Notes on Express.js in Node.js

Introduction to Express.js

- **Node.js** requires writing a lot of code for basic tasks like extracting the body of an incoming request or handling routing.
- **Express.js** is a **third-party framework** that simplifies these tasks, allowing developers to focus on **business logic** (the core functionality of the application).
- It provides a **rule set** and **utility functions** to write cleaner and more efficient code.

Key Concepts in Express.js

1. **Middleware**:
    - A core concept in Express.js.
    - Middleware functions are used to handle tasks like parsing request bodies, logging, or authentication.
    - They sit between the incoming request and the final response, allowing for modular and reusable code.

2. **Handling Requests and Responses**:
    - Express.js simplifies the process of handling **HTTP requests** (e.g., GET, POST) and sending **responses**.
    - Example:

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

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

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

3. **Routing**:
    - Express.js allows developers to define routes for different **URLs** or **paths** without writing multiple if statements.
    - Example:

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

app.post('/submit', (req, res) => {
    res.send('Form Submitted');
});
```

4. **Returning HTML Pages**:
    - Instead of embedding HTML directly in Node.js code, Express.js allows serving pre-prepared HTML files.
    - Example:

```
app.get('/home', (req, res) => {
   res.sendFile(__dirname + '/index.html');
});
```

Benefits of Using Express.js

- **Reduces boilerplate code**: Handles repetitive tasks like request parsing and routing.

- **Improves code organization**: Encourages modular and reusable code through middleware and routing.
- **Enhances productivity**: Allows developers to focus on application-specific logic rather than low-level details.

Key Vocabulary

- **Node.js**: A runtime environment for executing JavaScript on the server.
- **Express.js**: A web application framework for Node.js.
- **Middleware**: Functions that process requests before they reach the final route handler.
- **Routing**: Defining how an application responds to client requests for specific endpoints (URLs).
- **Business Logic**: The core functionality of an application, such as processing data or handling user interactions.

Example Code: Basic Express.js Setup

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

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

// Route for the home page
app.get('/', (req, res) => {
    res.send('Welcome to the Home Page!');
});

// Route for the about page
app.get('/about', (req, res) => {
```

```
    res.send('About Us');
});

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

Summary

- **Express.js** is a powerful framework built on top of **Node.js** that simplifies web development by handling common tasks like routing, request/response handling, and serving static files.
- It uses **middleware** to modularize code and provides tools to focus on **business logic**.
- By using Express.js, developers can write cleaner, more maintainable, and efficient code.

Notes on Express.js

What is Express.js?

- **Express.js** is a **framework** for **Node.js** that simplifies server-side logic.
- It helps developers avoid writing complex code for tasks like **parsing incoming requests** (e.g., extracting the body of a request).
- Without Express.js, developers would need to manually handle events like data and end, create buffers, and convert data to strings.

Why Use Express.js?

1. **Simplifies Complex Tasks**:
    - Handles repetitive tasks like **request parsing** and **routing**.
    - Allows developers to focus on **business logic** (the unique functionality of the application).
    - Example: Instead of manually parsing request bodies, Express.js makes it easy to integrate third-party packages for this purpose.
2. **Framework Benefits**:
    - Provides a **set of helper functions**, **tools**, and **rules** for structuring applications.
    - Offers a **clearly defined way** to write clean and maintainable code.
3. **Flexibility and Extensibility**:
    - Express.js is **highly flexible** and doesn't impose too many functionalities out of the box.
    - It is **extensible**, with thousands of **third-party packages** available to enhance functionality without extensive configuration.

Alternatives to Express.js

- **Vanilla Node.js**: Can be used for simpler applications or for developers seeking more control.
- **Other Frameworks**:
    - **Adonis.js**: Inspired by Laravel (PHP framework).

    - **Koa**: A lightweight alternative to Express.js.
    - **Sails.js**: A framework for building data-driven APIs.
- Express.js is the **most popular** and widely used framework for Node.js.

Key Vocabulary

- **Express.js**: A web application framework for Node.js.
- **Framework**: A set of tools, helper functions, and rules for building applications.
- **Business Logic**: The core functionality that defines an application's unique features.
- **Request Parsing**: Extracting and processing data from incoming HTTP requests.
- **Third-Party Packages**: External libraries or tools that can be integrated into a project.

Example Code: Installing and Using Express.js

1. **Installation**:

```
Bash:
    npm install express
```

2. **Basic Setup**:

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

    // Route for the home page
    app.get('/', (req, res) => {
        res.send('Hello, World!');
```

```
});


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

Summary

- **Express.js** simplifies server-side development by handling repetitive tasks and providing a structured way to build applications.
- It allows developers to focus on **business logic** while outsourcing **low-level details** like request parsing and routing.
- Express.js is **flexible**, **extensible**, and supported by a large ecosystem of third-party packages.
- While alternatives exist, Express.js remains the **most popular** framework for Node.js.

Installing and Setting Up Express.js

Installation of Express.js

1. **Command**: Install as a **production dependency** (essential for deployment):

   Bash:

   ```
   npm install --save express
   ```

   - **Why --save?** Express.js is a core part of the application, not just a development tool.
   - Updates dependencies in **package.json**.

Setting Up Express.js

1. **Importing**:

   Javascript:

   ```
   const express = require('express');
   const app = express(); // Initialize the Express app
   ```

   - express exports a **function** that initializes the framework.
   - The app constant manages Express.js logic (e.g., routing, middleware).

2. **Code Organization**:

   - Separate imports for clarity (optional but recommended):

   javascript

   ```
   // Core Node.js modules
   const http = require('http');

   // Third-party packages
   const express = require('express');
   ```

```
// Custom modules (if any)
// const myModule = require('./myModule');
```

Key Concepts

1. **Request Handler**:
   - The app object is a valid **request handler** and can be passed to http.createServer():

Javascript:

```
const server = http.createServer(app);
server.listen(3000);
```

   - Alternatively, use Express.js's built-in listen() method:

Javascript:

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

2. **Initial Behavior**:
   - Without defined routes, the app responds with no content but sets up Express.js's **request-handling pipeline**.

Example Code

Javascript:

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

// Basic server setup (no routes defined yet)
```

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

Summary

- **Express.js** is installed as a **production dependency** with npm install --save express.
- Initialize the app with const app = express();, which creates a **request handler** for managing server logic.
- The app object simplifies server creation (via app.listen()) and defines Express.js's structure for handling requests.
- No routes or logic are added initially, but Express.js sets up its foundational **request-handling workflow**.

Key Vocabulary

- **Production Dependency**: A package required for the application to run in production.
- **Request Handler**: Code that processes incoming HTTP requests (e.g., app in Express.js).
- **Express.js Pipeline**: The structured flow of middleware and routes for handling requests.

