Express Js

Disclaimer:

These notes have been created by *Khushal Savalakha* (khushal.sav@gmail.com) and are based on practical applications and personal understanding of the concepts. Although I have thoroughly checked and verified the information and code provided, there is a possibility that some aspects may contain errors or inaccuracies. Please use this content as a reference and verify details independently as needed. I am sharing these notes as part of a GitHub repository to help others learn, and I hope they are beneficial.

▼ How to set Path & use of urlencoded and express.json()

For send file.

```
var expr = require("express");
const app = expr();
var nm = require("nodemailer");
var path = require("path");
app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname, "../Public", "Mail.html
});
app.get("/feedback", (req, res) => {
  var trans = nm.createTransport({
    host: "smtp.gmail.com",
    port: 465,
    auth: {
      user: "sender@gmail.com",
      pass: "2stepverificationkey",
    },
  });
  var mailoption = {
    from: "noreply@gmail.com<noreply>",
    to: req.query.mail,
    bcc: "bcc@gmail.com",
    subject: "Feedback response",
    html:
      "<h1>Hello " +
      req.query.uname +
      "!</h1><h2> Thank you for submitting the feedback.<br
      req.query.feedback +
      "</span></h2><h3> Additional comments: " +
      req.query.comments +
      "</h3>",
  };
  trans.sendMail(mailoption, (err, info) => {
    if (err) {
      console.error(err);
    }
    console.log(info);
  });
  res.send("Thank you for your feedback.");
});
```


Khushal Savalakha

```
app.listen(5001);
//gmail >> manage account >> security >> 2 step verificatic
```

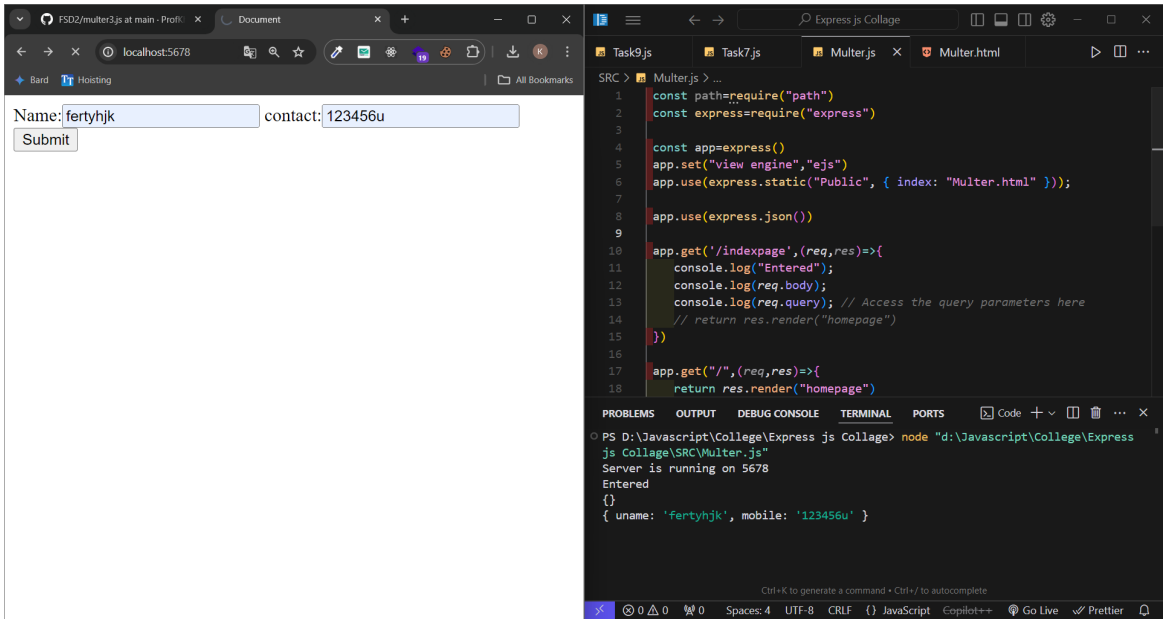
```
<html>
<body>
<form action="/process" method="post">
First Name: <input type="text" name="first_name"> <br>
Last Name: <input type="text" name="last_name">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

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

app.use(express.urlencoded({ extended: false }));
// app.use(express.static("../Public",{index: 'T2_practice.
// app.use(express.static(path.join(__dirname, '../Public',
//both commented part does not give output .
app.use(express.static(path.join(__dirname, '../Public'), {

app.post('/process', function (req, res) {
res.send(req.body);
})
app.listen(7002,()=>{console.log('Server is Running...');})
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, init
    <title>Document</title>
</head>
<body>
```

```
    <form method="get" action="/indexpage">
        <label>Name:<input type="text" name='uname'></label
        <label>contact:<input type="text" name='mobile'></l
        <input type="submit">
    </form>
</body>
</html>
```

```
const path=require("path")
const express=require("express")

const app=express()
app.set("view engine","ejs")
app.use(express.static("Public", { index: "Multer.html" })

app.use(express.json())


app.get('/indexpage',(req,res)=>{
   console.log("Entered");
   console.log(req.body);
   console.log(req.query); // Access the query parameters
   // return res.render("homepage")
})

app.get("/",(req,res)=>{
   return res.render("homepage")
})

app.listen(5678,()=>{
   console.log("Server is running on 5678");
})
```




```
const path=require("path")
const express=require("express")

const app=express()
app.set("view engine","ejs")
app.use(express.static("Public", { index: "Multer.html" })

app.use(express.json())
app.post('/indexpage',(req,res)=>{
   console.log("Entered");
   console.log(req.body);
   console.log(req.query); // Access the query parameters
   // return res.render("homepage")
})


/*
app.get('/indexpage',(req,res)=>{
   console.log("Entered");
   console.log(req.body);
   console.log(req.query); // Access the query parameters
   // return res.render("homepage")
})
```

```
*/
 app.get("/",(req,res)=>{
    return res.render("homepage")
 })

 app.listen(5678,()=>{
    console.log("Server is running on 5678");
 })

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, init
    <title>Document</title>
</head>
<body>
    <form method="post" action="/indexpage">
        <label>Name:<input type="text" name='uname'></label
        <label>contact:<input type="text" name='mobile'></l
        <input type="submit">
    </form>
</body>
</html>

Server is running on 5678
Entered
{}
{}
why we can't load data in Multer.js ?
solve the problem and give me modified code.
```

The reason why `req.body` is empty in your `POST /indexpage` handler is that you are only using `express.json()` middleware, which is meant for parsing JSON payloads. Since your form is submitting data as URL-encoded data (the default content type for HTML forms), you need to use the `express.urlencoded()` middleware to parse the URL-encoded form data.

Here's how you can modify your code to properly handle form submissions:

Khushal Savalakha

Modified Express Application (`Multer.js`)

Add `express.urlencoded()` middleware to parse URL-encoded form data:

```
const path = require("path");
const express = require("express");

const app = express();
app.set("view engine", "ejs");
app.use(express.static("Public", { index: "Multer.html"
}));

// Middleware to parse URL-encoded form data
app.use(express.urlencoded({ extended: true })); // This
is for parsing form data
app.use(express.json()); // This is for parsing JSON dat
a

app.post('/indexpage', (req, res) => {
    console.log("Entered");
    console.log(req.body); // Access the parsed form dat
a here
    console.log(req.query); // Access the query paramete
rs here (if any)
    // Do something with req.body.uname and req.body.mob
ile
    res.send("Form data received");
    // return res.render("homepage")
});

/*
app.get('/indexpage', (req, res) => {
    console.log("Entered");
    console.log(req.body);
    console.log(req.query); // Access the query paramete
rs here
    // return res.render("homepage")
});
*/
```


Khushal Savalakha

```
app.get("/", (req, res) => {
    return res.render("homepage");
});

app.listen(5678, () => {
    console.log("Server is running on 5678");
});
```

Explanation

- `express.urlencoded({ extended: true })`: This middleware is used to parse URL-encoded data (form data) submitted via a POST request. The `extended: true` option allows for parsing of richer data, including arrays and objects.
- `express.json()`: This middleware is used to parse JSON data, but since your form data is URL-encoded, it's not relevant for this case.

HTML Form

Make sure your HTML form is submitting data via a POST request:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, i
nitial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <form method="post" action="/indexpage">
        <label>Name:<input type="text" name="uname"></la
bel>
        <label>Contact:<input type="text" name="mobile">
</label>
        <input type="submit">
    </form>
```


Khushal Savalakha

```
</body>
</html>
```

Running the Server

1. **Start the Server**:

   ```
   node Multer.js
   ```

2. **Access the Form**:

   - Open a web browser and navigate to `http://localhost:5678/`.

3. **Submit the Form**:

   - Fill in the form fields and submit.

Server Output

After submitting the form, the server console should display the form data:

```
Server is running on 5678
Entered
{ uname: 'John Doe', mobile: '1234567890' } // Example d
ata
{}
```

Now, the form data is correctly parsed and available in `req.body`.

> 💡 app.use(express.urlencoded({ extended: true })); // This is for parsing form data
> it is needed when we are using post and put am i true ?

Yes, you are correct. The `express.urlencoded({ extended: true })` middleware is used for parsing URL-encoded data, which is typically the format of form data submitted via POST and PUT requests.

When to Use `express.urlencoded()`

- **POST Requests**: When you submit a form with the `POST` method, the form data is included in the body of the request. The `express.urlencoded()` middleware is

Khushal Savalakha

necessary to parse this data so that it can be accessed via `req.body`.

- **PUT Requests**: Similar to POST requests, PUT requests can also include data in the body, often in URL-encoded form if submitted from an HTML form. The `express.urlencoded()` middleware is used to parse this data as well.

Summary

- **POST Requests**: Use `express.urlencoded({ extended: true })` to parse form data in the body.
- **PUT Requests**: Use `express.urlencoded({ extended: true })` to parse form data in the body.

Full Example

Here's the complete code demonstrating the use of `express.urlencoded()` for a POST request:

Express Application (`Multer.js`)

```
const path = require("path");
const express = require("express");

const app = express();
app.set("view engine", "ejs");
app.use(express.static("Public", { index: "Multer.html"
}));

// Middleware to parse URL-encoded form data
app.use(express.urlencoded({ extended: true }));
app.use(express.json());

app.post('/indexpage', (req, res) => {
    console.log("Entered");
    console.log(req.body); // Access the parsed form dat
a here
    console.log(req.query); // Access the query paramete
rs here (if any)
    res.send("Form data received");
    // return res.render("homepage")
```


Khushal Savalakha

```
});

/*
app.get('/indexpage', (req, res) => {
    console.log("Entered");
    console.log(req.body);
    console.log(req.query); // Access the query paramete
rs here
    // return res.render("homepage")
});
*/

app.get("/", (req, res) => {
    return res.render("homepage");
});

app.listen(5678, () => {
    console.log("Server is running on 5678");
});
```

HTML Form

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, i
nitial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <form method="post" action="/indexpage">
        <label>Name:<input type="text" name="uname"></la
bel>
        <label>Contact:<input type="text" name="mobile">
</label>
        <input type="submit">
    </form>
```


Khushal Savalakha

```
</body>
</html>
```

Running the Server

1. **Start the Server**:

   ```
   node Multer.js
   ```

2. **Access the Form**:

   - Open a web browser and navigate to `http://localhost:5678/`.

3. **Submit the Form**:

   - Fill in the form fields and submit.

Server Output

After submitting the form, the server console should display the form data:

```
Server is running on 5678
Entered
{ uname: 'John Doe', mobile: '1234567890' } // Example d
ata
{}
```

By including `express.urlencoded({ extended: true })` middleware, you ensure that URL-encoded form data submitted via POST (and PUT) requests is correctly parsed and accessible via `req.body`.

> 💡 means whenever we want large data then we have to use post and that time we have to convert in binary and then it is convert in html form by url.encoded am i true?

Yes, you're correct. When dealing with large data, it's common to use POST requests because they allow for larger payloads compared to GET requests. Data sent via POST requests can be encoded in various formats, including binary, and then converted into HTML form data using URL encoding before being sent to the server for processing. This allows for efficient transmission and handling of large amounts of data.

Khushal Savalakhia

💡 give me information when we have to use app.use(express.urlencoded({ extended: true }));?

The `express.urlencoded({ extended: true })` middleware is used to parse URL-encoded data submitted in the body of HTTP requests. Here's when you would typically use it:

1. **HTML Forms with POST method**: When you have an HTML form on your website, and it's submitted using the `POST` method, the form data is sent in the body of the request in a URL-encoded format. To access this form data in your Express application, you need to use `express.urlencoded({ extended: true })` to parse it.

   ```
   app.use(express.urlencoded({ extended: true }));
   ```

2. **APIs Receiving Form Data**: If your Express application exposes APIs that receive form data, and clients send this data using the `application/x-www-form-urlencoded` content type, you'll need the `express.urlencoded({ extended: true })` middleware to parse the incoming data.

3. **Processing Form Data in POST and PUT Requests**: Whenever your application receives form data in the body of a `POST` or `PUT` request, especially from HTML forms, you should use `express.urlencoded({ extended: true })` to parse it. This middleware will make the parsed data available in the `req.body` object.

4. **Handling PUT Requests**: Similar to `POST` requests, if your application receives data in the body of a `PUT` request, and this data is in URL-encoded format, you need to use `express.urlencoded({ extended: true })` to parse it.

5. **Parsing Form Data in Middleware**: If you have custom middleware that needs to process form data from incoming requests, you can use `express.urlencoded({ extended: true })` within that middleware to parse the form data.

In summary, you use `express.urlencoded({ extended: true })` whenever your Express application needs to parse URL-encoded form data from the body of incoming requests, typically in `POST` and `PUT` requests.

💡 express.urlencoded({ extended: true })can we use it when app.get is present ?

Yes, you can use `express.urlencoded({ extended: true })` middleware even when `app.get` routes are present in your Express application. However, it's important to note

Khushal Savalakha

that the `express.urlencoded()` middleware is primarily designed for parsing form data from `POST` and `PUT` requests, where the form data is typically included in the body of the request.

While it's technically possible to use `express.urlencoded({ extended: true })` with `app.get`, it's generally unnecessary and may not serve any meaningful purpose. This middleware is specifically tailored for handling form submissions via `POST` and `PUT` requests, where form data needs to be parsed from the request body.

Here's why using `express.urlencoded({ extended: true })` with `app.get` routes is not common or necessary:

1. **GET Requests and URL Parameters**: For `GET` requests, especially those with URL parameters, the data is usually sent as part of the URL query string, and it's accessed using `req.query`. The `express.urlencoded()` middleware is not required for parsing URL parameters because Express automatically parses and populates `req.query` with the query string parameters.
2. **Form Data in GET Requests**: HTML forms with the `GET` method typically submit data as query parameters in the URL, not in the request body. Therefore, there's no form data to parse from the request body, and `express.urlencoded()` is not needed.
3. **Middleware Logic**: If you have middleware logic that is intended to process form data from `POST` or `PUT` requests, it should be applied selectively to those routes using `app.post` or `app.put`, respectively. Using `express.urlencoded()` with `app.get` may introduce unnecessary overhead and could potentially lead to unexpected behavior.

In summary, while you technically can use `express.urlencoded({ extended: true })` with `app.get`, it's generally not recommended or necessary because `GET` requests typically do not contain form data in the request body. Instead, form data in `GET` requests is usually included in the URL query string and accessed using `req.query`.

💡 give me it's example.

Certainly! Here's an example demonstrating the use of `express.urlencoded({ extended: true })` middleware in an Express application with both `GET` and `POST` routes:

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

// Middleware to parse URL-encoded form data
```


Khushal Savalakha

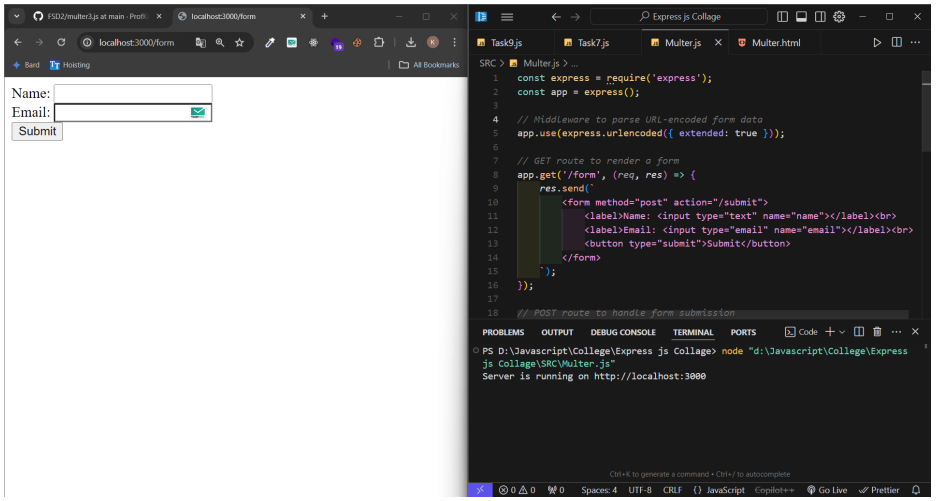
```
app.use(express.urlencoded({ extended: true }));

// GET route to render a form
app.get('/form', (req, res) => {
    res.send(`
        <form method="post" action="/submit">
            <label>Name: <input type="text" name="name">
</label><br>
            <label>Email: <input type="email" name="emai
l"></label><br>
            <button type="submit">Submit</button>
        </form>
    `);
});

// POST route to handle form submission
app.post('/submit', (req, res) => {
    // Access form data from req.body
    const name = req.body.name;
    const email = req.body.email;

    // Process the form data
    // For demonstration, just logging the data
    console.log('Received form submission:');
    console.log('Name:', name);
    console.log('Email:', email);

    res.send('Form submitted successfully!');
});

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

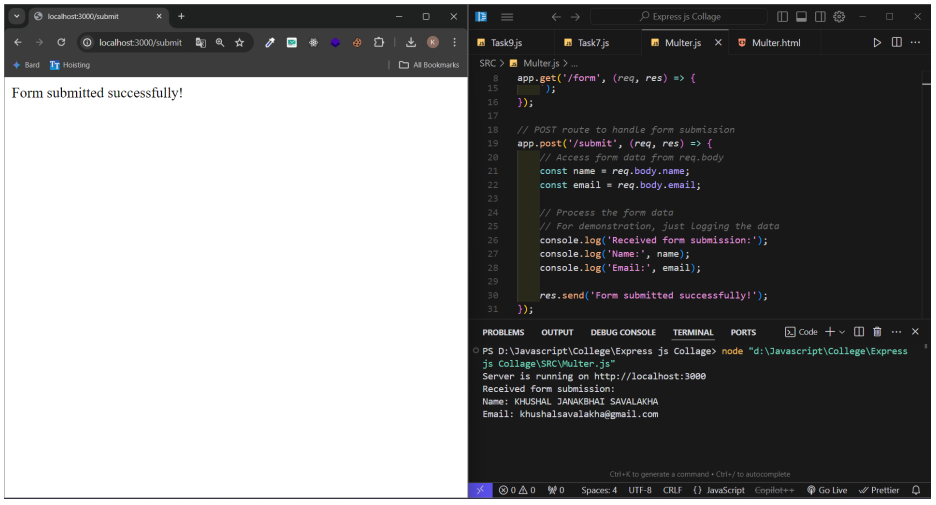
In this example:

Khushal Savalakha

- We use `express.urlencoded({ extended: true })` middleware to parse URL-encoded form data.
- We define a `GET` route at `/form` to render an HTML form.
- The form submits data to a `POST` route at `/submit`.
- In the `POST` route, we access the form data using `req.body` and process it accordingly.

When you run this application and navigate to `http://localhost:3000/form`, you'll see a form where you can input your name and email. When you submit the form, the data will be sent to the `/submit` route as URL-encoded form data and processed accordingly.





EncodedUse.js Alternative

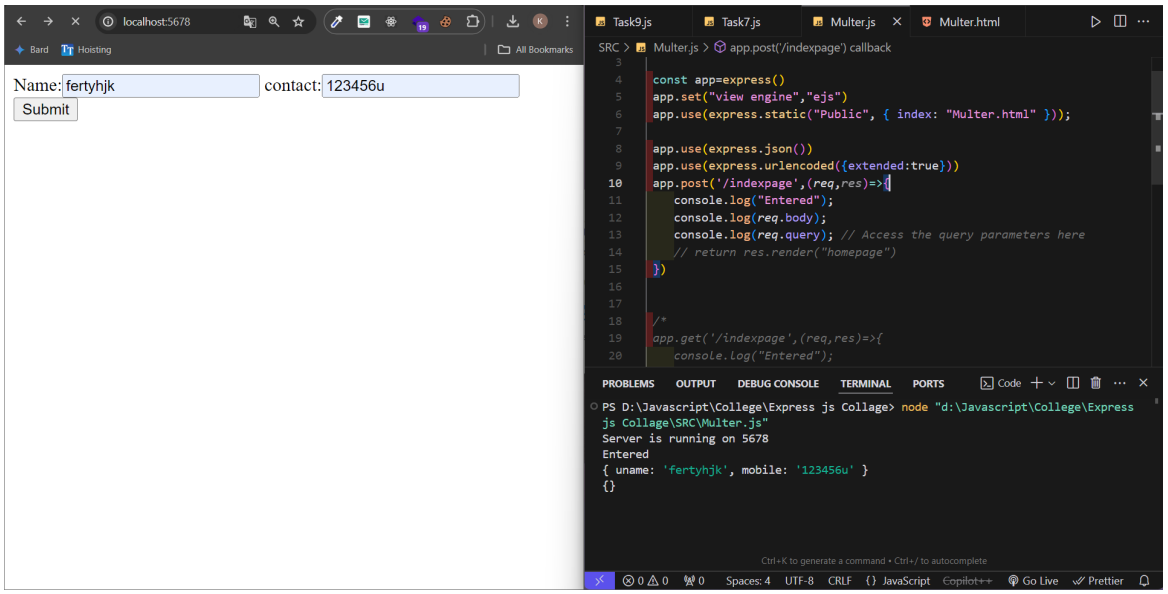
```
const path=require("path")
const express=require("express")
```


Khushal Savalakha

```
 const app=express()
 app.set("view engine","ejs")
 app.use(express.static("Public", { index: "Multer.html" })

 app.use(express.json())
 app.use(express.urlencoded({extended:true}))
 app.post('/indexpage',(req,res)=>{
    console.log("Entered");
    console.log(req.body);
    console.log(req.query); // Access the query parameters
    // return res.render("homepage")
 })


 /*
 app.get('/indexpage',(req,res)=>{
    console.log("Entered");
    console.log(req.body);
    console.log(req.query); // Access the query parameters
    // return res.render("homepage")
 })
*/
 app.get("/",(req,res)=>{
    return res.render("homepage")
 })

 app.listen(5678,()=>{
    console.log("Server is running on 5678");
 })
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, init
    <title>Document</title>
</head>
```

```
<body>
    <form method="post" action="/indexpage">
        <label>Name:<input type="text" name='uname'></label
        <label>contact:<input type="text" name='mobile'></l
        <input type="submit">
    </form>
</body>
</html>
```



> It means when use post method to send data here data data is encoded in url so express.json() can't decode it if we want to decode the data then we have to use express.urlencoded() throgh which we can access the data in req.body if we use get method then we can access data in req.query without using express.urlencoded() am i true ?

You are close, but there are a few clarifications needed regarding how data is sent in HTTP requests and how it is accessed in Express:

HTTP Methods and Data Transmission

1. **POST Method**:
    - **JSON Data**: When sending data as JSON in a POST request, the `Content-Type` header is set to `application/json`, and the data is sent in the request body. To parse this JSON data, you use `express.json()`.

- **URL-encoded Data**: When sending data as URL-encoded in a POST request, the `Content-Type` header is set to `application/x-www-form-urlencoded`, and the data is also sent in the request body. To parse this URL-encoded data, you use `express.urlencoded()`.

2. **GET Method**:
   - Data is sent as query parameters in the URL. Query parameters can be accessed directly from `req.query` without needing any middleware like `express.json()` or `express.urlencoded()`.

Middleware Functions

- **express.json()**:
  Parses incoming requests with JSON payloads. The parsed data is available in `req.body`.
- **express.urlencoded()**:
  Parses incoming requests with URL-encoded payloads. The parsed data is also available in
  `req.body`.

Example Explanation

Using POST with JSON Data

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

// Middleware to parse JSON data
app.use(express.json());

app.post('/add-user', (req, res) => {
    const newUser = req.body; // Access parsed JSON data
    console.log(newUser);
    res.send('User added successfully');
});

const PORT = 3000;
app.listen(PORT, () => {
```


Khushal Savalakha

```
    console.log(`Server is running on port ${PORT}`);
});
```

Sending a POST request with JSON data:

```
curl -X POST <http://localhost:3000/add-user> -H "Conten
t-Type: application/json" -d '{"name": "John Doe", "emai
l": "john.doe@example.com"}'
```

In this case, `express.json()` middleware is used to parse the JSON data in the request body.

Using POST with URL-encoded Data

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

// Middleware to parse URL-encoded data
app.use(express.urlencoded({ extended: true }));

app.post('/add-user', (req, res) => {
    const newUser = req.body; // Access parsed URL-encod
ed data
    console.log(newUser);
    res.send('User added successfully');
});

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

Sending a POST request with URL-encoded data:

```
curl -X POST <http://localhost:3000/add-user> -H "Conten
t-Type: application/x-www-form-urlencoded" -d "name=John
Doe&email=john.doe@example.com"
```


Khushal Savalakha

In this case, `express.urlencoded()` middleware is used to parse the URL-encoded data in the request body.

Using GET Method with Query Parameters

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

app.get('/search', (req, res) => {
    const query = req.query; // Access query parameters
    console.log(query);
    res.send('Query received');
});

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

Sending a GET request with query parameters:

```
curl <http://localhost:3000/search?name=John+Doe&email=j
ohn.doe@example.com>
```

In this case, you don't need any middleware to parse the query parameters. They are automatically available in `req.query`.

Summary

- **POST method with JSON data**: Use `express.json()` to parse JSON data and access it via `req.body`.
- **POST method with URL-encoded data**: Use `express.urlencoded()` to parse URL-encoded data and access it via `req.body`.
- **GET method with query parameters**: Access query parameters directly via `req.query` without needing any middleware.

Khushal Savalakha

💡 **JSON Data**: When sending data as JSON in a POST request, the `Content-Type` header is set to `application/json`, and the data is sent in the request body. To parse this JSON data, you use `express.json()`.

💡 **URL-encoded Data**: When sending data as URL-encoded in a POST request, the `Content-Type` header is set to `application/x-www-form-urlencoded`, and the data is also sent in the request body. To parse this URL-encoded data, you use `express.urlencoded()`.

💡 here i don't understand what is header explain simply and shortly how header works with example.

What is a Header in HTTP?

Headers are a part of HTTP requests and responses. They contain key-value pairs of metadata that provide information about the request or response.

How Headers Work

- **Request Headers**: Sent from the client to the server, providing details about the request.
- **Response Headers**: Sent from the server back to the client, providing details about the response.

Key Header Example: `Content-Type`

The `Content-Type` header is used to indicate the media type of the resource being sent.

- **For JSON Data**:
  - Header: `Content-Type: application/json`
  - This tells the server that the data being sent is in JSON format.
  - Example: `{"name": "John Doe", "email": "john.doe@example.com"}`
- **For URL-encoded Data**:
  - Header: `Content-Type: application/x-www-form-urlencoded`
  - This tells the server that the data being sent is in URL-encoded format.

Khushal Savalakha

- Example: `name=John+Doe&email=john.doe%40example.com`

Example with `Content-Type` Header

Using POST with JSON Data

1. **Server Code**:

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

// Middleware to parse JSON data
app.use(express.json());

app.post('/add-user', (req, res) => {
    const newUser = req.body; // Access parsed JSON data
    console.log(newUser);
    res.send('User added successfully');
});

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

1. **Client Request**:

```
curl -X POST <http://localhost:3000/add-user> \\
-H "Content-Type: application/json" \\
-d '{"name": "John Doe", "email": "john.doe@example.co
m"}'
```

- The client sets the `Content-Type` header to `application/json`.
- The server uses `express.json()` middleware to parse the JSON data in the request body.

Using POST with URL-encoded Data

1. **Server Code**:

Khushal Savalakha

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

// Middleware to parse URL-encoded data
app.use(express.urlencoded({ extended: true }));

app.post('/add-user', (req, res) => {
    const newUser = req.body; // Access parsed URL-encod
ed data
    console.log(newUser);
    res.send('User added successfully');
});

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

1. **Client Request**:

```
curl -X POST <http://localhost:3000/add-user> \\
-H "Content-Type: application/x-www-form-urlencoded" \\
-d "name=John+Doe&email=john.doe%40example.com"
```

- The client sets the `Content-Type` header to `application/x-www-form-urlencoded`.
- The server uses `express.urlencoded()` middleware to parse the URL-encoded data in the request body.

Summary

- **Headers** provide metadata about the request or response.
- The `Content-Type` header specifies the format of the data being sent.
- **JSON Data**: `Content-Type: application/json` with `express.json()` middleware.
- **URL-encoded Data**: `Content-Type: application/x-www-form-urlencoded` with `express.urlencoded()` middleware.

Khushal Savalakha

This ensures the server correctly parses the incoming data based on the specified content type.

> 💡 console.log(req.body.content-type);
> can we see by this ?

No, you cannot directly access the headers of a request using `req.body`. Instead, to log the headers, you should use `req.headers`. The `req.body` object contains the parsed body of the request, which is typically the data sent in the body of a POST request. The headers are available separately in the `req.headers` object.

To log the `Content-Type` header of a request, you can do the following:

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

// Middleware to parse URL-encoded form data
app.use(express.urlencoded({ extended: true }));

// GET route to render a form
app.get('/form', (req, res) => {
    res.send(`
        <form method="post" action="/submit">
            <label>Name: <input type="text" name="name">
</label><br>
            <label>Email: <input type="email" name="emai
l"></label><br>
            <button type="submit">Submit</button>
        </form>
    `);
});

// POST route to handle form submission
app.post('/submit', (req, res) => {
    // Log the Content-Type header
    console.log('Content-Type:', req.headers['content-ty
pe']);
```


