ReactJS

By : LAKSHMIKANT DESHPANDE

Prerequisites

- **Adding JavaScript to a Web Page**
- **Variables**
- **Data types**
- **Arrays**
- **Conditionals**
- **Loops**
- **Scope**
- **Object**
- **Functions**
- **Higher Order Function**
- **Destructuring and Spreading**
- **Functional Programming**
- **Classes**
- **Document Object Model(DOM)**

Prerequisites contd…

- **Scripting**
  - **Inline Script**
  - **Internal Script**
  - **External Script**
  - **Multiple External Scripts**
- **Array**
  - **How to create an empty array**
  - **How to create an array with values**
  - **Creating an array using split**
  - **Accessing array items using index**
  - **Modifying array element**
  - **Methods to manipulate array**
  - **Array of arrays**

Prerequisites contd…

- **Conditional Statements**
    - **If**
    - **If Else**
    - **If Else if Else**
    - **Switch**
    - **Ternary Operators**
- **Callback**
- **Returning function**
- **Setting time**
- **What is Destructuring?**
- **What can we destructure?**
- **Spread or Rest Operator**

Prerequisites contd…

- **Array Methods**
    - **forEach**
    - **map**
    - **filter**
    - **reduce**
    - **find**
    - **findIndex**
    - **some**
    - **every**

Prerequisites contd…

- **Defining a classes**
- **Class Instantiation**
- **Class Constructor**
- **Default values with constructor**
- **Class methods**
- **Properties with initial value**
- **getter**
- **setter**
- **Static method**
- **Inheritance**
- **Overriding methods**

Setting Up Your first React App

- Node
- Module
- Package
- Node Package Manager(NPM)
- Visual Studio Code
- Browser
- Visual Studio Extensions
- Create React App
    - React Boilerplate
    - Styles in JSX
    - Injecting data to JSX elements
    - Importing Media Objects in React

npm (Node Package Manager)

- **npm** is the world's largest **Software Library** (Registry)
- **npm** is also a software **Package Manager** and **Installer**
- The registry contains over 800,000 **code packages**.
- **Open-source** developers use **npm** to **share** software.
- Many organizations also use npm to manage private development.

Introduction to React

- What is React?
- Why use React?
- Setting up the development environment (Node.js, npm or yarn)
- Creating your first React component
- JSX syntax and its benefits
  - JSX Element
  - Commenting a JSX element
  - Rendering a JSX Element
  - Style and className in JSX
  - Injecting data to a JSX Element
- React's component-based architecture
- Virtual DOM and its advantages

What is React

- React is a Open-source JavaScript library created by **Facebook** i,e a **User Interface** (UI) library
- Designed for creating user interfaces.
- Focuses on building reusable UI components.
- Efficiently updates and re-renders components.
- React uses a declarative approach to describe the desired state of the UI, allowing for efficient management of the DOM through its virtual DOM reconciliation algorithm.
- Utilizes a virtual DOM for performance optimization.
- It is widely used for building web applications, particularly single-page applications (SPAs), and can also be employed to develop mobile applications using React Native.

Key features and concepts

- **Declarative**: React allows you to describe the UI based on how it should look at any given point in time, and it automatically updates and re-renders the components as needed when the underlying data changes.
- **Component-Based**: React applications are built using reusable components that encapsulate both UI and logic. These components can be composed together to form complex interfaces.
- **Virtual DOM**: React uses a virtual representation of the actual DOM (Document Object Model) to efficiently manage updates. Changes are first made to this virtual DOM, and then React optimizes the actual DOM manipulation to minimize performance overhead.
- **Unidirectional Data Flow**: React enforces a unidirectional data flow, where data flows from parent components to child components. This helps in managing data changes and simplifies debugging.
- **JSX (JavaScript XML)**: React uses JSX, a syntax extension for JavaScript, to write UI components in a format that resembles HTML. JSX gets transpiled into regular JavaScript for execution in the browser.

Key features and concepts contd…

- **State**: Components can have internal state that holds data specific to that component's functionality. When state changes, React re-renders the component and its children.
- **Props (Properties)**: Components can accept input data called props, which are passed down from parent components. Props are read-only and help to configure a component's behavior and appearance.
- **Lifecycle Methods**: React components have lifecycle methods that allow you to perform actions at different stages of a component's existence, such as when it's created, updated, or destroyed.
- **Hooks**: Hooks are functions that let you "hook into" React state and lifecycle features from functional components, enabling you to manage state and side effects without using class components.
- **Reconciliation**: React's reconciliation algorithm efficiently updates the UI by determining the minimum number of changes needed to reflect changes in the data. This contributes to better performance.

