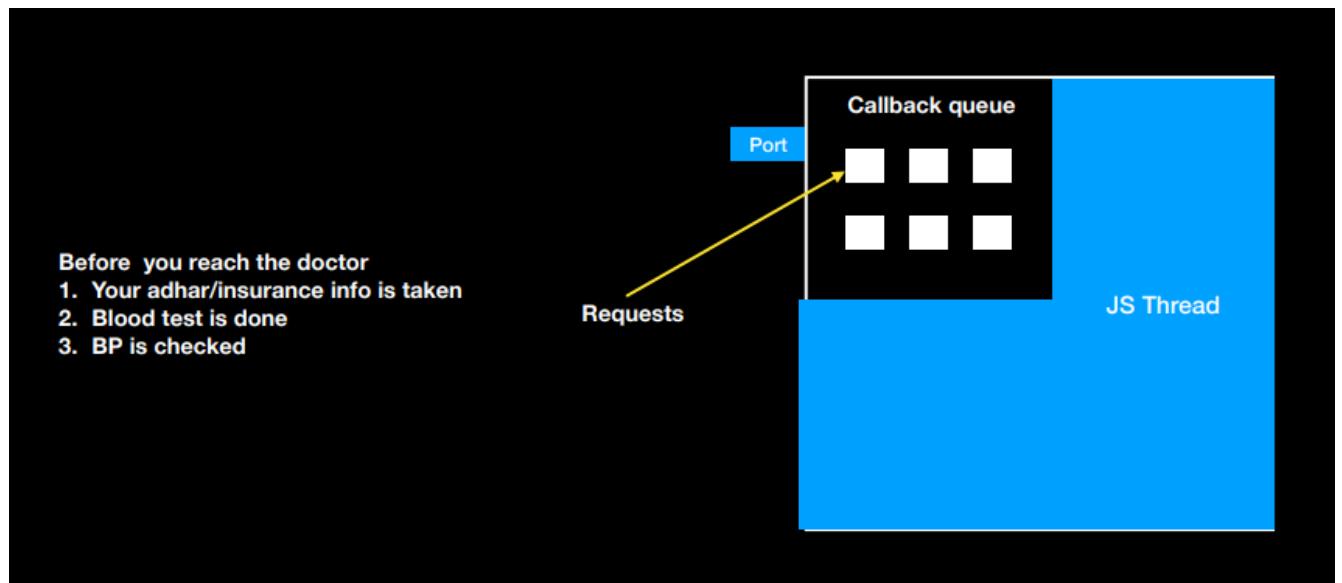
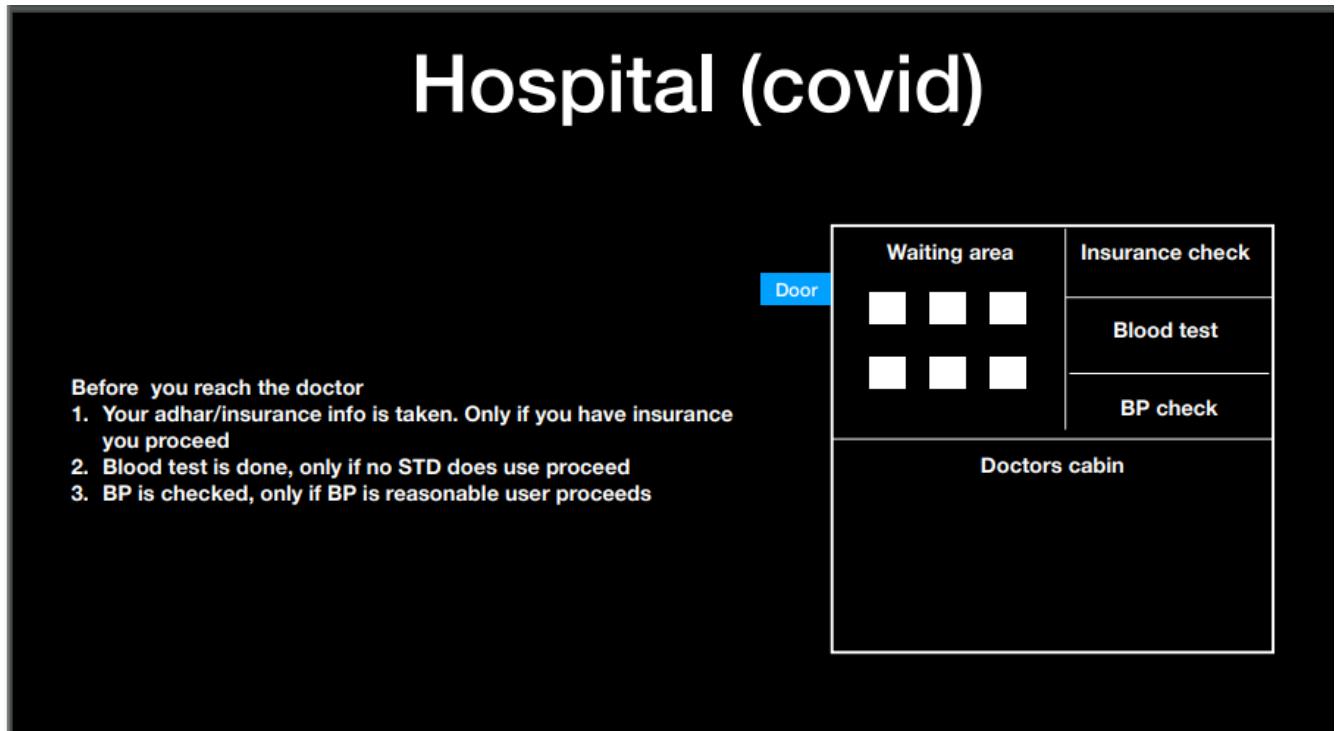