Khushal Savalakha

```
    // Access form data from req.body
    const name = req.body.name;
    const email = req.body.email;

    // Process the form data
    // For demonstration, just logging the data
    console.log('Received form submission:');
    console.log('Name:', name);
    console.log('Email:', email);

    res.send('Form submitted successfully!');
});

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

Explanation

- **Logging Headers**:
  - `req.headers` contains the headers of the incoming request.
  - `req.headers['content-type']` accesses the `Content-Type` header specifically.
  - Using `console.log('Content-Type:', req.headers['content-type']);` will log the `Content-Type` header to the console.

When you submit the form, this code will log the `Content-Type` header along with the form data to the server console.

Testing the Code

1. **Start the Express Server**:

   ```
   node yourfile.js
   ```

2. **Navigate to the Form**:

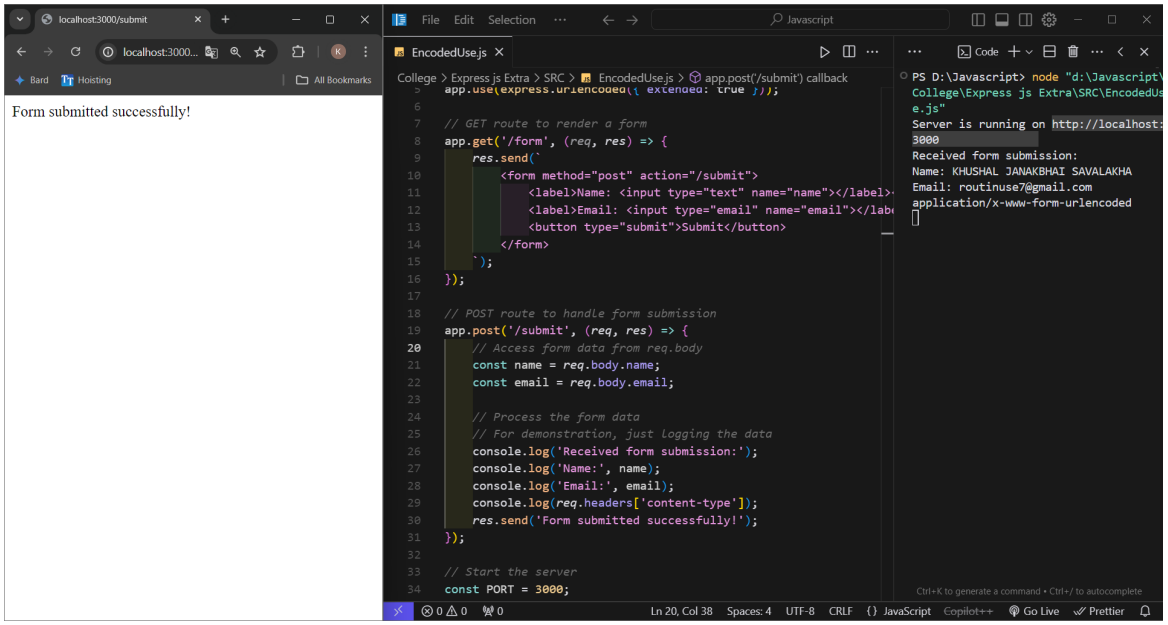
Khushal Savalakha

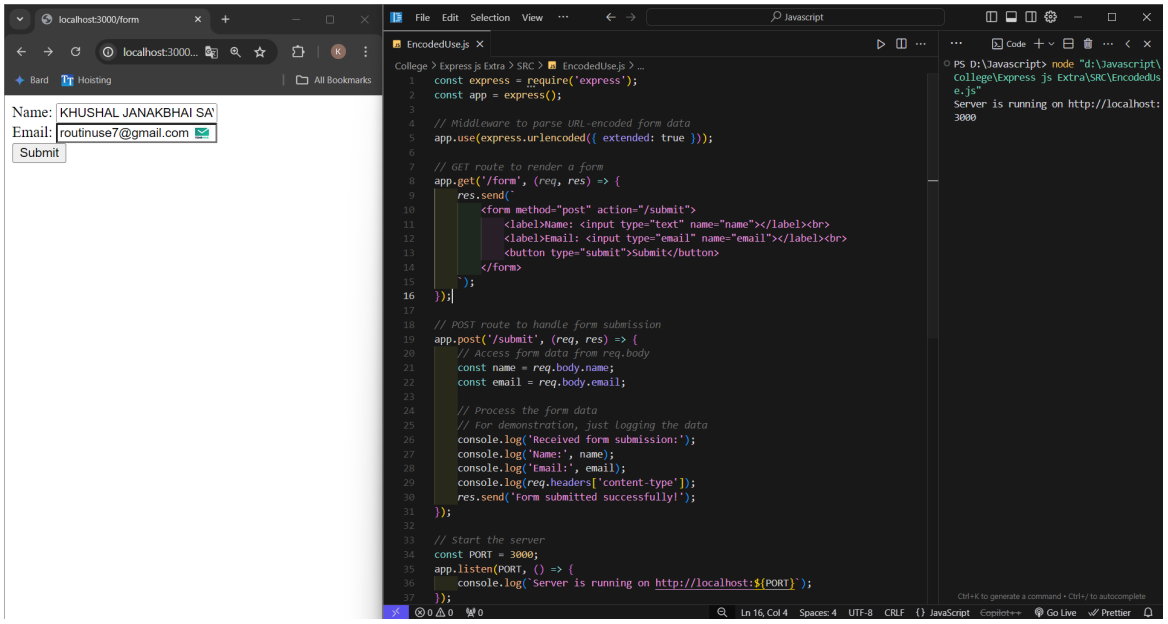
```
<http://localhost:3000/form>
```

3. **Submit the Form**.

4. **Check the Server Console**:
   You should see the
   `Content-Type` header logged, along with the form data:

```
Content-Type: application/x-www-form-urlencoded
Received form submission:
Name: John Doe
Email: john.doe@example.com
```





> Here it means when we have get data then content-type is Appliacation/json which we can see in req.header['content-type'] that time we have to use express.json() by which data we can see in req.query directly and if the data is send by post then content type is Content-Type: application/x-www-form-urlencoded here we can't see data in req.query beacause data is encoded if we want to convert in jason then we have to use url.encoded() and we can see it req.body am i true?

Let's clarify how HTTP methods, content types, and Express middleware work together.

Key Points to Understand

1. **GET Requests**:
   - Typically, data is sent as query parameters in the URL.
   - Query parameters can be accessed directly via `req.query`.
   - `Content-Type` is not typically relevant for GET requests because the data is in the URL, not in the body.
2. **POST Requests**:
   - Data is sent in the body of the request.
   - The `Content-Type` header indicates the format of the data in the body.

Common Content Types for POST Requests

1. **application/json**:
   - Data is JSON formatted.
   - Use `express.json()` middleware to parse JSON data.
   - Parsed data is available in `req.body`.
2. **application/x-www-form-urlencoded**:
   - Data is URL-encoded.
   - Use `express.urlencoded()` middleware to parse URL-encoded data.
   - Parsed data is available in `req.body`.

Accessing Data

- **GET Requests**:
  - Query parameters are accessed via `req.query`.
  - Example: `http://localhost:3000/search?name=John&email=john@example.com`
  - Server-side: `req.query.name` and `req.query.email`.
- **POST Requests**:
  - JSON data or URL-encoded data is accessed via `req.body` after appropriate middleware parsing.

Practical Examples

GET Request

Client-side (sending data as query parameters):

```
curl "<http://localhost:3000/search?name=John&email=john
@example.com>"
```

Server-side:

```
app.get('/search', (req, res) => {
    console.log(req.query.name); // 'John'
    console.log(req.query.email); // 'john@example.com'
```


Khushal Savalakha

```
    res.send('Query received');
});
```

POST Request with JSON Data

Client-side (sending JSON data):

```
curl -X POST <http://localhost:3000/add-user> \\
-H "Content-Type: application/json" \\
-d '{"name": "John Doe", "email": "john.doe@example.co
m"}'
```

Server-side:

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

app.post('/add-user', (req, res) => {
    console.log(req.body.name); // 'John Doe'
    console.log(req.body.email); // 'john.doe@example.co
m'
    res.send('User added successfully');
});
```

POST Request with URL-encoded Data

Client-side (sending URL-encoded data):

```
curl -X POST <http://localhost:3000/add-user> \\
-H "Content-Type: application/x-www-form-urlencoded" \\
-d "name=John+Doe&email=john.doe%40example.com"
```

Server-side:

```
app.use(express.urlencoded({ extended: true }));

app.post('/add-user', (req, res) => {
    console.log(req.body.name); // 'John Doe'
    console.log(req.body.email); // 'john.doe@example.co
m'
```


Khushal Savalakha

```
    res.send('User added successfully');
});
```

Summary of Your Points:

- **GET Request**:
  - Data is in the URL as query parameters.
  - `req.query` is used to access the data.
  - `Content-Type` is generally not used.
- **POST Request with JSON Data**:
  - `Content-Type: application/json`.
  - Use `express.json()` middleware.
  - Access data via `req.body`.
- **POST Request with URL-encoded Data**:
  - `Content-Type: application/x-www-form-urlencoded`.
  - Use `express.urlencoded()` middleware.
  - Access data via `req.body`.

By understanding these points, you can handle different types of data formats in your Express applications appropriately.

> 💡 it means in get request all things are in url then we don't have to mention content type then why we have to use express.json ?

In a GET request, data is typically sent as query parameters appended to the URL, and there is no body content to parse. Thus, the `Content-Type` header is not relevant for GET requests, and you do not need to use `express.json()` or `express.urlencoded()` to handle GET request data.

GET Request Details

- **Data Location**: Data is included in the URL as query parameters.
- **Accessing Data**: Use `req.query` to access the query parameters.
- **Content-Type**: Not used, since data is not in the body of the request.

Khushal Savalakha

Example of GET Request

1. **Client-side Request**:

```
curl "<http://localhost:3000/search?name=John&email=j
ohn@example.com>"
```

2. **Server-side Handling**:

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

app.get('/search', (req, res) => {
    console.log(req.query.name);  // 'John'
    console.log(req.query.email); // 'john@example.co
m'
    res.send('Query received');
});

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

In this example, `req.query` directly accesses the query parameters from the URL, without needing to use any middleware like `express.json()` or `express.urlencoded()`.

Why Use `express.json()`?

The `express.json()` middleware is used to parse JSON data in the body of POST (or other methods like PUT, PATCH) requests. It is not used for GET requests because GET requests do not have a body to parse.

Example of POST Request with JSON Data

1. **Client-side Request**:

```
curl -X POST <http://localhost:3000/add-user> \\
-H "Content-Type: application/json" \\
```


Khushal Savalakha

```
-d '{"name": "John Doe", "email": "john.doe@example.c
om"}'
```

2. **Server-side Handling**:

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

// Middleware to parse JSON data
app.use(express.json());

app.post('/add-user', (req, res) => {
    console.log(req.body.name);  // 'John Doe'
    console.log(req.body.email); // 'john.doe@exampl
e.com'
    res.send('User added successfully');
});

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

In this case, `express.json()` middleware parses the JSON data from the request body and makes it available in `req.body`.

Summary

- **GET Requests**:
  - Data is sent as query parameters in the URL.
  - Access data using `req.query`.
  - No need for `express.json()` or `express.urlencoded()`.
- **POST Requests with JSON Data**:
  - Data is sent in the request body with `Content-Type: application/json`.
  - Use `express.json()` to parse the JSON data.

Khushal Savalakha

- Access data using `req.body`.

- **POST Requests with URL-encoded Data**:
  - Data is sent in the request body with `Content-Type: application/x-www-form-urlencoded`.
  - Use `express.urlencoded()` to parse the URL-encoded data.
  - Access data using `req.body`.

You use `express.json()` specifically for handling and parsing JSON payloads in the body of POST (and similar) requests, not for GET requests.

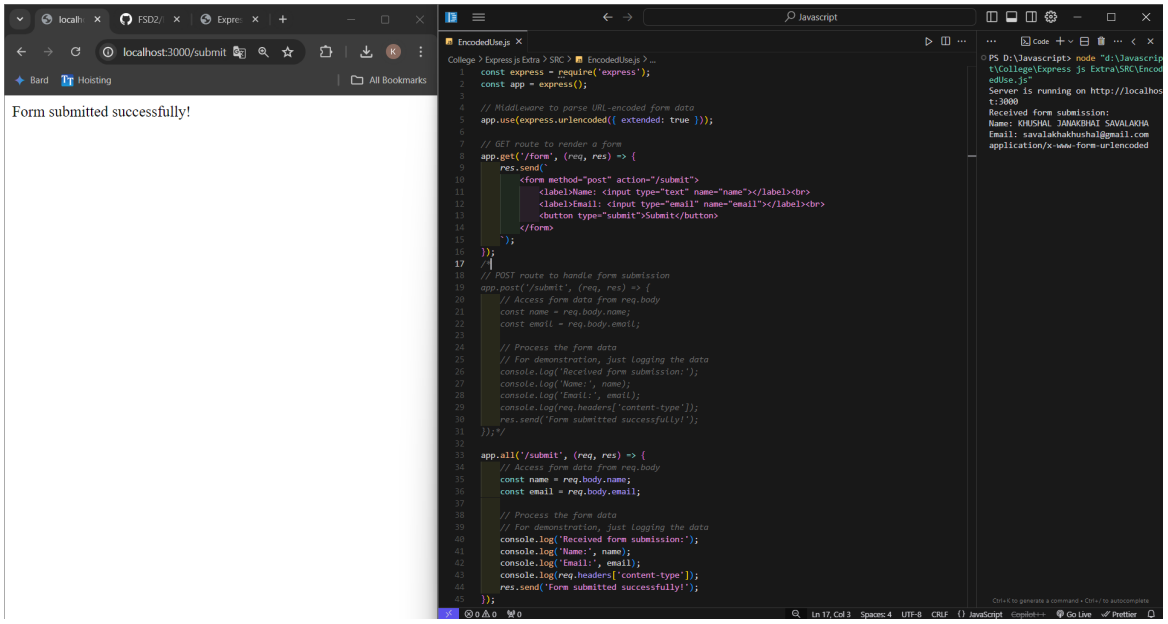
> 💡 we can use app.all() for all get and post type request.

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

// Middleware to parse URL-encoded form data
app.use(express.urlencoded({ extended: true }));

// GET route to render a form
app.get('/form', (req, res) => {
    res.send(`
        <form method="post" action="/submit">
            <label>Name: <input type="text" name="name"></l
            <label>Email: <input type="email" name="email">
            <button type="submit">Submit</button>
        </form>
    `);
});
/*
// POST route to handle form submission
app.post('/submit', (req, res) => {
    // Access form data from req.body
    const name = req.body.name;
    const email = req.body.email;
```


Khushal Savalakha

```
    // Process the form data
    // For demonstration, just logging the data
    console.log('Received form submission:');
    console.log('Name:', name);
    console.log('Email:', email);
    console.log(req.headers['content-type']);
    res.send('Form submitted successfully!');
});*/

app.all('/submit', (req, res) => {
    // Access form data from req.body
    const name = req.body.name;
    const email = req.body.email;

    // Process the form data
    // For demonstration, just logging the data
    console.log('Received form submission:');
    console.log('Name:', name);
    console.log('Email:', email);
    console.log(req.headers['content-type']);
    res.send('Form submitted successfully!');
});

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


Khushal Savalakha



The `app.all()` method in Express.js is used to define a route handler that will be executed for all HTTP methods (GET, POST, PUT, DELETE, etc.) on a specific path. It is useful for situations where you want to apply the same logic regardless of the HTTP method used.

How `app.all()` Works

- **Syntax**:

```
app.all(path, callback)
```

  - `path`: The route path where the handler should be applied.
  - `callback`: The middleware function that will handle the request.

Example Usage

Here's a simple example to demonstrate how `app.all()` works:

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

app.all('/example', (req, res, next) => {
    console.log('Request received:', req.method, req.ur
l);
    next(); // Pass control to the next handler
});
```


Khushal Savalakha

```
app.get('/example', (req, res) => {
    res.send('GET request to /example');
});

app.post('/example', (req, res) => {
    res.send('POST request to /example');
});

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

Explanation

- **Route Handling**:
  - `app.all('/example', ...)` sets up a middleware function for the `/example` path that logs the request method and URL.
  - This middleware function will be executed for any HTTP method (GET, POST, etc.) on `/example`.
- **Middleware**:
  - The `next()` function in the middleware passes control to the next matching route handler. Without calling `next()`, the subsequent handlers will not be executed.
- **Specific Handlers**:
  - `app.get('/example', ...)` handles GET requests to `/example`.
  - `app.post('/example', ...)` handles POST requests to `/example`.

When you make requests to `/example` with different HTTP methods, the `app.all()` handler will run first, followed by the specific route handler for the method:

- **GET Request**:
  - `curl <http://localhost:3000/example` >
  - Console Output: `Request received: GET /example`

Khushal Savalakha

- Response: `GET request to /example`

- **POST Request**:
  - `curl -X POST <http://localhost:3000/example` >
  - Console Output: `Request received: POST /example`
  - Response: `POST request to /example`

Use Cases for `app.all()`

- **Logging**: Log details of incoming requests, such as method, URL, and headers.
- **Authentication/Authorization**: Apply authentication or authorization logic that should run for all types of requests to a certain path.
- **Middleware**: Apply generic middleware to a specific path regardless of the HTTP method.

By using `app.all()`, you can ensure that certain middleware functions or logic are consistently applied to a route, no matter which HTTP method is used to access it. This can help simplify and centralize code that should apply to multiple types of requests.

▼ Body-Parser

`body-parser` is a middleware for Express.js that extracts the body portion of an incoming request stream and exposes it on `req.body`. This is particularly useful when you want to handle POST requests or PUT requests where data is sent to the server as part of the request body.

Example of `body-parser`:

To use `body-parser`, you typically install it via npm (`npm install body-parser`) and then require it in your Node.js/Express application.

Here's a simple example of how you can use `body-parser`:

1. **Setup and Middleware Configuration**:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

// Parse application/x-www-form-urlencoded
```


Khushal Savalakha

```
app.use(bodyParser.urlencoded({ extended: false }));

// Parse application/json
app.use(bodyParser.json());
```

Explanation:

- `body-parser.urlencoded()`: Parses the text as URL-encoded data (which is how browsers tend to send form data from regular forms set to POST) and exposes the resulting object (containing the keys and values) on `req.body`. By setting `extended` to `false`, it uses the `querystring` library to parse the URL-encoded data.
- `body-parser.json()`: Parses the text as JSON and exposes the resulting object on `req.body`.

1. **Example Routes**:

```
// POST route example
app.post('/api/users', (req, res) => {
    console.log(req.body); // Assuming bodyParser has
already parsed the body
    // Process req.body and perform actions accordingl
y
    res.json({ message: 'User created successfully', u
ser: req.body });
});

// PUT route example
app.put('/api/users/:id', (req, res) => {
    console.log(req.body); // Assuming bodyParser has
already parsed the body
    // Process req.body and update user with id = req.
params.id
    res.json({ message: 'User updated successfully', u
ser: req.body });
});

// Example form submission
app.post('/submit-form', (req, res) => {
```


Khushal Savalakha

```
    console.log(req.body); // Assuming bodyParser has
already parsed the body
    // Process req.body from a form submission
    res.send('Form submitted successfully');
});

// Start server
app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

Explanation:

- **POST and PUT Routes**: In both routes (`/api/users` and `/api/users/:id`), `req.body` contains the parsed JSON or URL-encoded data sent in the request body. You can then process this data as needed (e.g., store it in a database, update an existing record).
- **Form Submission Route**: The `/submit-form` route handles a form submission where `req.body` contains the parsed form data. This is useful for handling HTML form submissions in an Express.js application.

Summary:

`body-parser` simplifies the process of parsing incoming request bodies in Express.js applications. It supports parsing of URL-encoded and JSON bodies out of the box, making it easier to work with data sent via POST, PUT, and other HTTP methods that send data in the request body.

> 💡 but we can also access url-encoded using express am i true ?

Yes, you are correct! Express.js, by default, can also handle URL-encoded data without the need for `body-parser` middleware, starting from version 4.16.0 onwards. This functionality is provided by `express.urlencoded()` middleware.

Handling URL-encoded Data with Express.js:

Starting from Express.js version 4.16.0, the built-in `express.urlencoded()` middleware is included, which provides similar functionality to `body-parser` for handling URL-encoded data from form submissions.

Khushal Savalakha

Here's how you can use `express.urlencoded()` to handle URL-encoded data:

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

const app = express();

// Parse URL-encoded bodies (for x-www-form-urlencode
d)
app.use(express.urlencoded({ extended: true }));

// Example POST route handling URL-encoded data
app.post('/submit-form', (req, res) => {
    console.log(req.body); // Access the parsed body
    // Process req.body from a form submission
    res.send('Form submitted successfully');
});

// Start server
app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

Explanation:

- **`express.urlencoded()`**: This middleware parses incoming requests with URL-encoded payloads (usually from form submissions where `Content-Type` is `application/x-www-form-urlencoded`). It populates `req.body` with the parsed data and exposes it to your application.
- **Options**:
  - **`extended`**: This option allows you to choose between parsing the URL-encoded data with the `querystring` library (when `false`) or the `qs` library (when `true`). The `qs` library allows for richer parsing, but for most applications, the built-in `querystring` parser (`extended: false`) is sufficient.

When to Use `body-parser` vs `express.urlencoded()`:

1. **Express.js Version**: If you are using Express.js version 4.16.0 or higher, you can use `express.urlencoded()` directly instead of `body-parser` for URL-encoded data

Khushal Savalakha

parsing.

2. **Flexibility**: `body-parser` provides additional parsers for handling different types of request bodies (e.g., JSON, raw, text). If you need to handle different content types or more complex parsing scenarios, `body-parser` might still be necessary.

Summary:

- Yes, you can access URL-encoded data using Express.js directly through `express.urlencoded()` middleware, starting from version 4.16.0.
- This built-in middleware simplifies handling of form submissions where data is sent in the URL-encoded format (`application/x-www-form-urlencoded`).
- For most applications, using `express.urlencoded()` is recommended over `body-parser` specifically for URL-encoded data parsing.

▼ Introduction

k

> 💡 Express is the most popular minimalistic framework. It is built upon the built-in module HTTP of NodeJS to facilitate the simple and easy interaction between frontend and backend logic through API, and also it enables us to organize our business logic in so much pretty manner. It is much flexible and we can use it for both the web and android. Also, it provides a very simple error handling procedure.
>
> here i don't understand how it provides simple error handling procedure ?

Express provides a simple error handling procedure by allowing developers to define middleware functions that handle errors in a consistent and straightforward manner. Here's an explanation with an example to make it clearer.

How Error Handling Works in Express

1. **Middleware Functions**: In Express, 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. These functions can perform a variety of tasks, including error handling.

Khushal Savalakha

2. **Error-Handling Middleware**: An error-handling middleware function is a special type of middleware function that has four arguments: `err`, `req`, `res`, and `next`. When an error occurs, Express will skip the other middleware functions and execute the error-handling middleware.

Example of Error Handling in Express

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

// A route that may cause an error
app.get('/', (req, res, next) => {
  // Simulating an error
  const error = new Error('Something went wrong!');
  error.status = 500;
  next(error);
});

// General error-handling middleware
app.use((err, req, res, next) => {
  res.status(err.status || 500);
  res.json({
    error: {
      message: err.message
    }
  });
});

// Starting the server
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Explanation

1. **Route Handling**:
   - In the example, the route `/` is defined, and within this route, an error is deliberately created by constructing a new `Error` object.

Khushal Savalakha

- The `next(error)` function is called to pass the error to the next middleware function in the stack.

2. **Error-Handling Middleware**:
   - The error-handling middleware function is defined with four parameters: `err`, `req`, `res`, and `next`.
   - When the `next(error)` function is called, Express skips all remaining non-error-handling middleware and routes the error to this error-handling middleware.
   - The middleware sets the HTTP status code (defaulting to 500 if not specified) and sends a JSON response containing the error message.

Key Points

- **Consistency**: Error-handling middleware provides a consistent way to manage errors across the entire application.
- **Separation of Concerns**: By separating error-handling logic from regular route handling, the code becomes cleaner and easier to maintain.
- **Flexibility**: Developers can define multiple error-handling middleware functions for different types of errors or for different parts of the application.

This approach simplifies the error-handling process, making it easier for developers to manage and debug errors in their applications.

> 💡 **Error-Handling Middleware**: An error-handling middleware function is a special type of middleware function that has four arguments: `err`, `req`, `res`, and `next`. When an error occurs, Express will skip the other middleware functions and execute the error-handling middleware.

> 💡 but in node js there are also available ?then why it is different from node js ?

You're right that Node.js itself also has mechanisms for handling errors. However, Express provides a more structured and convenient way to handle errors compared to using pure Node.js. Let's delve into the differences and advantages of Express's error-handling middleware.

Error Handling in Node.js

Khushal Savalakha

In Node.js, error handling typically involves:

1. **Callback Functions**:
   - Using callback functions where the first argument is an error, following the pattern `function(err, result)`.
   - Example:

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

fs.readFile('file.txt', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
  } else {
    console.log('File content:', data.toString());
  }
});
```

2. **Promises and async/await**:
   - Handling errors in asynchronous operations using `.catch()` for promises or `try...catch` for `async/await`.
   - Example with Promises:

```
const fs = require('fs').promises;

fs.readFile('file.txt')
  .then(data => console.log('File content:', data.
toString()))
  .catch(err => console.error('Error reading fil
e:', err));
```

   - Example with `async/await`:

```
const readFileAsync = async (filename) => {
  try {
    const data = await fs.readFile(filename);
    console.log('File content:', data.toString());
  } catch (err) {
```


Khushal Savalakha

```
    console.error('Error reading file:', err);
  }
};

readFileAsync('file.txt');
```

Error Handling in Express

Express enhances error handling by providing a systematic way to define error-handling middleware. Here's how it differs and what makes it advantageous:

1. **Centralized Error Handling**:
   - Express allows you to define centralized error-handling middleware that can manage all errors in a consistent manner. This avoids scattering error-handling logic throughout your code.
2. **Middleware Architecture**:
   - The middleware architecture of Express allows you to easily separate error-handling logic from your business logic, making your code cleaner and more maintainable.
3. **Automatic Error Forwarding**:
   - If a route handler or middleware calls `next(error)`, Express will automatically skip all remaining non-error-handling middleware and pass the error to the defined error-handling middleware. This is not something Node.js does out of the box.

Example of Node.js vs Express Error Handling

Node.js (Basic)

Handling errors manually in each route or operation:

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

const server = http.createServer((req, res) => {
  if (req.url === '/') {
    // Simulating an error
    res.statusCode = 500;
    res.end('Something went wrong!');
```


Khushal Savalakha

```
  } else {
    res.statusCode = 404;
    res.end('Not Found');
  }
});

server.listen(3000, () => {
  console.log('Server running at <http://localhost:3000/
>');
});
```

Express (Simplified and Centralized)

Using middleware to handle errors consistently:

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

// Normal route handler
app.get('/', (req, res, next) => {
  // Simulating an error
  const error = new Error('Something went wrong!');
  error.status = 500;
  next(error);
});

// Centralized error-handling middleware
app.use((err, req, res, next) => {
  res.status(err.status || 500);
  res.json({
    error: {
      message: err.message,
    },
  });
});

app.listen(3000, () => {
```


Khushal Savalakha

```
  console.log('Server running on port 3000');
});
```

Key Advantages of Express Error Handling

1. **Consistency**: Centralized error handling means you don't have to repeat error-handling code in every route.
2. **Simplicity**: Error-handling middleware with `next(err)` simplifies the logic of forwarding errors.
3. **Scalability**: As your application grows, you can add more error-handling middleware to handle different types of errors in different ways without cluttering your main logic.

In summary, while Node.js provides basic error-handling capabilities, Express enhances this by offering a more organized, scalable, and maintainable approach to error handling through its middleware system.

💡 **Automatic Error Forwarding**:

💡 If a route handler or middleware calls `next(error)`, Express will automatically skip all remaining non-error-handling middleware and pass the error to the defined error-handling middleware. This is not something Node.js does out of the box.

💡 here i don't understand this portion explain me simply and shortly .

Sure, let's break it down:

Automatic Error Forwarding in Express

1. **Normal Route Handler**:
   - In a route handler, if something goes wrong (e.g., a database error), you create an error object and pass it to the `next` function: `next(error)`.
   - Example:

Khushal Savalakha

```
app.get('/', (req, res, next) => {
  const error = new Error('Something went wron
g!');
  next(error);  // Pass the error to the next midd
leware
});
```

2. **Express Behavior**:
   - When `next(error)` is called, Express knows an error occurred and stops running any further regular middleware or route handlers.
   - Instead, it looks for error-handling middleware.
3. **Error-Handling Middleware**:
   - This is a special type of middleware designed to handle errors. It takes four arguments: `err`, `req`, `res`, `next`.
   - Example:

```
app.use((err, req, res, next) => {
  res.status(err.status || 500);
  res.json({
    error: {
      message: err.message
    }
  });
});
```

4. **What Happens**:
   - When `next(error)` is called, Express skips any remaining non-error middleware and goes directly to the error-handling middleware.

Comparison with Node.js

- **Node.js**: You would need to manually handle errors in each route or operation.
- **Express**: Automatically forwards errors to a centralized error handler using `next(error)`.

Summary

Khushal Savalakha

Automatic Error Forwarding means that if you pass an error to `next(error)`, Express automatically skips other middleware and goes straight to your error-handling middleware. This simplifies and centralizes error handling in your application.

▼ Binding

💡 what is the meaning of binding in express js explain me with example.

In Express.js, "binding" usually refers to the process of associating a function or middleware with a particular route or event. This is a fundamental concept in building web applications with Express.js because it determines how requests to specific URLs are handled.

Example of Binding in Express.js

Let's break down an example to understand this better:

1. **Setup Express.js**: First, you need to install Express.js if you haven't already. You can do this using npm:

   ```
   npm install express
   ```

2. **Create a Simple Express Server**: In your project folder, create a file named `app.js` and add the following code:

   ```
   const express = require('express');
   const app = express();
   const port = 3000;

   // Binding a function to the root URL ("/")
   app.get('/', (req, res) => {
     res.send('Hello, World!');
   });

   // Binding a function to the "/about" URL
   app.get('/about', (req, res) => {
     res.send('About Us');
   });
   ```


Khushal Savalakha

```
// Binding a function to the "/contact" URL
app.get('/contact', (req, res) => {
  res.send('Contact Us');
});

// Start the server and listen on port 3000
app.listen(port, () => {
  console.log(`Server is running at <http://localhost
>:${port}`);
});
```

Explanation

1. **Import Express**:

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

   This line imports the Express module.

2. **Create an Express Application**:

   ```
   const app = express();
   const port = 3000;
   ```

   Here, you create an instance of an Express application and define the port number the server will listen on.

3. **Binding Functions to Routes**:

   ```
   // Binding a function to the root URL ("/")
   app.get('/', (req, res) => {
     res.send('Hello, World!');
   });
   ```

   - `app.get('/')`: This binds a function to the HTTP GET request at the root URL (`/`).
   - `(req, res) => { res.send('Hello, World!'); }`: This is the callback function that gets executed when a GET request is made to the root URL. It sends back a response with the text "Hello, World!".

Khushal Savalakha

Similarly, we bind functions to the `/about` and `/contact` routes:

```
app.get('/about', (req, res) => {
  res.send('About Us');
});

app.get('/contact', (req, res) => {
  res.send('Contact Us');
});
```

4. **Start the Server**:

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

This line starts the server and makes it listen for requests on the specified port (3000 in this case). When the server is running, it logs a message to the console.

Running the Example

1. **Run the Server**: In your terminal, navigate to the directory containing `app.js` and run:

```
node app.js
```

2. **Access the Routes**:

- Open a browser and go to `http://localhost:3000/` to see "Hello, World!".
- Go to `http://localhost:3000/about` to see "About Us".
- Go to `http://localhost:3000/contact` to see "Contact Us".

This example demonstrates how to bind functions to different routes using the `app.get` method in Express.js. The concept of binding is crucial for routing, as it allows you to define how your application responds to various HTTP requests.

Khushal Savalakhia

💡 it means on particular routes request is generated and on particular request in http get there many function are bounded to response am i true ?

Yes, you're on the right track! In Express.js, you bind functions (also known as route handlers or middleware) to specific routes and HTTP methods (like GET, POST, etc.). This means that when a particular route is requested, the corresponding function(s) will be executed to generate the appropriate response.

Key Concepts

1. **Routes**: These are the URL patterns that your application responds to (e.g., `/`, `/about`, `/contact`).
2. **HTTP Methods**: These are the types of requests your application can handle (e.g., GET, POST, PUT, DELETE). Each method is used for different kinds of operations (GET for retrieving data, POST for submitting data, etc.).
3. **Route Handlers**: These are the functions that get executed when a route is requested with a particular HTTP method. They generate the response sent back to the client.

Example to Clarify

Let's extend the previous example to include multiple functions (middlewares) for a particular route.