Express.js Middleware

Core Concept: Middleware

- **Middleware** in Express.js refers to functions that process incoming requests in a sequence (a "**pipeline**").
- Requests are **funneled through middleware functions** until a response is sent.
- Middleware allows splitting code into **modular, reusable blocks** instead of a single monolithic handler.

Key Mechanics of Middleware

1. **Adding Middleware**:
    - Use app.use() to register middleware functions.
    - Example:

```
javascript
    app.use((req, res, next) => {
      console.log('In the middleware!');
      next(); // Pass control to the next middleware
    });
```

2. **Arguments in Middleware**:
    - **req**: The **request** object (contains data about the incoming request).
    - **res**: The **response** object (used to send back data).
    - **next**: A **function** that passes the request to the next middleware in line.

3. **Critical Rules**:
    - **Call next()** to allow the request to proceed to the next middleware.
    - **Send a response** (e.g., res.send()) if you don't call next() (otherwise, the request hangs).

Example: Middleware Execution Flow

Javascript:

```
// Middleware 1
app.use((req, res, next) => {
  console.log('Middleware 1');
  next(); // Pass to Middleware 2
});

// Middleware 2
app.use((req, res, next) => {
  console.log('Middleware 2');
  res.send('Response sent!'); // End the request here
});
```

- **Result**:
    - Request passes through Middleware 1, then Middleware 2.
    - res.send() in Middleware 2 stops further processing.

Key Vocabulary

- **Middleware**: Functions that process requests in Express.js.
- **app.use()**: Method to register middleware.
- **next()**: Function to pass control to the next middleware.

- **Request Pipeline**: The sequential flow of middleware functions.

Common Pitfalls

- **Not calling next()**:
  - The request stalls indefinitely unless a response is sent.
  - Example of a **blocking middleware**:

```
Javascript:

    app.use((req, res, next) => {
      console.log('This blocks the request!');
      // No call to next() or res.send() → request hangs
    });
```

Summary

- **Middleware** is central to Express.js, enabling modular and organized request handling.
- Use app.use() to add middleware, **next()** to forward requests, and **res.send()** to terminate processing.
- Middleware executes **top-to-bottom**; order matters.
- Always ensure a response is sent or next() is called to avoid hanging requests.

Sending Responses in Express.js Middleware

Key Concepts

1. **Middleware Execution Flow**:
    - Requests travel through middleware functions **top-to-bottom**.
    - Use **next()** to pass control to the next middleware.
    - If no next() is called and no response is sent, the request **stalls indefinitely**.

2. **Sending Responses**:
    - Use **res.send()** to send a response and terminate the request.
    - **res.send()** automatically sets the **Content-Type** header based on the response body (e.g., text/html for HTML).
    - Example:

```
javascript
    app.use((req, res) => {
      res.send('<h1>Hello from Express</h1>');
    });
```

Key Features of res.send()

1. **Automatic Header Handling**:
    - Sets **Content-Type** based on the response body (e.g., text/html for HTML, application/json for JSON).
    - Example:

```
javascript
    res.send('<h1>Hello</h1>'); // Content-Type:
    text/html
    res.send({ message: 'Hello' }); // Content-
    Type: application/json
```

2. **Flexibility**:
    - Can send **any type of data** (e.g., HTML, JSON, plain text).
    - Example:

```
Javascript:
    res.send('Plain text response');
    res.send({ key: 'value' });
    res.send('<h1>HTML response</h1>');
```

3. **Manual Header Override**:
    - Use **res.setHeader()** to manually set headers if needed.
    - Example:

```
Javascript:
    res.setHeader('Content-Type', 'text/plain');
    res.send('Plain text response');
```

Middleware Behavior

1. **Terminating Middleware**:
    - If a middleware sends a response (e.g., res.send()), subsequent middleware **will not execute**.
    - Example:

```
Javascript:
  app.use((req, res) => {
    res.send('Response sent!'); // Terminates the
  request
  });

  app.use((req, res) => {
    console.log('This will not run');
```

```
});
```

2. **Non-Terminating Middleware**:
   - Use **next()** to pass control to the next middleware.
   - Example:

```
Javascript:
     app.use((req, res, next) => {
       console.log('Middleware 1');
       next(); // Pass to Middleware 2
     });

     app.use((req, res) => {
       res.send('Response from Middleware 2');
     });
```

Example Code

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

     // Middleware 1
     app.use((req, res, next) => {
       console.log('Middleware 1');
       next(); // Pass to Middleware 2
     });

     // Middleware 2
```

```
app.use((req, res) => {
  res.send('<h1>Hello from Express</h1>'); // Terminates the request
});

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

Key Vocabulary

- **Middleware**: Functions that process requests in Express.js.
- **res.send()**: Method to send a response and terminate the request.
- **next()**: Function to pass control to the next middleware.
- **Content-Type**: HTTP header indicating the type of response data.

Summary

- **Middleware** functions process requests in a **top-to-bottom** sequence.
- Use **res.send()** to send responses and terminate requests; it automatically sets **Content-Type**.
- Call **next()** to pass control to the next middleware; otherwise, the request stalls.
- **res.send()** simplifies response handling compared to manual methods like res.write() and res.end().

Express.js Internals and Code Optimization

Exploring Express.js Internals

1. **Open Source Code**:
    - Express.js is **open source**; its code is available on GitHub.
    - Key files:
        - **lib/response.js**: Contains the implementation of res.send().
        - **lib/application.js**: Contains the implementation of app.listen().

2. **How res.send() Works**:
    - **Checks the type of data** being sent (e.g., string, number, boolean, JSON).
    - **Automatically sets Content-Type** if not already defined:
        - **String**: Sets Content-Type: text/html.
        - **Number/Boolean**: Sets Content-Type: application/octet-stream (binary data).
        - **JSON**: Sets Content-Type: application/json.
    - Example:

```
Javascript:

    res.send('<h1>Hello</h1>'); // Content-Type: text/html

    res.send({ message: 'Hello' }); // Content-Type: application/json
```

3. **Why Dive into the Code?**
    - Helps understand **default behaviors** (e.g., automatic header setting).
    - Useful for debugging or customizing functionality.

Optimizing Server Setup

1. **Simplified Server Initialization**:
    - Instead of manually creating an HTTP server:

      Javascript:

      ```
      const http = require('http');
      const server = http.createServer(app);
      server.listen(3000);
      ```

    - Use **app.listen()**:

      Javascript:

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

    - **Internally**, app.listen() calls http.createServer() and passes the app object.

2. **Benefits**:
    - **Reduces boilerplate code**.
    - **Improves readability** and maintainability.

