22-09-2025

- In modern web applications, we need ways to:
    - Manage navigation between pages → **Routing**
    - Handle requests/responses before reaching final logic → **Middleware**
- In React, routing is handled by libraries like **React Router**.
- Middleware is not directly part of React (since React runs on client-side), but it comes into play when using **Node.js/Express as backend** or in **Redux middleware** for state management.

Middleware in React Applications

Definition:

Middleware is software component that sits **between** the request and the response, intercepting and modifying data flow.

In React context:

1. **Redux Middleware** → Intercepts actions before they reach reducers.
2. **Express Middleware** → Often used when React has a backend server.

1. Redux Middleware

- Redux middleware provides a way to **intercept actions** dispatched to the store.
- Common use cases:
    - Logging actions
    - Handling asynchronous operations (API calls)
    - Error handling

Example: Logging Middleware

```
// middleware/logger.js
const logger = store => next => action => {
  console.log('Dispatching:', action);
  let result = next(action);
```

```
  console.log('Next State:', store.getState());
  return result;
};

export default logger;
```

Applying Middleware

```
import { createStore, applyMiddleware } from "redux";
import rootReducer from "./reducers";
import logger from "./middleware/logger";

const store = createStore(rootReducer, applyMiddleware(logger));
```

👉 Every action will now be logged before and after execution.

1. **Dispatch an action** → goes into middleware first.
2. **Logger prints** "Dispatching: <action>".
3. **next(action)** forwards the action to reducer (or next middleware).
4. **Reducer updates state**.
5. **Logger prints** "Next State: <updated state>".
6. Returns result → dispatch works as usual.

👉 In short: **Action → Logger (before log) → Reducer → Logger (after log) → New State**.

2. Express Middleware in MERN stack

If you use React frontend + Express backend:

```
// server.js
const express = require("express");
const app = express();

// middleware example
app.use((req, res, next) => {
  console.log("Request URL:", req.url);
  next(); // pass control to next middleware/route
});

app.get("/", (req, res) => {
  res.send("Hello from Express + React backend");
});

app.listen(5000, () => console.log("Server running"));
```

Here's the **short flow** of your Express code:

1. **Client sends request** → Express receives it.
2. **Middleware runs first**
   - Logs: "Request URL: /".
   - Calls next() to pass control.
3. **Route handler matches** (app.get("/"))
   - Sends response "Hello from Express + React backend".
4. **Server listens** on port 5000.

👉 Flow in one line:
Request → Middleware (log + next) → Route Handler → Response.

Routing in React

Definition:

Routing allows navigation between different components/pages in a **Single Page Application (SPA)** without reloading the browser.

Library: React Router DOM

Install:

npm install react-router-dom

3.1 Basic Routing Example

```
import { BrowserRouter, Routes, Route, Link } from "react-router-dom";


function Home() {
  return <h2>Home Page</h2>;
}


function About() {
  return <h2>About Page</h2>;
}


export default function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link> |
        <Link to="/about">About</Link>
      </nav>


      <Routes>
```

```
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
    </Routes>
  </BrowserRouter>
 );
}
```

👉 **Key Concepts:**

- BrowserRouter – Wrapper that enables routing.
- Routes – Container for all routes.
- Route – Maps a path (/about) to a component.
- Link – Used for navigation without refreshing the page.

🚀 **How to Run This React Router Code**

1. **Create a React project (using Vite recommended)**

```
npm create vite@latest myapp
cd myapp
npm install
```

2. **Install React Router**

```
npm install react-router-dom
```

3. **Replace App.jsx** with your code:

```
import { BrowserRouter, Routes, Route, Link } from "react-router-dom";

function Home() {
 return <h2>Home Page</h2>;
}

function About() {
 return <h2>About Page</h2>;
```

```
}


export default function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link> |
        <Link to="/about">About</Link>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
}
```

4. **Start development server**

npm run dev

Open the given http://localhost:5173/ URL in your browser.

⚡ **Flow of Execution (Short)**

1. **Browser loads App** → Router is initialized.
2. **Navigation bar renders** with Home & About links.
3. When user clicks:
    - / → <Home /> component displays → **"Home Page"**.
    - /about → <About /> component displays → **"About Page"**.
4. All navigation happens **without page reload** (SPA behavior).

👉 In short:
App → BrowserRouter → Nav Links → Route Matching → Component Render.

3.2 Dynamic Routing Example

```
import { useParams } from "react-router-dom";

function User() {
  let { id } = useParams();
  return <h2>User Profile: {id}</h2>;
}

<Routes>
  <Route path="/user/:id" element={<User />} />
</Routes>
```

👉 Visiting /user/101 → Output: **User Profile: 101**

3.3 Nested Routes

```
function Dashboard() {
  return (
    <div>
      <h2>Dashboard</h2>
      <nav>
        <Link to="stats">Stats</Link>
        <Link to="settings">Settings</Link>
      </nav>
      <Outlet /> {/* Renders child routes */}
    </div>
  );
}

<Routes>
```

```
  <Route path="/dashboard" element={<Dashboard />}>
    <Route path="stats" element={<h3>Statistics</h3>} />
    <Route path="settings" element={<h3>Settings</h3>} />
  </Route>
</Routes>
```

---

4. Conclusion

- **Middleware** is an intermediate layer for logging, API handling, or security (Redux/Express).
- **Routing** in React (via React Router DOM) helps build **SPA navigation** with static, dynamic, and nested routes.
- Together, they make applications **scalable, modular, and user-friendly**.