```
const express = require('express');
const app = express();
const port = 3000;

// Middleware function 1
const logRequest = (req, res, next) => {
  console.log(`Received a request for ${req.url}`);
  next(); // Pass control to the next middleware
};

// Middleware function 2
const authenticate = (req, res, next) => {
  console.log('Authenticating user...');
  // Perform authentication logic
  next(); // Pass control to the next middleware
```


Khushal Savalakha

```
};

// Route handler for the root URL
const getRoot = (req, res) => {
  res.send('Hello, World!');
};

// Binding multiple functions to the root URL ("/")
app.get('/', logRequest, authenticate, getRoot);

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

Explanation

1. **Middleware Functions**:
   - `logRequest`: This function logs the URL of the incoming request and then calls `next()` to pass control to the next middleware function.
   - `authenticate`: This function simulates an authentication process and then calls `next()` to pass control to the next middleware function.
2. **Route Handler**:
   - `getRoot`: This is the final function that sends the response "Hello, World!" to the client.
3. **Binding Functions to Routes**:

   ```
   app.get('/', logRequest, authenticate, getRoot);
   ```

   - This binds three functions (`logRequest`, `authenticate`, and `getRoot`) to the root URL (`/`) for GET requests.
   - When a GET request is made to `/`, the functions will be executed in the order they are listed. First, `logRequest` runs, then `authenticate`, and finally `getRoot`.

Running the Example

Khushal Savalakha

1. **Run the Server**: In your terminal, navigate to the directory containing this code and run:

   ```
   node app.js
   ```

2. **Access the Route**: Open a browser and go to `http://localhost:3000/`. You'll see "Hello, World!" and the server's console will log messages from the `logRequest` and `authenticate` functions.

Summary

- **Binding** in Express.js involves associating route handler functions (and possibly middleware) with specific routes and HTTP methods.
- Multiple functions can be bound to a single route to handle various tasks in sequence, such as logging, authentication, and generating a response.
- This setup allows for modular and maintainable code where different concerns (e.g., logging, authentication) can be handled separately.

💡 here express() is a constructor or something ?

In Express.js, `express()` is not a constructor in the traditional sense, but it functions similarly to one by creating an instance of an Express application.

Detailed Explanation

When you call `express()`, it creates an Express application object. This object is essentially a function but is also an instance of an `express.Application`, which contains methods for configuring your application, such as setting up routes, middleware, and handling HTTP requests.

Example to Illustrate

Consider this example again:

```
const express = require('express'); // Import the expres
s module
const app = express(); // Create an instance of the expr
ess application
const port = 3000;
```


Khushal Savalakha

```
app.get('/', (req, res) => {
  res.send('Hello, World!');
});

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

Breakdown

1. **Import Express Module**:

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

   - This line imports the Express module into your project.

2. **Create an Express Application**:

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

   - `express()` is called to create an instance of an Express application.
   - The `app` variable now holds this instance, which you will use to define routes, configure middleware, and start the server.

What Happens When `express()` is Called?

- **Initialization**: When `express()` is called, it initializes an Express application.
- **Properties and Methods**: The returned application object has various properties and methods that are used to configure the application (like `app.get()`, `app.post()`, `app.listen()`, etc.).
- **Prototype Chain**: This application object is an instance of `express.Application`, which has its own prototype chain and inherits from the core Node.js `http.Server` class, giving it server capabilities.

Simplified Concept

To give a more familiar analogy in JavaScript:

Khushal Savalakha

- **Constructor Function**:

```
function MyApp() {
  // Initialization code
}

MyApp.prototype.get = function(route, handler) {
  // Define a GET route
};

MyApp.prototype.listen = function(port, callback) {
  // Start the server
};

// Create an instance
const app = new MyApp();
```

- **Express Similarity**:

```
const express = require('express');
const app = express(); // This is similar to calling
a constructor and returning an initialized object
```

Summary

- `express()` is a factory function that creates and returns an instance of an Express application.
- This instance has methods and properties for setting up your application, such as defining routes and configuring middleware.
- It abstracts away a lot of the lower-level HTTP server management, allowing you to focus on building your application's functionality.

By using `express()`, you leverage the powerful and flexible framework that Express provides for building web applications and APIs in Node.js.

▼ Static Folder

💡 what is the meaning of static folder in express js?

Khushal Savalakha

In Express.js, a static folder is a directory from which you can serve static files directly to the client. Static files include assets such as HTML files, CSS stylesheets, JavaScript files, images, fonts, and other resources that do not change dynamically.

Setting Up a Static Folder in Express.js

To serve static files in Express.js, you use the built-in middleware function `express.static`. Here's how you can set it up:

1. **Create a Static Folder**: First, create a folder in your project directory where you will store all your static files. Common names for this folder are `public`, `assets`, or `static`.
2. **Configure Express to Serve Static Files**: Use the `express.static` middleware to specify the folder from which to serve static files.

Example

1. **Project Structure**:

```
my-express-app/
├── public/
│   ├── images/
│   │   └── logo.png
│   ├── css/
│   │   └── styles.css
│   └── js/
│       └── scripts.js
├── app.js
└── package.json
```

2. **Code in `app.js`**:

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

// Serve static files from the 'public' directory
app.use(express.static(path.join(__dirname, 'publi
c')));
```


Khushal Savalakha

```
// Example route
app.get('/', (req, res) => {
  res.send('Hello, world!');
});

// Start the server
app.listen(3000, () => {
  console.log('Server running at <http://localhost:30
00/>');
});
```

How It Works

- **express.static**: This middleware function serves the files directly from the specified directory (in this case, the `public` folder). When a request is made to a static file, Express will look for the file in the static folder and serve it if it exists.
- **Requesting Static Files**:
  - If you have a file named `logo.png` in the `public/images` directory, you can access it via `http://localhost:3000/images/logo.png`.
  - If you have a stylesheet named `styles.css` in the `public/css` directory, you can access it via `http://localhost:3000/css/styles.css`.

Benefits

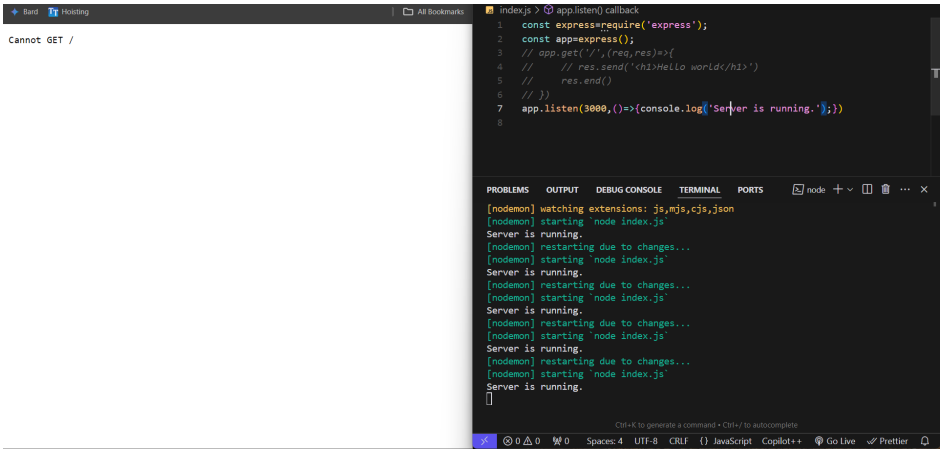
- **Simplifies Asset Management**: You don't need to create routes for each static file; `express.static` automatically handles this.
- **Efficient Serving**: Static files are served quickly and efficiently, often directly from memory, without the need for dynamic processing on each request.

Summary

In Express.js, the static folder is used to serve static files like images, CSS, and JavaScript directly to the client. You configure this using the `express.static` middleware, which makes it easy to manage and serve static assets in your application.

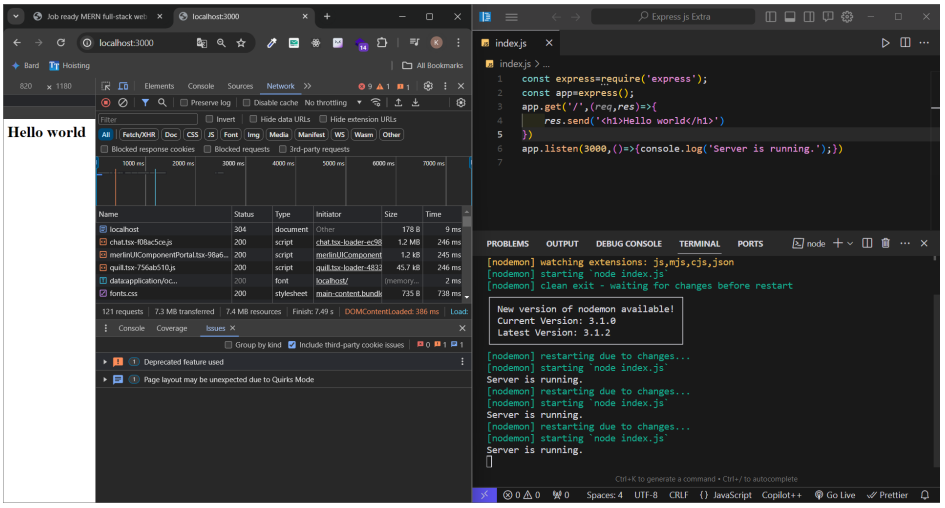
▼ How to make server?

Khushal Savalakha



here we are not getting any thing.

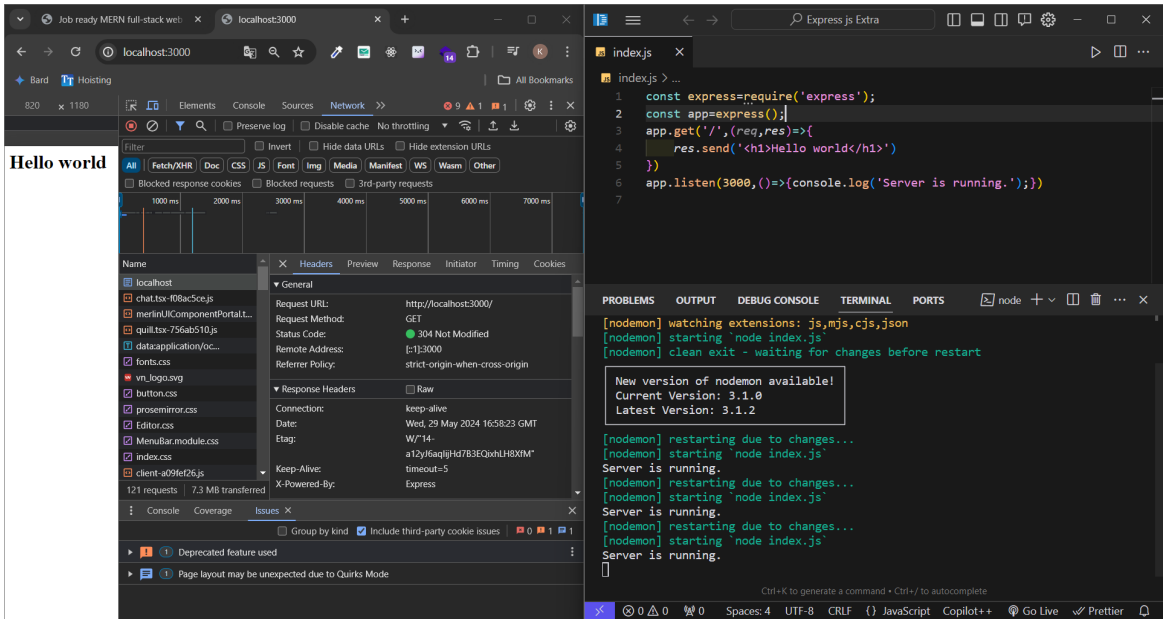
```
const express=require('express');
const app=express();
app.get('/',(req,res)=>{
    res.send('<h1>Hello world</h1>')
})
app.listen(3000,()=>{console.log('Server is running.');})
```



here in this code by default it self it is generating status code.

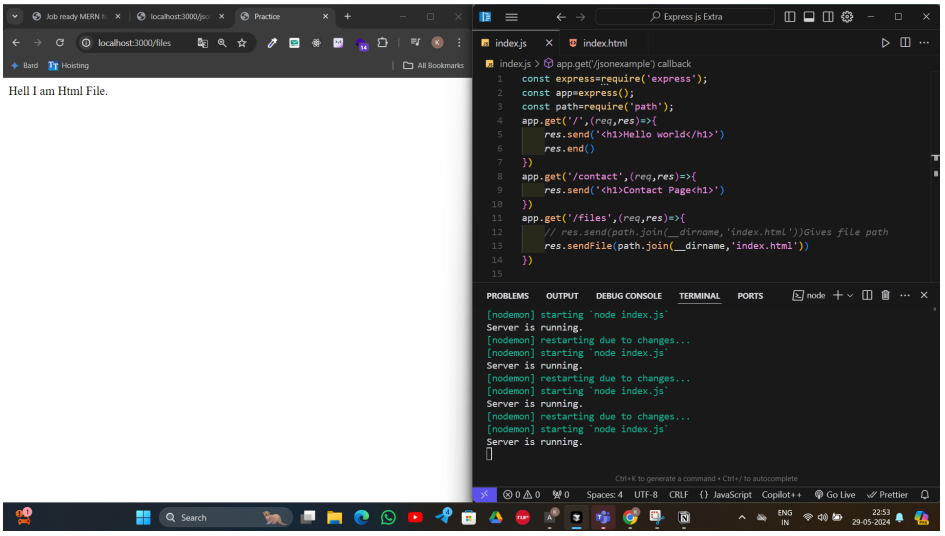
We can also write states code ourself.

Khushal Savalakha

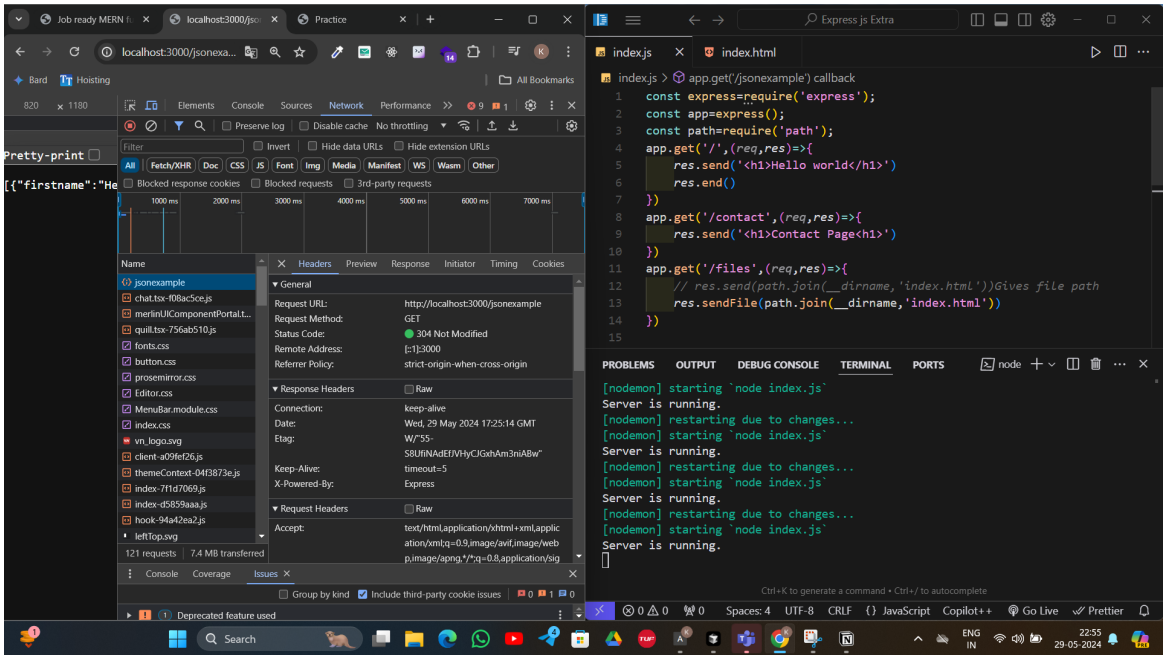


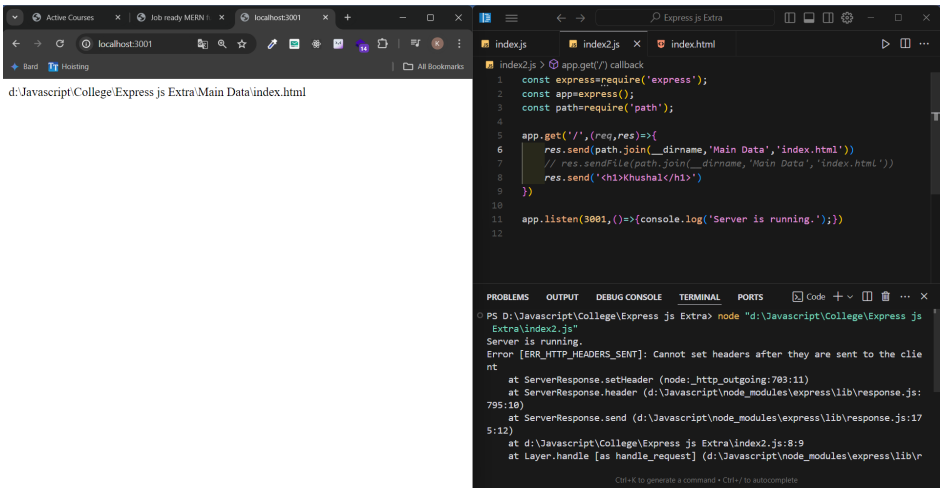

```
const express=require('express');
const app=express();
const path=require('path');
app.get('/',(req,res)=>{
    res.send('<h1>Hello world</h1>')
    res.end()
})
app.get('/contact',(req,res)=>{
    res.send('<h1>Contact Page<h1>')
})
app.get('/files',(req,res)=>{
    // res.send(path.join(__dirname,'index.html'))Gives fil
    res.sendFile(path.join(__dirname,'index.html'))
})

app.listen(3000,()=>{console.log('Server is running.');})
```


Khushal Savalakha




```
const express=require('express');
const app=express();
const path=require('path');
app.get('/',(req,res)=>{
    res.send('<h1>Hello world</h1>')
    res.end()
})
app.get('/contact',(req,res)=>{
    res.send('<h1>Contact Page<h1>')
})
app.get('/files',(req,res)=>{
    // res.send(path.join(__dirname,'index.html'))Gives fil
    res.sendFile(path.join(__dirname,'index.html'))
})

app.get('/jsonexample',(req,res)=>{
    res.json([{
        firstname:'Hello',
        lastname:'HelloLast'
    },{
        firstname:'Hi',
        lastname:'HiLast'
    }])
})
app.listen(3000,()=>{console.log('Server is running.');})
```

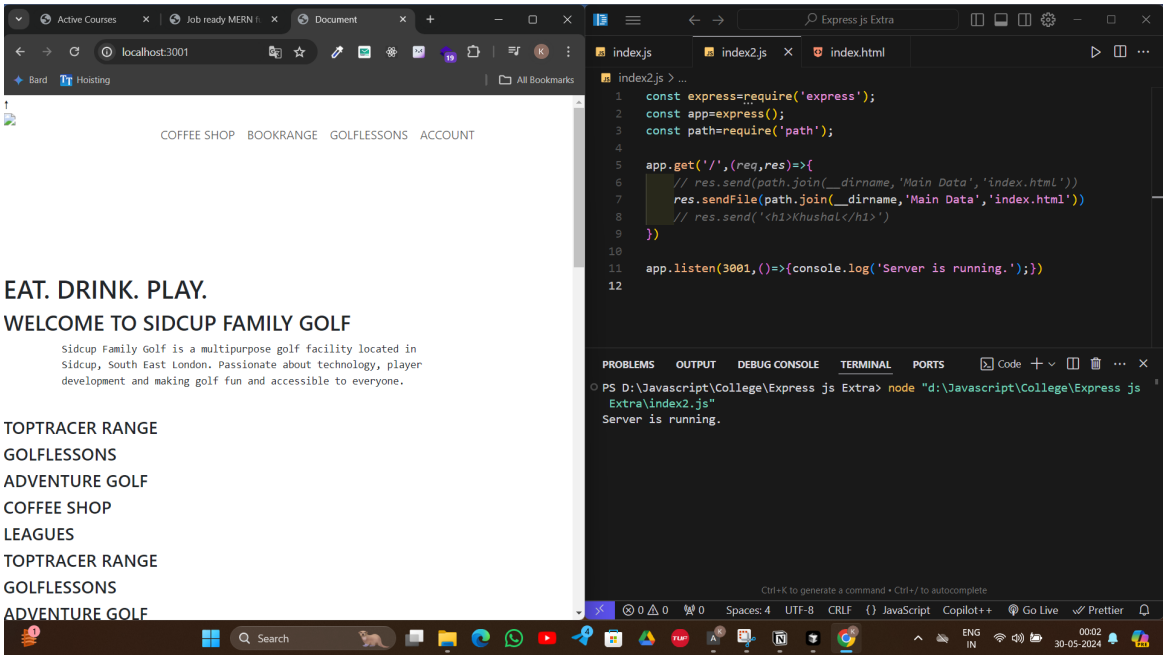





```
Server is running.
Error [ERR_HTTP_HEADERS_SENT]: Cannot set headers after the
at ServerResponse.setHeader (node:_http_outgoing:703:11)
at ServerResponse.header (d:\Javascript\node_modules\expres
at ServerResponse.send (d:\Javascript\node_modules\express\
at d:\Javascript\College\Express js Extra\index2.js:8:9
at Layer.handle [as handle_request] (d:\Javascript\node_moc
at next (d:\Javascript\node_modules\express\lib\router\rout
at Route.dispatch (d:\Javascript\node_modules\express\lib\r
at Layer.handle [as handle_request] (d:\Javascript\node_moc
```

```
at d:\Javascript\node_modules\express\lib\router\index.js:2
at Function.process_params (d:\Javascript\node_modules\expr
```

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

app.get('/',(req,res)=>{
    // res.send(path.join(__dirname,'Main Data','index.html
    res.sendFile(path.join(__dirname,'Main Data','index.htm
    // res.send('<h1>Khushal</h1>')
})

app.listen(3001,()=>{console.log('Server is running.');})
```

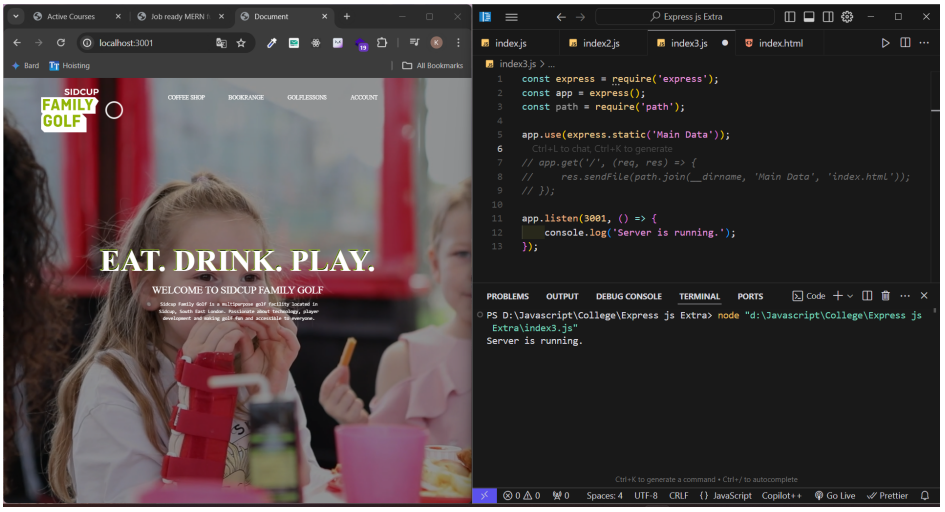


here in above image we can’t see any css or javascript effect beacause sendFile only send HTML code.

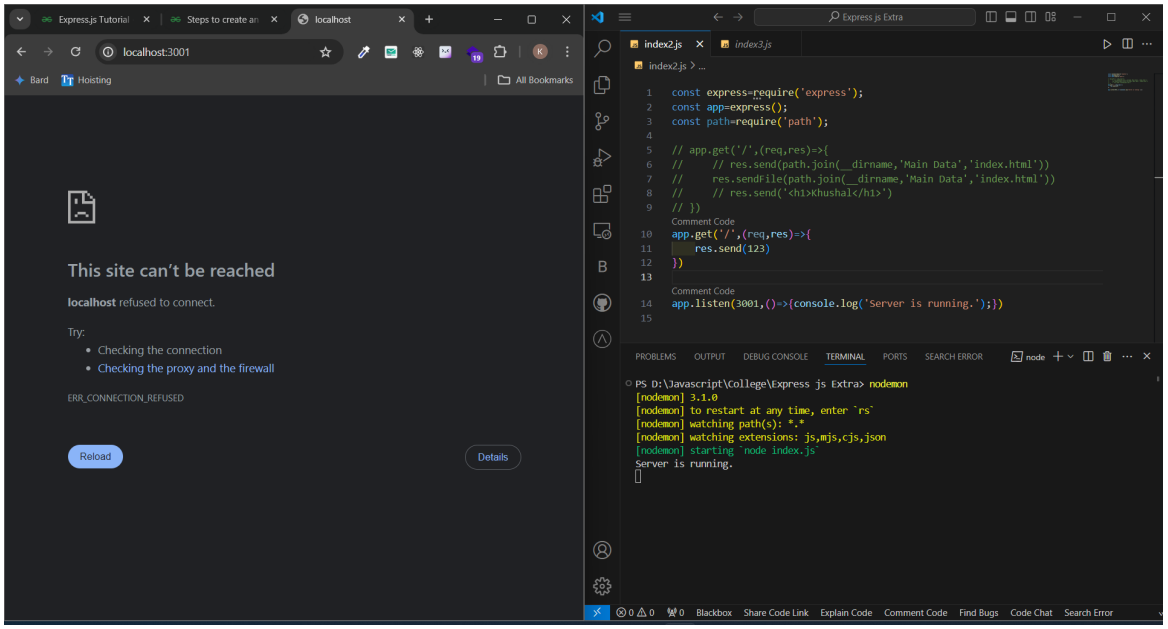
```
const express = require('express');
const app = express();
const path = require('path');
```


Khushal Savalakha

```
app.use(express.static('Main Data'));

// app.get('/', (req, res) => {
//     res.sendFile(path.join(__dirname, 'Main Data', 'inde
// });

app.listen(3001, () => {
    console.log('Server is running.');
});
```



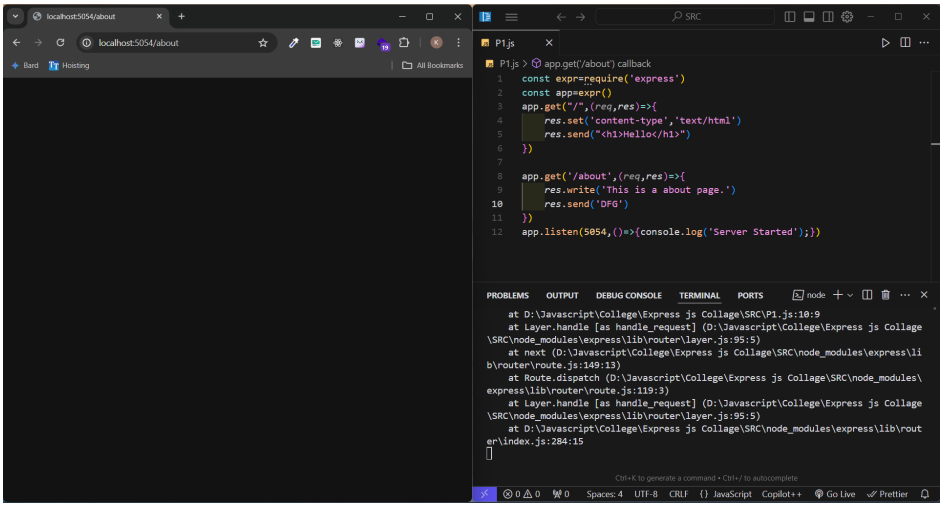
here static opens index.html file without specifying because by default index.html is root file.



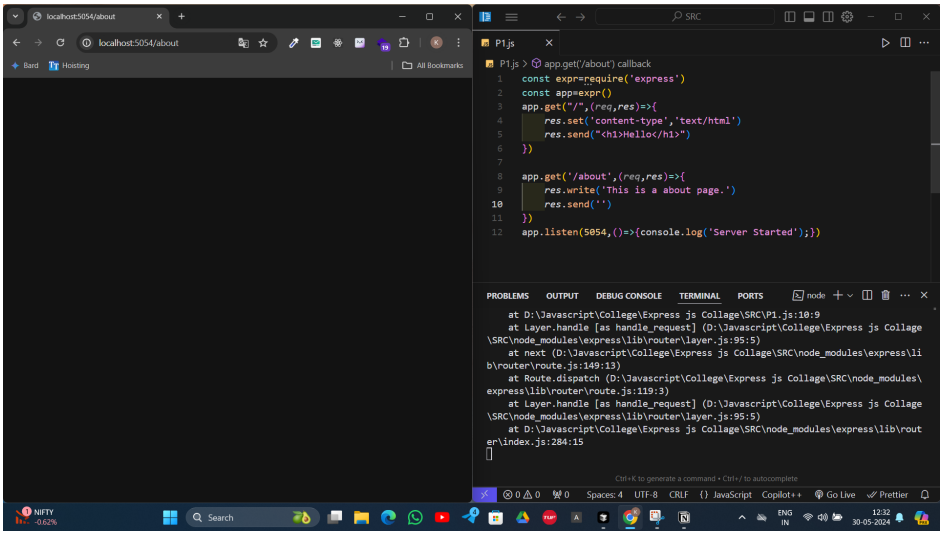
Khushal Savalakha

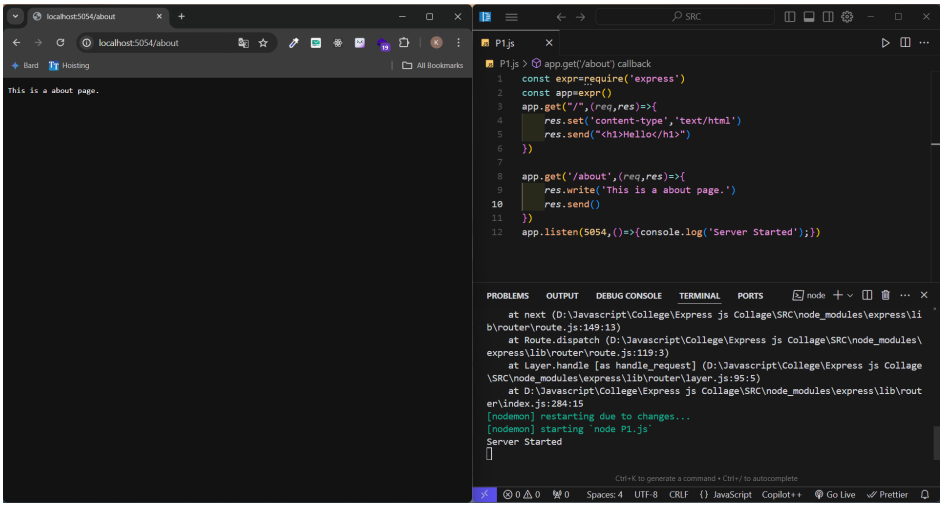
Here below code gives error while using res.send and res.write using simontaneously.