Example Code

Javascript:

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

```
// Middleware example
app.use((req, res, next) => {
  console.log('Middleware executed');
  next();
});

// Route handler
app.use((req, res) => {
  res.send('<h1>Hello from Express</h1>');
});

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

Key Vocabulary

- **res.send()**: Express.js method to send responses with automatic Content-Type handling.
- **app.listen()**: Simplified method to start an Express.js server.
- **Middleware**: Functions that process requests in a sequence.
- **Content-Type**: HTTP header indicating the type of response data.

Summary

- **Express.js Internals**:
  - **res.send()** automatically sets Content-Type based on the response data.

    - **app.listen()** simplifies server setup by internally calling http.createServer().

- **Code Optimization**:
    - Use **app.listen()** to reduce boilerplate and improve readability.

- **Middleware**:
    - Requests flow through middleware functions **top-to-bottom**.
    - Use **next()** to pass control or **res.send()** to terminate the request.

Next Steps:

- Learn how to handle **different routes** (e.g., /message, /home).
- Simplify reading **incoming request data**.

Routing and Middleware in Express.js

Handling Different Routes

1. **Filtering Requests by Path**:
    - Use **app.use(path, callback)** to filter requests based on the URL path.
    - The **path** argument specifies the starting part of the URL (e.g., /add-product).
    - Example:

```
javascript
    app.use('/add-product', (req, res) => {
      res.send('<h1>Add Product Page</h1>');
    });
```

2. **Middleware Execution Order**:
    - Middleware executes **top-to-bottom** in the file.
    - If a middleware sends a response (e.g., res.send()), subsequent middleware **will not execute**.
    - Use **next()** to pass control to the next middleware.

3. **Default Path (/)**:
    - Middleware without a path filter (or with /) runs for **all requests**.
    - Example:

```
Javascript:
    app.use((req, res, next) => {
      console.log('This always runs!');
      next(); // Pass to the next middleware
    });
```

Example Code: Routing with Middleware

Javascript:

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

// Middleware for all requests
app.use((req, res, next) => {
  console.log('This always runs!');
  next();
});

// Route for /add-product
app.use('/add-product', (req, res) => {
  res.send('<h1>Add Product Page</h1>');
});

// Default route (/)
app.use((req, res) => {
  res.send('<h1>Hello from Express</h1>');
});

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

Key Concepts

1. **Path Filtering**:
    - The **path** in app.use(path, callback) matches the **starting part** of the URL.
    - Example: /add-product matches /add-product, /add-product/123, etc.

2. **Middleware Order Matters**:
    - Place **general middleware** (e.g., logging) at the top to run for all requests.
    - Place **specific routes** (e.g., /add-product) below general middleware.

3. **Sending Responses**:
    - Use **res.send()** to send a response and terminate the request.
    - Avoid calling **next()** after sending a response to prevent errors.

Common Pitfalls

1. **Overlapping Paths**:
    - Ensure specific routes are placed **before general routes** to avoid unintended matches.
    - Example:

```
Javascript:
    app.use('/add-product', (req, res) => {
      res.send('<h1>Add Product Page</h1>');
    });

    app.use((req, res) => {
      res.send('<h1>Hello from Express</h1>');
```

```
});
```

2. **Multiple Responses**:
    - Sending more than one response per request results in an **error**.
    - Example of **incorrect usage**:

```
Javascript:
    app.use((req, res) => {
      res.send('First response');
      res.send('Second response'); // Error!
    });
```

Key Vocabulary

- **Middleware**: Functions that process requests in Express.js.
- **app.use(path, callback)**: Method to filter requests by URL path.
- **next()**: Function to pass control to the next middleware.
- **Path Filtering**: Matching requests based on the starting part of the URL.

Summary

- Use **app.use(path, callback)** to handle specific routes (e.g., /add-product).
- Middleware executes **top-to-bottom**; order matters for routing and functionality.
- Send **one response per request** using res.send() and avoid calling next() afterward.
- Place **general middleware** at the top and **specific routes** below to ensure proper request handling.

Next Steps:

- Learn how to handle **dynamic routes** (e.g., /products/:id).
- Simplify reading **incoming request data** (e.g., query parameters, request body).

Handling Incoming Requests in Express.js

Handling POST Requests and Parsing Request Bodies

1. **Returning HTML Forms**:
    - Use **res.send()** to return an HTML form in the response.
    - Example:

```
javascript
    app.use('/add-product', (req, res) => {
      res.send(`
        <form action="/product" method="POST">
          <input type="text" name="title"
placeholder="Product Title">
          <button type="submit">Add
Product</button>
        </form>
      `);
    });
```

2. **Handling Form Submissions**:
    - Use **app.use(path, callback)** to handle POST requests to a specific path (e.g., /product).
    - Example:

```
Javascript:
    app.use('/product', (req, res) => {
      console.log(req.body); // Log the parsed
request body
      res.redirect('/'); // Redirect to the home
page
    });
```

3. **Parsing Request Bodies**:
    - Use **body-parser** to parse incoming request bodies.
    - Install **body-parser**:

      Bash:

      ```
      npm install --save body-parser
      ```

    - Configure **body-parser**:

      Javascript:

      ```
      const bodyParser = require('body-parser');
      app.use(bodyParser.urlencoded({ extended: false }));
      ```

    - **req.body** will contain the parsed form data as a JavaScript object (e.g., { title: 'Book' }).

Example Code: Handling POST Requests

Javascript:

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

// Middleware to parse request bodies
app.use(bodyParser.urlencoded({ extended: false }));

// Route to display the form
app.use('/add-product', (req, res) => {
  res.send(`
    <form action="/product" method="POST">
```

```
      <input type="text" name="title"
placeholder="Product Title">
      <button type="submit">Add Product</button>
    </form>
  `);
});

// Route to handle form submissions
app.use('/product', (req, res) => {
  console.log(req.body); // Log the parsed request
body
  res.redirect('/'); // Redirect to the home page
});

// Default route
app.use('/', (req, res) => {
  res.send('<h1>Hello from Express</h1>');
});

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

Key Concepts

1. **Form Handling**:
    - Use **<form>** tags to create HTML forms.
    - Set **action** to the URL where the form data should be sent.
    - Set **method** to POST for form submissions.

2. **Request Body Parsing**:
    - **body-parser** middleware parses form data and populates **req.body**.
    - **req.body** contains key-value pairs based on the form inputs (e.g., { title: 'Book' }).
    -
3. **Redirecting Responses**:
    - Use **res.redirect(path)** to redirect users to another route.

Common Pitfalls

1. **Missing Body Parser**:
    - Without **body-parser**, **req.body** will be undefined.
    - Ensure **body-parser** is configured before route handlers.

2. **Incorrect Form Attributes**:
    - Ensure the **action** and **method** attributes in the form match the route and HTTP method in Express.

3. **Order of Middleware**:
    - Place **body-parser** middleware **before** route handlers to ensure request bodies are parsed.

Key Vocabulary

- **body-parser**: Middleware to parse incoming request bodies.
- **req.body**: Object containing parsed form data.
- **res.redirect(path)**: Method to redirect users to another route.
- **Form Handling**: Managing user input via HTML forms and processing submissions.

Summary

- Use **res.send()** to return HTML forms and handle user input.
- Install and configure **body-parser** to parse form data into **req.body**.
- Use **res.redirect()** to redirect users after processing form submissions.
- Ensure middleware order is correct (e.g., **body-parser** before route handlers).

Next Steps:

- Learn how to handle **JSON data** and **file uploads** in Express.js.
- Explore **route-specific middleware** for more advanced routing.

Filtering Requests by HTTP Method in Express.js

Filtering Requests by HTTP Method

1. **Problem**:
   - **app.use()** executes for **all HTTP methods** (e.g., GET, POST).
   - Example: A middleware for /product would run for both GET and POST requests.

2. **Solution**:
   - Use **app.get()** and **app.post()** to filter requests by HTTP method.
   - **app.get(path, callback)**: Executes only for **GET requests**.
   - **app.post(path, callback)**: Executes only for **POST requests**.

3. **Example**:

Javascript:

```
// Handle GET requests to /product
app.get('/product', (req, res) => {
  res.send('This is a GET request to /product');
});

// Handle POST requests to /product
app.post('/product', (req, res) => {
  console.log(req.body); // Log parsed request body
  res.redirect('/');
});
```

Key Concepts

1. **HTTP Methods**:
    - **GET**: Used to retrieve data.
    - **POST**: Used to submit data.
    - Other methods: **DELETE**, **PATCH**, **PUT** (used for APIs, not HTML forms).

2. **Method-Specific Middleware**:
    - **app.get()**: Filters for GET requests.
    - **app.post()**: Filters for POST requests.
    - **app.use()**: Executes for all HTTP methods.

3. **Request Body Parsing**:
    - Use **body-parser** to parse incoming request bodies for POST requests.
    - Example:

```
Javascript:

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

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

Example Code: Handling GET and POST Requests

```
Javascript:

    const express = require('express');

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

    const app = express();


    // Middleware to parse request bodies
```

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

// Route to display the form (GET request)
app.get('/add-product', (req, res) => {
  res.send(`
    <form action="/product" method="POST">
      <input type="text" name="title" placeholder="Product Title">
      <button type="submit">Add Product</button>
    </form>
  `);
});

// Route to handle form submissions (POST request)
app.post('/product', (req, res) => {
  console.log(req.body); // Log the parsed request body
  res.redirect('/');
});

// Default route (GET request)
app.get('/', (req, res) => {
  res.send('<h1>Hello from Express</h1>');
});

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

Key Vocabulary

- **app.get()**: Middleware for handling **GET requests**.
- **app.post()**: Middleware for handling **POST requests**.
- **HTTP Methods**: Actions like GET, POST, DELETE, PATCH, PUT.
- **body-parser**: Middleware to parse incoming request bodies.

Summary

- Use **app.get()** and **app.post()** to filter requests by HTTP method.
- **app.use()** executes for all HTTP methods,
  while **app.get()** and **app.post()** are method-specific.
- Combine **body-parser** with **app.post()** to handle form submissions and parse request bodies.
- Ensure proper routing by placing method-specific middleware before general middleware.

Next Steps:

- Explore handling **DELETE**, **PATCH**, and **PUT** requests for APIs.
- Learn how to serve static files (e.g., CSS, images) in Express.js.

Organizing Routes with Express Router

Why Organize Routes?

- As applications grow, keeping all routes in a single file (e.g., **app.js**) becomes **unmanageable**.
- **Splitting routes** into multiple files improves **readability**, **maintainability**, and **scalability**.
- Common practice: Store route-related code in a **routes** folder.

Using Express Router

1. **What is Express Router?**
    - A **mini Express app** that can be plugged into the main app.
    - Allows defining routes in separate files and exporting them.

2. **Steps to Use Express Router**:
    - **Create a router**:

      Javascript:

      ```
      const express = require('express');
      const router = express.Router();
      ```

    - **Define routes**: Use **router.get()**, **router.post()**, etc., instead of **app.get()**, **app.post()**.
    - **Export the router**:

      Javascript:

      ```
      module.exports = router;
      ```

    - **Import and use the router in the main app**:

```
javascript
const adminRoutes = require('./routes/admin');
```

```
app.use(adminRoutes);
```

Example Code: Organizing Routes

1. **routes/admin.js**:

```
javascript:
     const express = require('express');
     const router = express.Router();

     // GET request to /add-product
     router.get('/add-product', (req, res) => {
       res.send(`
         <form action="/product" method="POST">
           <input type="text" name="title"
placeholder="Product Title">
           <button type="submit">Add Product</button>
         </form>
       `);
     });

     // POST request to /product
     router.post('/product', (req, res) => {
       console.log(req.body); // Log parsed request
body
       res.redirect('/');
     });

     module.exports = router;
```

2. **routes/shop.js**:

   javascript:

```
const express = require('express');
const router = express.Router();

// GET request to /
router.get('/', (req, res) => {
  res.send('<h1>Hello from Express</h1>');
});

module.exports = router;
```

3. **app.js**:

   javascript:

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

// Middleware to parse request bodies
app.use(bodyParser.urlencoded({ extended: false
}));

// Import routes
const adminRoutes = require('./routes/admin');
const shopRoutes = require('./routes/shop');

// Use routes
```

```
app.use(adminRoutes);
app.use(shopRoutes);

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

Key Concepts

1. **Express Router**:
   - A **pluggable mini app** for defining routes in separate files.
   - Use **router.get()**, **router.post()**, etc., to define method-specific routes.

2. **Route Organization**:
   - Store related routes in separate files (e.g., **admin.js**, **shop.js**).
   - Export routers and import them into the main app.

3. **Order of Middleware**:
   - The order in which routes are registered (**app.use()**) matters.
   - **Exact matching** is enforced by **router.get()**, **router.post()**, etc.

Key Vocabulary

- **Express Router**: A feature to define and organize routes in separate files.
- **router.get()**: Method to handle **GET requests** in a router.
- **router.post()**: Method to handle **POST requests** in a router.
- **Middleware Order**: The sequence in which middleware and routes are registered.

Summary

- Use **Express Router** to split routes into separate files for better organization.
- Define routes using **router.get()**, **router.post()**, etc., and export the router.
- Import and use routers in the main app with **app.use()**.
- Pay attention to the **order of middleware** and route registration to ensure proper request handling.

Next Steps:

- Explore **route prefixes** (e.g., /admin/add-product) for better route organization.
- Learn how to handle **404 errors** and other edge cases in Express.js.

