reasons:

1. **Type Safety**: Zod allows developers to define schemas for data structures, such as objects, arrays, and primitive types, in a type-safe manner. This ensures that data passed through APIs or processed within the backend adheres to the specified schema, reducing the likelihood of runtime errors.
2. **Validation**: Zod provides powerful validation capabilities, allowing developers to validate input data against defined schemas. This helps ensure that incoming data meets certain criteria or constraints before being processed further, enhancing the robustness and reliability of the backend application.
3. **Declarative Syntax**: Zod offers a declarative syntax for defining schemas, making it easy to express complex data structures and validation rules concisely and clearly. This improves code readability and maintainability, as developers can easily understand the intended structure and constraints of the data.
4. **Integration with TypeScript**: Zod seamlessly integrates with TypeScript, leveraging its static type-checking capabilities to provide compile-time validation of data schemas. This enables early detection of potential errors and ensures type safety throughout the development process.

So instead of doing this:-



We do this:-



EXPLANATION:-

Basic usage

Creating a simple string schema

```
import { z } from "zod";

// creating a schema for strings
const mySchema = z.string();

// parsing
mySchema.parse("tuna"); // => "tuna"
mySchema.parse(12); // => throws ZodError

// "safe" parsing (doesn't throw error if validation fails)
mySchema.safeParse("tuna"); // => { success: true; data: "tuna" }
mySchema.safeParse(12); // => { success: false; error: ZodError }
```

As we can see, we have created our own schema, which requires input as a string, so if we send a number instead of a string, then it shows an error.

We can also design for objects like this:-

```
const schema = zod.object({
    email: zod.string(),
    password: z.string(),
    country: z.literal("IN").or(z.literal("US")),
    kidneys: z.array(z.number())
})
```

AUTHENTICATION

As you can tell by now, anyone can send requests to your backend
They can just go to postman and send a request
How do you ensure that this user has access to a certain resource?

Dumb way - Ask user to send username and password in all requests as headers

Slightly better way -

1. Give the user back a token on signup/signin
2. Ask the user to send back the token in all future requests
3. When the user logs out, ask the user to forget the token (or revoke it from the backend)

Library we need to get comfortable in - jsonwebtokens