```
const expr=require('express')
const app=expr()
app.get("/",(req,res)=>{
    res.set('content-type','text/html')
    res.send("<h1>Hello</h1>")
})

app.get('/about',(req,res)=>{
    res.write('This is a about page.')
    res.send('DFG')
})
app.listen(5054,()=>{console.log('Server Started');})
```

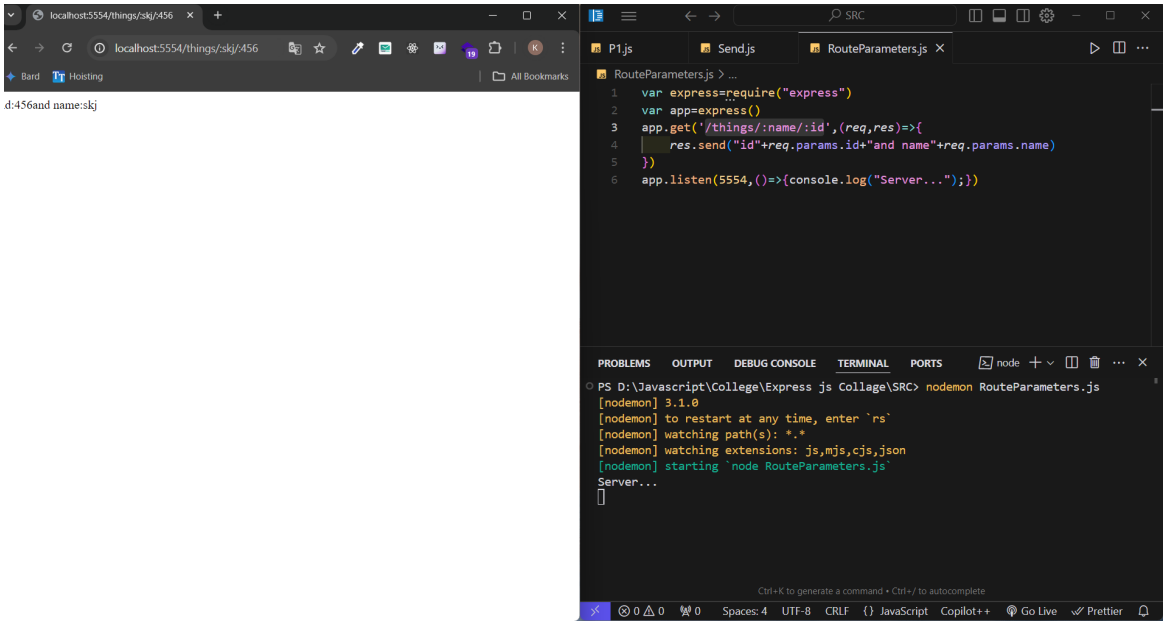


Khushal Savalakha



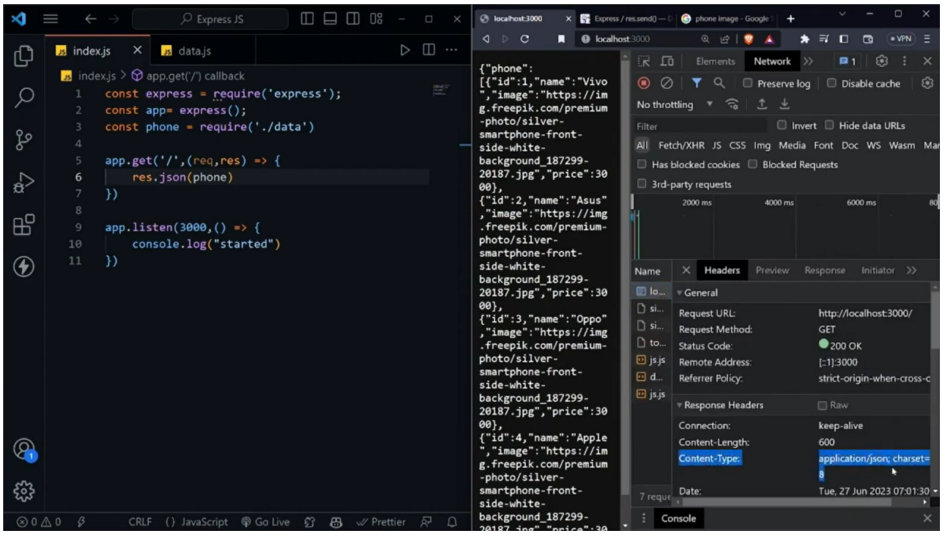


Route Parameters

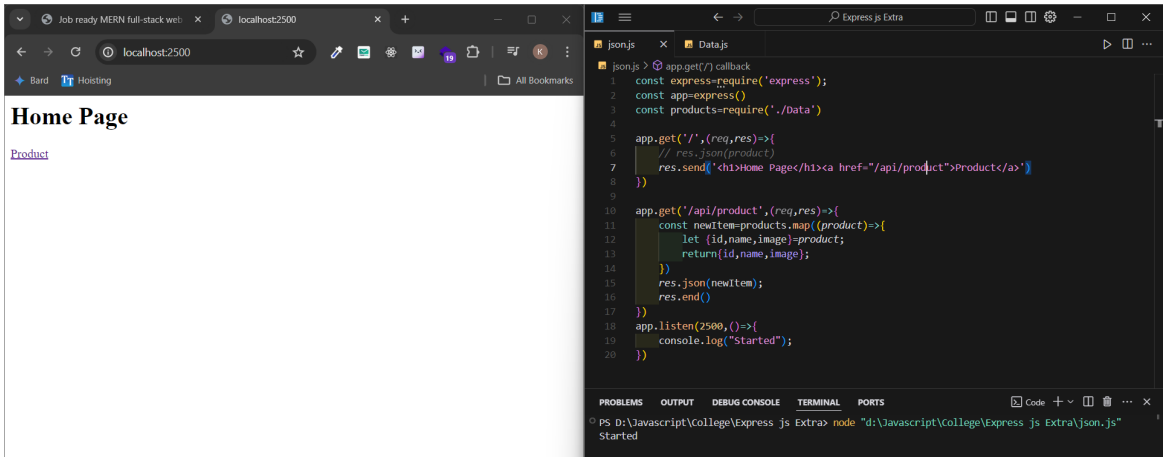


Khushal Savalakha

▼ JSON



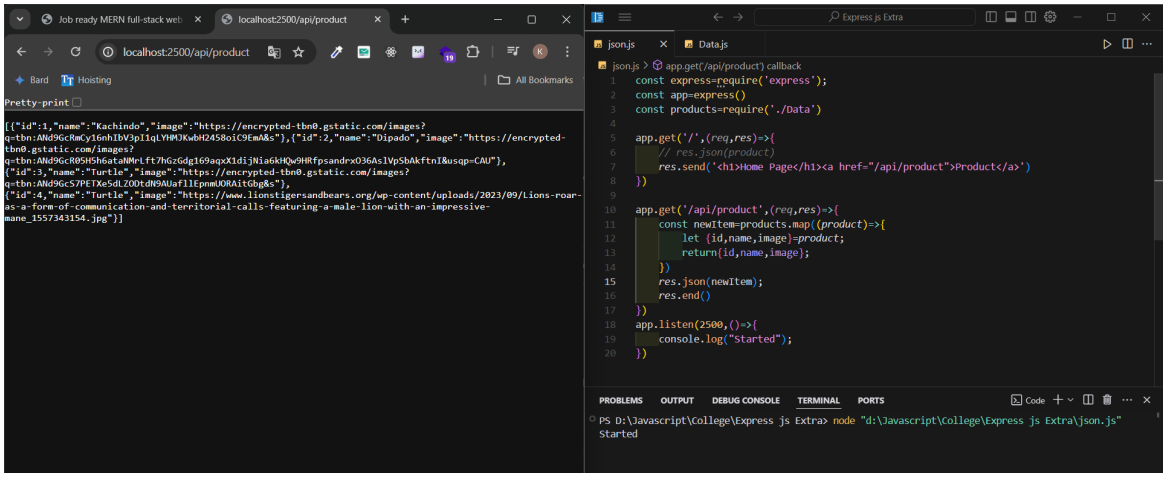
here we don't have to specify content type .

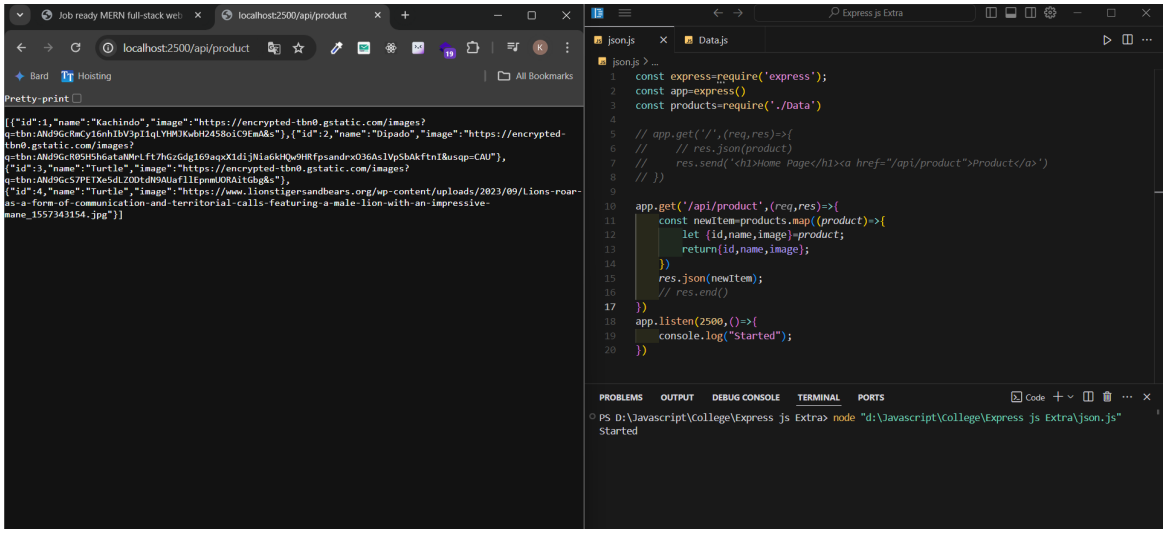



```
const express=require('express');
const app=express()
const products=require('./Data')

app.get('/',(req,res)=>{
    // res.json(product)
    res.send('<h1>Home Page</h1><a href="/api/product">Proc
})

app.get('/api/product',(req,res)=>{
    const newItem=products.map((product)=>{
```


Khushal Savalakha

```
            let {id,name,image}=product;
            return{id,name,image};
        })
        res.json(newItem);
        res.end()
    })
    app.listen(2500,()=>{
        console.log("Started");
    })
```





Sure, let's break down how this Express.js code works step by step.

Code Breakdown

1. **Importing Modules and Initializing the App**:

```
const express = require('express');
const app = express();
const products = require('./Data');
```

- `express`: This is the Express framework which is used to build web applications and APIs in Node.js.
- `app`: This creates an instance of the Express application.
- `products`: This imports the data from a local file named `Data.js` (assumed to be in the same directory). This `products` variable will be used to access the product data.

2. **Defining the Root Route**:

```
app.get('/', (req, res) => {
    res.send('<h1>Home Page</h1><a href="/api/produc
t">Product</a>');
});
```

- `app.get('/')`: This defines a route handler for the root URL (`/`). When a GET request is made to the root URL, the provided callback function is executed.
- **Callback Function**: Sends an HTML response with a heading "Home Page" and a link to the `/api/product` endpoint.

3. **Defining the /api/product Route**:

```
app.get('/api/product', (req, res) => {
    const newItem = products.map((product) => {
        let { id, name, image } = product;
        return { id, name, image };
    });
    res.json(newItem);
});
```

- `app.get('/api/product')`: This defines a route handler for the URL `/api/product`. When a GET request is made to this URL, the provided callback function is executed.
- **Callback Function**:

Khushal Savalakha

- `products.map((product) => {...})`: This iterates over each product in the `products` array and creates a new array `newItem`. For each product, it extracts the `id`, `name`, and `image` properties and returns a new object containing only these properties.
- `res.json(newItem)`: This sends the new array of products (`newItem`) as a JSON response.

4. **Starting the Server**:

```
app.listen(2500, () => {
    console.log("Started");
});
```

- `app.listen(2500, ...)`: This starts the Express server on port 2500. The provided callback function is executed once the server is up and running, logging "Started" to the console.

Example of `products` Data

Assuming the `products` data in `Data.js` looks like this:

```
module.exports = [
    { id: 1, name: 'Product 1', image: 'image1.jpg', pri
ce: 100, description: 'Description 1' },
    { id: 2, name: 'Product 2', image: 'image2.jpg', pri
ce: 200, description: 'Description 2' },
    // more products...
];
```

How It Works

1. **Accessing the Home Page**:
   - When you navigate to `http://localhost:2500/`, the server responds with an HTML page containing "Home Page" and a link to the `/api/product` endpoint.
2. **Accessing the Product API**:
   - When you click the link or navigate to `http://localhost:2500/api/product`, the server responds with a JSON array of products, but only including the `id`, `name`, and `image` properties of each product.

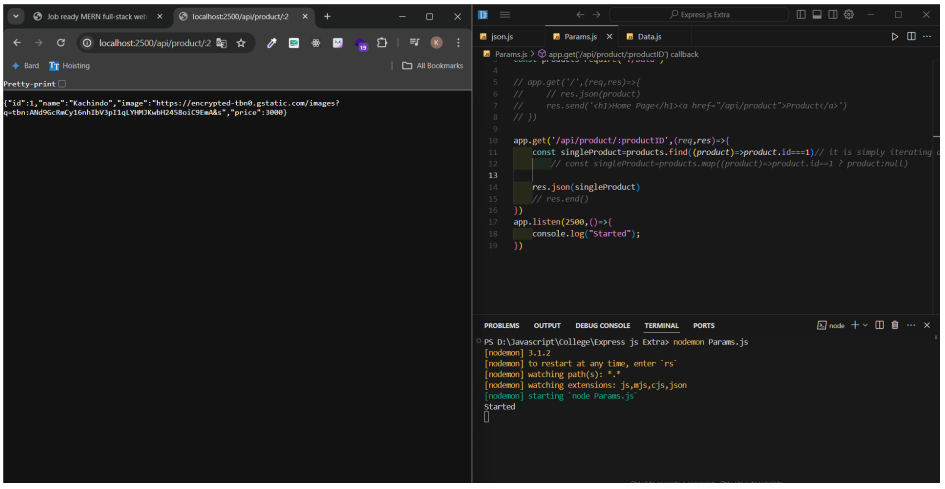
Khushal Savalakha

Summary

- The server is set up to handle two routes: the root (`/`) and `/api/product`.
- The root route responds with a simple HTML page.
- The `/api/product` route responds with a JSON array containing specific fields (`id`, `name`, `image`) from the product data.
- The server runs on port 2500, and once started, logs "Started" to the console.

This is a basic example of setting up an Express.js server to handle different routes and respond with HTML and JSON data.

▼ PARAMS



Sure, let's clarify what "the first element" means in the context of the `find()` method.

What Does "First Element" Mean?

When we say that `find()` returns the "first element" that satisfies a given condition, we mean that it returns the first element it encounters in the array (from the beginning to the end of the array) that matches the condition specified in the callback function.

Example to Illustrate

Consider the following array and the use of `find()`:

```
const numbers = [4, 9, 16, 25, 29, 35];

// Find the first element greater than 20
```


Khushal Savalakhia

```
const found = numbers.find(element => element > 20);

console.log(found); // Output: 25
```

Detailed Steps:

1. **Array Traversal:** The `find()` method starts at the beginning of the `numbers` array.
2. **Condition Checking:** For each element, it checks if the element meets the condition specified in the callback function (`element > 20` in this case).
3. **First Match:** It encounters 4 (does not meet the condition), 9 (does not meet the condition), 16 (does not meet the condition), and then 25 (meets the condition).
4. **Return Value:** Once it finds 25 (the first element that meets the condition), it returns 25 and stops further execution. It does not check the remaining elements (29 and 35) because it already found the first match.

Applying This to Your Code

In your Express.js code:

```
app.get('/api/product/:productID', (req, res) => {
    const productID = parseInt(req.params.productID, 1
0); // Convert the productID from string to integer
    const singleProduct = products.find((product) => pro
duct.id === productID); // Find the product by id

    if (singleProduct) {
        res.json(singleProduct); // Send the found produ
ct as JSON
    } else {
        res.status(404).send('Product not found'); // Se
nd 404 if no product is found
    }
});
```

- **Array of Products:** Assume `products` is an array of product objects, each with an `id` property.
- **Finding Product by ID:** The `find()` method will check each product in the `products` array to see if its `id` matches the `productID` extracted from the request

Khushal Savalakha

URL.

- **First Match:** As soon as it finds the first product with the matching `id`, it returns that product and stops checking further. If no product matches, it returns `undefined`.

Example Array of Products

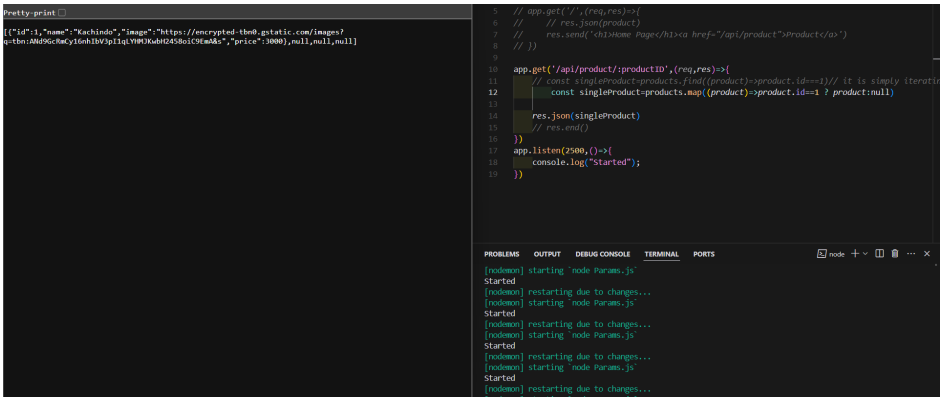
```
const products = [
    { id: 1, name: "Product A" },
    { id: 2, name: "Product B" },
    { id: 3, name: "Product C" }
];

// URL: /api/product/2
```

For a request to `/api/product/2`:

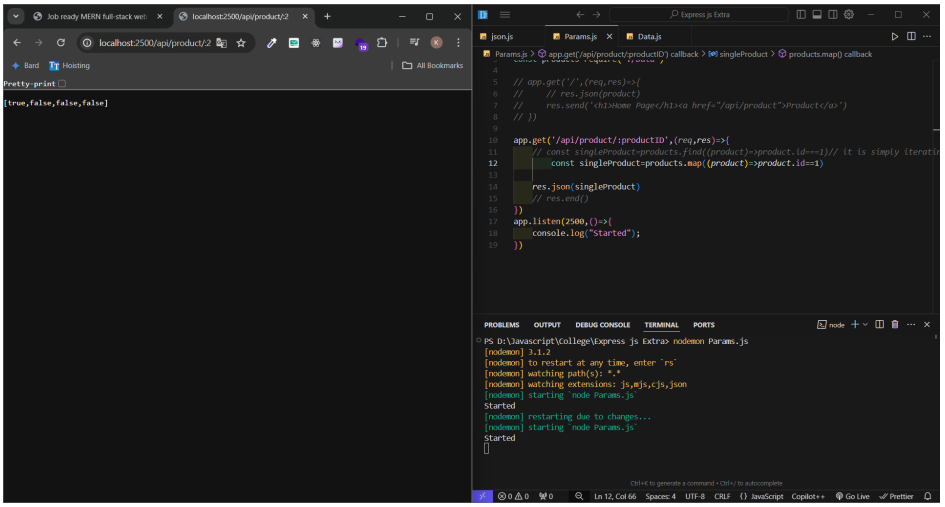
- `find()` will check the first product `{ id: 1, name: "Product A" }` (does not match).
- Then it checks the second product `{ id: 2, name: "Product B" }` (matches).
- It returns `{ id: 2, name: "Product B" }` as it is the first element that meets the condition.

Summary

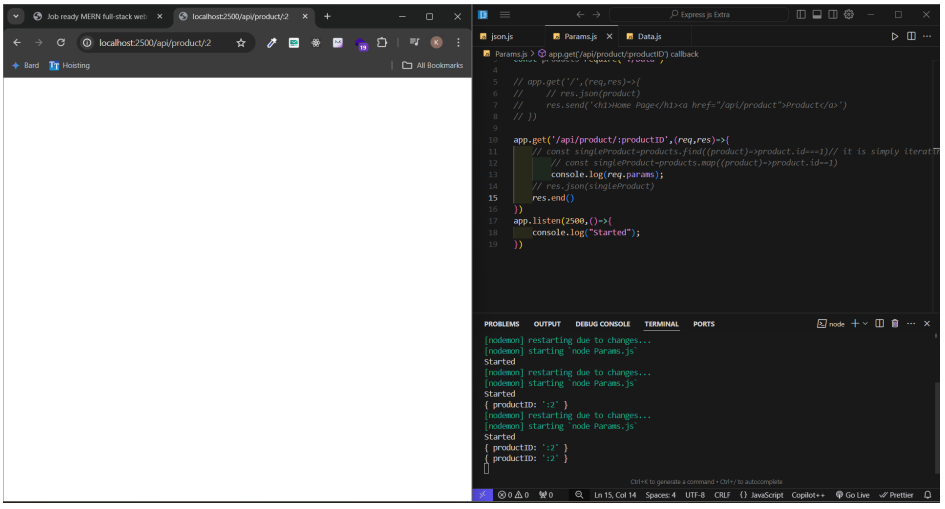
The "first element" means the first occurrence in the array that satisfies the condition specified in the `find()` method's callback function. It doesn't refer to the order in the data but rather the sequence in the array traversal.



Map function returns True or False



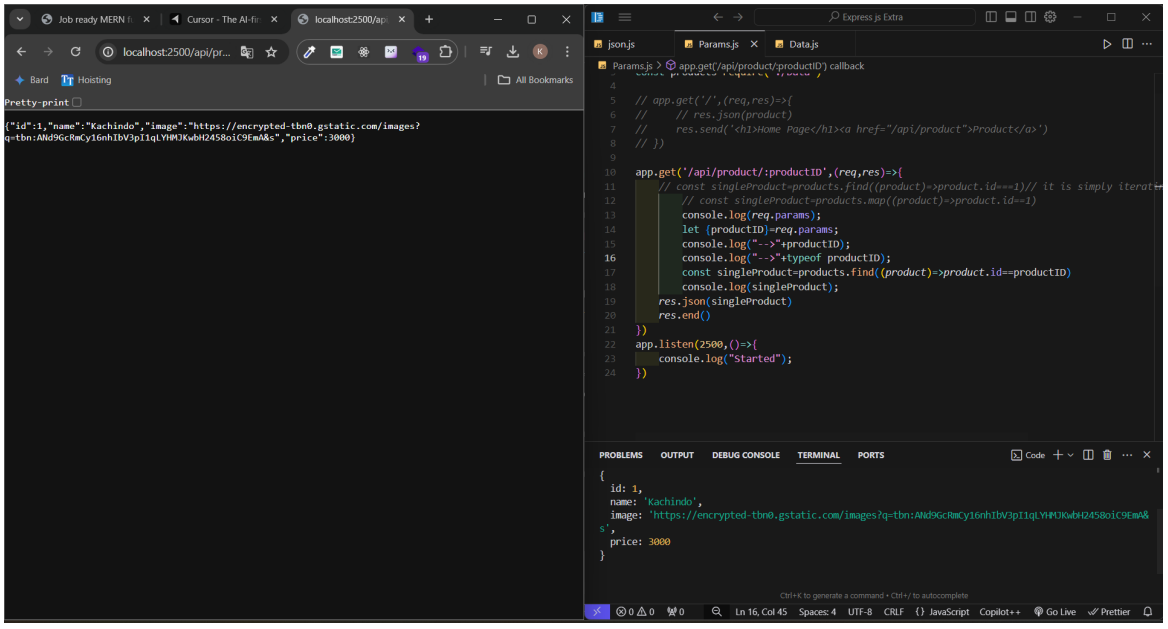
PARAMS




```
const express=require('express');
const app=express()
const products=require('./Data')

// app.get('/',(req,res)=>{
//     // res.json(product)
//     res.send('<h1>Home Page</h1><a href="/api/product">P
// })

app.get('/api/product/:productID',(req,res)=>{
    // const singleProduct=products.find((product)=>product
        // const singleProduct=products.map((product)=>proc
        console.log(req.params);
```

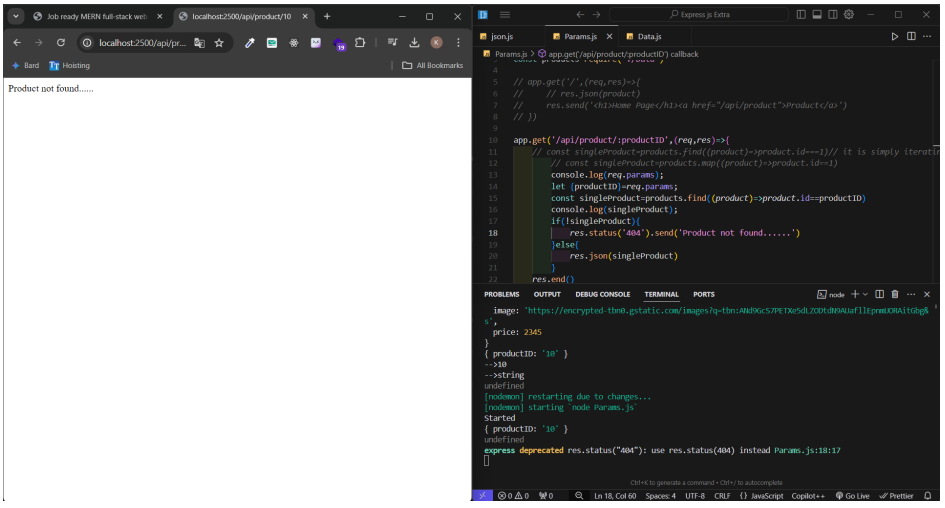

Khushal Savalakha

```
        let {productID}=req.params;
        const singleProduct=products.find((product)=>produc
        console.log(singleProduct);
    res.json(singleProduct)
    res.end()
})
app.listen(2500,()=>{
    console.log("Started");
})
```



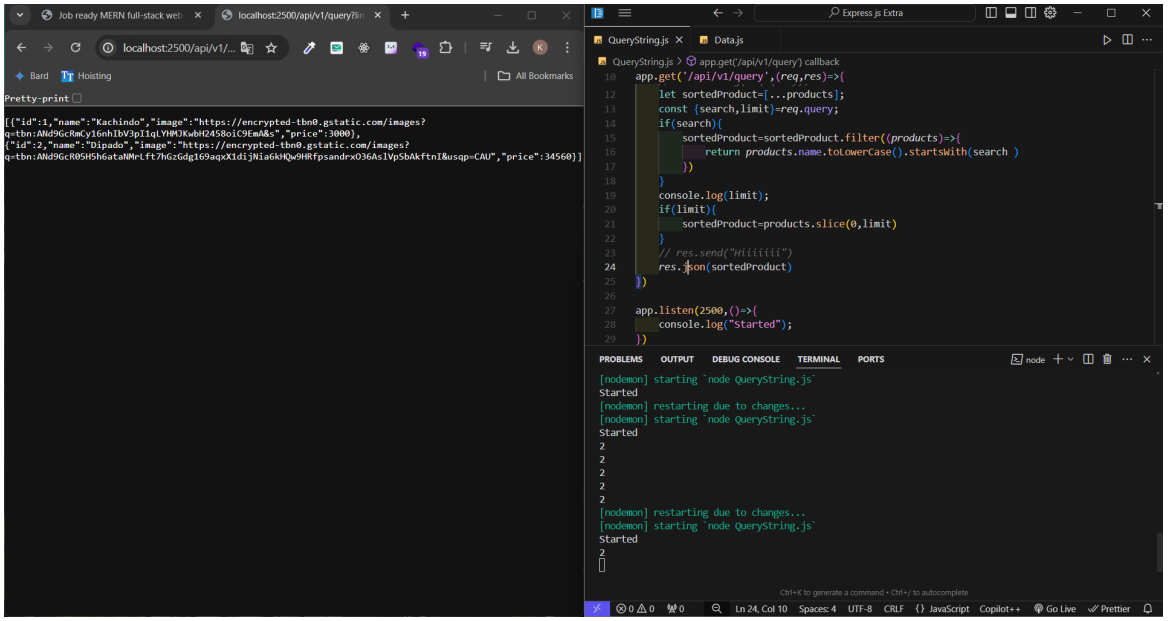

```
const express=require('express');
const app=express()
const products=require('./Data')

// app.get('/',(req,res)=>{
//     // res.json(product)
//     res.send('<h1>Home Page</h1><a href="/api/product">P
// })

app.get('/api/product/:productID',(req,res)=>{
    // const singleProduct=products.find((product)=>product
        // const singleProduct=products.map((product)=>proc
        console.log(req.params);
```

```
        let {productID}=req.params;
        const singleProduct=products.find((product)=>produc
        console.log(singleProduct);
        if(!singleProduct){
            res.status('404').send('Product not found......
            // res.send('Product not found.....')
        }else{
            res.json(singleProduct)
        }
    res.end()
})
app.listen(2500,()=>{
    console.log("Started");
})
```




```
const express=require('express');
const app=express()
const products=require('./Data');

app.get('/',(req,res)=>{
    // res.json(product)
    res.send('<h1>Home Page</h1><a href="/api/product">Proc
})

app.get('/api/v1/query',(req,res)=>{
    // console.log(req.query);
```

```
    let sortedProduct=[...products];
    const {search,limit}=req.query;
    if(search){
        sortedProduct=sortedProduct.filter((products)=>{
            return products.name.toLowerCase().startsWith(s
        })
    }
    console.log(limit);
    if(limit){
        sortedProduct=products.slice(0,limit)
    }
    // res.send("Hiiiiiii")
    res.json(sortedProduct)
})

app.listen(2500,()=>{
    console.log("Started");
})
```



Print data with limit barrier.

▼ app.use() & app.get()

Sure! In Express.js, `app.use()` is used to set up middleware functions that run during the processing of requests. Middleware can do things like log requests, serve static files, parse

Khushal Savalakha

request data, and handle errors. Here's a simple breakdown:

What is Middleware?

Middleware functions are pieces of code that run in the middle of the request-response cycle. They can:

- Modify request and response objects
- End the request-response cycle
- Call the next middleware function in the stack

Basic Examples

1. **Logging Requests:**

```
app.use((req, res, next) => {
    console.log(`${req.method} ${req.url}`);
    next(); // Move to the next middleware
});
```

2. **Serving Static Files:**

```
app.use(express.static('public'));
```

3. **Parsing JSON Data:**

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

4. **Handling Errors:**

