# React – Json-Server And Firebase Real Time Database,Routing

## Routing in React (React Router)

### What is React Router? How does it handle routing in single-page applications?

**React Router** is a popular library used with React to handle **client-side routing** in **single-page applications (SPAs)**. In SPAs, the app loads a single HTML file, and navigation between different “pages” happens dynamically **without reloading the browser**.

**How React Router works:**

- It **listens to URL changes** in the browser (like clicking a link or changing the address bar).
- Instead of letting the browser send a new request to the server, React Router **intercepts the URL change**, **renders the appropriate React component**, and **updates the DOM**.
- This keeps the user experience fast and seamless, without a full page reload.

### Difference between BrowserRouter, Route, Link, and Switch components

✅ **BrowserRouter**

- A **router provider component** that uses the HTML5 history API (pushState, popstate) to keep the UI in sync with the URL.
- It **wraps your entire app** (or the parts that need routing) so routing context is available.

Example:-

```
import { BrowserRouter } from 'react-router-dom';

<BrowserRouter>

  <App />

</BrowserRouter>
```

✅ **Route**

- Defines **what component should render for a specific path**.
- Takes path and element (or component in older versions) props.

Example:-

```
<Route path="/about" element={<About />} />
```

✅ **Link**

- A React component that renders as an **anchor tag (<a>)**, but prevents a full page reload.
- Uses client-side navigation: changes the URL and lets React Router render the new component.

Example:-

```
<Link to="/about">Go to About</Link>
```

✅ **Switch** (React Router v5) / **Routes** (React Router v6+)

- In **v5**, Switch renders **only the first matching <Route>** among its children.
- In **v6**, Switch was replaced by Routes, which automatically renders the best match and has a slightly different API.

Example:-

```
<Switch>

  <Route path="/about" component={About} />

  <Route path="/contact" component={Contact} />

</Switch>
```

# React – JSON-server and Firebase Real Time Database

### What do you mean by RESTful web services?

RESTful web services are APIs that follow **REST (Representational State Transfer)** principles:

- They use standard HTTP methods (**GET**, **POST**, **PUT**, **DELETE**) to perform CRUD operations on resources.
- Data is often exchanged in **JSON** format.
- Resources are identified by unique URLs.
- They are **stateless**, meaning each request is independent and contains all necessary information.

### What is Json-Server? How do we use it in React?

**Json-Server** is a **mock REST API** tool that lets you quickly spin up a full fake API using a simple db.json file.

**How to use in React:**

1. Install globally:

```
npm install -g json-server
```

2.Create a db.json file with sample data.

3.Start server:

```
json-server --watch db.json --port 3001
```

4.In React, fetch data from http://localhost:3001/your-endpoint.

## How do you fetch data from a Json-server API in React? Role of fetch() or axios()

You fetch data using:

- **fetch()** (built-in browser API) or
- **axios()** (third-party library with simpler syntax and extra features).

**Example using fetch():**

## How do you fetch data from a Json-server API in React? Role of fetch() or axios()

You fetch data using:

- **fetch()** (built-in browser API) or
- **axios()** (third-party library with simpler syntax and extra features).

**Example using fetch():**

```
useEffect(() => {

  fetch('http://localhost:3001/users')

    .then(res => res.json())

    .then(data => setUsers(data))

    .catch(err => console.error(err));

}, []);
```

**Example using axios():**

```
import axios from 'axios';

useEffect(() => {
  axios.get('http://localhost:3001/users')
    .then(res => setUsers(res.data))
    .catch(err => console.error(err));
}, []);
```

### What is Firebase? What features does it offer?

**Firebase** is a **Backend-as-a-Service (BaaS)** platform by Google, used to build web and mobile apps.

**Key features:**

- **Authentication:** User login/signup with email, Google, etc.
- **Firestore/Realtime Database:** Cloud-based NoSQL databases.
- **Cloud Storage:** Upload/store files.
- **Cloud Functions:** Serverless backend logic.
- **Hosting:** Deploy web apps.
- **Analytics, Push Notifications, Crash Reporting**, and more.