ZOD and Middleware

Lets see the example of the hospital.



Equivalent code:

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

app.get('/health-checkup', function(req, res) {
    // do health checkup
    res.send("Your heart is healthy")
})
```

How do you do

1. Auth checks? (Does this user have funds to visit the doctor)
2. Ensure input by the user is valid (BP / blood tests)

Middleware use is to do **prechecks** (either authentication and input validation(have they entered correct input))

- Before we proceed, lets add constraints to our route
1. User needs to send a kidneyId as a query param which should be a number from 1-2 (humans only has 2 kidneys)
 2. User should send a username and password in headers

```
index.js > ...
2 const express = require("express");
3
4 const app = express();
5
6 v app.get("/health-checkup", function (req, res) {
7 | // do health checks here
8 | res.send("Your heart is healthy");
9 });
10
```

There are three ways to send input:

1. query parameter
2. headers (we can see this in POSTMAN)

3. body

Lets first see the non-efficient way to do it

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

app.get('/health-checkup', function(req, res) {
    // do health checkup
    const kidneyId = req.query.kidneyId;
    const username = req.headers.username;
    const password = req.headers.password;

    if (username != "thapa" && password != "pass") {
        res.status(401).json({
            message: "User does not exist"
        })
        return;
    }
    // early returning
    if(kidneyId !=1 && kidneyId !=2){
        res.status(411).json({
            message:"wrong inputs"
        })
        return;
    }
    //Do something with the kidney here

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

This will check only if we have a single user, but LinkedIn has millions of users.

```
Week3 > Zod_Middleware > practice.js > app.get('/health-checkup') callback
```

```
1 const express = require('express');
2 const app = express();
3
4 app.get('/health-checkup',function(req,res){
5     // do health checkup
6     const kidneyId = req.query.kidneyId;
7     const username = req.headers.username;
8     const password = req.headers.password;
9
10    if (username != "thapa" && password != "pass"){
11        res.status(401).json({
12            message: "User does not exist"
13        })
14        return;
15    }
16    if(kidneyId !=1 && kidneyId !=2){
17        res.status(411).json({
18            message:"wrong inputs"
19        })
20        return;
21    }
22    // do something with kidney here
23
24    res.send("Your heart is healthy")
25})
26
```

user check

input validation

Now see the different responses

Observe the Different status code return

1. 200

GET <http://localhost:3000/health-checkup?kidneyId=2>

Send

Params • Auth Headers (9) Body Pre-req. Tests Settings • Cookies

<input checked="" type="checkbox"/> Accept-Encoding ⓘ	gzip, deflate, br
<input checked="" type="checkbox"/> Connection ⓘ	keep-alive
<input checked="" type="checkbox"/> Authorization	123456778
<input checked="" type="checkbox"/> username	thapa
<input checked="" type="checkbox"/> password	pass

Body

Pretty Raw Preview Visualize JSON

200 OK 6 ms 263 B Save Response

