React Theory Questions

Routing in React (React Router)

Question 1: What is React Router? How does it handle routing in single-page applications?

React Router is a declarative routing library for React applications that enables navigation between different components/pages without page refreshes. It's the standard routing solution for React single-page applications (SPAs).

How it handles routing in SPAs:

- **Client-side routing**: React Router intercepts browser navigation events and renders different components based on the URL, without making server requests
- **History API**: Uses the browser's History API to manipulate the URL and maintain navigation history
- **Component-based routing**: Maps URL paths to React components, allowing conditional rendering based on the current route
- **Nested routing**: Supports hierarchical routing structures where routes can have child routes
- **Dynamic routing**: Routes are rendered as components, making them dynamic and allowing for programmatic navigation

The router listens to URL changes and renders the appropriate component tree while maintaining the SPA experience where only the content changes, not the entire page.

Question 2: Explain the difference between BrowserRouter, Route, Link, and Switch components in React Router.

BrowserRouter:

- Top-level router component that uses HTML5 history API
- Provides routing context to all child components
- Manages browser history and URL synchronization
- Should wrap your entire application or the routing section

Route:

- Defines a mapping between a URL path and a component
- Renders a component when the current URL matches its path
- Can accept props like path, component, render, and exact

- Supports path parameters and query strings

Link:

- Replaces anchor tags (<a>) for navigation within the SPA
- Prevents full page reloads and uses client-side routing
- Accepts a to prop that specifies the destination path
- Maintains SPA behavior while updating the URL

Switch (replaced by Routes in v6):

- Renders only the first Route that matches the current location
- Prevents multiple routes from rendering simultaneously
- Ensures exclusive routing - only one route component renders at a time
- In React Router v6, Switch has been replaced by Routes

React – JSON-server and Firebase Real Time Database

Question 1: What do you mean by RESTful web services?

RESTful web services are web services that follow the REST (Representational State Transfer) architectural style. They are designed around resources and use standard HTTP methods for communication.

Key characteristics:

- **Stateless**: Each request contains all information needed to process it
- **Resource-based**: Everything is treated as a resource with unique URLs
- **HTTP methods**: Uses standard HTTP verbs (GET, POST, PUT, DELETE, PATCH)
- **Uniform interface**: Consistent way of interacting with resources
- **JSON/XML**: Typically uses JSON for data exchange
- **Cacheable**: Responses can be cached for better performance

HTTP methods in REST:

- GET: Retrieve data
- POST: Create new resources
- PUT: Update entire resources

- PATCH: Partial updates
- DELETE: Remove resources

Question 2: What is Json-Server? How we use in React?

JSON Server is a simple tool that creates a full fake REST API from a JSON file in seconds. It's primarily used for prototyping, testing, and development.

Features:

- Creates RESTful endpoints automatically
- Supports GET, POST, PUT, PATCH, DELETE operations
- Provides filtering, sorting, and pagination
- Zero configuration required
- Perfect for frontend development when backend isn't ready

How to use in React:

1. **Installation:**

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

2. **Create db.json:**

```
{
  "users": [
    { "id": 1, "name": "John", "email": "john@example.com" },
    { "id": 2, "name": "Jane", "email": "jane@example.com" }
  ]
}
```

3. **Start server:**

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

4. **Use in React:**

- Automatically creates endpoints like /users, /users/1
- Supports full CRUD operations
- Can be consumed using fetch() or axios

Question 3: How do you fetch data from a Json-server API in React? Explain the role of fetch() or axios() in making API requests.