### Importance of handling errors and loading states in React when working with APIs

When fetching data:

- APIs can fail (network issues, bad responses). **Error handling** lets you show meaningful messages to users instead of a broken app.
- **Loading states** keep users informed while data is being fetched — improves user experience and avoids showing empty components.

Example:

```
const [loading, setLoading] = useState(true);
const [error, setError] = useState(null);
```

```
useEffect(() => {
  fetch('/api/data')
    .then(res => res.json())
    .then(data => {
      setData(data);
      setLoading(false);
    })
    .catch(err => {
      setError(err.message);
      setLoading(false);
    });
}, []);
```

# Context API

## What is the Context API in React? How is it used to manage global state across multiple components?

The **Context API** in React is a built-in way to **share global data** (like user authentication, theme, or language settings) between multiple components **without prop drilling** (passing props manually through every level).

**How it works:**

- You create a **context object** using createContext().
- Provide this context at a higher level in your component tree using a **Provider** component.
- Consume (read) the context data in nested components using useContext() or <Context.Consumer>.

This makes it easier to manage and update shared state across many components in a clean way.

## How createContext() and useContext() are used in React for sharing state

- **createContext()**: Creates a context object that holds your global state and provides two components:
  - **Provider:** Makes state available to child components.
  - **Consumer:** (or useContext) Lets you read the context in any child component.
- **useContext()**: A React hook used inside functional components to **access the current context value**.

**Example:**

```
import React, { createContext, useContext, useState } from 'react';

//         Create Context

const ThemeContext = createContext();

//         Create Provider

export const ThemeProvider = ({ children }) => {

  const [theme, setTheme] = useState('light');

  return (

    <ThemeContext.Provider value={{ theme, setTheme }}>

      {children}

    </ThemeContext.Provider>

  );

};

//         Consume Context in a child component

const ThemeSwitcher = () => {

  const { theme, setTheme } = useContext(ThemeContext);

 return (

    <button onClick={() => setTheme(theme === 'light' ? 'dark' : 'light')}>

      Current theme: {theme}

    </button>

  );
};

//        Use Provider in App

export default function App() {

  return (

    <ThemeProvider>

      <ThemeSwitcher />

    </ThemeProvider>

  );

}
```

## State Management (Redux, Redux-Toolkit or Recoil)

### What is Redux, and why is it used in React applications? Core concepts: actions, reducers, store

**Redux** is a **state management library** often used with React to handle **global application state** in a predictable way.

**Why use Redux in React apps?**

- Helps manage complex state shared across many components.
- Makes state updates predictable via **pure functions (reducers)**.
- Provides powerful dev tools (time-travel debugging, centralized state inspection).

**Core Concepts:**
✅ **Store:**

- A single **JavaScript object** holding the entire application state.
- Created using createStore() (or configureStore() in Redux Toolkit).

✅ **Actions:**

- Plain JavaScript objects describing **what happened** (type + payload).
- Example: { type: 'INCREMENT' }.

✅ **Reducers:**

- Pure functions that **take current state + action → return new state**.
- Example:

```
function counterReducer(state = 0, action) {
  switch(action.type) {

    case 'INCREMENT': return state + 1;

    default: return state;

  }

}
```

## How does Recoil simplify state management in React compared to Redux?

**Recoil** is a state management library built **specifically for React**, offering simpler and more **React-native** patterns than Redux:

✅ **Key differences & advantages:**

- No boilerplate actions/reducers — instead, you define **atoms** (pieces of state) and **selectors** (derived/computed state).
- Works **seamlessly with React hooks** — you read/write state using useRecoilState() or useRecoilValue().
- Supports **derived state** and **async queries** (via selectors) out of the box.
- State is **decentralized** (multiple atoms vs. one big store), making it easier to split and scale.

**Example vs Redux:**

- **Redux:** Setup involves actions, reducers, dispatch, middleware.
- **Recoil:** You create an atom, then directly use it with hooks in components.