```
1 {  
2   "message": "Kidney is fine"  
3 }
```

2. 411

GET <http://localhost:3000/health-checkup?kidneyId=3>

Send

Params • Auth Headers (9) Body Pre-req. Tests Settings • Cookies

<input checked="" type="checkbox"/> Accept-Encoding ⓘ	gzip, deflate, br
<input checked="" type="checkbox"/> Connection ⓘ	keep-alive
<input checked="" type="checkbox"/> Authorization	123456778
<input checked="" type="checkbox"/> username	thapa
<input checked="" type="checkbox"/> password	pass

Body

Pretty Raw Preview Visualize JSON

411 Length Required 10 ms 274 B Save Response

```
1 {  
2   "message": "wrong inputs"  
3 }
```

3. 403

GET <http://localhost:3000/health-checkup?kidneyId=3> Send

Params • Auth Headers (9) Body Pre-req. Tests Settings • Cookies

<input checked="" type="checkbox"/> Accept-Encoding	①	gzip, deflate, br
<input checked="" type="checkbox"/> Connection	①	keep-alive
<input checked="" type="checkbox"/> Authorization		123456778
<input checked="" type="checkbox"/> username		thapaa
<input checked="" type="checkbox"/> password		pass

Body [Save Response](#)

Pretty Raw Preview Visualize JSON [Copy](#) [Search](#)

```
1 "message": "User does not exist"
2
3
```

What if I tell you to introduce another route that does Kidney replacement and Input need to be same?

Then if we do it without the middleware we have to write the following code

In this solution we are repeating code violating (**DRY**).

Writing this logic for suppose 20 route is bad and if we have to change something then this we tiresome

```
index.js > f app.put("/replace-kidney") callback > ...
2 const express = require("express");
3
4 const app = express();
5
6 v app.get("/health-checkup", function (req, res) {
7   // do health checks here
8   const kidneyId = req.query.kidneyId;
9   const username = req.headers.username;
10  const password = req.headers.password;
11
12 v  if (username != "harkirat" && password != "pass") {
13 v    res.status(403).json({
14     |   msg: "User doesnt exist",
15   });
16    return;
17  }
18
19 v  if (kidneyId != 1 && kidneyId != 2) {
20 v    res.status(411).json({
21     |   msg: "wrong inputs",
22   });
23    return;
24  }
25  // do something with kidney here
26
27  res.send("Your heart is healthy");
28 });
29
30 v app.put("/replace-kidney", function (req, res) {
31   // do health checks here
32   const kidneyId = req.query.kidneyId;
33   const username = req.headers.username;
34   const password = req.headers.password;
35
36 v   if (username != "harkirat" && password != "pass") {
37 v     res.status(403).json({
38       msg: "User doesnt exist",
39     });
40     return;
41   }
42
43 v   if (kidneyId != 1 && kidneyId != 2) {
44 v     res.status(411).json({
45       msg: "wrong inputs",
46     });
47     return;
48   }
49 Q // do kidney replacement logic here
50
51  res.send("Your heart is healthy");
52 });


```

Upto one extent we will solve **DRY** by making function and doing function call inside the routes.

Slightly better solution will be this.

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

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

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

    if (!usernameValidator(req.query.username, req.query.password)) {
        res.status(401).json({
            message: "User does not exist"
        })
        return;
    }
    if(!kidneyIdValidator(kidneyId)) {
        res.status(411).json({
            message:"wrong inputs"
        })
        return;
    }
    //Do something with the kidney here
```

```

    res.send("Your health 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({
      message: "User does not exist"
    })
    return;
  }

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

  res.send("Kidney replaced successfully")
})

app.listen(3000)

```

Any change need to be done only on functions now and now we dont have to do it inside the route

Optimal Solution is **Middleware**

Middleware

We can give a range of callback function while making a route

Example

```
app.get("/health-checkup", function(req, res) {  
    // , function (req, res) {  
    // )  
})
```

Now how will we know which function will be called?

```
const express = require('express');  
const app = express();  
  
app.get("/health-checkup", function() {  
    console.log("Hi from the first function")  
}, function(req, res) {  
    console.log("Hi from the second function")  
})  
  
app.listen(3000)
```

Terminal:

:

```
PS C:\Users\NTC\Desktop\Dev\Week3> node  
"c:\Users\NTC\Desktop\Dev\Week3\Zod_Middleware\practice.js"  
Hi from the first function
```

Why doesn't the second function is called

Now actually in the function parameter inside the route, we also pass the next, we will call next() when everything is alright in the first function and then it will call the second function.

