React

Reading Material

PW SKILLS

Topics Covered

- Understanding the Concept
  - React
    - Advantages
    - Limitations
  - JSX
  - Virtual DOM
  - Components in React
  - class components and functional components.
    - Props and State
    - React Hooks
  - Event Handling in React
  - Handling API Integration
    - Axios
    - Fetch
  - State Management
    - Redux toolkit
    - Context API
  - Routing in React
    - React router DOM
    - Optimize React app

Understanding the Concept

React

React is a JavaScript library for creating user interfaces, great for single-page apps. It's component-based, meaning you build UI pieces you can reuse. Key features include server-side rendering, using virtual DOM for faster updates, and unidirectional data flow.

Advantages

MVC stands for Model View Controller:-

- React utilizes a virtual DOM for rendering, enhancing efficiency by creating a virtual representation of the real DOM. This speeds up UI rendering compared to traditional methods.
- React offers a gentle learning curve, making it accessible for those with basic JavaScript knowledge to build web applications.
- React is SEO-friendly, supporting server-side rendering and enabling search engine navigation, enhancing app visibility.
- React's component-based architecture promotes reusable and independent code components, accelerating the development pace.
- React boasts a vast ecosystem of libraries, empowering developers to choose tools and libraries tailored to their project needs.

Limitations:-

- The learning curve can be steep for complex features.
- React focuses only on the view layer, needing extra tools for full app development.
- Mixing HTML and JavaScript in JSX might confuse some developers.
- Large apps may suffer performance issues due to virtual DOM.
- Routing and form handling require extra libraries or custom solutions.
- Managing a global state can get messy without tools like Redux.
- Accessibility features need manual implementation.
- Limited built-in support for server-side rendering compared to some frameworks.

JSX

JSX, short for JavaScript XML, lets us write HTML directly inside JavaScript. It's like a shortcut that replaces React.createElement() with a more readable syntax. Instead of creating elements with React.createElement(), we can simply use HTML-like syntax in our JavaScript code. This makes it easier to create and manage UI components in React.

Virtual Dom

The **Virtual DOM (VDOM)** is an in-memory representation of **Real DOM**. The representation of a UI is kept in memory and synced with the "real" DOM. It's a step that happens between the render function being called and the displaying of elements on the screen. This entire process is called **reconciliation**.

Components in React

components are the building blocks of user interfaces. There are two main types of components:

class components and functional components.

Class Components:

- Class components are ES6 classes that extend from React. Component.
- They have a render() method that returns the JSX (JavaScript XML) to define the UI.
- Class components have access to lifecycle methods like componentDidMount, componentDidUpdate, etc.
- State and lifecycle methods can be used in class components.

Example:

```
import React, { Component } from 'react';

class MyClassComponent extends Component {
  render() {
    return <h1>Hello, I'm a class component!</h1>;
  }
}
export default MyClassComponent;
```

Functional Components:

- Functional components are JavaScript functions that return JSX.
- They are simpler and more lightweight compared to class components.
- Functional components are also known as stateless or presentational components.
- With the introduction of React Hooks, functional components can now manage state and use lifecycle features.

Example:

```
import React from 'react';

const MyFunctionalComponent = () => {
  return <h1>Hello, I'm a functional component!</h1>;
};
export default MyFunctionalComponent;
```

Props and State

Props:

Props (short for properties) are a way of passing data from parent to child components in React. They are immutable, meaning they cannot be modified by the child component. Props are passed down the component tree in a unidirectional flow.

- Props are like messages sent from parent components to child components through JSX attributes.
- Props are like read-only data for child components; they can't be altered once received.
- In functional components, props are directly accessible as arguments; in class components, they're accessed via this.props.
- Whenever props change, the child component re-renders to reflect the new data.
- React follows a one-way data flow, ensuring data moves from parent to child components via props, promoting a predictable structure.