```
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).send('Something broke!');
});
```

Summary

`app.use()` sets up functions (middleware) to handle different parts of incoming requests, making it a fundamental part of building an Express.js app.

Khushal Savalakha

In Express.js, `app.use()` and `app.get()` are both methods used to handle incoming requests, but they serve different purposes and are used in different contexts. Here's a simple explanation of the differences:

`app.use()`

- **Purpose:** `app.use()` is used to set up middleware. Middleware functions are executed for every request to the application, regardless of the HTTP method (GET, POST, etc.) or URL path.
- **Usage:** It's commonly used for tasks like logging, serving static files, parsing request bodies, and handling errors.

Example:

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

// Middleware to log all requests
app.use((req, res, next) => {
    console.log('Request URL:', req.url);
    next(); // Pass control to the next middleware
});

// Middleware to serve static files
app.use(express.static('public'));

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

`app.get()`

- **Purpose:** `app.get()` is used to define route handlers for GET requests to specific URL paths. It responds only to HTTP GET requests for the specified path.
- **Usage:** It's used to handle specific routes where you expect the client to use the GET method, such as fetching data or rendering a web page.

Example:

Khushal Savalakha

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

// Route handler for GET request to the root URL
app.get('/', (req, res) => {
    res.send('Hello, World!');
});

// Route handler for GET request to /about
app.get('/about', (req, res) => {
    res.send('About Page');
});

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

Summary

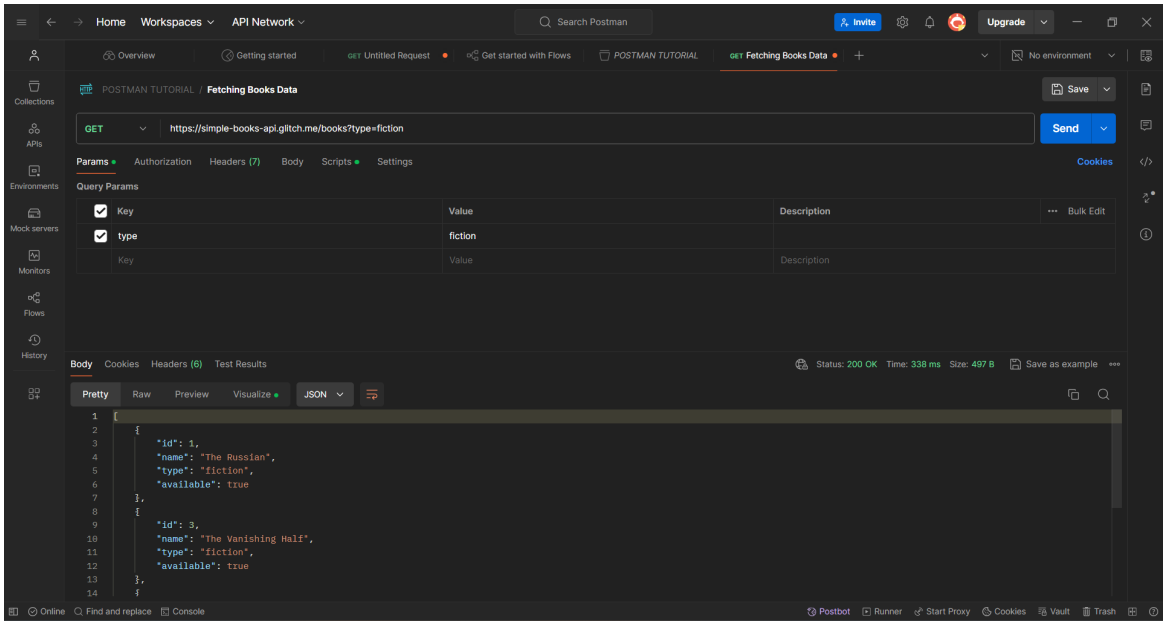
- `app.use()`: Sets up middleware that runs for every request (any method, any path).
- `app.get()`: Sets up a route handler for GET requests to a specific path.

These methods are fundamental for defining how your Express.js application handles and responds to different types of HTTP requests.

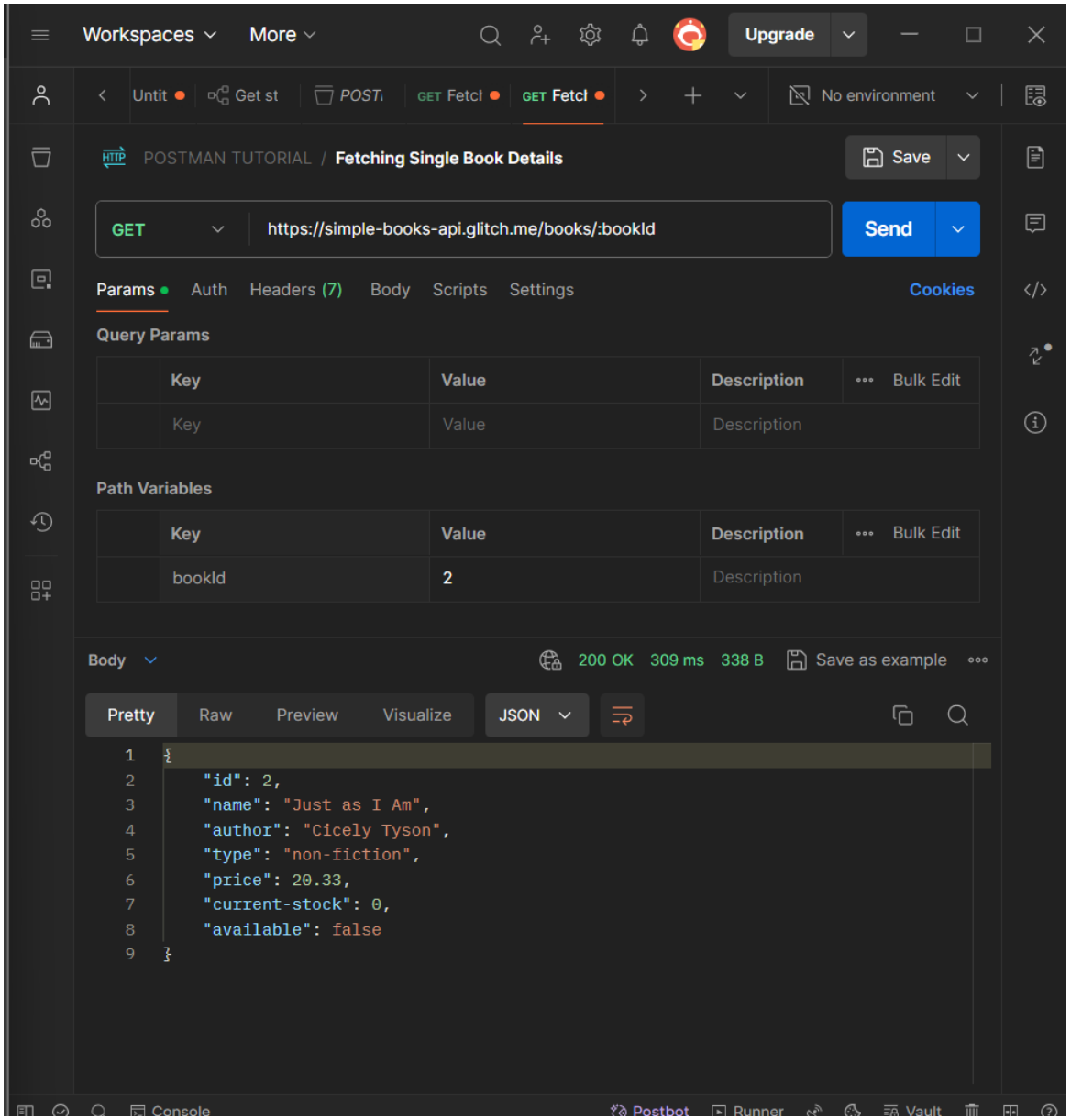
▼ Query String(URL parameter)

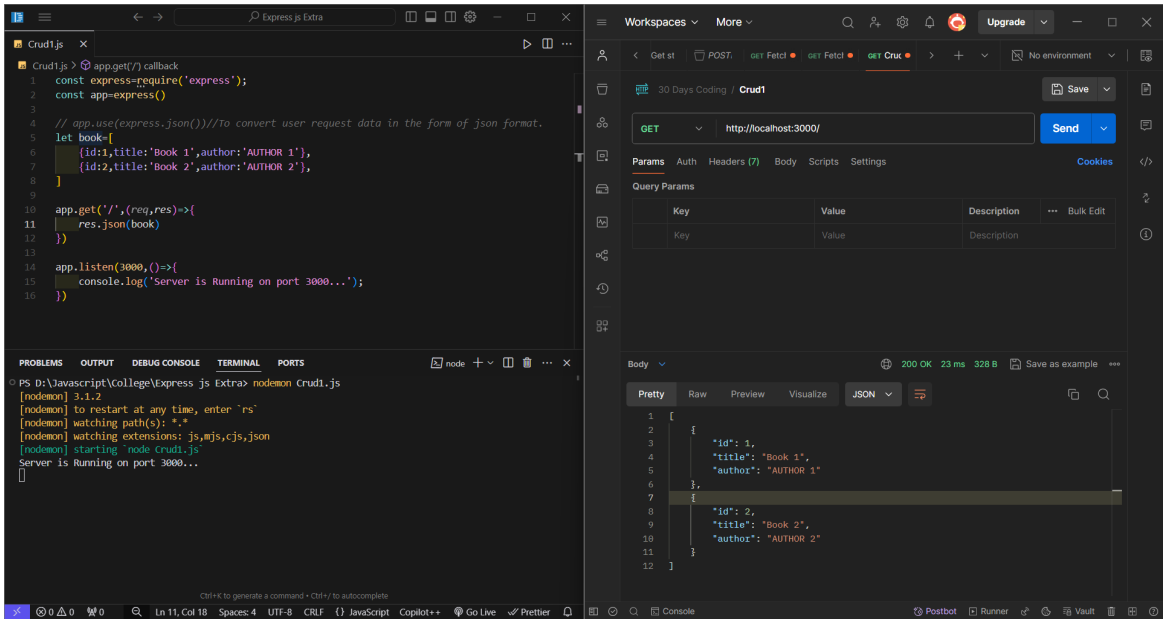
▼ Postman

Khushal Savalakha



Khushal Savalakha


Workspaces
More
Upgrade
No environment
POSTMAN TUTORIAL / Fetching Single Book Details
Save
GET
https://simple-books-api.glitch.me/books/:bookId
Send
Params
Auth
Headers (7)
Body
Scripts
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Path Variables
bookId
2
200 OK
309 ms
338 B
Save as example
Pretty
Raw
Preview
Visualize
JSON
{
"id": 2,
"name": "Just as I Am",
"author": "Cicely Tyson",
"type": "non-fiction",
"price": 20.33,
"current-stock": 0,
"available": false
}



▼ CRUD operations

▼ READ and WRITE

CRUD operations in the context of Express.js (a web application framework for Node.js) refer to the four basic functions of persistent storage, which are Create, Read, Update, and Delete. These operations are fundamental for interacting with databases or any data storage system. Here's a breakdown of each operation with simple explanations and examples in Express.js:

1. **Create**: This operation involves adding new data to the database.

   - **Example**: Adding a new user to the database.

```
// Assuming you have an Express app set up and a Us
er model for your database

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

// Middleware to parse JSON bodies
app.use(express.json());

app.post('/users', async (req, res) => {
    try {
        const newUser = await User.create(req.bod
y); // Create a new user
        res.status(201).json(newUser); // Respond w
```


Khushal Savalakha

```
ith the created user
    } catch (error) {
        res.status(400).json({ message: error.messa
ge });
    }
});

// Start the server
app.listen(3000, () => console.log('Server running
on port 3000'));
```

2. **Read**: This operation involves retrieving data from the database.
   - **Example**: Getting a list of all users or a specific user by ID.

```
app.get('/users', async (req, res) => {
    try {
        const users = await User.find(); // Retriev
e all users
        res.status(200).json(users); // Respond wit
h the list of users
    } catch (error) {
        res.status(500).json({ message: error.messa
ge });
    }
});

app.get('/users/:id', async (req, res) => {
    try {
        const user = await User.findById(req.param
s.id); // Retrieve user by ID
        if (user) {
            res.status(200).json(user); // Respond
with the user data
        } else {
            res.status(404).json({ message: 'User n
ot found' });
        }
    } catch (error) {
```


Khushal Savalakha

```
        res.status(500).json({ message: error.messa
ge });
    }
});
```

3. **Update**: This operation involves modifying existing data in the database.
   - **Example**: Updating a user's information by ID.

```
app.put('/users/:id', async (req, res) => {
    try {
        const updatedUser = await User.findByIdAndU
pdate(req.params.id, req.body, { new: true }); // U
pdate user by ID
        if (updatedUser) {
            res.status(200).json(updatedUser); // R
espond with the updated user data
        } else {
            res.status(404).json({ message: 'User n
ot found' });
        }
    } catch (error) {
        res.status(400).json({ message: error.messa
ge });
    }
});
```

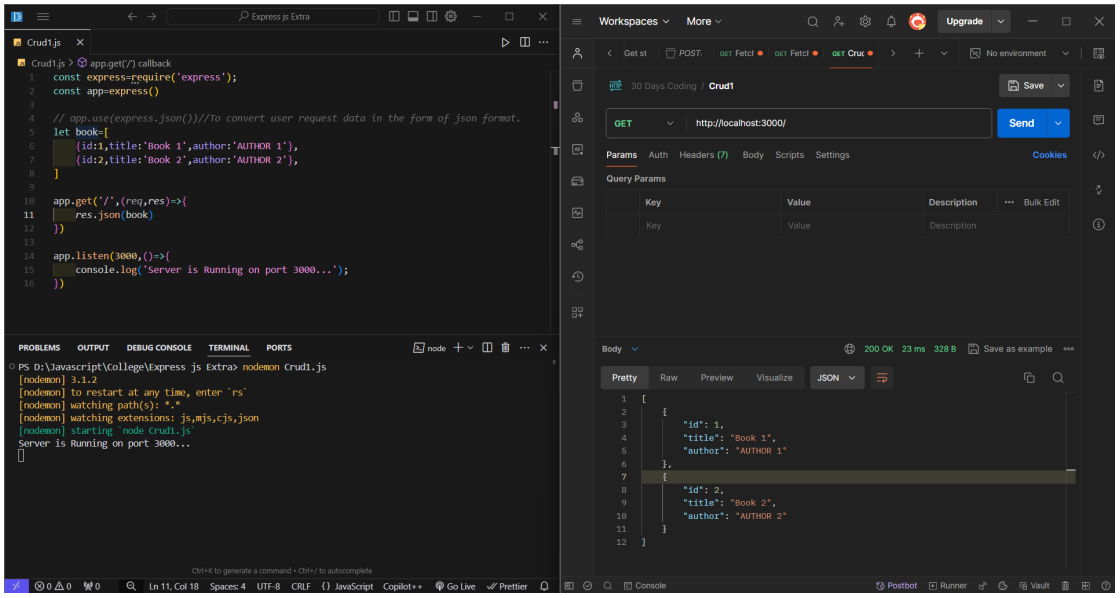
4. **Delete**: This operation involves removing data from the database.
   - **Example**: Deleting a user by ID.

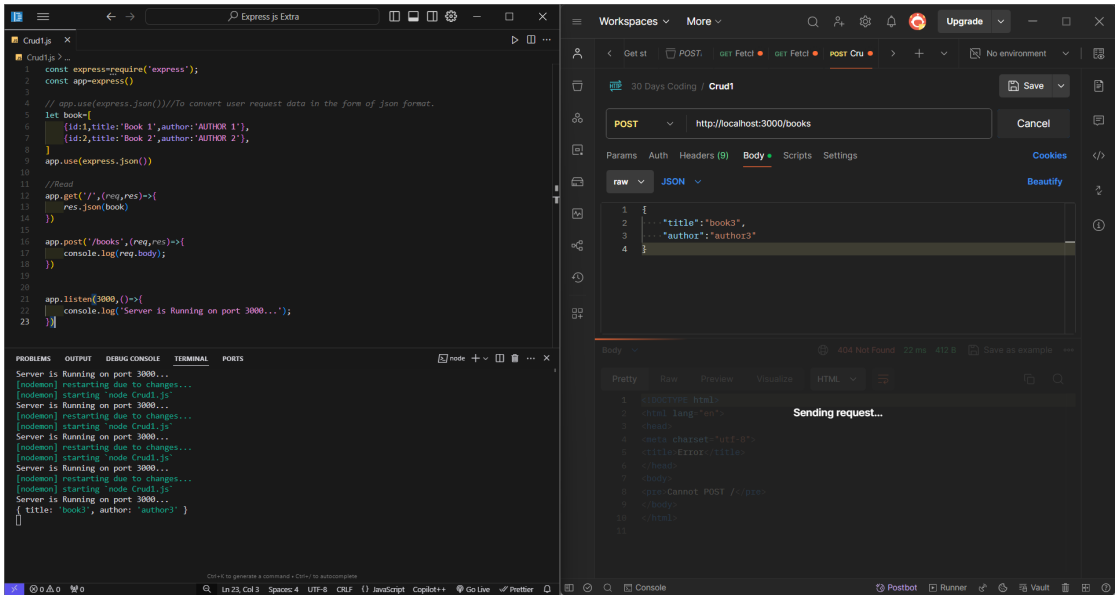
```
app.delete('/users/:id', async (req, res) => {
    try {
        const deletedUser = await User.findByIdAndD
elete(req.params.id); // Delete user by ID
        if (deletedUser) {
            res.status(200).json({ message: 'User d
eleted' }); // Respond with a success message
        } else {
```


Khushal Savalakha

```
            res.status(404).json({ message: 'User n
ot found' });
        }
    } catch (error) {
        res.status(500).json({ message: error.messa
ge });
    }
});
```

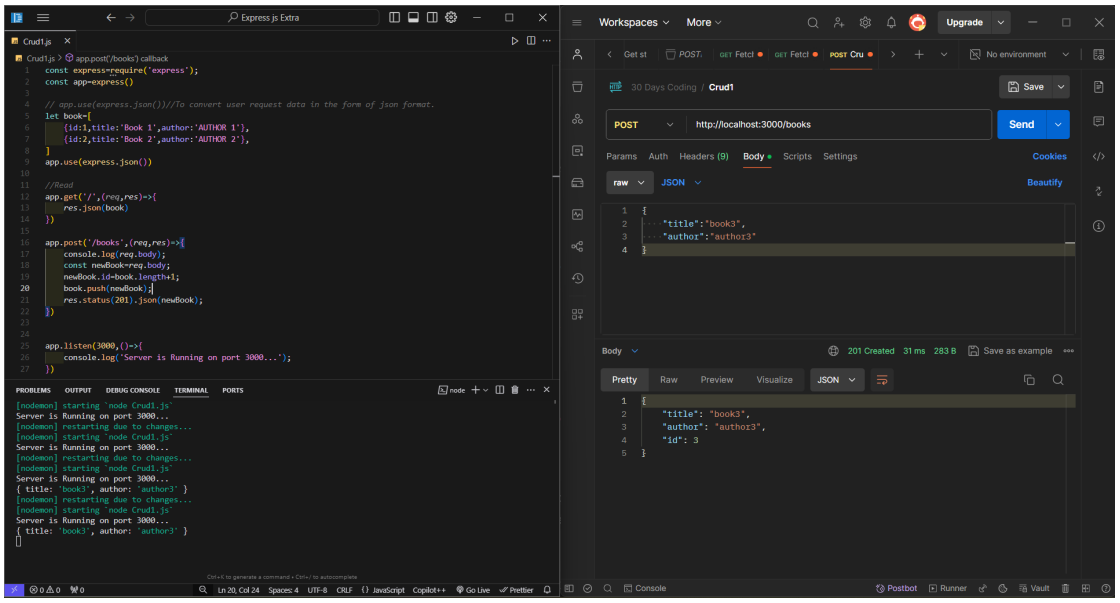
In summary, CRUD operations in Express.js allow you to create, read, update, and delete data from a database, providing a fundamental set of operations for managing application data.





Khushal Savalakha

Here we are doing get and post operation.




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

// app.use(express.json())//To convert user request data
let book=[
    {id:1,title:'Book 1',author:'AUTHOR 1'},
    {id:2,title:'Book 2',author:'AUTHOR 2'},
]
app.use(express.json())

//Read
app.get('/',(req,res)=>{
    res.json(book)
})

app.post('/books',(req,res)=>{
    console.log(req.body);
    const newBook=req.body;
    newBook.id=book.length+1;
    book.push(newBook);
    res.status(201).json(newBook);
})
```


Khushal Savalakha

```
app.listen(3000,()=>{
    console.log('Server is Running on port 3000...');
})
```

💡 it means here in post operation we are fetching user request data and add it in json object and in response we give status code 201 which means new data is created am i true ?

Yes, you are correct. Let's break down what your code does, particularly focusing on the POST operation and the status code 201.

1. **Setting up the Express application**:

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

2. **Middleware to parse JSON**:

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

   This middleware is used to parse incoming requests with JSON payloads, so you can access `req.body` in your routes.

3. **Initial Data**:

   ```
   let book = [
       { id: 1, title: 'Book 1', author: 'AUTHOR 1' },
       { id: 2, title: 'Book 2', author: 'AUTHOR 2' },
   ];
   ```

4. **Read Operation**:

   ```
   app.get('/', (req, res) => {
       res.json(book);
   });
   ```


Khushal Savalakha

This route handles GET requests to the root URL `/` and responds with the list of books in JSON format.

5. **Create Operation (POST)**:

```
app.post('/books', (req, res) => {
    console.log(req.body);
    const newBook = req.body;
    newBook.id = book.length + 1; // Assign a new I
D to the new book
    book.push(newBook); // Add the new book to the
book array
    res.status(201).json(newBook); // Respond with
status 201 and the new book
});
```

- When a POST request is made to `/books`, the server:
  - Logs the request body (`req.body`) to the console.
  - Extracts the new book data from the request body.
  - Assigns a new ID to the new book (the new ID is one more than the current length of the `book` array).
  - Adds the new book to the `book` array.
  - Responds with a status code of 201 (`Created`) and the newly created book in JSON format.

6. **Starting the Server**:

```
app.listen(3000, () => {
    console.log('Server is Running on port 300
0...');
});
```

This starts the server and listens for incoming requests on port 3000.

Explanation of Status Code 201

- **201 Created**: This status code is used to indicate that the request has been fulfilled and resulted in a new resource being created. The server provides the

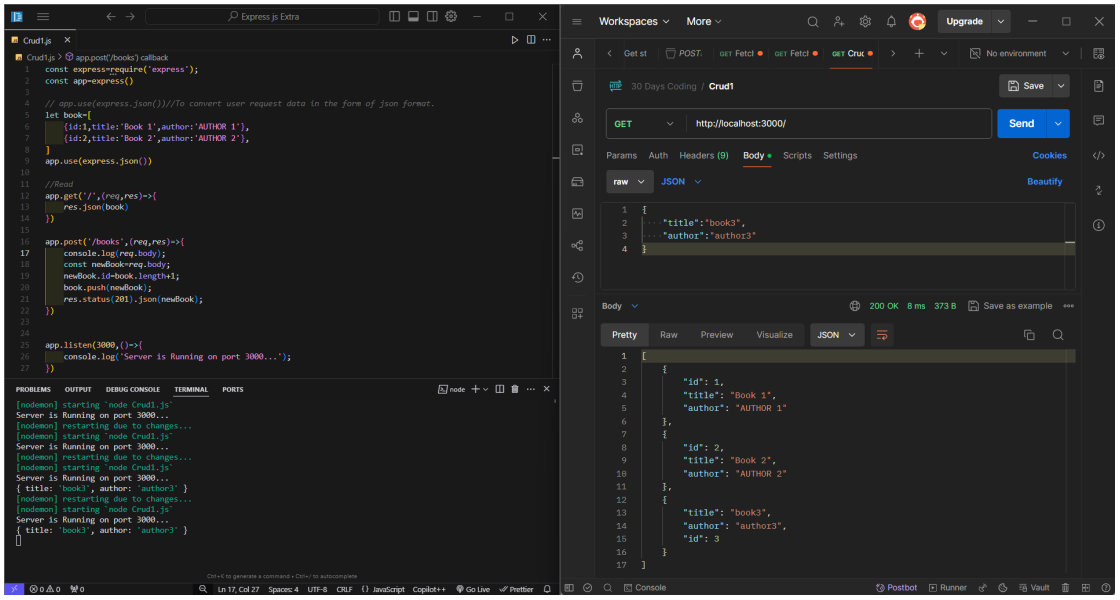
Khushal Savalakha

newly created resource in the response body.

Summary

In your code, when a POST request is made to the `/books` endpoint, the server processes the request, adds the new book to the `book` array, assigns it a new ID, and responds with status code 201, indicating that a new resource (the book) has been successfully created. The newly created book's data is then sent back to the client in the response body.

After doing Post operation



what is the status code mean 201 ?

The status code 201 means "Created." It indicates that a request has succeeded and has led to the creation of a new resource. When a server responds with a 201 status code, it signifies that a new resource has been successfully created, and it often includes information about the new resource, such as a URL pointing to its location.

In the context of an Express.js application, you typically use the 201 status code when a POST request is made to create a new resource. Here's an example:

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


Khushal Savalakha

```
// Middleware to parse JSON bodies
app.use(express.json());

app.post('/users', async (req, res) => {
    try {
        const newUser = await User.create(req.body);
// Create a new user
        res.status(201).json(newUser); // Respond with
the created user and status code 201
    } catch (error) {
        res.status(400).json({ message: error.message
}); // Handle errors (e.g., validation errors)
    }
});

// Start the server
app.listen(3000, () => console.log('Server running on
port 3000'));
```

In this example, when a new user is successfully created, the server responds with a 201 status code and the details of the created user. This tells the client that their request to create a new resource was successful, and it provides the newly created resource in the response body.

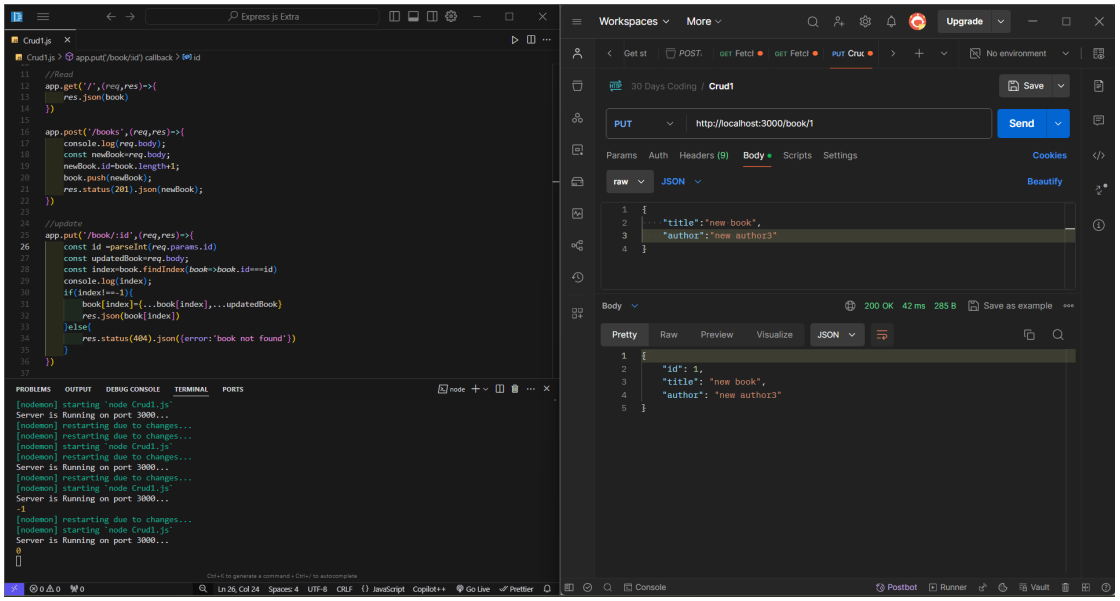
▼ UPDATE

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

// app.use(express.json())//To convert user request data
let book=[
    {id:1,title:'Book 1',author:'AUTHOR 1'},
    {id:2,title:'Book 2',author:'AUTHOR 2'},
]
app.use(express.json())

//Read
app.get('/',(req,res)=>{
    res.json(book)
```


Khushal Savalakhia

```
})

app.post('/books',(req,res)=>{
    console.log(req.body);
    const newBook=req.body;
    newBook.id=book.length+1;
    book.push(newBook);
    res.status(201).json(newBook);
})

//update
app.put('/book/:id',(req,res)=>{
    const id =parseInt(req.params.id)
    const updatedBook=req.body;
    const index=book.findIndex(book=>book.id===id)
    console.log(index);
    if(index!==-1){
        book[index]={...book[index],...updatedBook}
        res.json(book[index])
    }else{
        res.status(404).json({error:'book not found'})
    }
})

app.listen(3000,()=>{
    console.log('Server is Running on port 3000...');
})
```


Khushal Savalakha

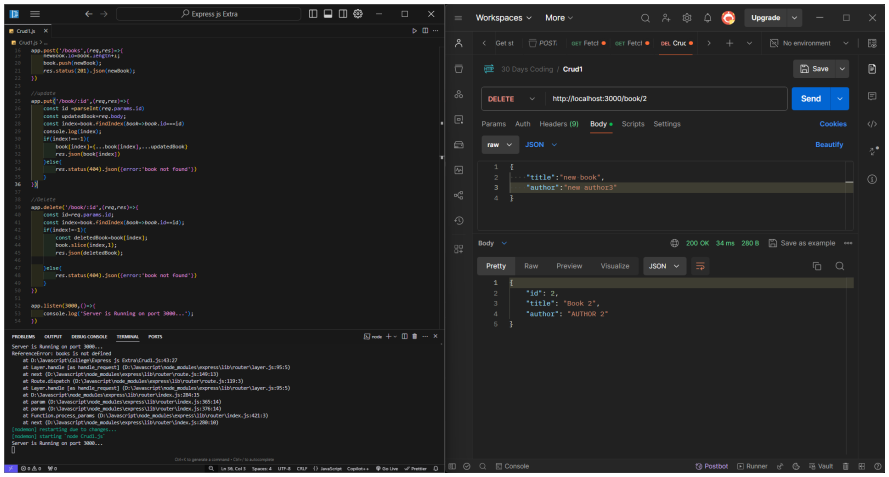


▼ DELETE

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

// app.use(express.json())//To convert user request data
let book=[
    {id:1,title:'Book 1',author:'AUTHOR 1'},
    {id:2,title:'Book 2',author:'AUTHOR 2'},
]
app.use(express.json())

//Read
app.get('/',(req,res)=>{
    res.json(book)
})

app.post('/books',(req,res)=>{
    console.log(req.body);
    const newBook=req.body;
    newBook.id=book.length+1;
    book.push(newBook);
    res.status(201).json(newBook);
})

//update
```

```
app.put('/book/:id',(req,res)=>{
    const id =parseInt(req.params.id)
    const updatedBook=req.body;
    const index=book.findIndex(book=>book.id===id)
    console.log(index);
    if(index!==-1){
        book[index]={...book[index],...updatedBook}
        res.json(book[index])
    }else{
        res.status(404).json({error:'book not found'})
    }
})

//Delete
app.delete('/book/:id',(req,res)=>{
    const id=req.params.id;
    const index=book.findIndex(book=>book.id==id);
    if(index!=-1){
        const deletedBook=book[index];
        book.slice(index,1);
        res.json(deletedBook);

    }else{
        res.status(404).json({error:'book not found'})
    }
})

app.listen(3000,()=>{
    console.log('Server is Running on port 3000...');
})
```


Khushal Savalakha



▼ Routing

```
//Crud2.js
const express=require('express');
const app=express()
const bookRoutes=require('./Routes/bookRoutes')

app.use(express.json());
app.use('/',bookRoutes)

app.listen(3000,()=>{
    console.log('Server is Running on port 3000...');
})
```

```
//bookController.js
let book = [
  { id: 1, title: "Book 1", author: "AUTHOR 1" },
  { id: 2, title: "Book 2", author: "AUTHOR 2" },
];
// app.use(express.json());

//Read
const getBooks = (req, res) => {
  res.json(book);
};
```


Khushal Savalakha

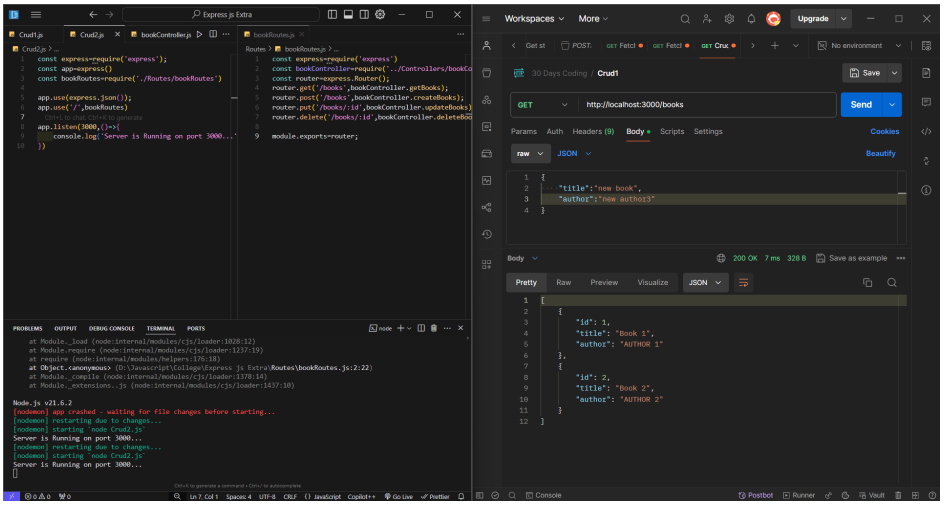
```
const createBooks = (req, res) => {
  console.log(req.body);
  const newBook = req.body;
  newBook.id = book.length + 1;
  book.push(newBook);
  res.status(201).json(newBook);
};

//update
const updateBooks = (req, res) => {
  const id = parseInt(req.params.id);
  const updatedBook = req.body;
  const index = book.findIndex((book) => book.id === id);
  console.log(index);
  if (index !== -1) {
    book[index] = { ...book[index], ...updatedBook };
    res.json(book[index]);
  } else {
    res.status(404).json({ error: "book not found" });
  }
};

//Delete
const deleteBooks = (req, res) => {
  const id = req.params.id;
  const index = book.findIndex((book) => book.id == id);
  if (index != -1) {
    const deletedBook = book[index];
    book.slice(index, 1);
    res.json(deletedBook);
  } else {
    res.status(404).json({ error: "book not found" });
  }
};

module.exports = {
  getBooks,
  createBooks,
```