Example

```
const express = require('express');  
const app = express();  
  
app.get("/health-checkup", function(req, res, next) {  
    console.log("First function")  
    next()  
})
```

```
},function(req,res) {
    console.log("Second function")
    res.send("Doone")
})
app.listen(3000)
```

Terminal

The screenshot shows the VS Code interface with the 'TERMINAL' tab selected. The terminal window displays the following text:

```
TC\Desktop\Dev\Week3\Zod_Middleware\practice.js"
PS C:\Users\NTC\Desktop\Dev\Week3> node "c:\Users\N
TC\Desktop\Dev\Week3\Zod_Middleware\practice.js"
First function
Second function
[]
```

Below the terminal, there are several status indicators: initialSetup*, Run Testcases, 0 errors, 0 warnings, 0 Prettier.

Actual syntax of route handler

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

// middleware just like any other function
function userMiddleware(req,res,next){
    let username = req.query.username;
    let password = req.query.password;
    if(username != "thapa" || password != "pass") {
        res.status(403).json({
            // if condition is not met get lost
            msg:"Invalid user",
        });
    } else{
        // on the route definition after this function pass next() then
        // only any other function after it defined will be called
        next();
    }
}
```

```

        }
    } ;

function kidneyMiddleware(req, res, next) {
    if(kidneyId !=1 && kidneyId !=2) {
        res.status(403).json({
            msg: "Incorrect input",
        });
    } else{
        next();
    }
}

app.get("/health-checkup",userMiddleware,kidneyMiddleware,
function(req,res) {
    // do something with health checkups here
    res.send("Your Basic health is okay")
})

app.get("/kidney-check",userMiddleware,kidneyMiddleware,
function(req,res) {
    // do something with kidney here
    res.send("Kidney check-up is done")
})

app.get("/heart-check",userMiddleware,function(req,res) {
    // do some heart check ups
    res.send("Your heart is healthy")
})

app.listen(3000)

```

Q/Na

1. We cannot respond to http twice
2. Express library is not promisified
3. res.send() is to send text

4. rate limiting : if we allow a person to send only certain no. of request this also comes in middleware
5. We dont need to return in middleware
6. Generally , ideally res.send() wil be last but still after this we can log something

7. Finding no. of request from middleware

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

// rate listening
let numberOfRequests = 0;

// popular example of middlepoint
function calculateRequest(req, res, next) {
  numberOfRequests++;
  // counting the load in our system
  console.log("Number of requests: ", numberOfRequests);
  next();
}

app.get("/health-checkup", calculateRequest, function(req, res) {

})
app.get("/kidney-check", calculateRequest, function(req, res) {

})
app.listen(3000)
```

Calling any of the two routes will count no. of requests.

Terminal:

```
PS C:\Users\NTC\Desktop\Dev\Week3> node "c:\Users\NTC\Desktop\Dev\Week3\Zod_Middleware\practice.js"
Number of requests: 1
Number of requests: 2
Number of requests: 3
Number of requests: 4
```

app.use()

Example To get the post request handler body. **app.use(express.json())**

If we use this the first thing which will be done is that request will go to wherever the this function returns (it act and get post body)

Here we are calling `express.json()` because it return a function.

It simply means that this middleware will be called everywhere.

Usecase:

If we know that a middleware will be used everywhere then we will use

app.use(middlewareName)

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

let numberOfRequests = 0;
function calculateRequest(req, res, next) {
    numberOfRequests++;
    console.log("Number of requests: ", numberOfRequests);
    next();
}
app.use(calculateRequest);
// now this middleware will be called for every request, after this line

app.post("/health-checkup", function(req, res) {
    res.json({
        message:"Health checkup is done"
    })
})
app.get("/kidney-check", function(req, res) {
})
app.listen(3000)
```

Even though we haven't pass the middleware or function calculateRequest it still counts the no. of request when ever we pass any of two routes (eq <http://localhost:3000/health-checkup>)

Terminal:

```
PS C:\Users\NTC\Desktop\Dev\Week3> node "c:\Users\NTC\Desktop\Dev\Week3\Zod_Middleware\practice.js"
Number of requests: 1
Number of requests: 2
Number of requests: 3
Number of requests: 4
```

If we dont use next() then response ("Health checkup is done") will not be sent.

Why we have to specify **body** only? Not req.headers or req.query?? Why!!!

Simply because we dont know what will body contain it can be html, json, text.

We use **app.use(express.json())** that is **express.json()** middleware because we are saying that we are expecting body as json please handle it.

Assignment

Other use cases of middleware (assignment) =

1. Count the number of requests
2. Find the average time your server is taking to handle requests

Why do we need input validation??

Let's see an example

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

app.post("/health-checkup", function(req, res) {
    // do something with the health checkup
    const kidneys = req.body.kidneys;
    const kidneyLength = kidneys.length;

    res.send("You have kidney is " + kidneyLength)
});
app.listen(3000)
```