State:

State is a built-in feature of React components that represents the mutable data managed by the component itself. Unlike props, which are passed down from parent components, the state is managed internally by the component and can be updated using setState() method.

- Each component manages its own state separately.
- The state can be updated using special functions provided by React.
- Typically set up in class components either in the constructor or as a class property.
- Functional components can now have a state too, thanks to hooks like useState().
- Any change to state causes the component to rerender and update the UI.
- State is local, meaning it's confined to the component where it's defined and doesn't affect other components.

React Hooks

React Hooks are functions that enable functional components to use state and lifecycle features previously only available in class components. Introduced in React 16.8, Hooks provides a more straightforward and concise way to manage component state and side effects in functional components, making them more powerful and expressive.

useState: Allows functional components to manage state.

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

function Counter() {
  const [count, setCount] = useState(0);

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

useEffect: Performs side effects in functional components.

```
import React, { useEffect, useState } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}
>Increment</button>
    </div>
  );
}
```

useContext: Allows functional components to consume context.

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

const ThemeContext = React.createContext('light');

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button>{theme}</button>;
}
}
```

useReducer: Manages state in a more complex manner.

```
import React, { useReducer } from 'react';

const initialState = { count: 0 };
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}
function Counter() {
  const [state, dispatch] = useReducer(reducer,
initialState);

  return (
    <div>
      Count: {state.count}
      <button onClick={() => dispatch({ type: 'increment'
})}>+</button>
      <button onClick={() => dispatch({ type: 'decrement'
})}>-</button>
    </div>
  );
```

useCallback: Memoizes callback functions to prevent unnecessary re-renders.

```
import React, { useState, useCallback } from 'react';

function MemoizedComponent() {
  const [count, setCount] = useState(0);
  const handleClick = useCallback(() => {
    setCount(count + 1);
  }, [count]);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
}
```

useMemo: Memoizes expensive calculations to optimize performance.

```
import React, { useState, useMemo } from 'react';

function ExpensiveComponent() {
  const [count, setCount] = useState(0);
  const expensiveCalculation = useMemo(() => {
    // Perform expensive calculation here
    return count * 2;
  }, [count]);

  return (
    <div>
      <p>Expensive Calculation: {expensiveCalculation}</p>
      <button onClick={() => setCount(count + 1)}
>Increment</button>
    </div>
  );
}
```

useRef: Allows access to a mutable ref object.

```
import React, { useRef } from 'react';

function TextInputWithFocusButton() {
  const inputRef = useRef();

  const handleClick = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={handleClick}>Focus Input</button>
    </div>
  );
}
```

useLayoutEffect: Synchronizes with the DOM after all updates.

```
import React, { useLayoutEffect, useState } from 'react';

function LayoutEffectExample() {
  const [width, setWidth] = useState(0);

  useLayoutEffect(() => {
    setWidth(document.documentElement.clientWidth);
  }, []);
 return <div>Window Width: {width}</div>;
}
```

Event Handling in React:-

React provides a way to handle user interactions, like clicks or inputs, using event handlers. Just like in regular HTML, you can attach event listeners to React elements using special attributes like onClick or onChange.

When an event occurs, React invokes the corresponding event handler function. You define these event handler functions within your component, and they can perform actions like updating state or triggering other functions.

React's synthetic event system ensures consistent behavior across different browsers and normalizes event handling.

- **Handling API Integration**

Axios:

Description: Axios is a popular Promise-based HTTP client for making asynchronous HTTP requests in JavaScript environments, particularly in web browsers and Node.js.

Features:

- Simplified API for sending HTTP requests and handling responses.
- Built-in support for interceptors, allowing you to modify request and response configurations globally.
- Automatic transformation of request and response data between JSON and JavaScript objects.
- Automatic handling of common HTTP errors.
- Support for cancellation of requests.