Handling 404 Errors in Express.js

Handling Unhandled Routes

1. **Problem**:
    - Requests to **unhandled routes** (e.g., /random-path) result in an error or no response.
    - Need to return a **404 error page** for such routes.

2. **Solution**:
    - Add a **catch-all middleware** at the **bottom** of the middleware stack to handle unhandled routes.
    - Use **app.use()** without a path filter to match all requests.

Key Concepts

1. **Middleware Execution Order**:
    - Requests are funneled through middleware **top-to-bottom**.
    - If no middleware handles the request, it reaches the **catch-all middleware**.

2. **404 Error Handling**:
    - Use **res.status(404)** to set the HTTP status code to **404**.
    - Use **res.send()** to return a custom error page.

Example Code: Adding a 404 Error Page

Javascript:

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

```
const app = express();

// Middleware to parse request bodies
app.use(bodyParser.urlencoded({ extended: false }));

// Import routes
const adminRoutes = require('./routes/admin');
const shopRoutes = require('./routes/shop');

// Use routes
app.use(adminRoutes);
app.use(shopRoutes);

// Catch-all middleware for 404 errors
app.use((req, res) => {
  res.status(404).send(`
    <h1>Page Not Found</h1>
    <p>The page you are looking for does not
exist.</p>
  `);
});

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

Key Vocabulary

- **Catch-all Middleware**: Middleware that handles all unhandled requests.
- **res.status(code)**: Method to set the HTTP status code (e.g., **404** for "Not Found").
- **res.send()**: Method to send a response to the client.
- **Middleware Execution Order**: The sequence in which middleware processes requests.

Summary

- Add a **catch-all middleware** at the **bottom** of the middleware stack to handle **unhandled routes**.
- Use **res.status(404)** to set the status code to **404** and **res.send()** to return a custom error page.
- Ensure the catch-all middleware is placed **after all other routes** to avoid overriding valid routes.

Next Steps:

- Explore serving **static files** (e.g., CSS, images) for a more polished 404 page.
- Learn how to handle **500 errors** (server errors) in Express.js.

Route Prefixing with Express Router

Route Prefixing

1. **Problem**:
    - Routes in a file often share a **common starting path** (e.g., /admin/add-product, /admin/edit-product).
    - Repeating the prefix (e.g., /admin) in every route is **redundant**.

2. **Solution**:
    - Use **route prefixing** to define a common starting path in the main app file (e.g., **app.js**).
    - Example:

      ```
      Javascript:

          app.use('/admin', adminRoutes);
      ```

    - This ensures all routes in **adminRoutes** are prefixed with /admin.

Key Concepts

1. **Route Prefixing**:
    - Adds a **common starting path** to all routes in a router file.
    - The prefix is **stripped** when matching routes in the router file.

2. **How It Works**:
    - A request to /admin/add-product is matched to /add-product in the **adminRoutes** file.
    - The /admin prefix is **ignored** during route matching in the router file.

3. **Benefits**:
    - **Reduces redundancy** by avoiding repetition of the common path in every route.
    - Improves **code organization** and **readability**.

Example Code: Route Prefixing

1. **routes/admin.js**:

```
javascript:

    const express = require('express');

    const router = express.Router();


    // GET request to /add-product (full path:
    /admin/add-product)

    router.get('/add-product', (req, res) => {

      res.send(`

        <form action="/admin/add-product"
    method="POST">

          <input type="text" name="title"
    placeholder="Product Title">

          <button type="submit">Add Product</button>

        </form>

      `);

    });


    // POST request to /add-product (full path:
    /admin/add-product)

    router.post('/add-product', (req, res) => {

      console.log(req.body); // Log parsed request
    body
```

```
  res.redirect('/');
});

module.exports = router;
```

2. **app.js**:

javascript:

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

// Middleware to parse request bodies
app.use(bodyParser.urlencoded({ extended: false
}));

// Import routes
const adminRoutes = require('./routes/admin');
const shopRoutes = require('./routes/shop');

// Use routes with prefixes
app.use('/admin', adminRoutes); // Prefix: /admin
app.use(shopRoutes); // No prefix

// Catch-all middleware for 404 errors
app.use((req, res) => {
  res.status(404).send(`
    <h1>Page Not Found</h1>
```

```
    <p>The page you are looking for does not
exist.</p>
  `);
});

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

Key Vocabulary

- **Route Prefixing**: Adding a common starting path to all routes in a router file.
- **app.use(path, router)**: Method to apply a prefix to all routes in a router.
- **Stripped Prefix**: The prefix is ignored during route matching in the router file.

Summary

- Use **route prefixing** to avoid repeating a common starting path in every route.
- Define the prefix in the main app file (e.g., **app.use('/admin', adminRoutes)**).
- The prefix is **stripped** when matching routes in the router file (e.g., /admin/add-product becomes /add-product).
- Improves **code organization** and reduces redundancy.

Next Steps:

- Explore **nested routers** for more complex route structures.
- Learn how to handle **dynamic routes** (e.g., /products/:id) in Express.js.

Serving HTML Files in Express.js

Serving HTML Files

1. **Problem**:
    - Returning **dummy HTML content** in responses is not scalable or maintainable.
    - Need to serve **real HTML files** for a professional application.

2. **Solution**:
    - Store HTML files in a **views** folder (or any folder of your choice).
    - Use **res.sendFile()** to serve these files in response to requests.

Key Concepts

1. **Folder Structure**:
    - Create a **views** folder to store HTML files (e.g., **shop.html**, **add-product.html**).
    - This aligns with the **MVC (Model-View-Controller)** pattern, where **views** represent the user interface.

2. **HTML Files**:
    - **add-product.html**: Contains a form for adding products.
    - **shop.html**: Displays a list of products (to be populated dynamically later).

3. **Serving Files**:
    - Use **res.sendFile()** to send HTML files as responses.
    - Example:

      Javascript:

```
    res.sendFile(path.join(__dirname, 'views',
'shop.html'));
```

Example Code: Serving HTML Files

1. **Folder Structure**:

```
project/
├── views/
│   ├── shop.html
│   └── add-product.html
├── routes/
│   ├── admin.js
│   └── shop.js
└── app.js
```

2. **views/add-product.html**:

```
html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-
width, initial-scale=1.0">
  <title>Add Product</title>
</head>
<body>
  <header>
    <nav>
      <ul>
```

```
            <li><a href="/">Shop</a></li>
            <li><a href="/add-product">Add
        Product</a></li>
          </ul>
        </nav>
      </header>
      <main>
        <form action="/add-product" method="POST">
          <input type="text" name="title"
        placeholder="Product Title">
          <button type="submit">Add Product</button>
        </form>
      </main>
    </body>
    </html>
```

Run HTML

3. **views/shop.html**:

html:

```
    <!DOCTYPE html>
    <html lang="en">
    <head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-
    width, initial-scale=1.0">
      <title>Shop</title>
    </head>
    <body>
```

```
  <header>
    <nav>
      <ul>
        <li><a href="/">Shop</a></li>
        <li><a href="/add-product">Add
Product</a></li>
      </ul>
    </nav>
  </header>
  <main>
    <h1>My Products</h1>
    <!-- Product list will be added dynamically
later -->
  </main>
</body>
</html>
```

Run HTML

4. **routes/admin.js**:

```
javascript:
    const express = require('express');
    const path = require('path');
    const router = express.Router();

    // GET request to /add-product
    router.get('/add-product', (req, res) => {
      res.sendFile(path.join(__dirname,
    '../views/add-product.html'));
```

```
});

// POST request to /add-product
router.post('/add-product', (req, res) => {
  console.log(req.body); // Log parsed request body
  res.redirect('/');
});

module.exports = router;
```

5. **routes/shop.js**:

   javascript

```
const express = require('express');
const path = require('path');
const router = express.Router();

// GET request to /
router.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, '../views/shop.html'));
});

module.exports = router;
```

6. **app.js**:

   javascript:

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

```
const app = express();

// Middleware to parse request bodies
app.use(bodyParser.urlencoded({ extended: false }));

// Import routes
const adminRoutes = require('./routes/admin');
const shopRoutes = require('./routes/shop');

// Use routes
app.use('/admin', adminRoutes);
app.use(shopRoutes);

// Catch-all middleware for 404 errors
app.use((req, res) => {
  res.status(404).sendFile(path.join(__dirname, 'views', '404.html'));
});

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

Key Vocabulary

- **views Folder**: Stores HTML files representing the user interface.
- **res.sendFile()**: Method to send an HTML file as a response.

- **MVC Pattern**: A design pattern separating an application into **Model**, **View**, and **Controller**.

Summary

- Store HTML files in a **views** folder for better organization.
- Use **res.sendFile()** to serve HTML files in response to requests.
- Align your folder structure with the **MVC pattern** for scalability.
- Serve **add-product.html** and **shop.html** for their respective routes.

Next Steps:

- Add **styling** to the HTML files using CSS.
- Learn how to use **templating engines** (e.g., EJS, Pug) for dynamic content.

Serving HTML Files in Express.js

Serving HTML Files

1. **Problem**:
    - Need to serve **real HTML files** instead of dummy HTML content for a professional application.
2. **Solution**:
    - Use **res.sendFile()** to serve HTML files stored in the **views** folder.
    - Construct the file path using **path.join()** to ensure compatibility across operating systems.

Key Concepts

1. **Folder Structure**:
    - Store HTML files in a **views** folder (e.g., **shop.html**, **add-product.html**).
    - Example:

```
project/
├── views/
│   ├── shop.html
│   └── add-product.html
├── routes/
│   ├── admin.js
│   └── shop.js
└── app.js
```

2. **Serving Files**:
    - Use **res.sendFile()** to send HTML files as responses.
    - Example:

```
javascript

    res.sendFile(path.join(__dirname,
'../views/shop.html'));
```

3. **Path Construction**:
    - Use **path.join()** to build file paths that work on both **Linux** and **Windows**.
    - **__dirname**: A global variable in Node.js that provides the absolute path to the current file's directory.
    - Use **../** to navigate up one directory level.

Example Code: Serving HTML Files

1. **routes/shop.js**:

```
javascript:

    const express = require('express');

    const path = require('path');

    const router = express.Router();


    // GET request to /

    router.get('/', (req, res) => {

      res.sendFile(path.join(__dirname,
'../views/shop.html'));

    });


    module.exports = router;
```

2. **routes/admin.js**:

```
javascript:
```

```
const express = require('express');
const path = require('path');
const router = express.Router();

// GET request to /add-product
router.get('/add-product', (req, res) => {
  res.sendFile(path.join(__dirname, '../views/add-product.html'));
});

// POST request to /add-product
router.post('/add-product', (req, res) => {
  console.log(req.body); // Log parsed request body
  res.redirect('/');
});

module.exports = router;
```

3. **app.js**:

javascript:

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

// Middleware to parse request bodies
app.use(bodyParser.urlencoded({ extended: false }));
```

```
// Import routes
const adminRoutes = require('./routes/admin');
const shopRoutes = require('./routes/shop');

// Use routes
app.use('/admin', adminRoutes);
app.use(shopRoutes);

// Catch-all middleware for 404 errors
app.use((req, res) => {
  res.status(404).sendFile(path.join(__dirname,
'views', '404.html'));
});

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

Key Vocabulary

- **views Folder**: Stores HTML files representing the user interface.
- **res.sendFile()**: Method to send an HTML file as a response.
- **path.join()**: Method to construct file paths compatible across operating systems.
- **__dirname**: Global variable providing the absolute path to the current file's directory.

Summary

- Store HTML files in a **views** folder for better organization.
- Use **res.sendFile()** to serve HTML files in response to requests.
- Construct file paths using **path.join()** and **__dirname** for cross-platform compatibility.
- Serve **shop.html** and **add-product.html** for their respective routes.

Next Steps:

- Add a **404 error page** (e.g., **404.html**) and serve it for unhandled routes.
- Add **styling** to the HTML files using CSS.
- Learn how to use **templating engines** (e.g., EJS, Pug) for dynamic content.

Serving a 404 Error Page in Express.js

Handling 404 Errors

1. **Problem**:
    - Requests to **unhandled routes** (e.g., /random-path) need to return a **404 error page**.

2. **Solution**:
    - Create a **404.html** file in the **views** folder.
    - Use **res.sendFile()** to serve this file for unhandled routes.
    - Set the **HTTP status code** to **404** using **res.status(404)**.

Key Concepts

1. **404 Error Page**:
    - A custom HTML page displayed when a requested resource is not found.
    - Example:

```
html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport"
content="width=device-width, initial-
scale=1.0">
  <title>Page Not Found</title>
</head>
<body>
```

```
    <h1>Page Not Found</h1>
  </body>
  </html>
```

Run HTML

2. **Serving the 404 Page**:
   - Use **res.sendFile()** to serve the **404.html** file.
   - Construct the file path using **path.join()** and **__dirname**.

3. **Setting the Status Code**:
   - Use **res.status(404)** to set the HTTP status code to **404**.