Let's send post request

The screenshot shows the Postman application interface. At the top, it displays a POST request to the URL `http://localhost:3000/health-checkup`. Below the URL, there are tabs for Params, Auth, Headers (11), Body (selected), Pre-req., Tests, and Settings. Under the Body tab, the content type is set to JSON. The request body is defined as:

```
1 {
2     "kidneys": [1,2]
3 }
```

At the bottom of the interface, the response status is shown as 200 OK with a response time of 52 ms and a size of 251 B. The response body is displayed as:

```
1 Your kidney length is 2
```

It is our job to handle errors , since backend is in the web then user can send post request with rubbish thing in the body not according to the type mentioned and 100 different things. **We must handle it!**

There can be several different error etc.

The screenshot shows a Postman interface. At the top, there's a 'POST' method dropdown and a URL field containing 'http://localhost:3000/health-checkup'. On the right, there's a 'Send' button. Below the URL, there are tabs for 'Params', 'Auth', 'Headers (11)', 'Body', 'Pre-req.', 'Tests', and 'Settings'. The 'Body' tab is active and has a 'raw' dropdown set to 'JSON'. The JSON payload is:

```
1 {  
2   "kidneys":123  
3 }
```

At the bottom, there's a 'Body' dropdown, a status bar showing '200 OK 8 ms 259 B', and a 'Save Response' button. Below the status bar, there are buttons for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'HTML'. The 'Pretty' button is selected. The response body is:

```
1 Your kidney length is undefined
```

Hence we must do input validation , **to prevent server from crashing.**

By seeing error Random error good can crash the backend server.

We have to check. Ten different check whether there is some input or not, is it array of numbers , or they send array or not .

If we simply write input validation by doing if else then it will get ugly and look unclean. We will use the **zod** library

```
if(!kidneys){  
    res.status(400).send("Please provide kidneys")  
}
```

Error handling middleware

There is another type of problem is that the if we dont do anything person who send request can see the exception message send by the system. This will expose **backend logic. Express also take care of it in production one.**

Another way is .

global caches.

Code:

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

app.use(express.json());
app.post("/health-checkup",function(req,res){
    // do something with the health checkup
    const kidneys = req.body.kidneys;
    const kidneyLength = kidneys.length;

    res.send("Your kidney length is " + kidneyLength)
});

// global caches
// another middleware that take 4 input except of three
// whenever there is any exception in routes it is send here
// by this we hide the exception from the user
app.use(function(err,req,res,next){
    res.json({
        message:"Sorry something went wrong with our system"
    })
})

app.listen(3000)
```

For example here in the below example we have send an empty json

```
errorCount++;
```

We keep track of how many time exception were thrown and also log the exception and once we get suppose 100+ exception then we inform developer of it . To fix the system

The screenshot shows a Postman interface with a POST request to `http://localhost:3000/health-checkup`. The Body tab is selected, showing a JSON response:

```
1 {  
2   "message": "Sorry something went wrong with our system"  
3 }
```

The response status is 200 OK.

Error handling middleware: is a special type of middleware function in Express that has four arguments instead of three ('(err, req, res, next)'). Express recognize it as an error-handling middleware because of these four arguments.
Global caches : help you give the user a better error message

```
js index.js > app.post("/health-checkup") callback
1 const express = require("express");
2
3 const app = express();
4
5 app.post("/health-checkup", function (req, res) {
6   // do something with kidney here
7   const kidneys = req.body.kidneys;
8   const kidneyLength = kidneys.length;
9
10  res.send("Your kidney length is " + kidneyLength);
11});
12
13 app.use(error, req, res, next) => {
14   // console.error(error); // Log the error for debugging
15   res.status(500).send('An internal server error occurred');
16 };
17
18 app.listen(3000);
```

What is the use of next

It can be that one function logs the error to the server

The second function can do is that it count no. of exception and inform the developer when exception floods to much

The third function can simply display some error message to the user

Zod

How can we do better input validation??

How can you do better input validation?

This is very hard to scale
What if you expect a complicated input?