Using fetch():

```
// GET request
const fetchUsers = async () => {
  try {
    const response = await fetch('http://localhost:3001/users');
    const data = await response.json();
    setUsers(data);
  } catch (error) {
    console.error('Error:', error);
  }
};

// POST request
const createUser = async (userData) => {
  try {
    const response = await fetch('http://localhost:3001/users', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(userData),
    });
    const newUser = await response.json();
    return newUser;
  } catch (error) {
    console.error('Error:', error);
  }
};
```

Using axios:

```
import axios from 'axios';


// GET request
const fetchUsers = async () => {
  try {
    const response = await axios.get('http://localhost:3001/users');
    setUsers(response.data);
  } catch (error) {
    console.error('Error:', error);
  }
};


// POST request
const createUser = async (userData) => {
  try {
    const response = await axios.post('http://localhost:3001/users', userData);
    return response.data;
  } catch (error) {
    console.error('Error:', error);
  }
};
```

Role of fetch() and axios:

- **fetch()**: Built-in browser API for making HTTP requests
- **axios**: Third-party library with additional features
- Both handle asynchronous HTTP requests
- Convert responses to usable JavaScript objects
- Support various HTTP methods and configurations

- Enable communication between React frontend and APIs

Question 4: What is Firebase? What features does Firebase offer?

Firebase is Google's Backend-as-a-Service (BaaS) platform that provides various backend services for web and mobile applications.

Key features:

- **Realtime Database**: NoSQL database with real-time synchronization
- **Cloud Firestore**: Flexible, scalable NoSQL document database
- **Authentication**: User authentication with multiple providers
- **Cloud Storage**: File storage and serving
- **Hosting**: Web hosting for static and dynamic content
- **Cloud Functions**: Serverless functions
- **Analytics**: App usage analytics
- **Cloud Messaging**: Push notifications
- **Performance Monitoring**: App performance insights
- **Crashlytics**: Crash reporting and analysis

Benefits:

- Real-time data synchronization
- Offline support
- Automatic scaling
- Security rules
- Easy integration with web and mobile apps
- No server management required

Question 5: Discuss the importance of handling errors and loading states when working with APIs in React

Error Handling:

- **User Experience**: Provides meaningful feedback when something goes wrong
- **Debugging**: Helps identify and fix issues during development
- **Graceful degradation**: App continues to function even when API calls fail
- **Security**: Prevents sensitive error information from being exposed

Loading States:

- **User Feedback**: Indicates that data is being fetched
- **Prevents confusion**: Users know the app is working
- **Improves perceived performance**: Makes app feel more responsive
- **Prevents multiple requests**: Disables buttons during API calls

Implementation example:

```
const [data, setData] = useState(null);
const [loading, setLoading] = useState(false);
const [error, setError] = useState(null);


const fetchData = async () => {
  setLoading(true);
  setError(null);
  try {
    const response = await fetch('/api/data');
    if (!response.ok) throw new Error('Failed to fetch');
    const result = await response.json();
    setData(result);
  } catch (err) {
    setError(err.message);
  } finally {
    setLoading(false);
  }
};
```

Context API

Question 1: What is the Context API in React? How is it used to manage global state across multiple components?

Context API is React's built-in solution for managing global state and sharing data across multiple components without prop drilling.

Purpose:

- Eliminates prop drilling (passing props through multiple component levels)
- Provides a way to share state globally
- Creates a "global" state that any component can access
- Reduces component coupling

How it manages global state:

- Creates a context that holds shared data
- Provides the data through a Provider component
- Allows any nested component to consume the data
- Updates to context trigger re-renders in consuming components

Use cases:

- User authentication state
- Theme preferences
- Language/locale settings
- Shopping cart data
- Any data needed by multiple components

Question 2: Explain how createContext() and useContext() are used in React for sharing state.

createContext():

- Creates a new context object
- Returns an object with Provider and Consumer components
- Takes an optional default value parameter

useContext():