Example Code: Serving a 404 Error Page

1. **Folder Structure**:

```
project/
├── views/
│   ├── shop.html
│   ├── add-product.html
│   └── 404.html
├── routes/
│   ├── admin.js
│   └── shop.js
└── app.js
```

2. **views/404.html**:

```html
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Page Not Found</title>
</head>
<body>
  <h1>Page Not Found</h1>
</body>
</html>
```

Run HTML

3. **app.js**:

javascript

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

// Middleware to parse request bodies
app.use(bodyParser.urlencoded({ extended: false }));

// Import routes
const adminRoutes = require('./routes/admin');
const shopRoutes = require('./routes/shop');
```

```
// Use routes
app.use('/admin', adminRoutes);
app.use(shopRoutes);

// Catch-all middleware for 404 errors
app.use((req, res) => {
  res.status(404).sendFile(path.join(__dirname, 'views', '404.html'));
});

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

Key Vocabulary

- **404 Error Page**: A custom HTML page displayed when a requested resource is not found.
- **res.sendFile()**: Method to send an HTML file as a response.
- **res.status(code)**: Method to set the HTTP status code (e.g., **404** for "Not Found").
- **path.join()**: Method to construct file paths compatible across operating systems.

Summary

- Create a **404.html** file in the **views** folder to handle unhandled routes.
- Use **res.sendFile()** to serve the **404.html** file for unhandled routes.
- Set the **HTTP status code** to **404** using **res.status(404)**.
- Construct the file path using **path.join()** and **__dirname** for cross-platform compatibility.

Next Steps:

- Add **styling** to the **404.html** page using CSS.
- Explore serving **static files** (e.g., CSS, images) in Express.js.
- Learn how to use **templating engines** (e.g., EJS, Pug) for dynamic content.

Path Management in Express.js

Managing Paths to the Root Directory

1. **Problem**:
    - Constructing paths to the **root directory** (e.g., **views** folder) in multiple files can be **repetitive** and **error-prone**.

2. **Solution**:
    - Create a **helper function** to dynamically determine the root directory path.
    - Use **path.dirname()** and **process.mainModule.filename** to get the root directory path.

Key Concepts

1. **Root Directory Path**:
    - The path to the **root folder** of the project (where **app.js** is located).
    - Use **process.mainModule.filename** to get the path to the file that started the application (e.g., **app.js**).
    - Use **path.dirname()** to get the directory name of a file path.

2. **Helper Function**:
    - Create a **util/path.js** file to export the root directory path.
    - Example:

```
javascript

    const path = require('path');

    module.exports =
    path.dirname(process.mainModule.filename);
```

3. **Using the Helper Function**:
    - Import the root directory path in route files (e.g., **shop.js**, **admin.js**).
    - Use it to construct paths to files in the **views** folder.

Example Code: Managing Paths

1. **Folder Structure**:

```
project/
├── util/
│   └── path.js
├── views/
│   ├── shop.html
│   ├── add-product.html
│   └── 404.html
├── routes/
│   ├── admin.js
│   └── shop.js
└── app.js
```

2. **util/path.js**:

```
javascript:
    const path = require('path');
    module.exports =
    path.dirname(process.mainModule.filename);
```

3. **routes/shop.js**:

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

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

const rootDir = require('../util/path');

const router = express.Router();


// GET request to /

router.get('/', (req, res) => {

  res.sendFile(path.join(rootDir, 'views',
'shop.html'));

});


module.exports = router;
```

4. **routes/admin.js**:

   javascript:

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

const path = require('path');

const rootDir = require('../util/path');

const router = express.Router();


// GET request to /add-product

router.get('/add-product', (req, res) => {

  res.sendFile(path.join(rootDir, 'views', 'add-
product.html'));

});


// POST request to /add-product

router.post('/add-product', (req, res) => {
```

```
  console.log(req.body); // Log parsed request
body
  res.redirect('/');
});


module.exports = router;
```

Key Vocabulary

- **Root Directory**: The main folder of the project (e.g., where **app.js** is located).
- **process.mainModule.filename**: Global variable providing the path to the file that started the application.
- **path.dirname()**: Method to get the directory name of a file path.
- **Helper Function**: A reusable function to simplify repetitive tasks (e.g., path construction).

Summary

- Use **process.mainModule.filename** and **path.dirname()** to dynamically determine the root directory path.
- Create a **helper function** in **util/path.js** to export the root directory path.
- Import and use the root directory path in route files to construct paths to the **views** folder.
- This approach ensures **cleaner** and **more maintainable** code.

Next Steps:

- Add **styling** to the HTML files using CSS.
- Explore serving **static files** (e.g., CSS, images) in Express.js.
- Learn how to use **templating engines** (e.g., EJS, Pug) for dynamic content.

Adding CSS Styling to Express.js Applications

Adding CSS Styling

1. **Problem**:
    - **Inline styles** (using <style> tags) in HTML files are **not scalable** or maintainable.
    - Need to serve **external CSS files** for better organization and reusability.
2. **Solution**:
    - Move CSS styles to **external files** (e.g., **main.css**).
    - Serve static files (e.g., CSS, images) using **express.static()**.

Key Concepts

1. **External CSS Files**:
    - Store CSS styles in a **public/css** folder (e.g., **main.css**).
    - Example:

```
css
     /* main.css */
          body {
            font-family: sans-serif;
            margin: 0;
            padding: 0;
          }
          .main-header {
            width: 100%;
            height: 3.5rem;
            background-color: #ffd700;
            padding: 0 1.5rem;
```

```
}
.main-header__nav {
  height: 100%;
  display: flex;
  align-items: center;
}
.main-header__item-list {
  list-style: none;
  margin: 0;
  padding: 0;
  display: flex;
}
.main-header__item {
  margin: 0 1rem;
}
.main-header__item a {
  text-decoration: none;
  color: black;
}
.main-header__item a:hover,
.main-header__item a:active {
  color: #800080;
}
```

2. **Serving Static Files**:
    - Use **express.static()** to serve static files (e.g., CSS, images).
    - Example:

Javascript:

```
app.use(express.static(path.join(__dirname,
'public')));
```

3. **Linking CSS Files**:
   - Link the external CSS file in the HTML <head> section.
   - Example:

     html

     ```
     <link rel="stylesheet" href="/css/main.css">
     ```

     Run HTML

Example Code: Serving Static Files

1. **Folder Structure**:

```
project/
├── public/
│   └── css/
│       └── main.css
├── views/
│   ├── shop.html
│   ├── add-product.html
│   └── 404.html
├── routes/
│   ├── admin.js
│   └── shop.js
└── app.js
```

2. **public/css/main.css**:

```
css
      body {
        font-family: sans-serif;
        margin: 0;
        padding: 0;
      }

      .main-header {
        width: 100%;
        height: 3.5rem;
        background-color: #ffd700;
        padding: 0 1.5rem;
      }

      .main-header__nav {
        height: 100%;
        display: flex;
        align-items: center;
      }

      .main-header__item-list {
        list-style: none;
        margin: 0;
        padding: 0;
        display: flex;
      }
```

```
.main-header__item {
  margin: 0 1rem;
}

.main-header__item a {
  text-decoration: none;
  color: black;
}

.main-header__item a:hover,
.main-header__item a:active {
  color: #800080;
}
```