```
if (kidneyId != 1 && kidneyId != 2) {  
    return false;  
}
```

It is quite hard to track no. of input validation

Here comes the input validation library

Zod is one of the popular library

Zod

Schema validation

We install it by doing

npm install zod

By using Zod we are parsing the data from the end user its like are you (the user) sending me the correct data this is what **Zod** does

Code:

```
const express = require('express');  
const zod = require('zod');  
const app = express();  
const schema = zod.array(zod.number())  
//This much is enough to make zod understand what be schema(structure or basically type of input) of our input, for example here we want input as an array of numbers
```

```
app.use(express.json());
app.post("/health-checkup", function(req, res) {
    // do something with the health checkup
    const kidneys = req.body.kidneys;
    const response = schema.safeParse(kidneys);

    res.send({
        response
    })
});

app.listen(3000)
```

The screenshot shows the Postman application interface. At the top, there is a header bar with 'POST' selected, the URL 'http://localhost:3000/health-checkup', a 'Send' button, and a dropdown menu. Below the header, there are tabs for 'Params', 'Auth', 'Headers (11)', 'Body', 'Pre-req.', 'Tests', and 'Settings'. The 'Params' tab is currently active. Under 'Query Params', there is a table with one row containing 'Key' and 'Value' columns, both of which are empty.

In the main body area, there is a 'Body' dropdown set to 'Pretty'. To the right of the dropdown are status indicators: a globe icon, '200 OK', '12 ms', '402 B', and a 'Save Response' button. Below these, there are tabs for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'JSON'. The 'Pretty' tab is selected. The JSON response is displayed in a code editor-like format:

```
1  {
2      "response": {
3          "success": false,
4          "error": {
5              "issues": [
6                  {
7                      "code": "invalid_type",
8                      "expected": "array",
9                      "received": "undefined",
10                     "path": [],
11                     "message": "Required"
12                 }
13             ],
14             "name": "ZodError"
15         }
16     }
```

Zod give us everything , validation success or not and bunch of error message .

It is very useful to send error messages to the user.

Lets observe the information send by the zod and then we do some changes in code

```
app.post("/health-checkup", function(req, res) {
    // do something with the health checkup
    const kidneys = req.body.kidneys;
    const response = schema.safeParse(kidneys);

    if(!response.success) {
        res.status(411).json({
            message:"Invalid input"
        })
    } else{
        res.send({
            response
        })
    }
}) ;
```

And now lets call the it in postman

1. response.success is false

The screenshot shows a Postman interface with the following details:

- Method:** POST
- URL:** http://localhost:3000/health-checkup
- Body:** JSON (empty)
- Response Status:** 411 Length Required
- Response Body:** {"message": "Invalid input"}

2. response.success is true

The screenshot shows the Postman interface. At the top, it says "POST http://localhost:3000/health-checkup". Below that, under "Body", the "JSON" tab is selected, showing the following JSON payload:

```
1 {  
2   "kidneys": [1,2]  
3 }
```

At the bottom, the response is displayed as:

```
1 {  
2   "response": {  
3     "success": true,  
4     "data": [  
5       1,  
6       2  
7     ]  
8   }  
9 }
```

The status bar at the bottom indicates a 200 OK response with 19 ms latency and 277 B size.

Describing the schema is crucial and hard part.

Eg email : this is suppose to be string with an @ symbol in between

password: atleast 8 letters

country: "IN","US"

Lets write zod schema for these

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

```
// literal is used to specify that the input should be exactly the
// same as the value passed in the literal
// usage of "or" is the syntax of zod
kidneys: zod.array(zod.number())
})
```

Simple example

```
const zod = require('zod');
//If this is an array of strings,return true, else return false
// function validateInput(arr) {
// if(typeof arr != "object" && arr.length >=1 && typeof item != "string") {
//
// }
// }

// using zod
function validateInput(arr) {
    const schema = zod.array(zod.number());
    const response = schema.safeParse(arr);
    return response.success;

}
console.log(validateInput([1,2,3,"sbc",5]))
```

Terminal

False

Write zod schema for the following.

email : this is supposed to be a string with an @ symbol in between
password: at least 8 letters

```
const z = require("zod")
const schema = z.object({
    email: z.string().email(),
    password: z.string().min(8)
})
```

<https://zod.dev/?id=introduction> To learn more about zod

Example:

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

function validateInput(obj) {
    const schema = zod.object({
        email: zod.string().email(),
        password: zod.string().min(8).max(20),
    })
    const response = schema.safeParse(obj);
    console.log(response);
}

validateInput({
    email:"abcdefg@gmail.com",
    password:"abcdefg"
})
validateInput({
    email:"abcdefg@gmailcom",
    password:"abcdefg"
})
validateInput({
    email:"abcdefg@gmail.com",
    password:"abcdefghabcdefghabcdefg"
})
```