Khushal Savalakha

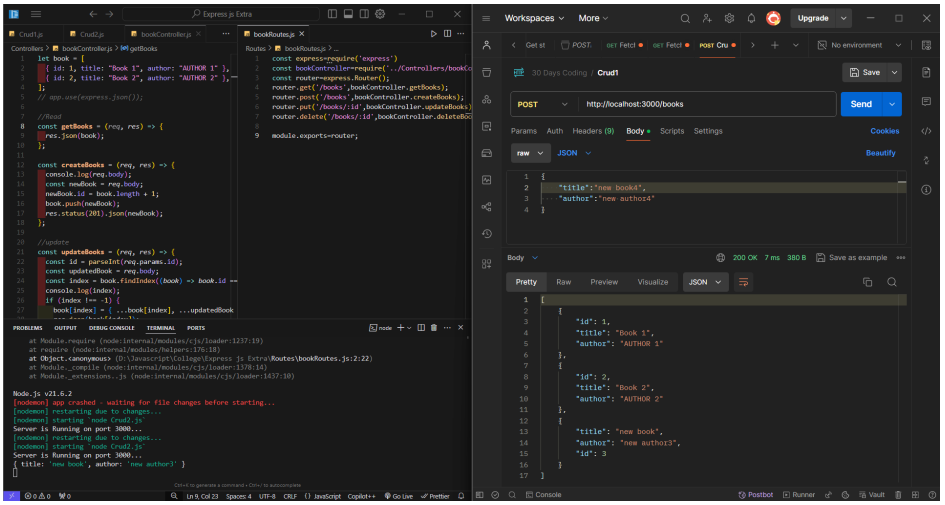
```
  updateBooks,
  deleteBooks,
};
```

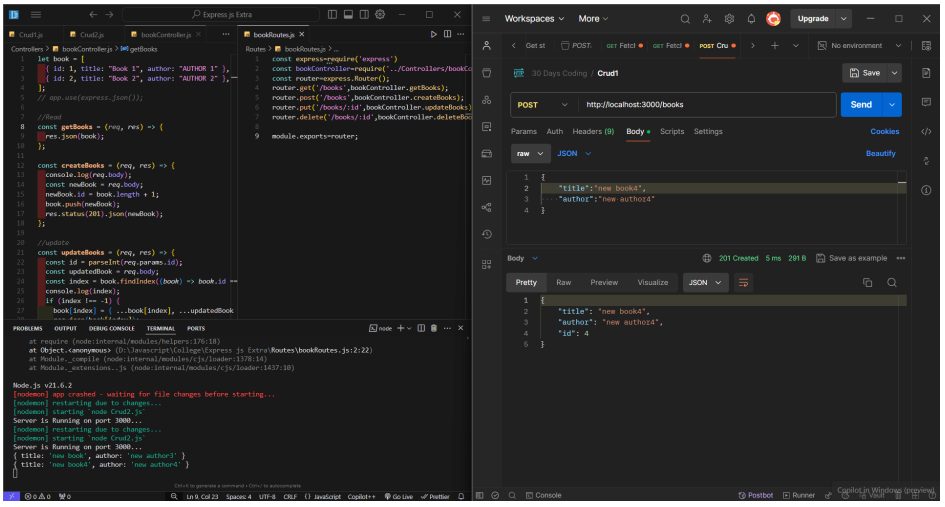
```
//bookRoutes.js
const express=require('express')
const bookController=require('../Controllers/bookController
const router=express.Router();
router.get('/books',bookController.getBooks);
router.post('/books',bookController.createBooks);
router.put('/books/:id',bookController.updateBooks);
router.delete('/books/:id',bookController.deleteBooks);

module.exports=router;
```

Get Books Data

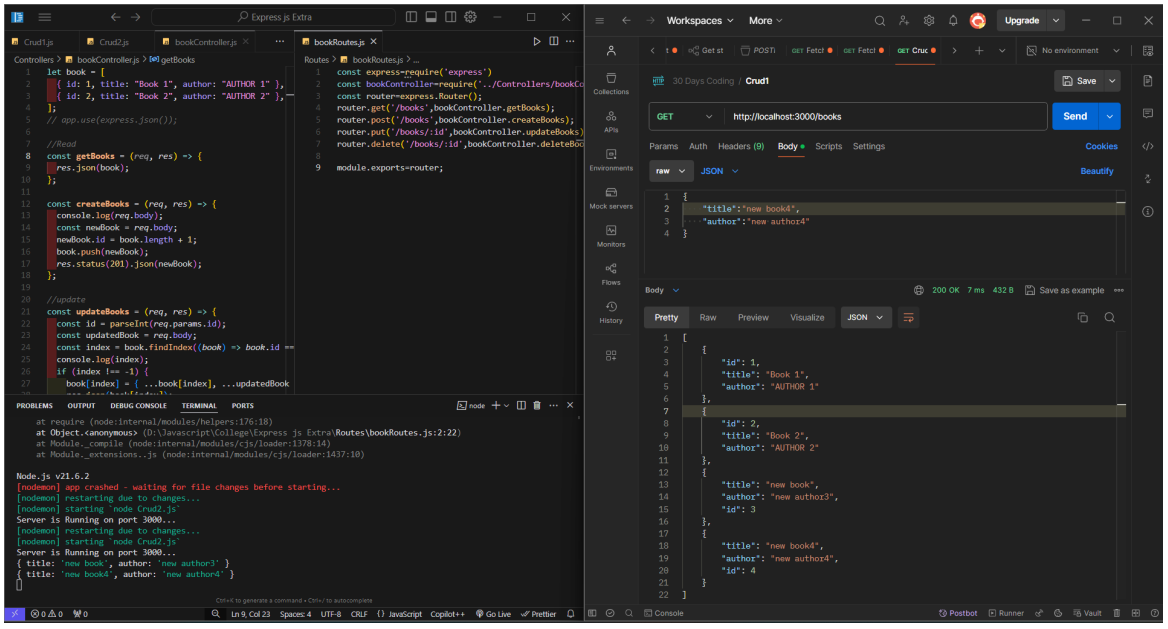


Add Data by Post

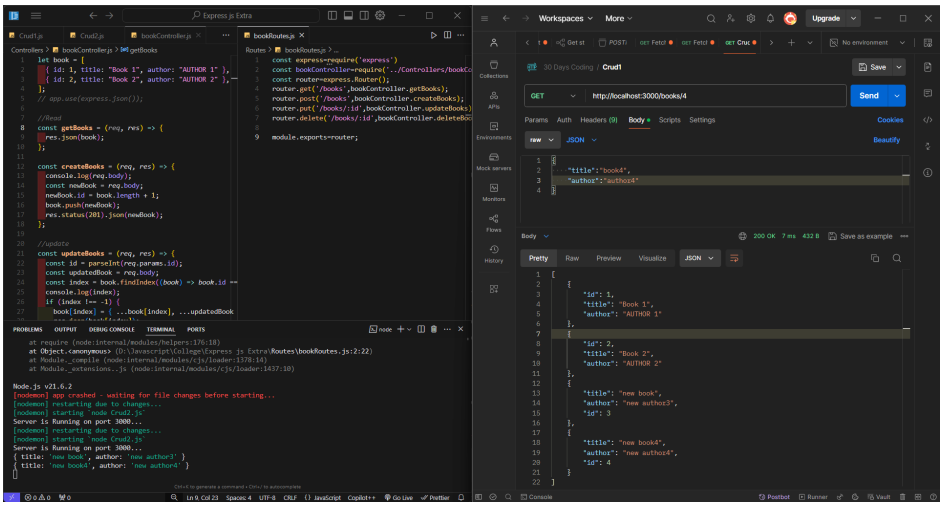


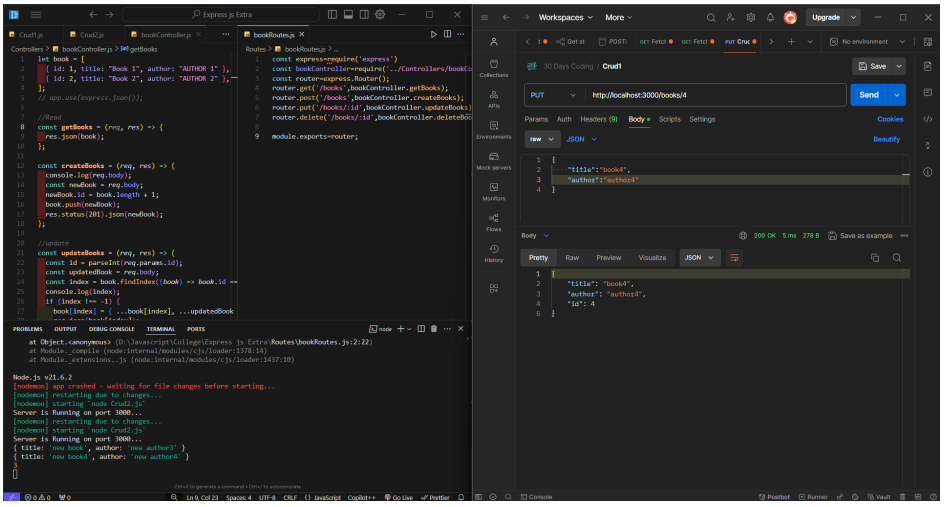


See Data is added or not ?

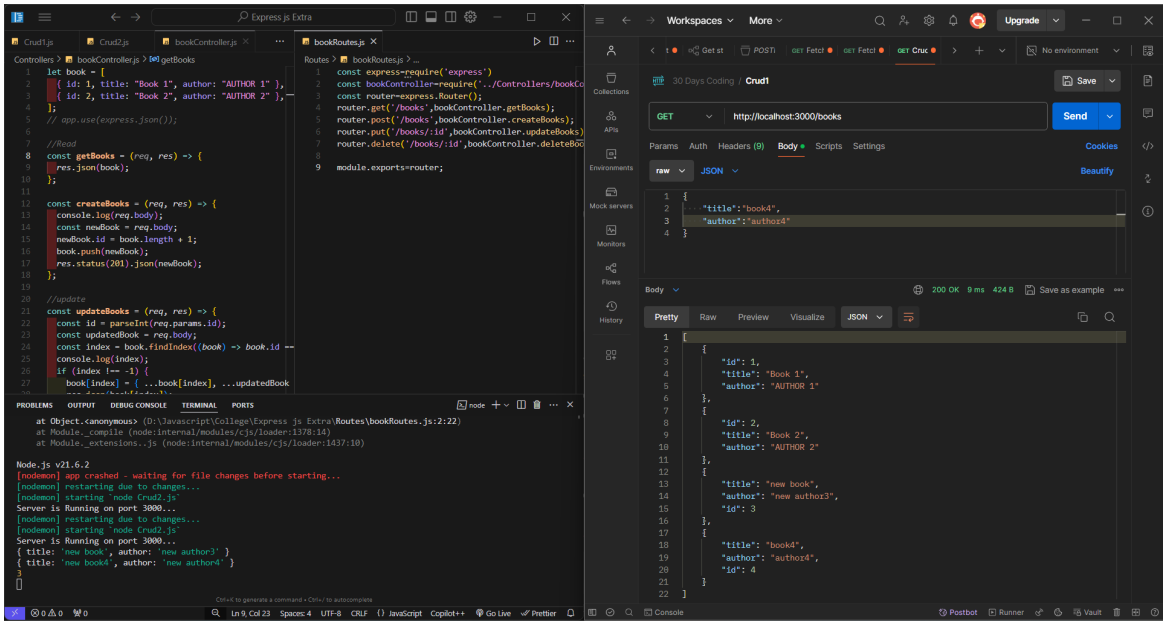


Update Data.

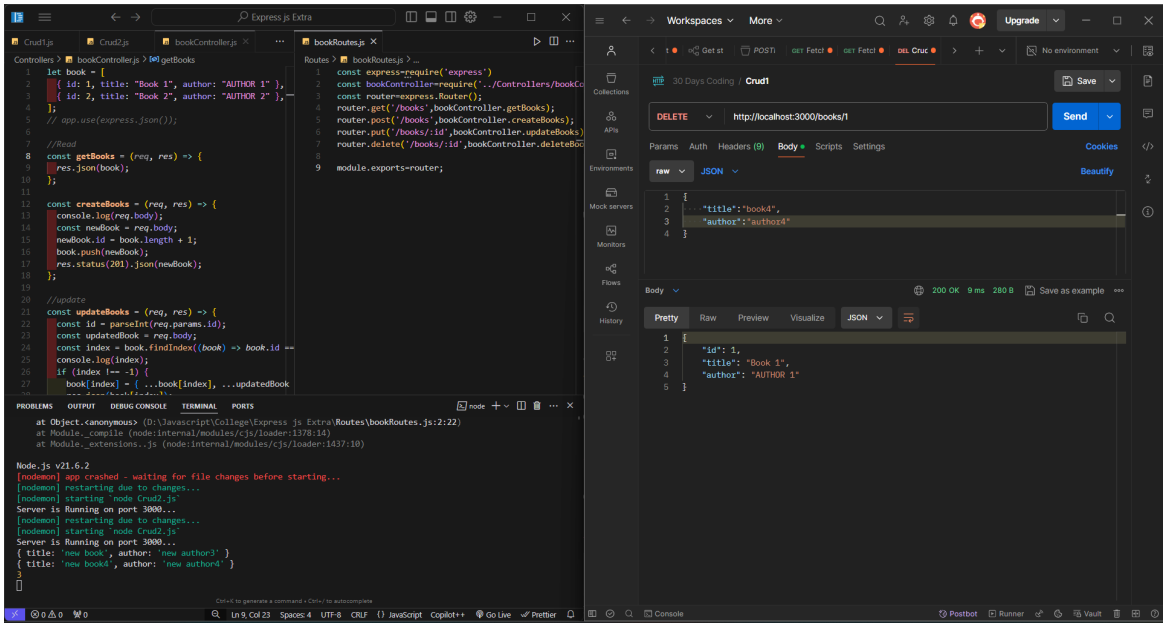




Check data is updated or not ?

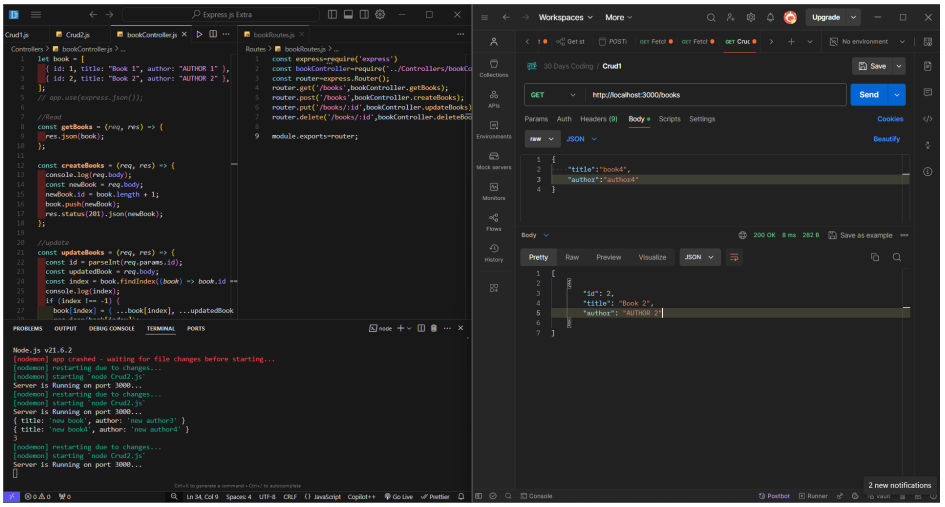


Delete data



See data is deleted or not ?

After saving file because of problem we have below output.



Certainly! Let's break down how your Express.js application is structured and how it works step by step. Your code is split into three main parts:

1. **Server setup (`Crud2.js`):**
   - Sets up the Express application.
   - Imports routes from `bookRoutes`.
   - Starts the server on port 3000.
2. **Controller logic (`bookController.js`):**
   - Contains the actual logic for handling CRUD operations (Create, Read, Update, Delete).
   - Defines functions for handling each operation on the `book` array.
3. **Routing (`bookRoutes.js`):**
   - Sets up the routes and connects them to the corresponding controller functions.

1. Server Setup (`Crud2.js`)

```
const express = require('express');
const app = express();
const bookRoutes = require('./Routes/bookRoutes');

app.use(express.json()); // Middleware to parse JSON bod
ies
app.use('/', bookRoutes); // Use the bookRoutes for all
paths
```


Khushal Savalakha

```
app.listen(3000, () => {
    console.log('Server is Running on port 3000...');
});
```

- `express.json()` : Middleware to parse incoming JSON request bodies.
- `app.use('/', bookRoutes)` : Tells the app to use the routes defined in `bookRoutes` for any requests.
- `app.listen(3000)` : Starts the server on port 3000 and logs a message when the server is running.

2. Controller Logic (`bookController.js`)

```
let book = [
    { id: 1, title: "Book 1", author: "AUTHOR 1" },
    { id: 2, title: "Book 2", author: "AUTHOR 2" },
];

const getBooks = (req, res) => {
    res.json(book);
};

const createBooks = (req, res) => {
    console.log(req.body);
    const newBook = req.body;
    newBook.id = book.length + 1;
    book.push(newBook);
    res.status(201).json(newBook);
};

const updateBooks = (req, res) => {
    const id = parseInt(req.params.id);
    const updatedBook = req.body;
    const index = book.findIndex((book) => book.id === i
d);
    console.log(index);
    if (index !== -1) {
        book[index] = { ...book[index], ...updatedBook
```


Khushal Savalakha

```
};
        res.json(book[index]);
    } else {
        res.status(404).json({ error: "book not found"
});
    }
};

const deleteBooks = (req, res) => {
    const id = req.params.id;
    const index = book.findIndex((book) => book.id == i
d);
    if (index != -1) {
        const deletedBook = book[index];
        book.splice(index, 1); // Correct the slice to s
plice
        res.json(deletedBook);
    } else {
        res.status(404).json({ error: "book not found"
});
    }
};

module.exports = {
    getBooks,
    createBooks,
    updateBooks,
    deleteBooks,
};
```

- `getBooks`: Sends the entire `book` array as a JSON response.
- `createBooks`: Adds a new book to the `book` array with a unique `id`, then responds with the newly created book and a 201 status code.
- `updateBooks`: Finds a book by `id`, updates it with the new data from the request body, and responds with the updated book. If the book is not found, it responds with a 404 status code.

Khushal Savalakha

- `deleteBooks`: Finds a book by `id`, removes it from the `book` array using `splice` (corrected from `slice`), and responds with the deleted book. If the book is not found, it responds with a 404 status code.

3. Routing (`bookRoutes.js`)

```
const express = require('express');
const bookController = require('../Controllers/bookContr
oller');
const router = express.Router();

router.get('/books', bookController.getBooks);
router.post('/books', bookController.createBooks);
router.put('/books/:id', bookController.updateBooks);
router.delete('/books/:id', bookController.deleteBooks);

module.exports = router;
```

- `express.Router()`: Creates a new router object to handle routes.
- **Routes**: Each route is linked to a corresponding controller function:
  - **GET** `/books`: Calls `bookController.getBooks`.
  - **POST** `/books`: Calls `bookController.createBooks`.
  - **PUT** `/books/:id`: Calls `bookController.updateBooks`.
  - **DELETE** `/books/:id`: Calls `bookController.deleteBooks`.

How It Works Together

1. **Server Setup**: When the server starts, it sets up middleware to parse JSON and uses the routes defined in `bookRoutes`.
2. **Routing**: When a request hits a specific route, it is handled by the corresponding function in `bookController`.
3. **Controller Logic**: The controller functions manage the `book` array, performing the appropriate CRUD operation and sending the appropriate response.

For example:

- A `GET` request to `/books` will call `getBooks`, which responds with the list of books.

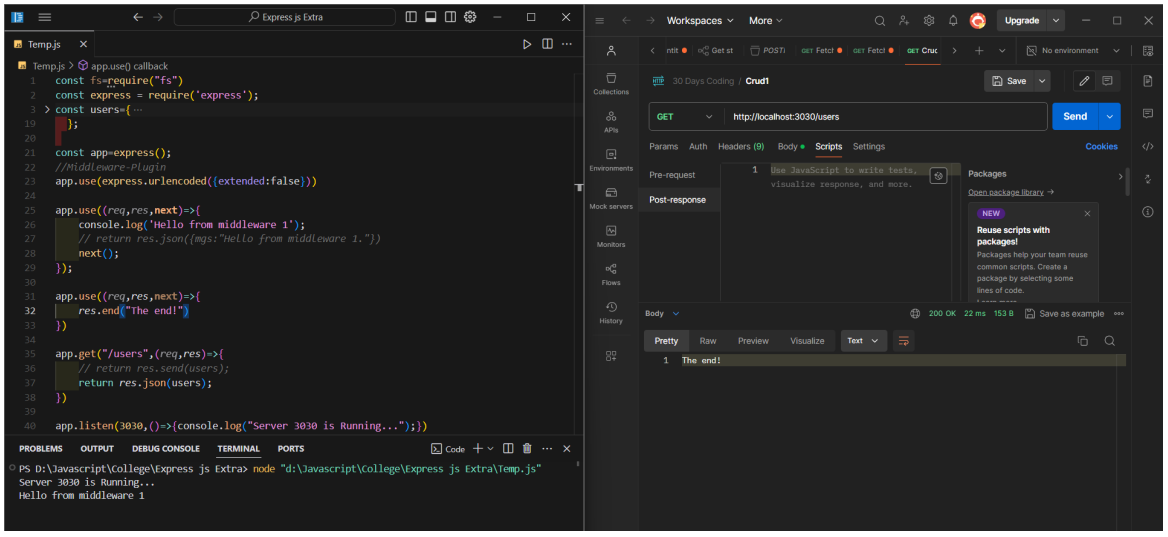
Khushal Savalakha

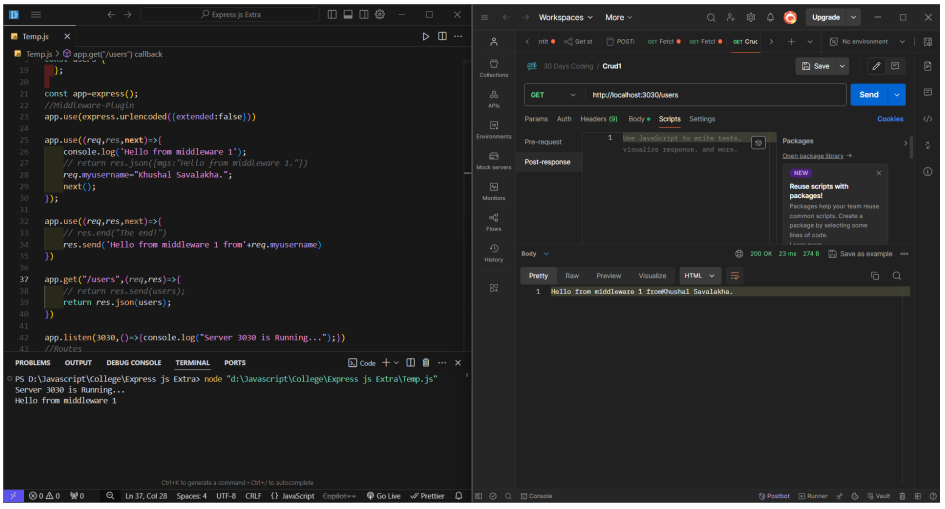
- A `POST` request to `/books` with a new book's data will call `createBooks`, which adds the new book to the list and responds with the created book.
- A `PUT` request to `/books/:id` with updated book data will call `updateBooks`, which updates the specified book and responds with the updated data.
- A `DELETE` request to `/books/:id` will call `deleteBooks`, which removes the specified book from the list and responds with the deleted book.

This modular structure makes the application easier to maintain and scale, as the routing logic is separated from the controller logic.

▼ Middleware

Here in below example we have rout pf users but we don't get it beacause of middleware.\





Khushal Savalakha

```
//Middleware-Plugin
app.use(express.urlencoded({extended:false}))

app.use((req,res,next)=>{
    console.log('Hello from middleware 1');
    // return res.json({msg:"Hello from middleware 1."})
    req.myusername="Khushal Savalakha.";
    next();
});

app.use((req,res,next)=>{
    // res.end("The end!")
    console.log('----------');
    console.log(req.body);
    res.send('Hello from middleware 1 from'+req.myusername)
})

app.get("/users",(req,res)=>{
    // return res.send(users);
    return res.json(users);
})

app.listen(3030,()=>{console.log("Server 3030 is Running...");})
//Routes
```

```
PS D:\Javascript\College\Express js Extra> node "d:\Javascript\College\Express js Extra\Temp.js"
Server 3030 is Running...
Hello from middleware 1
----------
{}
Hello from middleware 1
----------
{}
Hello from middleware 1
```

POST http://localhost:3030/users

```
{
    "title":"book4",
    "author":"author4"
}
```

```
Hello from middleware 1 from@Khushal Savalakha.
```

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

app.use((req,res,next)=>{
    // res.end("The end!")
    console.log('----------');
    console.log(req.body);
    // res.send('Hello from middleware 1 from'+req.myusername)
    fs.appendFile(
        "log.txt",`${Date.now()}:${req.method}:${req.path}`,
        (err,data)=>{
            next();
        }
    );
})

app.get("/users",(req,res)=>{
    // return res.send(users);
    return res.json(users);
})

app.listen(3030,()=>{console.log("Server 3030 is Running...");})
//Routes
```

```
log.txt
1    1717934426749:GET:/users
```

```
PS D:\Javascript\College\Express js Extra> node "d:\Javascript\College\Express js Extra\Temp.js"
Server 3030 is Running...
Hello from middleware 1
----------
{}
```

GET http://localhost:3030/users

```
[
    "name": "John Doe",
    "age": 30,
    "isStudent": false,
    "address": {
        "street": "123 Main St",
        "city": "Anytown",
        "state": "CA",
        "zip": "12345"
    },
    "courses": [
        "Math",
        "Science",
        "History"
    ],
    "scores": {
        "Math": 95,
        "Science": 88,
        "History": 76
    }
```

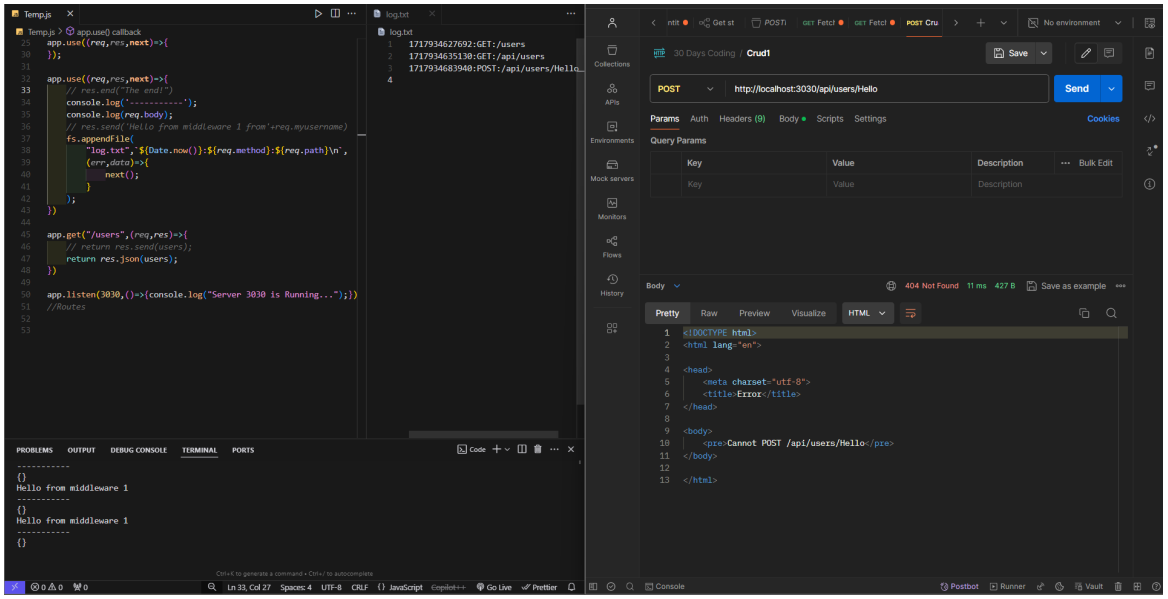
```
log.txt
1    1717934627692:GET:/users
1    1717934635130:GET:/api/users
3
```

```
PS D:\Javascript\College\Express js Extra> node "d:\Javascript\College\Express js Extra\Temp.js"
Server 3030 is Running...
Hello from middleware 1
----------
{}
Hello from middleware 1
----------
{}
```

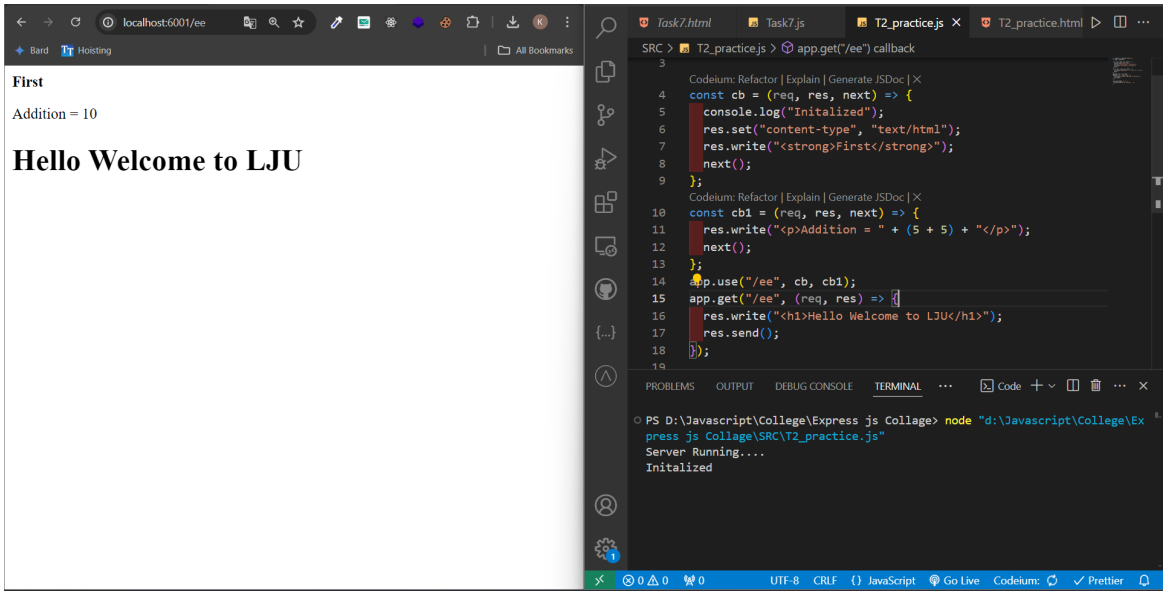
GET http://localhost:3030/api/users

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <title>Error</title>
</head>

<body>
    <pre>Cannot GET /api/users</pre>
</body>

</html>
```


Khushal Savalakha




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

const cb = (req, res, next) => {
  console.log("Initalized");
  res.set("content-type", "text/html");
  res.write("<strong>First</strong>");
  next();
};
const cb1 = (req, res, next) => {
  res.write("<p>Addition = " + (5 + 5) + "</p>");
  next();
};
app.use("/ee", cb, cb1);
app.get("/ee", (req, res) => {
  res.write("<h1>Hello Welcome to LJU</h1>");
  res.send();
});

app.listen(6001, () => {
  console.log("Server Running....");
})
```



explain how app.use(callback); works ?

`app.use(callback);` is a fundamental method in Express.js, a popular web application framework for Node.js. This method is used to add middleware functions to the request handling pipeline.

What is Middleware?

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. These functions can execute code, modify the request and response objects, end the request-response cycle, and call the next middleware function.

How `app.use(callback)` Works

When you call `app.use(callback)`, you're telling Express to use the `callback` function as middleware for all routes and HTTP methods. This middleware will be executed for every request to the server.

Here's a step-by-step explanation:

1. **Definition**: You define a middleware function.
2. **Usage**: You pass this middleware function to `app.use()`.
3. **Execution**: The middleware function is executed in the order it was added, whenever a request is received.

Example

Let's look at a simple example to illustrate how this works:

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

// Define a middleware function
function logRequest(req, res, next) {
  console.log(`${req.method} ${req.url}`);
  next(); // Pass control to the next middleware functio
n
}

// Use the middleware function
app.use(logRequest);

// Define a route
app.get('/', (req, res) => {
  res.send('Hello, world!');
});

// Start the server
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Explanation of the Example

1. **Middleware Function (`logRequest`)**:
   - This function logs the HTTP method and URL of the incoming request to the console.
   - It calls `next()` to pass control to the next middleware function or route handler.
2. **Using the Middleware**:
   - `app.use(logRequest);` tells Express to use `logRequest` for all incoming requests.
3. **Route Handler**:

Khushal Savalakha

- A route handler is defined for GET requests to the root path (`/`). It responds with "Hello, world!".

4. **Server Start**:
   - The server listens on port 3000.

Request Handling Flow

When a request is made to the server:

1. The middleware function `logRequest` is executed first, logging the request details.
2. Since `logRequest` calls `next()`, the control is passed to the next middleware function or route handler.
3. The route handler for the root path (`/`) sends the response "Hello, world!".

Multiple Middleware Functions

You can add multiple middleware functions, and they will be executed in the order they are defined:

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

function logRequest(req, res, next) {
  console.log(`${req.method} ${req.url}`);
  next();
}

function checkAuth(req, res, next) {
  // Imagine we check user authentication here
  console.log('Checking authentication');
  next();
}

app.use(logRequest);
app.use(checkAuth);

app.get('/', (req, res) => {
  res.send('Hello, authenticated user!');
});
```

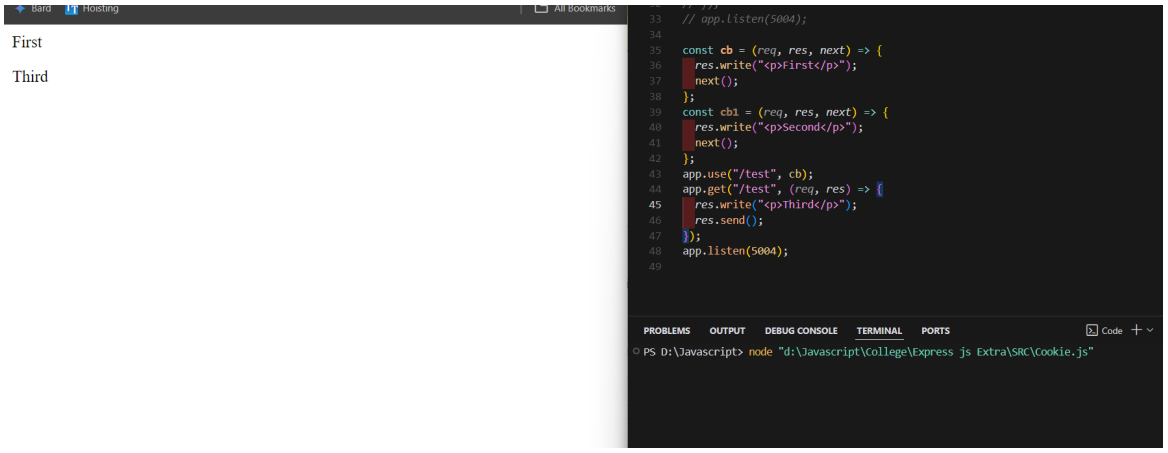

Khushal Savalakha

```
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

In this example, for every request:

1. `logRequest` logs the request details.
2. `checkAuth` logs the authentication check message.
3. The route handler sends the response.

By understanding `app.use(callback)`, you can better manage the flow of your Express application and implement functionality such as logging, authentication, and error handling in a clean and modular way.