3. **app.js**:

javascript:

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

// Middleware to parse request bodies
app.use(bodyParser.urlencoded({ extended: false
}));

// Serve static files (e.g., CSS, images)
```

```
app.use(express.static(path.join(__dirname,
'public')));

// Import routes
const adminRoutes = require('./routes/admin');
const shopRoutes = require('./routes/shop');

// Use routes
app.use('/admin', adminRoutes);
app.use(shopRoutes);

// Catch-all middleware for 404 errors
app.use((req, res) => {
  res.status(404).sendFile(path.join(__dirname,
'views', '404.html'));
});

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

4. **views/shop.html**:

```
html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
```

```
  <meta name="viewport" content="width=device-
width, initial-scale=1.0">
  <link rel="stylesheet" href="/css/main.css">
  <title>Shop</title>
</head>
<body>
  <header class="main-header">
    <nav class="main-header__nav">
      <ul class="main-header__item-list">
        <li class="main-header__item"><a href="/"
class="active">Shop</a></li>
        <li class="main-header__item"><a
href="/admin/add-product">Add Product</a></li>
      </ul>
    </nav>
  </header>
  <main>
    <h1>My Products</h1>
  </main>
</body>
</html>
```

Run HTML

Key Vocabulary

- **Static Files**: Files like CSS, images, and JavaScript that don't change dynamically.
- **express.static()**: Middleware to serve static files in Express.js.
- **External CSS**: CSS styles stored in separate files for better organization and reusability.

Summary

- Move CSS styles to **external files** (e.g., **main.css**) for better maintainability.
- Use **express.static()** to serve static files (e.g., CSS, images) from the **public** folder.
- Link external CSS files in the HTML <head> section using **<link>**.
- This approach ensures **cleaner code** and **better scalability**.

Next Steps:

- Add **JavaScript functionality** to the application.
- Explore using **templating engines** (e.g., EJS, Pug) for dynamic content.
- Learn how to handle **file uploads** and **user authentication** in Express.js.

Serving Static Files in Express.js

Why Serve Static Files?

- **Inline CSS/JS** in HTML files is **not scalable** or maintainable.
- **Static files** (CSS, images, JS) need to be served externally for better organization.

Key Concepts

1. **Static Files**:
    - Files like **CSS**, **JavaScript**, and **images** that don't change dynamically.
    - Store in a **public** folder (conventionally named).

2. **express.static() Middleware**:
    - Built-in Express.js middleware to serve static files.
    - Syntax:

      Javascript:

      ```
      app.use(express.static(path.join(__dirname, 'public')));
      ```

    - **public** folder becomes accessible to clients (e.g., /css/main.css maps to public/css/main.css).

3. **Path Handling**:
    - Use **path.join()** to construct OS-agnostic paths.
    - Example:

      Javascript:

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

      app.use(express.static(path.join(__dirname, 'public')));
      ```

Example Code

1. **Folder Structure**:

```
project/
├── public/
│   └── css/
│       ├── main.css
│       └── product.css
├── views/
│   ├── shop.html
│   ├── add-product.html
│   └── 404.html
└── app.js
```

2. **app.js (Middleware Setup)**:

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

    // Serve static files from the 'public' folder
    app.use(express.static(path.join(__dirname,
    'public')));

    // Other middleware and routes...
    app.listen(3000);
```

3. **HTML File Linking**:
    - In **shop.html**:

```
html
    <head>
      <link rel="stylesheet"
    href="/css/main.css">
    </head>
```

Run HTML

    - In **add-product.html**:

```
html
    <head>
      <link rel="stylesheet"
    href="/css/main.css">
      <link rel="stylesheet"
    href="/css/product.css">
    </head>
```

Run HTML

Key Vocabulary

- **express.static()**: Middleware to serve static files (CSS, JS, images).
- **Static Files**: Non-dynamic files served directly to clients.
- **public Folder**: Directory for static files accessible to users.

Summary

- Use **express.static()** to serve static files from the **public** folder.
- Link external CSS/JS files in HTML using relative paths (e.g., **/css/main.css**).

- Improves **code maintainability** and **scalability** by separating static assets.

Express.js Fundamentals

Core Concepts of Express.js

1. **Express.js Overview**:
    - **Express.js** is a **Node.js framework** that builds on Node.js core modules (e.g., **path**).
    - Provides **utility functions** and a **structured approach** to building web applications.

2. **Middleware**:
    - **Middleware functions** process incoming requests and responses.
    - Key components:
        - **req**: The **request object** (incoming data).
        - **res**: The **response object** (outgoing data).
        - **next**: A function to pass control to the **next middleware**.
    - **Rules**:
        - Call **next()** to forward the request to the next middleware.
        - Do **not call next()** if sending a response (e.g., using **res.send()**).

3. **Request Filtering**:
    - Use **app.use(path, callback)** to filter requests by **path**.
    - Use **app.get()**, **app.post()**, etc., to filter requests by **HTTP method**.
    - **Exact matching**: **app.get()** matches paths exactly, while **app.use()** matches the **beginning** of the path.

4. **Express Router**:
    - Use **express.Router()** to split routes into multiple files.

- Export the router and use it in the main app with **app.use()**.
- Example:

```
Javascript:
    const router = express.Router();
    router.get('/path', (req, res) => { ... });
    module.exports = router;
```

5. **Serving Static Files**:
   - Use **express.static()** to serve static files (e.g., CSS, JS, images).
   - Example:

```
Javascript:
    app.use(express.static(path.join(__dirname, 'public')));
```

   - Files in the **public** folder are accessible via URLs (e.g., /css/main.css).

Key Vocabulary

- **Middleware**: Functions that process requests and responses in Express.js.
- **req**: The **request object** containing incoming data.
- **res**: The **response object** used to send data back to the client.
- **next**: Function to pass control to the next middleware.
- **express.Router()**: A mini Express app for organizing routes into separate files.
- **express.static()**: Middleware to serve static files (e.g., CSS, JS, images).

Summary

- **Express.js** is a **Node.js framework** that simplifies web development with middleware and routing.

- **Middleware** processes requests and responses, with **next()** forwarding control to the next middleware.
- Use **app.use()**, **app.get()**, and **app.post()** to filter requests by **path** and **HTTP method**.
- Organize routes with **express.Router()** for better code structure.
- Serve static files (e.g., CSS, JS) using **express.static()**.

Next Steps:

- Learn to **render dynamic content** using templating engines (e.g., EJS, Pug).
- Explore **database integration** (e.g., MongoDB, MySQL) for storing and retrieving data.
- Implement **user authentication** and **session management**.
- Handle **file uploads** and **data validation**.