Terminal:

```
PS C:\Users\NTC\Desktop\Dev\Week3> node "c:\Users\NTC\Desktop\Dev\Week3\Zod_Middleware\practice.js"
{
  success: true,
  data: { email: 'abcdefg@gmail.com', password: 'abcdefg' }
}
{ success: false, error: [Getter] }
{ success: false, error: [Getter] }
PS C:\Users\NTC\Desktop\Dev\Week3> []
```

Understanding zod in Http server

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

function validateInput(obj) {
  const schema = zod.object({
    email: zod.string().email(),
    password: zod.string().min(8)
  })
  const response = schema.safeParse(obj);
  console.log(response);
  return response;
}

app.post("/login", function(req, res) {
  const response = validateInput(req.body);
  if(!response.success) {
    res.json({
      msg:"Your inputs are invalid"
    })
    return;
  }
})
app.listen(3000)
```

Q/na

1. Why do we call express.json() like this inside app.use?

```
app.use(express.json())
```

We use it because it is something like this

```
function middleware(){  
    return function(req, res, next){  
  
    }  
}
```

2. app.use () put in all the routes

- 3.

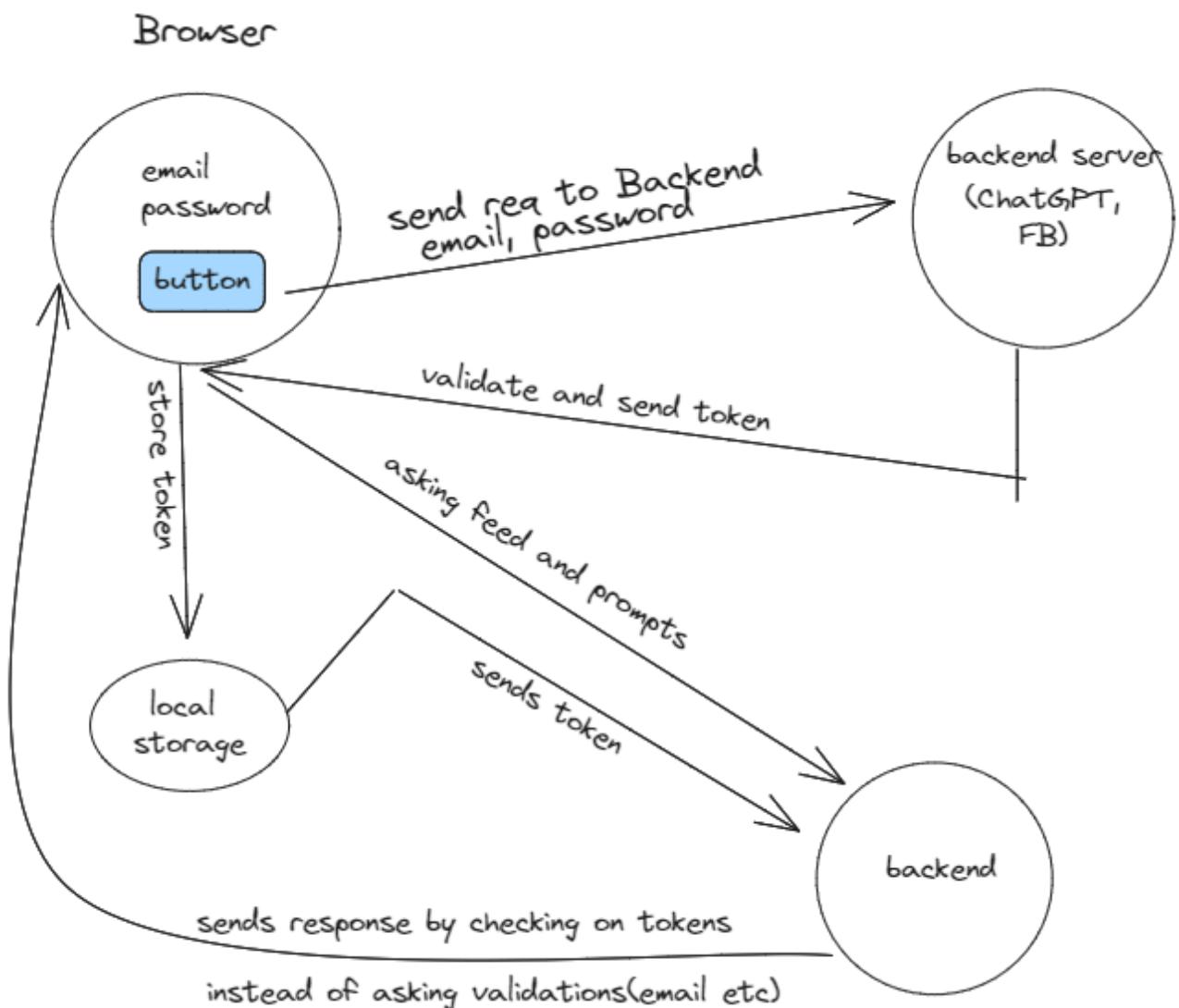
Intro to Authentication

As we 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 the 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)



Note: Check 1:47:00 to see how you can use somebody else token and use it to send a request from POSTMAN or elsewhere

Q/Na

1. What will the zod send when we input the wrong email address?

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

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