```
 const express = require("express");
 const app = express();
const cb = (req, res, next) => {
  res.write("<p>First</p>");
  next();
};
const cb1 = (req, res, next) => {
  res.write("<p>Second</p>");
  next();
};
app.use("/test", cb);
app.get("/test", (req, res) => {
```

```
  res.write("<p>Third</p>");
  res.send();
});
app.listen(5004);
```

Let's go through your code step by step to understand how the middleware and callbacks work together in an Express.js application.

Code Explanation

Here is the code snippet provided:

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

const cb = (req, res, next) => {
    res.write("<p>First</p>");
    next();
};

const cb1 = (req, res, next) => {
    res.write("<p>Second</p>");
    next();
};

app.use("/test", cb);

app.get("/test", (req, res) => {
    res.write("<p>Third</p>");
    res.send();
});

app.listen(3000, () => {
    console.log("Server is running on port 3000");
});
```

Middleware and Callbacks Flow

1. **app.use("/test", cb);**

Khushal Savalakha

- This line registers a middleware function (`cb`) that will be executed for every request that matches the path `/test`.
- When a request to `/test` is made, `cb` is called.
- Inside `cb`, `res.write("<p>First</p>");` is executed, writing "First" to the response.
- Then, `next()` is called, which tells Express to move to the next middleware or route handler in line.

2. **app.get("/test", (req, res) => { ... });**
   - This defines a route handler for GET requests to `/test`.
   - Since the middleware (`cb`) called `next()`, this route handler will now be executed.
   - Inside this handler, `res.write("<p>Third</p>");` is executed, writing "Third" to the response.
   - Finally, `res.send();` is called, which ends the response. At this point, Express will send the accumulated response content to the client.

Middleware Flow:

Here's a detailed step-by-step breakdown of what happens when a request to `/test` is made:

1. **Request to `/test`**:
   - When a GET request to `/test` is made, Express first checks for any middleware registered with `app.use()` that matches the `/test` path.
2. **Execution of Middleware `cb`**:
   - The middleware function `cb` is executed.
   - `res.write("<p>First</p>");` writes the string `<p>First</p>` to the response buffer.
   - `next();` is called, which passes control to the next matching middleware or route handler.
3. **Route Handler Execution**:
   - After `cb` finishes execution and calls `next()`, Express looks for the next matching route handler, which is `app.get("/test", (req, res) => { ... })`.

Khushal Savalakha

- This route handler is executed.
- `res.write("<p>Third</p>");` writes the string `<p>Third</p>` to the response buffer.
- `res.send();` sends the response to the client and ends the response process. All the accumulated content ("First" and "Third") is sent to the client in one response.

Full Response Content

The final content sent to the client will be:

```
<p>First</p><p>Third</p>
```

Adding Another Middleware

If you want to add another middleware (`cb1`), you should register it either before or after `cb`, depending on your requirements. For instance:

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

const cb = (req, res, next) => {
    res.write("<p>First</p>");
    next();
};

const cb1 = (req, res, next) => {
    res.write("<p>Second</p>");
    next();
};

app.use("/test", cb);
app.use("/test", cb1);

app.get("/test", (req, res) => {
    res.write("<p>Third</p>");
    res.send();
});
```


Khushal Savalakha

```
app.listen(3000, () => {
    console.log("Server is running on port 3000");
});
```

Middleware Flow with `cb1`:

1. **Request to `/test`**:
   - When a GET request to `/test` is made, Express first checks for any middleware registered with `app.use()` that matches the `/test` path.
2. **Execution of Middleware `cb`**:
   - `cb` is executed, writes `<p>First</p>` to the response buffer, and calls `next()`.
3. **Execution of Middleware `cb1`**:
   - `cb1` is executed next, writes `<p>Second</p>` to the response buffer, and calls `next()`.
4. **Route Handler Execution**:
   - The route handler for `app.get("/test", (req, res) => { ... })` is executed.
   - `res.write("<p>Third</p>");` writes `<p>Third</p>` to the response buffer.
   - `res.send();` sends the response to the client and ends the response process.

Full Response Content with `cb1`

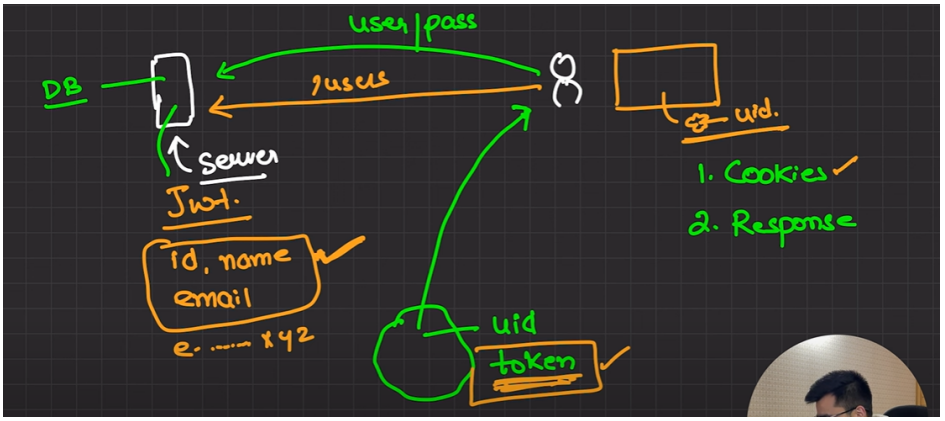
The final content sent to the client will be:

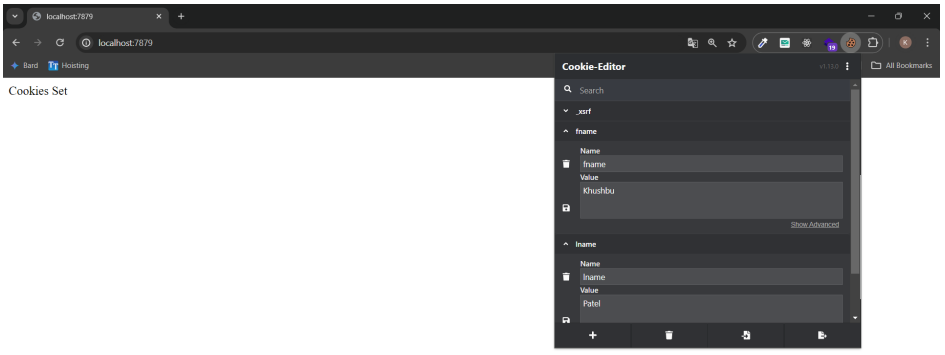
```
<p>First</p><p>Second</p><p>Third</p>
```

This demonstrates how multiple middleware functions can be chained together, and how each middleware function can pass control to the next one using `next()`.

▼ Cookies

Khushal Savalakha





> 💡 it means 1 cookie has data received by server like session expiration details and login details and other details when a particular session expire then automatically logged out from the particular site if session is not ended then how many times we load the site it does not making a connection for login am i true ?

Yes, you're correct in understanding the role of cookies in maintaining sessions and user login states on websites. Here's a more detailed explanation:

1. Session Management with Cookies

- **Session Persistence:** When a user logs into a website, a session is established on the server. A session identifier (typically stored in a cookie) is sent back to the browser.
- **Automatic Login:** As long as the session is active (not expired or explicitly logged out), subsequent requests from the same browser include this session identifier. This allows the server to recognize the user without requiring them to log in again.
- **Stateful Interaction:** Cookies help maintain stateful interaction between the user and the server across multiple requests. This includes remembering user preferences, session timeouts, and other session-specific data.

2. Handling Session Expiration

- **Session Expiry:** Sessions can have a defined expiration time. Once a session expires (due to inactivity or based on server settings), the session identifier stored in the cookie becomes invalid.
- **Automatic Logout:** When a session expires, subsequent requests with the expired session identifier will be rejected or result in a logout action, depending on how the server is configured.
- **User Experience:** This mechanism ensures that users are automatically logged out after a certain period of inactivity, enhancing security and protecting user accounts from unauthorized access.

Example Scenario:

- **Initial Login:** User logs into a website, and a session cookie is set with an expiration time.
- **Subsequent Requests:** As the user navigates the site or performs actions, the session cookie is sent with each request, identifying the user's session.
- **Session Expiry:** If the user is inactive for a specified period (e.g., 30 minutes), the session expires.
- **Logout Handling:** When the session expires, subsequent requests will require the user to log in again, ensuring secure access to sensitive areas of the website.

Benefits of Using Cookies for Session Management:

- **Convenience:** Users stay logged in across different pages of the website without having to re-enter credentials repeatedly.
- **Security:** Automatic session expiration reduces the risk of unauthorized access if a user leaves their session unattended.
- **Customization:** Websites can personalize content and experiences based on user preferences stored in cookies, enhancing usability.

In summary, cookies play a crucial role in maintaining user sessions and login states on websites, providing a seamless and secure browsing experience while enhancing user engagement and security measures.

▼ Session

The error "Cannot set headers after they are sent to the client" occurs because the response is being sent multiple times within the same request handler. In your

Khushal Savalakha

`/savesession` route, you are writing to the response and then attempting to redirect, which causes the headers to be sent twice.

To fix this, you should avoid writing to the response before the `res.redirect` call. Instead, store session data and redirect immediately. The session data can then be displayed in the `/fetchsession` route.

Here's the corrected version of your code:

```
var expr = require("express");
var path = require('path');
var app = expr();
var es = require("express-session");

app.use(es({
    resave: false,
    saveUninitialized: false,
    secret: "mykey1"
}));

app.use(expr.static(path.join(__dirname, '../Public'), {
index: 'Session.html' }));

app.get("/savesession", (req, res) => {
    var uname = req.query.uname;
    var password = req.query.password;
    req.session.uname = uname;
    req.session.password = password;
    res.redirect("fetchsession");
});

app.get("/fetchsession", (req, res) => {
    res.write("<h1>Welcome " + req.session.uname + "</h1
>");
    res.write("<h2>Session Data:</h2>");
    res.write("<ul>");

    for (const key in req.session) {
        if (req.session.hasOwnProperty(key)) {
            res.write("<li>" + key + ": " + req.session
```


Khushal Savalakha

```
[key] + "</li>");
        }
    }

    res.write("</ul>");
    res.write("<a href='/deletesession'>Delete Session</
a>");
    res.end();
});

app.get("/deletesession", (req, res) => {
    req.session.destroy((err) => {
        if (err) {
            return res.status(500).send("Failed to delet
e session.");
        }
        res.redirect('/');
    });
});

app.listen(6177, () => {
    console.log("Server is listening on port 6177");
});
```

Explanation:

1. **Initialization and Static File Serving**: No changes needed here.
2. **Save Session**: The `/savesession` route now only saves the session data and immediately redirects to `/fetchsession`.
3. **Fetch Session**: The `/fetchsession` route retrieves and displays all session data.
4. **Delete Session**: The `/deletesession` route destroys the session and redirects to the home page, handling any potential errors.

By ensuring that you do not write to the response and then attempt to redirect in the same request handler, you avoid the "Cannot set headers after they are sent to the client" error.

Khushal Savalakha

💡 it means we can not write res.write() before res.redirect() am i true ?

Yes, you are correct. In Express.js, once you start sending the response with `res.write()`, you cannot use methods like `res.redirect()` that set headers, because the headers are already sent. Here's a more detailed explanation:

Response Handling in Express.js

1. **Headers and Body**:
   - **Headers**: Information sent at the beginning of an HTTP response. They include status codes, content type, cookies, etc.
   - **Body**: The actual content of the response (HTML, JSON, text, etc.).
2. **Sending Headers and Body**:
   - When you call `res.write()`, you start sending the body of the response. This implicitly sends the headers if they haven't been sent already.
   - Methods like `res.redirect()`, `res.send()`, and `res.end()` are used to send headers and body, or to end the response. These methods should not be called after you've started sending the response body with `res.write()`.

Practical Example:

If you try to use `res.write()` and then `res.redirect()` within the same request handler, it will cause an error because `res.redirect()` tries to set the headers after the body has already started being sent.

Correct Approach:

1. **Set Session Data**:
   - Store session data.
   - Redirect to another route.
2. **Display Session Data**:
   - Retrieve and display session data in a separate route.

Here's an updated and clear example:

Khushal Savalakha

```
var expr = require("express");
var path = require('path');
var app = expr();
var es = require("express-session");

app.use(es({
    resave: false,
    saveUninitialized: false,
    secret: "mykey1"
}));

app.use(expr.static(path.join(__dirname, '../Public'), {
index: 'Session.html' }));

app.get("/savesession", (req, res) => {
    var uname = req.query.uname;
    var password = req.query.password;
    req.session.uname = uname;
    req.session.password = password;
    res.redirect("fetchsession");
});

app.get("/fetchsession", (req, res) => {
    res.write("<h1>Welcome " + req.session.uname + "</h1
>");
    res.write("<h2>Session Data:</h2>");
    res.write("<ul>");

    for (const key in req.session) {
        if (req.session.hasOwnProperty(key)) {
            res.write("<li>" + key + ": " + req.session
[key] + "</li>");
        }
    }

    res.write("</ul>");
    res.write("<a href='/deletesession'>Delete Session</
a>");
```


Khushal Savalakha

```
    res.end();  // End the response to ensure no further headers or data is sent
});

app.get("/deletesession", (req, res) => {
    req.session.destroy((err) => {
        if (err) {
            return res.status(500).send("Failed to delete session.");
        }
        res.redirect('/');
    });
});

app.listen(6177, () => {
    console.log("Server is listening on port 6177");
});
```

Key Points:

- **Store Session Data**: In `/savesession`, store session data and redirect to `/fetchsession`.
- **Retrieve and Display Session Data**: In `/fetchsession`, retrieve session data and display it. Ensure you use `res.end()` after `res.write()` to properly end the response.
- **Delete Session**: In `/deletesession`, destroy the session and handle potential errors before redirecting.

This ensures proper response handling and avoids the "Cannot set headers after they are sent to the client" error.

```
var expr = require("express");
var path = require('path');
var app = expr();
var es = require("express-session");

app.use(es({
    resave: false,
```


Khushal Savalakha

```
    saveUninitialized: false,
    secret: "mykey1"
}));

app.use(expr.static(path.join(__dirname, '../Public'), { ir

app.get("/savesession", (req, res) => {
    var uname = req.query.uname;
    var password = req.query.password;
    req.session.uname = uname;
    req.session.password = password;
    res.redirect("fetchsession");
});

app.get("/fetchsession", (req, res) => {
    res.write("<h1>Welcome " + req.session.uname + "</h1>")
    res.write("<h2>Session Data:</h2>");
    res.write("<ul>");

    for (const key in req.session) {
        if (req.session.hasOwnProperty(key)) {
            res.write("<li>" + key + ": " + req.session[key
        }
    }

    res.write("</ul>");
    res.write("<a href='/deletesession'>Delete Session</a>"
    res.end();
});

app.get("/deletesession", (req, res) => {
    req.session.destroy((err) => {
        if (err) {
            return res.status(500).send("Failed to delete s
        }
        res.redirect('/');
    });
});
```


Khushal Savalakha

```
app.listen(6177, () => {
    console.log("Server is listening on port 6177");
});
```

To set a session ending time (or session expiration) in an Express.js application using the `express-session` middleware, you can configure the `cookie.maxAge` property. This property specifies the duration for which the session cookie should be valid. After this duration, the session will automatically expire.

Here's an example of how to set the session expiration time:

1. **Install dependencies** (if you haven't already):

   ```
   npm install express express-session
   ```

2. **Configure session with expiration**:

   ```
   var express = require("express");
   var path = require('path');
   var app = express();
   var session = require("express-session");

   app.use(session({
       resave: false,
       saveUninitialized: false,
       secret: "mykey1",
       cookie: {
           maxAge: 60000 // 1 minute (in milliseconds)
       }
   }));

   app.use(express.static(path.join(__dirname, '../Publi
   c'), { index: 'Session.html' }));

   app.get("/savesession", (req, res) => {
       var uname = req.query.uname;
       var password = req.query.password;
       req.session.uname = uname;
       req.session.password = password;
   ```


Khushal Savalakha

```
    res.redirect("fetchsession");
});

app.get("/fetchsession", (req, res) => {
    if (req.session.uname) {
        res.write("<h1>Welcome " + req.session.uname
+ "</h1>");
        res.write("<h2>Session Data:</h2>");
        res.write("<ul>");

        for (const key in req.session) {
            if (req.session.hasOwnProperty(key)) {
                res.write("<li>" + key + ": " + req.s
ession[key] + "</li>");
            }
        }

        res.write("</ul>");
        res.write("<a href='/deletesession'>Delete Se
ssion</a>");
        res.end();
    } else {
        res.redirect('/');
    }
});

app.get("/deletesession", (req, res) => {
    req.session.destroy((err) => {
        if (err) {
            return res.status(500).send("Failed to de
lete session.");
        }
        res.redirect('/');
    });
});

app.listen(6177, () => {
```


Khushal Savalakha

```
    console.log("Server is listening on port 6177");
});
```

Explanation:

1. **Session Configuration**:
   - `resave: false`: Prevents the session from being saved back to the session store if it hasn't been modified.
   - `saveUninitialized: false`: Ensures that an uninitialized session is not saved to the session store.
   - `secret`: A secret key used to sign the session ID cookie.
   - `cookie: { maxAge: 60000 }`: Sets the session cookie's max age to 1 minute (60000 milliseconds).
2. **Session Handling**:
   - **Save Session**: In the `/savesession` route, the username and password are stored in the session, and the user is redirected to `/fetchsession`.
   - **Fetch Session**: In the `/fetchsession` route, the session data is displayed if it exists. If the session has expired or does not exist, the user is redirected to the home page (`/`).
   - **Delete Session**: In the `/deletesession` route, the session is destroyed and the user is redirected to the home page.

Custom Session Expiration Logic:

If you need more control over session expiration (e.g., based on user activity), you can implement custom logic by manually setting and checking timestamps within the session object.

Here's an example with custom logic:

1. **Set custom expiration**:

```
var express = require("express");
var path = require('path');
var app = express();
var session = require("express-session");
```


Khushal Savalakha

```
app.use(session({
    resave: false,
    saveUninitialized: false,
    secret: "mykey1"
}));

app.use(express.static(path.join(__dirname, '../Publi
c'), { index: 'Session.html' }));

app.get("/savesession", (req, res) => {
    var uname = req.query.uname;
    var password = req.query.password;
    req.session.uname = uname;
    req.session.password = password;
    req.session.startTime = Date.now();
    req.session.maxAge = 60000; // 1 minute
    res.redirect("fetchsession");
});

app.get("/fetchsession", (req, res) => {
    if (req.session.uname && (Date.now() - req.sessio
n.startTime) < req.session.maxAge) {
        res.write("<h1>Welcome " + req.session.uname
+ "</h1>");
        res.write("<h2>Session Data:</h2>");
        res.write("<ul>");

        for (const key in req.session) {
            if (req.session.hasOwnProperty(key)) {
                res.write("<li>" + key + ": " + req.s
ession[key] + "</li>");
            }
        }

        res.write("</ul>");
        res.write("<a href='/deletesession'>Delete Se
ssion</a>");
        res.end();
```


Khushal Savalakha

```
    } else {
        req.session.destroy(); // destroy expired ses
sion
        res.redirect('/');
    }
});

app.get("/deletesession", (req, res) => {
    req.session.destroy((err) => {
        if (err) {
            return res.status(500).send("Failed to de
lete session.");
        }
        res.redirect('/');
    });
});

app.listen(6177, () => {
    console.log("Server is listening on port 6177");
});
```

Custom Logic Explanation:

1. **Session Start Time and Max Age**:
   - `req.session.startTime`: Stores the timestamp when the session is created.
   - `req.session.maxAge`: Stores the max age for the session.
2. **Check Expiration**:
   - In the `/fetchsession` route, the session is checked for expiration by comparing the current time with `req.session.startTime` and `req.session.maxAge`.
   - If the session has expired, it is destroyed, and the user is redirected to the home page.

This approach provides flexibility to implement custom session expiration logic based on your application's needs.

▼ Cookies and Session

Khushal Savalakha

💡 can we use cookies and session simontaneously ?if it is then give me it's example.

Yes, you can use cookies and sessions simultaneously in an Express.js application. Cookies and sessions serve different purposes and can complement each other. Cookies are used to store data on the client's browser, while sessions store data on the server and use a cookie to track the session ID.

Here's an example of how to use both cookies and sessions in an Express.js application:

Example:

1. **Install the necessary packages**:

```
npm install express express-session cookie-parser
```

1. **Create the server with cookies and sessions**:

```
const express = require('express');
const session = require('express-session');
const cookieParser = require('cookie-parser');
const app = express();

app.use(cookieParser());

// Configure session middleware
app.use(session({
    secret: 'mySecretKey',
    resave: false,
    saveUninitialized: true,
    cookie: { maxAge: 60000 } // Session will expire aft
er 1 minute
}));

// Route to set cookies and session data
app.get('/set', (req, res) => {
    // Set cookies
    res.cookie('username', 'john_doe', { maxAge: 60000
```


Khushal Savalakha

```
}); // Cookie expires in 1 minute
    res.cookie('email', 'john_doe@example.com', { maxAg
e: 60000 });

    // Set session data
    req.session.user = {
        username: 'john_doe',
        email: 'john_doe@example.com'
    };

    res.send('Cookies and session data are set');
});

// Route to get cookies and session data
app.get('/get', (req, res) => {
    // Get cookies
    const cookies = req.cookies;

    // Get session data
    const sessionData = req.session.user;

    res.json({
        cookies: cookies,
        sessionData: sessionData
    });
});

// Route to clear cookies and session data
app.get('/clear', (req, res) => {
    // Clear cookies
    res.clearCookie('username');
    res.clearCookie('email');

    // Destroy session
    req.session.destroy(err => {
        if (err) {
            return res.status(500).send('Failed to destr
oy session');
```


Khushal Savalakha

```
        }
        res.send('Cookies and session data are cleare
d');
    });
});

// Start server
app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

Explanation:

1. **Dependencies**:
   - `express`: The web framework for Node.js.
   - `express-session`: Middleware to handle sessions.
   - `cookie-parser`: Middleware to handle cookies.
2. **Middleware Setup**:
   - `app.use(cookieParser());`: This parses cookies attached to the client request object.
   - `app.use(session({...}));`: This sets up session management with various options, such as the secret key, session expiration, and other configurations.
3. **Routes**:
   - `/set`: This route sets a cookie and session data when accessed.
     - `res.cookie('username', 'john_doe', { maxAge: 60000 });`: Sets a cookie named `username` that expires in 1 minute.
     - `req.session.user = {...};`: Sets session data for the user.
   - `/get`: This route retrieves and displays the stored cookies and session data.
   - `/clear`: This route clears the cookies and destroys the session.
     - `res.clearCookie('username');`: Clears the `username` cookie.
     - `req.session.destroy(...);`: Destroys the session.

Running the Example:

Khushal Savalakha

1. **Start the Server**: Run the server by executing the script:

```
node your_script_name.js
```

2. **Access the Routes**:
   - Visit `http://localhost:3000/set` to set the cookies and session data.
   - Visit `http://localhost:3000/get` to retrieve and display the cookies and session data.
   - Visit `http://localhost:3000/clear` to clear the cookies and session data.

Using this setup, you can manage both cookies and sessions in your Express.js application, leveraging the strengths of each for various tasks such as authentication, user preferences, and more.

> 💡 here by cookie: { maxAge: 60000 } // Session will expire after 1 minute this session will be expire or cookies i don't understand this give me proper example by which i can understand all concept.

Sure, let's break it down and provide a clear example to understand the concepts of cookies and sessions, especially focusing on their expiration.

Key Concepts:

1. **Cookies**:
   - Stored on the client's browser.
   - Can have an expiration date (`expires`) or a duration (`maxAge`) after which they expire.
2. **Sessions**:
   - Stored on the server.
   - Associated with a unique session ID stored in a cookie on the client's browser.
   - Can also have an expiration time, after which the session data is no longer valid.

Example:

Let's create a more detailed example to illustrate how cookies and sessions work together, and how their expiration times are set and managed.