```
import axios from 'axios';

axios.get('/api/data')
.then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error('Error fetching data:', error);
  });
```

Fetch:

1. **Description:** Fetch is a modern API for making HTTP requests introduced in ES6 and built into web browsers. It provides a more powerful and flexible alternative to XMLHttpRequest (XHR).
2. **Features:**

- Built-in browser API for making asynchronous network requests.
- Promise-based interface for handling responses.
- Supports modern features like streaming responses, request and response headers, and FormData.
- No need for external libraries or dependencies.

```
fetch('/api/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error('Error fetching data:', error);
  });
```

Axios provides a higher-level abstraction with additional features and a more user-friendly API, while Fetch is a built-in browser feature with a lower-level interface that's more verbose but provides fine-grained control over requests and responses.

State Management Tools :

Redux Toolkit:

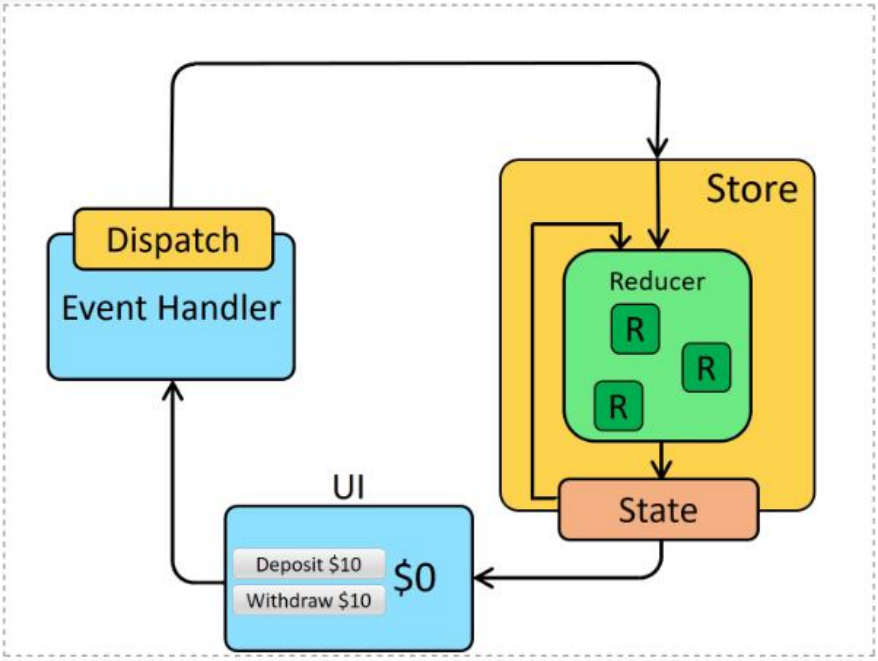
- **Purpose:** Redux Toolkit is an opinionated, batteries-included toolset for efficient Redux development. It aims to simplify Redux workflows, making managing application states easier and more intuitive.

- **Use Cases:**

Redux Toolkit is suitable for applications with complex state management needs, similar to Redux.
It's particularly beneficial for developers who want the benefits of Redux but with reduced boilerplate and simplified syntax.

Key Features:

- **Simplified syntax:** Redux Toolkit provides a set of utility functions that abstract away much of the boilerplate associated with Redux, such as creating action creators and reducers.
- **Immutable updates:** Redux Toolkit encourages immutable updates to the state, helping to ensure predictability and easier debugging.
- **Built-in middleware:** It comes pre-configured with common Redux middleware like Redux Thunk for handling asynchronous logic.
- **DevTools integration:** Redux Toolkit seamlessly integrates with Redux DevTools Extension, providing powerful debugging capabilities out of the box.



Context API:

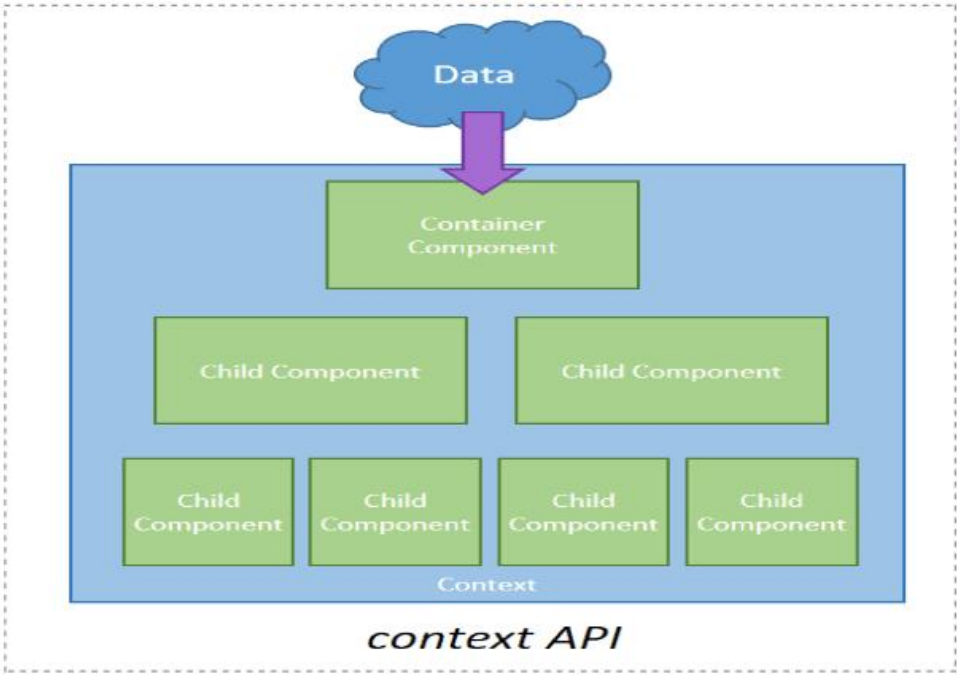