- Hook that allows components to consume context values

- Takes a context object as parameter
- Returns the current context value

Implementation example:

```
// 1. Create Context
const ThemeContext = createContext();

// 2. Create Provider Component
const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState('light');

  const toggleTheme = () => {
    setTheme(prev => prev === 'light' ? 'dark' : 'light');
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};

// 3. Consume Context in Components
const ThemedButton = () => {
  const { theme, toggleTheme } = useContext(ThemeContext);

  return (
    <button
      onClick={toggleTheme}
```

```
      style={{
        backgroundColor: theme === 'light' ? 'white' : 'black',
        color: theme === 'light' ? 'black' : 'white'
      }}
    >
      Toggle Theme
    </button>
  );
};

// 4. Wrap App with Provider
const App = () => (
  <ThemeProvider>
    <ThemedButton />
  </ThemeProvider>
);
```

State Management (Redux, Redux-Toolkit, Recoil)

Question 1: What is Redux, and why is it used in React applications? Explain the core concepts of actions, reducers, and the store.

Redux is a predictable state container for JavaScript applications, commonly used with React for managing application state.

Why Redux is used:

- **Predictable state updates**: Unidirectional data flow
- **Centralized state**: Single source of truth
- **Debugging**: Time-travel debugging with Redux DevTools
- **Testing**: Pure functions make testing easier

- **Large applications**: Better state management for complex apps
- **State persistence**: Easy to persist and rehydrate state

Core concepts:

1. Store:

- Single object that holds the entire application state
- Created using createStore() or configureStore()
- Provides methods like getState(), dispatch(), and subscribe()

2. Actions:

- Plain JavaScript objects that describe what happened
- Must have a type property
- Can contain additional data (payload)
- Action creators are functions that return actions

```
// Action
const incrementAction = { type: 'INCREMENT' };


// Action creator
const increment = (amount) => ({
  type: 'INCREMENT',
  payload: amount
});
```

3. Reducers:

- Pure functions that specify how state changes in response to actions
- Take current state and action as parameters
- Return new state object
- Must not mutate the original state

```
const counterReducer = (state = { count: 0 }, action) => {
  switch (action.type) {
    case 'INCREMENT':
```

```
    return { count: state.count + (action.payload || 1) };
  case 'DECREMENT':
    return { count: state.count - 1 };
  default:
    return state;
 }
};
```

Data flow:

1. Component dispatches an action
2. Store passes action to reducer
3. Reducer returns new state
4. Store updates and notifies subscribers
5. Components re-render with new state

Question 2: How does Recoil simplify state management in React compared to Redux?

Recoil is Facebook's experimental state management library designed specifically for React applications.

How Recoil simplifies state management:

1. Atomic State:

- State is broken into small, independent atoms
- Components subscribe only to atoms they need
- Reduces unnecessary re-renders

2. Less Boilerplate:

- No actions, reducers, or complex setup
- Direct state updates using hooks
- More intuitive for React developers

3. Built-in Async Support:

- Native support for async operations
- Selectors can be async

- Built-in loading and error states

4. Component-level State:

- State can be local to component trees
- No need for global state containers
- Better encapsulation

Comparison:

Redux:

```
// Action
const increment = () => ({ type: 'INCREMENT' });

// Reducer
const counterReducer = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    default:
      return state;
  }
};

// Component
const Counter = () => {
  const count = useSelector(state => state.counter);
  const dispatch = useDispatch();

  return (
    <div>
      <span>{count}</span>
```

```
      <button onClick={() => dispatch(increment())}>
        Increment
      </button>
    </div>
  );
};
```

Recoil:

```
// Atom
const countState = atom({
  key: 'countState',
  default: 0,
});

// Component
const Counter = () => {
  const [count, setCount] = useRecoilState(countState);

  return (
    <div>
      <span>{count}</span>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </div>
  );
};
```

Recoil advantages:

- More React-like API

- Better performance with granular subscriptions
- Easier async state management
- Less ceremony and boilerplate
- Built-in developer tools
- Time-travel debugging capabilities

When to choose:

- **Redux**: Large, complex applications with predictable state patterns
- **Recoil**: React-specific apps wanting simpler state management with modern features