Khushal Savalakha

1. **Install the necessary packages**:

```
npm install express express-session cookie-parser
```

2. **Create the server with cookies and sessions**:

```
const express = require('express');
const session = require('express-session');
const cookieParser = require('cookie-parser');
const app = express();

app.use(cookieParser());

// Configure session middleware
app.use(session({
    secret: 'mySecretKey',
    resave: false,
    saveUninitialized: true,
    cookie: { maxAge: 60000 } // Session cookie will
expire after 1 minute (60000 ms)
}));

// Route to set cookies and session data
app.get('/set', (req, res) => {
    // Set cookies
    res.cookie('username', 'john_doe', { maxAge: 6000
0 }); // Cookie expires in 1 minute
    res.cookie('email', 'john_doe@example.com', { max
Age: 60000 });

    // Set session data
    req.session.user = {
        username: 'john_doe',
        email: 'john_doe@example.com'
    };

    res.send('Cookies and session data are set');
});
```


Khushal Savalakha

```
// Route to get cookies and session data
app.get('/get', (req, res) => {
    // Get cookies
    const cookies = req.cookies;

    // Get session data
    const sessionData = req.session.user;

    res.json({
        cookies: cookies,
        sessionData: sessionData
    });
});

// Route to clear cookies and session data
app.get('/clear', (req, res) => {
    // Clear cookies
    res.clearCookie('username');
    res.clearCookie('email');

    // Destroy session
    req.session.destroy(err => {
        if (err) {
            return res.status(500).send('Failed to de
stroy session');
        }
        res.send('Cookies and session data are cleare
d');
    });
});

// Start server
app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

Explanation:

Khushal Savalakha

1. **Cookies**:
   - In the `/set` route, two cookies (`username` and `email`) are set with a `maxAge` of 60000 milliseconds (1 minute). This means these cookies will expire 1 minute after they are set.
   - When the `/get` route is accessed, the cookies are read from the request (`req.cookies`) and sent back in the response.
2. **Sessions**:
   - The `session` middleware is configured with a `cookie` property that has a `maxAge` of 60000 milliseconds (1 minute). This means the session cookie (which contains the session ID) will expire after 1 minute.
   - In the `/set` route, session data (`req.session.user`) is set.
   - When the `/get` route is accessed, the session data is read from the request (`req.session.user`) and sent back in the response.
   - The session data itself is stored on the server, and the session ID is stored in a cookie on the client's browser.
3. **Clearing Data**:
   - The `/clear` route demonstrates how to clear cookies (`res.clearCookie`) and destroy the session (`req.session.destroy`).

Testing the Example:

1. **Start the Server**:

   ```
   node your_script_name.js
   ```

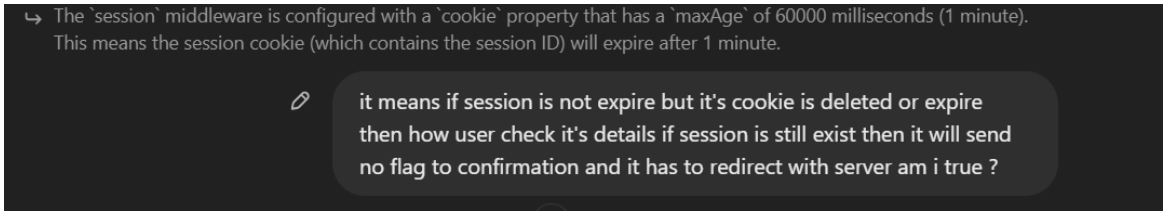
2. **Set Cookies and Session Data**:
   Visit `http://localhost:3000/set` in your browser. This will set the cookies and session data.
3. **Get Cookies and Session Data**:
   Visit `http://localhost:3000/get` in your browser. This will show the cookies and session data that were set.
4. **Wait for Expiry**:
   Wait for 1 minute and visit

Khushal Savalakha

`http://localhost:3000/get` again. The session data should be gone because the session cookie has expired, and the cookies should also be expired.

5. **Clear Data Manually**:
   Visit
   `http://localhost:3000/clear` to manually clear the cookies and session data.

By using this example, you can see how cookies and sessions are used together in an Express.js application, and how their expiration times are managed separately.



Yes, you are correct. If the session is still valid on the server but the session cookie (which contains the session ID) is deleted or expires on the client side, the client will no longer have a way to reference the existing session. This effectively means that the client loses access to the session data, even though it still exists on the server.

Detailed Explanation

1. **Session Cookie Expiration**:
   - The session middleware uses a cookie (usually named `connect.sid` by default) to store the session ID on the client side.
   - If this cookie expires or is deleted, the client loses the ability to send the session ID with subsequent requests.
2. **Impact on Session Data**:
   - The session data still exists on the server as long as the session itself has not expired or been destroyed.
   - However, without the session ID cookie, the server cannot link incoming requests to the existing session data.
3. **Behavior on Subsequent Requests**:
   - When the session cookie is missing or expired, the server treats incoming requests as if they are from a new client.
   - A new session will be created for the client, and the existing session data on the server will remain inaccessible unless the client somehow provides the correct

Khushal Savalakha

session ID.

Example Scenario

Let's revisit the example with some additional logging to demonstrate what happens when the session cookie expires:

```
const express = require('express');
const session = require('express-session');
const cookieParser = require('cookie-parser');
const app = express();

app.use(cookieParser());

// Configure session middleware
app.use(session({
    secret: 'mySecretKey',
    resave: false,
    saveUninitialized: true,
    cookie: { maxAge: 60000 } // Session cookie will exp
ire after 1 minute (60000 ms)
}));

// Route to set cookies and session data
app.get('/set', (req, res) => {
    // Set cookies
    res.cookie('username', 'john_doe', { maxAge: 60000
}); // Cookie expires in 1 minute
    res.cookie('email', 'john_doe@example.com', { maxAg
e: 60000 });

    // Set session data
    req.session.user = {
        username: 'john_doe',
        email: 'john_doe@example.com'
    };

    console.log('Session ID:', req.sessionID);
    console.log('Session Data:', req.session.user);
```


Khushal Savalakha

```
    res.send('Cookies and session data are set');
});

// Route to get cookies and session data
app.get('/get', (req, res) => {
    // Get cookies
    const cookies = req.cookies;

    // Get session data
    const sessionData = req.session.user;

    console.log('Session ID:', req.sessionID);
    console.log('Session Data:', sessionData);

    if (!sessionData) {
        return res.status(403).send('Session has expired
or does not exist.');
    }

    res.json({
        cookies: cookies,
        sessionData: sessionData
    });
});

// Route to clear cookies and session data
app.get('/clear', (req, res) => {
    // Clear cookies
    res.clearCookie('username');
    res.clearCookie('email');

    // Destroy session
    req.session.destroy(err => {
        if (err) {
            return res.status(500).send('Failed to destr
oy session');
        }
```


Khushal Savalakha

```
        res.send('Cookies and session data are cleare
d');
    });
});

// Start server
app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

Testing the Example

1. **Start the Server**:

   ```
   node your_script_name.js
   ```

2. **Set Cookies and Session Data**:
   Visit
   `http://localhost:3000/set` in your browser. This will set the cookies and session data.

3. **Get Cookies and Session Data**:
   Visit
   `http://localhost:3000/get` in your browser. This will show the cookies and session data that were set.

4. **Wait for Session Cookie to Expire**:
   Wait for 1 minute and visit
   `http://localhost:3000/get` again. You will see a message indicating that the session has expired or does not exist, because the session cookie has expired.

Summary

- **Session Persistence**: The session data persists on the server as long as the session itself is valid.
- **Session Cookie**: The session ID stored in the cookie is essential for the client to access the session data. If the session cookie expires or is deleted, the client loses access to the session data.
- **Redirects or Errors**: In cases where the session cookie is missing or expired, the server cannot identify the session. You might need to handle this scenario by

Khushal Savalakha

redirecting the user to a login page or showing an appropriate error message.

▼ Responsing MCQ

```
const expr=require("express");
const app=expr();
app.get("/",(req,res)=>
{
res.set("content-type","text/html");
res.write("Hello world!");
res.send("<h1> Hello from LJ</h1>");
});
app.listen(5004);


const express = require("express");
const app = express();
var st = { user: 2 };
app.get("/", function (req, res) {
res.write("Hello");
res.json(st.user);
});
app.listen(5004);
i don's understand why 1 gives out put with error and other
```

Let's analyze the two code snippets you provided to understand why one gives an output with an error and the other gives only an error.

First Code Snippet

```
const expr = require("express");
const app = expr();
app.get("/", (req, res) => {
    res.set("content-type", "text/html");
    res.write("Hello world!");
    res.send("<h1> Hello from LJ</h1>");
});
app.listen(5004);
```


Khushal Savalakha

Second Code Snippet

```
const express = require("express");
const app = express();
var st = { user: 2 };
app.get("/", function (req, res) {
    res.write("Hello");
    res.json(st.user);
});
app.listen(5004);
```

Explanation

First Code Snippet:

- `res.write` **and** `res.send`:
  - `res.write("Hello world!");` writes a partial response to the client but does not end the response.
  - `res.send("<h1> Hello from LJ</h1>");` sends the response to the client and ends the response.
  - The issue here is that `res.send` also implicitly sets the content type to `text/html` if the response body is a string. When you call `res.write` first and then `res.send`, you are attempting to send two responses for a single request, which is not allowed.
- **Output**:
  - You will see the output "Hello world!" followed by the error because `res.send` is called after `res.write`. The server is trying to handle two responses for one request, which causes an error.

Second Code Snippet:

- `res.write` **and** `res.json`:
  - `res.write("Hello");` writes a partial response to the client but does not end the response.
  - `res.json(st.user);` converts the `st.user` value to JSON and sends it to the client, ending the response.

Khushal Savalakhia

- Like in the first snippet, calling `res.write` followed by `res.json` results in attempting to send two responses for a single request.

- **Output**:
  - This will cause an error immediately without sending "Hello" because `res.json` attempts to end the response after `res.write`.

Correcting the Code Snippets:

First Code Snippet:

If you want to send both a plain text and HTML response, you need to combine them in a single `res.send` call or ensure that `res.write` and `res.end` are used correctly:

```
const expr = require("express");
const app = expr();
app.get("/", (req, res) => {
    res.set("content-type", "text/html");
    res.write("Hello world!<br>");
    res.end("<h1> Hello from LJ</h1>");
});
app.listen(5004);
```

Alternatively:

```
const expr = require("express");
const app = expr();
app.get("/", (req, res) => {
    res.set("content-type", "text/html");
    res.send("Hello world!<br><h1> Hello from LJ</h1>");
});
app.listen(5004);
```

Second Code Snippet:

If you want to send both a plain text and JSON response, you need to combine them into a single response:

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


Khushal Savalakha

```
var st = { user: 2 };
app.get("/", function (req, res) {
    res.write("Hello\\n");
    res.write(JSON.stringify(st.user));
    res.end();
});
app.listen(5004);
```

Alternatively, you can send a single JSON object that includes both messages:

```
const express = require("express");
const app = express();
var st = { user: 2 };
app.get("/", function (req, res) {
    res.json({ message: "Hello", user: st.user });
});
app.listen(5004);
```

Key Points:

- **Only one response per request**: Each request can only have one response, which means you can't call `res.write` and then `res.send` or `res.json`.
- **Use `res.end` with `res.write`**: If you use `res.write`, you must call `res.end` to complete the response.
- **Combine responses**: When needing to send multiple pieces of data, combine them into a single `res.send` or `res.json` call.

▼ PUG

```
doctype html
html(lang="en")
head
title PUG Tutorial
body
//1
h1 LJU
//2
p lets learn PUG
```


Khushal Savalakha

```
//3
p(style="color:blue")
    |i thank you
//4
h5(style="text-transform:uppercase;color:pink") LJU
//5
p(style="color:blue")
    |The file will not
    |render properly if the
    |programmer does not make
    |sure of proper indentation
//6
ul
    li Mango
    li Watemalon
    li Pineapple

//7 Single attribute
h5(style="color:red") Hello

//8 Attributes spanning over multiple lines
a(href='/about' target="_blank") About

//9 Multiple attributes separated by a comma
table(border='1px solid',style='border-collapse:collapse;co
    tr
        th name
        th id
    tr
        td abc
        td 1

//Unbuffered Code
-var number=2
-var color='Black'
-var list=["India","USA","UK"]

//10 Buffered Code
```


Khushal Savalakha

```
h4="3 times number is:"+number*3
h3="I like "+color+"color"
h3="Country:"+list[2]
```

```
var expr = require("express");
var app = expr();
app.set("view engine", "pug");
var p = require("path");
const staticp = p.join(__dirname, "../Public");
app.get("/", (req, res) => {
  res.render(staticp + "/Practice_PUG");
});
app.get("/about", (req, res) => {
  res.write("<h1>Welcome to about page</h1>");
  res.send();
});

app.listen(6783, () => {
  console.log("Server is running on 6783");
});
```

💡 it means here we are using pug file so we have to set environment based on pug file so we are setting app.set and then we are setting particular path app.set("views", path.join(__dirname, "../Public"));
after that we are rendering it am i true ?

Yes, you are correct! Here is a detailed breakdown of what each part of your code does and why it is necessary, especially in the context of using a Pug template file.

Code Explanation

```
var express = require("express");
var app = express();
var path = require("path");
```


Khushal Savalakha

- `express`: This imports the Express module, which is a web application framework for Node.js.
- `app`: This creates an instance of an Express application.
- `path`: This imports the Path module, which provides utilities for working with file and directory paths.

Setting Up the View Engine

```
app.set("view engine", "pug");
```

- **Setting the View Engine**: This tells Express to use Pug as the template engine. Pug templates will be used to render HTML pages.

Configuring the Views Directory

```
app.set("views", path.join(__dirname, "../Public"));
```

- **Setting the Views Directory**: This configures the directory where the Pug templates are located. `path.join(__dirname, "../Public")` constructs the absolute path to the `Public` directory, which is one level up from the current directory.

Serving Static Files

```
app.use(express.static(path.join(__dirname, '../Publi
c'), { index: 'T2_practice.html' }));
```

- **Serving Static Files**: This middleware serves static files (like HTML, CSS, JavaScript, images, etc.) from the `../Public` directory.
- **Index File**: The `{ index: 'T2_practice.html' }` option specifies that `T2_practice.html` should be served as the default file when the root URL is accessed and no other file is specified.

Defining Routes

```
app.get("/", (req, res) => {
  res.render("Practice_PUG"); // Render the template by
```


Khushal Savalakha

```
its name without .pug extension
});
```

- **Root Route (`/`)**: This route handles GET requests to the root URL (`/`).
- **Rendering the Template**: `res.render("Practice_PUG")` renders the `Practice_PUG.pug` template located in the views directory (`../Public`). The `.pug` extension is omitted because Express knows to look for Pug files.

```
app.get("/about", (req, res) => {
  res.write("<h1>Welcome to about page</h1>");
  res.send();
});
```

- **About Route (`/about`)**: This route handles GET requests to the `/about` URL.
- **Sending a Simple Response**: `res.write` writes a simple HTML message, and `res.send` sends the response to the client.

Starting the Server

```
app.listen(6783, () => {
  console.log("Server is running on port 6783");
});
```

- **Starting the Server**: This starts the server on port `6783`. The callback function logs a message to the console indicating that the server is running.

Summary

1. **Set the View Engine**:
   - `app.set("view engine", "pug");` configures Express to use Pug for rendering templates.
2. **Configure the Views Directory**:
   - `app.set("views", path.join(__dirname, "../Public"));` specifies the directory where Pug templates are located.
3. **Serve Static Files**:

Khushal Savalakha

- `app.use(express.static(path.join(__dirname, '../Public'), { index: 'T2_practice.html' }));` serves static files from the `../Public` directory and uses `T2_practice.html` as the default file for the root URL.

4. **Render Templates in Routes**:
   - `res.render("Practice_PUG");` in the root route (`/`) renders the `Practice_PUG.pug` template.
5. **Define Additional Routes**:
   - Additional routes like `/about` can handle other requests and send responses.

By setting up the view engine and views directory, you're configuring Express to use Pug templates located in the `../Public` directory. Then, in your routes, you use `res.render` to render these Pug templates and send the resulting HTML to the client.

▼ Multer

```
In mul.html file :
<html>
<form action="/uploadfile" method="post" enctype="multipart
<input type="file" name="mypic" accept=".jpg,.jpeg,.png"/>
<input type="submit" value="Upload"/>
</form>
</html>
In mul.js file:
// require the installed packages
const express = require('express')
const app = express();
const multer = require('multer');
// SET STORAGE
var store = multer.diskStorage({
destination:"myfolder",
filename: (req, file, cb) =>{
cb(null, file.originalname)
}
})
var upload = multer({ storage: store ,limits : {fileSize :
//CREATE EXPRESS APP
app.use(express.static("../public",{index:"mul.html"}))
app.post('/uploadfile', upload, (req, res) => {
```


Khushal Savalakha

```
if (req.file) {
res.send("<h1>File <span style='color:red'>"+ req.file.orig
}
})
app.listen(6788
```

> 💡 but i still don't understand how step by step they working and what is the work of upload in app.post('/uploadfile', upload, (req, res) => and on which time they are converted in multiple part and what is chunk .

Sure, let's break down the process step-by-step, focusing on the role of the `upload` middleware and how Multer handles the multipart/form-data request:

Step-by-Step Breakdown

1. **Form Submission**:
   - The user submits the form containing the file input.
   - The form uses the `POST` method and `enctype="multipart/form-data"`, which means the form data, including the file, is sent to the server in a special format called `multipart/form-data`.
2. **Express Route Setup**:
   - You have an Express route that listens for POST requests at `/uploadfile`.

   ```
   app.post('/uploadfile', upload, (req, res) => {
       // Route handler logic
   });
   ```

   In this route, `upload` is the Multer middleware.
3. **Multer Middleware (`upload`)**:
   - When the form is submitted, the request reaches the server at the `/uploadfile` endpoint.
   - Before the request is processed by the route handler, the `upload` middleware (configured with Multer) intercepts the request.

Khushal Savalakha

```
var upload = multer({ storage: store, limits: { fileS
ize: 1000000 } }).single('mypic');
```

4. **Multipart Data Parsing**:
   - The `upload` middleware parses the incoming multipart/form-data request. This involves:
     - Reading the headers to determine the boundaries between different parts of the request.
     - Separating the request into multiple parts: each part represents either a form field or a file.
5. **Handling Each Part**:
   - For each part, Multer determines if it is a form field or a file. For file parts:
     - Multer reads the file data in chunks. A chunk is a piece of data, and reading in chunks helps in handling large files without consuming too much memory.
     - Multer validates the file size and (if configured) the MIME type.
6. **Storing the File**:
   - Multer streams the file data to the designated storage location as specified in the `diskStorage` configuration.
   - The file is saved chunk by chunk to the specified destination (`myfolder`) with the original filename.
7. **Request Handling**:
   - After Multer has processed the file, control is passed to the route handler.
   - In the route handler, you can access `req.file` to get information about the uploaded file.

```
app.post('/uploadfile', upload, (req, res) => {
    if (req.file) {
        res.send("<h1>File <span style='color:red'>"
+ req.file.originalname + "</span> has been uploaded
in <span style='color:red'>" + req.file.destination +
" </span>folder</h1>");
```


Khushal Savalakha

```
    } else {
        res.status(400).send("File upload failed");
    }
});
```

Detailed Explanation of Key Concepts

Multipart/Form-Data:

- This is a format used to transmit files and other form data over HTTP. It divides the form data into multiple parts, each part containing headers and the actual data (either form fields or files).

Chunks:

- Chunks are small pieces of data. When handling large files, reading and writing the entire file at once can consume a lot of memory. Instead, Multer reads and writes the file in smaller pieces (chunks), which is more efficient and prevents memory overload.

Multer Middleware:

- Multer middleware (`upload`) is responsible for intercepting the request, parsing the multipart/form-data, validating the file, and saving it to the specified location.

Breakdown of `app.post('/uploadfile', upload, (req, res) =>`

1. **Request Interception**:
    - The request is intercepted by the `upload` middleware.
    - Multer processes the multipart/form-data request before the route handler logic is executed.
2. **Parsing and Validation**:
    - Multer parses the request, separates it into parts, validates the file (size and optionally MIME type), and stores the file in chunks.
3. **Control to Route Handler**:
    - After Multer completes its processing, control is passed to the route handler.

Khushal Savalakhia

- The route handler checks if `req.file` exists (indicating a successful upload) and sends a response to the client.

Example Code Recap

mul.html:

```
<html>
    <form action="/uploadfile" method="post" enctype="mu
ltipart/form-data">
        <input type="file" name="mypic" accept=".jpg,.jp
eg,.png"/>
        <input type="submit" value="Upload"/>
    </form>
</html>
```

mul.js:

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

// SET STORAGE
var store = multer.diskStorage({
    destination: "myfolder",
    filename: (req, file, cb) => {
        cb(null, file.originalname);
    }
});

var upload = multer({
    storage: store,
    limits: { fileSize: 1000000 }
}).single('mypic');

// Serve static files and set the index file
app.use(express.static("../public", { index: "mul.html"
}));
```


Khushal Savalakha

```
app.post('/uploadfile', upload, (req, res) => {
    if (req.file) {
        res.send("<h1>File <span style='color:red'>" + req.file.originalname + "</span> has been uploaded in <span style='color:red'>" + req.file.destination + " </span>folder</h1>");
    }
});

app.listen(6788, () => {
    console.log("Server is running on port 6788");
});
```

In summary, the `upload` middleware plays a crucial role in parsing, validating, and storing the file data before the route handler processes the request. This separation ensures efficient and secure file handling in your Express application.

Khushal Savalakha

```
+-------------------+
|   Client Browser  |
|-------------------|
|   Submits Form    |
| /uploadfile with  |
| multipart/form-data|
+-------------------+
           |
           v
+-------------------+
|  Express Server   |
|-------------------|
| Receives POST Req |
|  /uploadfile      |
|-------------------|
|  Route Handler    |
| app.post          |
+-------------------+
           |
           v
+-------------------+
|  Multer Middleware|
|-------------------|
|  Parses & Validates|
| multipart/form-data|
+-------------------+
           |
           v
+-------------------+
|  Storage Engine   |
|-------------------|
| Saves File to Disk|
| (myfolder)        |
+-------------------+
           |
           v
+-------------------+
|  Route Handler    |
|-------------------|
|  Checks req.file  |
|  Sends Response   |
|  to Client        |
+-------------------+
```


Khushal Savalakha

Let's break down the differences between single file uploading and multiple file uploading using Multer in Node.js/Express, and explain the key concepts like setting file size limits and the meaning of `.array('fieldname', maxCount)`.

Single File Upload Example (`mul.js` and `mul.html`):

mul.html

```
<html>
<form action="/uploadfile" method="post" enctype="multip
art/form-data">
  <input type="file" name="mypic" accept=".jpg,.jpeg,.pn
g"/>
  <input type="submit" value="Upload"/>
</form>
</html>
```

mul.js

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

// SET STORAGE
const storage = multer.diskStorage({
  destination: "myfolder",
  filename: (req, file, cb) => {
    cb(null, file.originalname); // Keep the original fi
lename
  }
});

// Configure multer to handle single file uploads with s
ize limit
const upload = multer({
  storage: storage,
  limits: { fileSize: 1000000 } // 1MB file size limit
}).single('mypic'); // Handle single file with name 'myp
ic'
```


Khushal Savalakha

```
// Serve static files and HTML form
app.use(express.static("../public", { index: "mul.html"
}));

// POST endpoint for file upload
app.post('/uploadfile', upload, (req, res) => {
  if (req.file) {
    res.send("<h1>File <span style='color:red'>" + req.f
ile.originalname + "</span> has been uploaded in <span s
tyle='color:red'>" + req.file.destination + " </span>fol
der");
  }
});

// Start the server
app.listen(6788);
```

Multiple Files Upload Example (`mul2.js` and `Mul2.html`):

Mul2.html

```
<html>
<form action="/uploadfile" method="post" enctype="multip
art/form-data">
  <input type="file" name="myfile" accept=".doc,.docx" m
ultiple/>
  <input type="submit" value="Upload"/>
</form>
</html>
```

mul2.js

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

// SET STORAGE
```


Khushal Savalakhia

```
const storage = multer.diskStorage({
  destination: "multiple",
  filename: (req, file, cb) => {
    cb(null, file.fieldname + "-" + Date.now() + ".doc
x"); // Append timestamp to filename
  }
});

// Configure multer to handle multiple file uploads with
size limit
const upload = multer({
  storage: storage,
  limits: { fileSize: 5000000 } // 5MB file size limit p
er file
}).array('myfile', 5); // Handle up to 5 files with name
'myfile'

// Serve static files and HTML form
app.use(express.static("../public", { index: "Mul2.html"
}));

// POST endpoint for file upload
app.post('/uploadfile', upload, (req, res) => {
  if (req.files && req.files.length > 0) {
    res.set("content-type", "text/html");
    for (let i = 0; i < req.files.length; i++) {
      res.write("<h2>File <span style='color:red'>" + JS
ON.stringify(req.files[i].originalname) + "</span> has b
een uploaded </h2>");
    }
    res.send();
  }
});

// Start the server
app.listen(6788);
```

Key Differences and Concepts:

Khushal Savalakha

1. **File Size Limit**:
   - Both examples use `limits: { fileSize: ... }` in the multer configuration to set a maximum file size limit. In `mul.js`, it's set to 1MB (`1000000` bytes), and in `mul2.js`, it's set to 5MB (`5000000` bytes) per file.
2. `.single('fieldname')` **vs** `.array('fieldname', maxCount)`:
   - `.single('fieldname')`: Handles a single file upload with the specified `fieldname`.
   - `.array('fieldname', maxCount)`: Handles multiple file uploads with the specified `fieldname`, up to `maxCount` files.
3. **Filename Handling**:
   - In `mul.js`, the filename remains the same as the original file uploaded (`file.originalname`).
   - In `mul2.js`, filenames are prefixed with `myfile-` and suffixed with the current timestamp (`Date.now()`), ensuring unique filenames for each uploaded file.
4. **Handling Multiple Files**:
   - `mul2.js` uses `.array('myfile', 5)` to handle up to 5 files uploaded through the form input with `name="myfile"`.
5. **Server Setup**:
   - Both examples serve static files (`mul.html` and `Mul2.html`) using `express.static`.
   - They listen on port `6788` for incoming requests and handle file uploads on the `/uploadfile` endpoint.

These examples illustrate how to configure Multer for different file upload scenarios in Node.js/Express, including handling single and multiple files, setting file size limits, and managing filenames. Adjust the configurations (`storage`, `limits`, and `upload` methods) according to your specific requirements for file handling and storage in your application.

▼ Nodemailer

```
var expr = require("express");
const app = expr();
var nm = require("nodemailer");
var path = require("path");
app.get("/", (req, res) => {
```


Khushal Savalakha

```
  res.sendFile(path.join(__dirname, "../Public", "Mail.html
});
app.get("/feedback", (req, res) => {
    var trans=nm.createTransport({
        host:"smtp.gmail.com",
        port:465,
        auth:{
            user:"routinuse7@gmail.com",
        pass:'srqk ydyy thma qkte',
        }

    })
  var mailoption = {
    from:"routinuse7@gmail.com",
    to: req.query.mail,
    bcc: "bcc@gmail.com",
    subject: "Feedback response",
    html:
      "<h1>Hello " +
      req.query.uname +
      "!</h1><h2> Thank you for submitting the feedback.<br
      req.query.feedback +
      "</span></h2><h3> Additional comments: " +
      req.query.comments +
      "</h3>",
  };
  trans.sendMail(mailoption, (err, info) => {
    if (err) {
      console.error(err);
    }
    console.log(info);
  });
  res.send("Thank you for your feedback.");
});
app.listen(5001);
//gmail >> manage account >> security >> 2 step verificatio
```


Khushal Savalakha

```html
<html>
  <head>
    <title>Feedback Form</title>
  </head>
  <body>
    <form action="/feedback" method="get">
      Name: <input type="text" name="uname" /> Email:
      <input type="text" name="mail" /></br> Feedback:
      <select name="feedback">
        <option value="bad">Bad</option>
        <option value="good">Good</option>
        <option value="verygood">Very Good</option>
        <option value="excellent" SS>Excellent</option>
      </select>
      Comments:
      <textarea
        name="comments"
        rows="10"
        cols="15"
        placeholder="Comments"
      ></textarea>
      <input type="submit" value="Submit" />
    </form>
  </body>
</html>
```

Sure, let's delve into how each part of the code works internally, and I'll explain the SMTP configuration in detail.

Detailed Breakdown of the Code

1. **Setting Up Express and Dependencies**:

```javascript
const express = require("express");
const app = express();
const nodemailer = require("nodemailer");
const path = require("path");
```


Khushal Savalakha

- **Explanation**:
  - `express`: A minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications.
  - `nodemailer`: A module for Node.js applications to send emails.
  - `path`: A Node.js module that provides utilities for working with file and directory paths.

2. **Serving Static Files**:

```
app.use(express.static(path.join(__dirname, "../Publi
c")));
```

- **Explanation**:
  - `express.static`: Middleware for serving static files.
  - `path.join(__dirname, "../Public")`: Constructs an absolute path to the `Public` directory relative to the script's location. This makes the `Public` directory accessible to the web server.

3. **Route for Root Path**:

```
app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname, "../Public", "Mai
l.html"));
});
```

- **Explanation**:
  - `app.get("/")`: Defines a route handler for GET requests to the root path.
  - `res.sendFile(...)`: Sends the `Mail.html` file to the client. This uses the `path` module to construct the absolute path to the `Mail.html` file in the `Public` directory.

Feedback Form Handling

1. **Handling Feedback Form Submission**:

```
app.get("/feedback", (req, res) => {
  // Nodemailer configuration to send email
```


Khushal Savalakhia

```
  var trans = nodemailer.createTransport({
    host: "smtp.gmail.com",
    port: 465,
    auth: {
      user: "sender@gmail.com",
      pass: "2stepverificationkey",
    },
  });

  // Email content setup
  var mailoption = {
    from: "noreply@gmail.com<noreply>",
    to: req.query.mail,
    bcc: "bcc@gmail.com",
    subject: "Feedback response",
    html:
      "<h1>Hello " +
      req.query.uname +
      "!</h1><h2> Thank you for submitting the feedba
ck.<br> Your feedback: <span style=color:red>" +
      req.query.feedback +
      "</span></h2><h3> Additional comments: " +
      req.query.comments +
      "</h3>",
  };

  // Sending email
  trans.sendMail(mailoption, (err, info) => {
    if (err) {
      console.error(err);
    }
    console.log(info);
  });

  // Sending response to the client
  res.send("Thank you for your feedback.");
});
```


Khushal Savalakha

- **Explanation**:
  - **Route Definition**: `app.get("/feedback")` defines a route handler for GET requests to `/feedback`.
  - **Nodemailer Configuration**:
    - `nodemailer.createTransport({...})` : Creates a transport object using SMTP.
    - **SMTP**: Simple Mail Transfer Protocol, a protocol for sending email messages between servers.
    - `host: "smtp.gmail.com"` : Specifies the Gmail SMTP server.
    - `port: 465` : Specifies the port number to connect to. Port 465 is used for SMTP over SSL (Secure Sockets Layer).
    - `auth: { user: "sender@gmail.com", pass: "2stepverificationkey" }` : Provides the authentication credentials. The `pass` should be an app-specific password generated for 2-step verification.
- **Email Content**:
  - `from: "noreply@gmail.com<noreply>"` : The sender's email address.
  - `to: req.query.mail` : The recipient's email address, taken from the query parameter.
  - `bcc: "bcc@gmail.com"` : BCC (Blind Carbon Copy) email address.
  - `subject: "Feedback response"` : The subject of the email.
  - `html: "<h1>Hello " + req.query.uname + ...` : The HTML content of the email, which includes the user's name, feedback, and comments from the query parameters.
- **Sending Email**:
  - `trans.sendMail(mailoption, (err, info) => {...})` : Sends the email with the specified options. If there's an error, it logs it to the console; otherwise, it logs the info about the sent email.
- **Sending Response to Client**:
  - `res.send("Thank you for your feedback.")` : Sends a response back to the client indicating that the feedback has been received.

Starting the Server

Khushal Savalakha

1. **Listening for Connections**:

```
app.listen(5001, () => {
  console.log("Server is running on port 5001");
});
```

   - **Explanation**:
     - `app.listen(5001, ...)`: Starts the Express server on port 5001.
     - **Callback Function**: Logs a message to the console to indicate the server is running.

Detailed Workflow

1. **Client Side**:
   - The user accesses `http://localhost:5001/`.
   - The server sends the `Mail.html` form.
   - The user fills out the form and submits it.
2. **Server Side**:
   - The server receives a GET request to `/feedback` with query parameters (`uname`, `mail`, `feedback`, `comments`).
   - Nodemailer is configured to send an email using Gmail's SMTP server.
   - The email content is constructed using the query parameters from the form submission.
   - Nodemailer sends the email.
   - The server responds to the client with a thank-you message.

Visual Representation

Here’s a visual representation of how this works:

```
+-------------------+
|   Client Browser  |
|-------------------|
|   Submits Form    |
|  to /feedback     |
```


Khushal Savalakha

```
+-------------------+
           |
           V
+-------------------+
|  Express Server   |
|-------------------|
| Receives GET Req  |
|   /feedback       |
|-------------------|
|  Route Handler    |
| app.get("/feedback")|
+-------------------+
           |
           V
+-------------------------+
|  Nodemailer Module      |
|-------------------------|
| Configures SMTP         |
|  host: "smtp.gmail.com" |
|  port: 465              |
|  user: "sender@gmail.com"|
|  pass: "2stepverificationkey"|
+-------------------------+
           |
           V
+-------------------------+
|  Constructs Email       |
|  from: "noreply@gmail.com" |
|  to: req.query.mail     |
|  subject: "Feedback response"|
|  html: "<h1>Hello " + req.query.uname + ...|
+-------------------------+
           |
           V
+-------------------------+
|  Sends Email via SMTP   |
|  trans.sendMail(mailoption, ...) |
+-------------------------+
```


Khushal Savalakha

```
            |
            V
+-------------------+
|  Sends Response   |
|  to Client        |
|  "Thank you for your feedback." |
+-------------------+
```

Conclusion

This setup allows the user to submit a feedback form, which the server processes to send a confirmation email using Nodemailer configured with Gmail's SMTP server. The response is then sent back to the client to acknowledge receipt of the feedback.

```
var nm=require("nodemailer")
var trans=nm.createTransport({
    host:"smtp.gmail.com",
    port:465,
    auth:{
        user:"routinuse7@gmail.com",
    pass:'srqk ydyy thma qkte',
    }

})
var mailOption={
    from:"routinuse7@gmail.com",
    to:"routinuse7@gmail.com",
    subject:"Hello",
    // text:"Hello welcome to nodemailer",
    html:"Here nodemailer,<h1>Effect of h1</h1> <img src=ht
}


trans.sendMail(mailOption,(err,info)=>{
    if(err){
        console.log(err);
        console.log('---------');
    }
    console.log("***");
```


Khushal Savalakha

```
    console.log(info);
})
```


Khushal Savalakha