- **Purpose:** The Context API in React allows for sharing data between components without the need for prop drilling. It's particularly useful for managing simpler state needs or sharing data between closely related components.

- **Use Cases:**

Similar to Redux Toolkit, the Context API is suitable for managing application state, albeit for simpler state management needs.
It's beneficial when prop drilling becomes cumbersome, especially in deeply nested component structures.

Key Features:

- **Provider/Consumer pattern:** Components can access context data using a Provider component to set the context value and a Consumer component to access it, similar to Redux Toolkit's store and selectors.
- **Built-in to React:** Like Redux Toolkit, the Context API is lightweight and built into React, eliminating the need for additional dependencies.
- **Good for simpler state management needs:** While Redux Toolkit is geared towards more complex state management scenarios, the Context API shines in managing simpler state or UI-related state.



Routing in React:

- **React router DOM:-**

React Router DOM, on the other hand, is a routing library for React applications. It allows you to handle navigation and URL changes in a React application by mapping components to different routes. Together, Redux and React Router DOM enable you to manage the application state and handle navigation in a predictable and efficient way.

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

Optimize React app:

To Optimize the React app:-

- **Code Splitting:** Break your code into smaller parts and load them when necessary using tools like React.lazy() and dynamic imports.
- **Bundle Size Analysis:** Use tools like Webpack Bundle Analyzer to check bundle size, identify large dependencies, and find alternatives.
- **Minification and Compression:** Ensure your code is minified and served with compression to reduce file size. Tools like Webpack or create-react-app handle this automatically.
- **Image Optimization:** Compress and optimize images to reduce size without losing quality, using tools like ImageOptimize or webpack image loaders.
- **Performance Monitoring:** Monitor app performance with tools like Lighthouse or Chrome DevTools, and optimize based on identified bottlenecks.
- **Server-Side Rendering (SSR) or Static Site Generation (SSG):** Implement SSR or SSG for faster initial load times and better SEO using libraries like Next.js.