✅ T**asks for students**:

1. Create a logger middleware in Redux.
2. Build a small React app with Home, About, and Contact pages using React Router.
3. Implement a dynamic user profile page with /user/:id.

The Request and Response Objects in React

Introduction

- In web development, **client (frontend)** and **server (backend)** communicate using **HTTP requests and responses**.
- React, being a **client-side library**, relies on APIs (backend services) to fetch or send data.
- The **request object** contains information we send to the server, while the **response object** contains what the server sends back.

The Request Object

- Represents the **information sent from React to the server**.
- Includes:
    1. **URL/Endpoint** – the address of the server resource.
    2. **HTTP Method** – GET, POST, PUT, DELETE, etc.
    3. **Headers** – metadata (e.g., Content-Type, Authorization).
    4. **Body (Payload)** – data sent to the server (usually JSON in React apps).

Example (React using Fetch – Sending a POST request):

```
function sendData() {
  fetch("https://api.example.com/users", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({
      name: "Ishan",
      email: "ishan@example.com",
    }),
  });
```

}

👉 Here:

- Request **method**: POST
- Request **headers**: Content-Type → application/json
- Request **body**: {name, email}

3. The Response Object

- Represents the **data returned by the server** after a request.
- Contains:
    1. **Status Code** (200, 404, 500, etc.)
    2. **Headers** (e.g., Content-Type: application/json)
    3. **Body** – the actual data (often JSON for APIs).

Example (Handling Response in React):

```
function fetchData() {
  fetch("https://api.example.com/users")
    .then(response => {
      if (!response.ok) {
        throw new Error("Network response was not ok");
      }
      return response.json();  // parsing response body
    })
    .then(data => {
      console.log("Data received:", data);
    })
    .catch(error => {
      console.error("Fetch error:", error);
    });
}
```

👉 Here:

- Response **status** checked with response.ok.
- Response **body** parsed as JSON.
- Data handled in .then().

4. Lifecycle of Request–Response in React

1. **User Action** (e.g., clicking a button, submitting a form).
2. **React Makes a Request** (via fetch or axios).
3. **Server Processes Request** and returns a response.
4. **React Handles Response** (updates state/UI accordingly).

5. Practical Example (React Component)

```
import { useState, useEffect } from "react";

function Users() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(res => res.json())  // response object → JSON
      .then(data => setUsers(data)); // update state with response data
  }, []);

  return (
    <div>
      <h2>Users List</h2>
      <ul>
        {users.map(u => <li key={u.id}>{u.name}</li>)}
```

```
    </ul>
  </div>
 );
}

export default Users;
```

👉 Flow:

- Request → fetch users
- Response → JSON data
- React updates **state** → UI re-renders

6. Key Points for Students

- React does not create request/response objects itself; it uses **browser APIs** (fetch) or libraries (axios).
- **Request Object** = what we send (method, headers, body).
- **Response Object** = what we receive (status, headers, body).
- Always handle **errors** and **loading states** for good UX.

✅ **Conclusion:**
In React, handling request and response objects is essential for working with APIs. Requests carry data from frontend to backend, and responses bring data back. Understanding both helps in building dynamic, data-driven applications.

🚀 How to Create and Run React App (Request & Response Example)

1. Create a New React Project

We'll use **Vite** (faster than CRA). Run these commands in your terminal:

```
# 1. Create a new project
npm create vite@latest myapp

# 2. Go inside project folder
cd myapp

# 3. Install dependencies
npm install

# 4. Start development server
npm run dev
```

👉 It will show a local URL like http://localhost:5173 in your terminal.
Open it in your browser.

---

2. Project File Structure

Inside your project folder, you'll see:

```
myapp/
 ├ src/
 |  ├ App.jsx
 |  ├ main.jsx
 |  └ index.css
 ├ package.json
 └ vite.config.js
```

We'll put our **Request-Response example** inside App.jsx.

---

3. Add Request–Response Code

Open src/App.jsx and replace everything with this code:

```
import { useState, useEffect } from "react";


function App() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);


  useEffect(() => {
    // Request: fetch users data
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(response => {
        if (!response.ok) {
          throw new Error("Request failed with status " + response.status);
        }
        return response.json(); // Response body → JSON
      })
      .then(data => {
        setUsers(data); // Save response data in state
        setLoading(false);
      })
      .catch(error => {
        console.error("Error fetching data:", error);
        setLoading(false);
      });
  }, []);
```

```
  return (
    <div>
      <h1>Users List (Request & Response Example)</h1>

      {loading ? (
        <p>Loading...</p>
      ) : (
        <ul>
          {users.map(user => (
            <li key={user.id}>
              {user.name} ({user.email})
            </li>
          ))}
        </ul>
      )}
    </div>
  );
}

export default App;
```

---

4. Run the App

Now start the server:

npm run dev

👉 Open the link shown in terminal (http://localhost:5173).
You'll see:

- **Loading...** (initially, while request is being made)
- Then the **list of users** (response data from API).

---

5. Flow of Execution

1. React app starts.
2. useEffect runs → sends **HTTP Request** to API.
3. Server responds with **Response Object** (JSON data).
4. Response is parsed → saved in users state.
5. React re-renders → UI updates with user list.