```
const response = schema.parse("abccxsd")
console.log(response.errors);
```

Terminal

```
.....
.....
issues: [
  {
    validation: 'email',
    code: 'invalid_string',
    message: 'Invalid email',
    path: []
  }
],
addIssue: [Function (anonymous)],
addIssues: [Function (anonymous)]
}
```

2. Understanding global caches

Let see an example

Code:

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

app.get('/', function(req, res) {
  console.log(req.body.kidneys.length);
  res.json({
    msg: "done"
  })
})
```

```
)  
  
app.listen(3000)
```

Now if we send a get request, since get request is trying to access something undefined it will throw an error



A screenshot of a terminal window showing a stack trace for a `TypeError`. The error message is: `TypeError: Cannot read properties of undefined (reading 'kidneys')`. The stack trace shows the error occurred at `practice.js:6:26`, which is part of the `Layer.handle` function. It traces back through several layers of the Express.js middleware stack, including `next`, `Route.dispatch`, `Layer.handle`, `Function.process_params`, `next`, `expressInit`, and finally `Layer.handle`.

This is a bad error to send to user. We keep the code as it is but at the end add `app.use(function(err, req, res, next))`

```
app.use(function(err, req, res, next) {  
  res.send({  
    message:"internal error"  
  })  
})
```

Terminal:

The screenshot shows the Postman application interface. At the top, there's a header with 'GET' and 'http://localhost:3000/'. Below it, a navigation bar includes 'Params', 'Auth', 'Headers (9)', 'Body' (which is underlined), 'Pre-req.', 'Tests', and 'Settings'. To the right of the navigation bar are 'Cookies' and 'Beautify' buttons. The main area has tabs for 'raw' and 'JSON' (which is selected). A large text input field contains the number '1'. Below the input field, there's a status bar showing '200 OK' and '56 ms'. On the left, there's a 'Body' dropdown with 'Pretty', 'Raw', 'Preview' (which is selected), and 'Visualize' options. The preview section shows the JSON response:

```
{"message": "internal error"}
```

3. How can we make sure that email end with google.com

`z.string().email().endsWith('@google.com')`

4. How can we use bunch of middleware together

```
const middleware = [express.json(), userValidator, kidneyValidator];

app.get('/', ...middleware, function(req, res) {})
```

5. req in global can be use many usecase eg Which user is facing problem ,

or which ip

6. Whenever we are sending sensitive data we send via headers

For get request we use query

For post request we use body

7. We can use same backend for app and web

8. Sending data from middleware

```
function middleware(req, res, next){
```

```
    req.user=1
```

```
}
```

```
app2.get("/", middleware, funciton(req, res){
```

```
    console.log(req.user);
```

```
})
```

9. Middleware are synchronous they are asynchronous only if there is a asynchronous function like setTimeout inside it.

Eg

```
function middleware(req, res, next){
```

```
    req.user = 1;
```

```
    setTimeout(function(){
```

```
        next()
```

```
    }, 1000)
```

```
}
```

10. Where we write middleware is important , generally the thing is that first we have the initial middleware and then route and then global cache. A exception in any route can only reach middleware below (app.use()) not above hence the position of global catch is important

- 11.What is the use of next() in route handler?

```
12. app.get("/", ...middleware, function(req, res){
```

```
    res.json({
```

```
        msg: "Done"
```

```
    })
```

```
})
```

Although we may think that res.json() will do early return but it doesn't it is just a function. We as developers should not write anything generally after res.json() but this function doesn't early return

```
13. function middleware(req,res,next){  
    if(abc){  
        res.send()  
        next()  
        // this will result in error, because we send and call the function  
    also  
    } else{  
    }  
}
```

Another example (correct usage)

```
if(abc){  
    res.send()  
} else{  
    next()  
}
```

14. We can also use try catch in route