Key features and concepts contd…

- **Virtualization**: Lists or collections of items can be efficiently rendered using techniques like "virtualization," where only the visible items are rendered, reducing the strain on the rendering process.
- **Server-Side Rendering (SSR)**: React supports server-side rendering, allowing you to render React components on the server and send the pre-rendered HTML to the client. This improves initial load times and SEO.
- **Community and Ecosystem**: React has a vast ecosystem of libraries, tools, and extensions that help with routing, state management, styling, testing, and more. It's maintained by Facebook and has a large and active community.
- **Single Page Applications (SPAs)**: React is often used in creating SPAs, where a single HTML page loads dynamically updated content as the user interacts with the application, providing a smoother user experience.
- **React Native**: React can also be used to build mobile applications through React Native, which allows you to write components using React syntax and have them compiled to native mobile components.

Key features and concepts contd…

- **Open Source**: React is open-source, and its code is available on platforms like GitHub, making it accessible for contributions, customization, and improvements.
- **Component Reusability**: React's modular nature and component-based architecture promote code reusability, making it easier to maintain and scale applications.
- **Context API**: React provides a Context API that enables the sharing of state data across components without the need to pass props explicitly through every level of the component tree.
- **Error Boundaries**: React allows you to define error boundaries around components to catch and handle errors that occur during rendering, preventing the entire application from crashing.
- **Pure Components**: React's PureComponent class performs shallow comparisons on props and state to determine if a component needs to re-render, optimizing performance by avoiding unnecessary re-renders.

Key features and concepts contd…

- **Higher-Order Components (HOCs)**: HOCs are functions that take a component and return an enhanced version of that component. They are used for code reuse, logic abstraction, and cross-cutting concerns.
- **React Router**: React Router is a popular library for handling routing within a React application, allowing you to create navigational components and manage different views.
- **State Management Libraries**: While React's built-in state management is suitable for many scenarios, more complex applications often use external state management libraries like Redux or MobX to manage global application state.
- **CSS-in-JS**: React supports various approaches to styling, including using traditional CSS classes, inline styles, and CSS-in-JS libraries like Styled Components and Emotion.
- **Component Testing**: React applications can be thoroughly tested using tools like Jest and React Testing Library, which provide utilities for writing unit and integration tests for React components.

Key features and concepts contd…

- **Immutability**: React encourages the use of immutability, where data is not modified directly but instead replaced with new copies. This ensures predictability and helps in optimizing re-renders.
- **Webpack and Babel**: When setting up a React project, tools like Webpack and Babel are commonly used to bundle and transpile the code, making it browser-compatible and optimizing performance.
- **Suspense and Lazy Loading**: React's Suspense API allows you to suspend rendering while fetching asynchronous data, and lazy loading enables the loading of components only when they are needed, reducing initial bundle size.
- **Accessibility (A11y)**: React supports building accessible applications by providing tools and guidelines for creating UIs that are usable by individuals with disabilities.
- **Code Splitting**: Code splitting involves breaking down your application code into smaller chunks that are loaded on demand, reducing the initial load time and improving performance.

Key features and concepts contd…

- **Strict Mode**: React's Strict Mode is a development mode that highlights potential issues and optimizations, helping developers catch and fix problems early.
- **Community Conferences and Meetups**: The React community hosts conferences (like ReactConf) and local meetups where developers can learn, share experiences, and stay up-to-date with the latest trends and techniques.
- **TypeScript Integration**: While React can be used with plain JavaScript, TypeScript, a statically typed superset of JavaScript, is also commonly used to provide better code analysis, debugging, and scalability.
- **Compatibility with Other Libraries/Frameworks**: React can be integrated with other libraries and frameworks. For instance, it can be combined with libraries like D3 for data visualization or integrated into frameworks like Next.js for server-rendered React applications.

What is Babel?

- Babel is a JavaScript transcompiler that can translate markup or programming languages into JavaScript.
- With Babel, we can use the newest features of JavaScript (ES6 - ECMAScript 2015).
- Babel is available for different conversions. React uses Babel to convert JSX into JavaScript.
- <script type="text/babel"> is needed for using Babel.

What is JSX?

- JSX stands for **J**ava**S**cript **X**ML.
- JSX (JavaScript XML) is a syntax extension for JavaScript that allows you to write HTML-like code within your JavaScript code. It's commonly used with React.js to define the structure and content of UI components.
- JSX is an XML/HTML like extension to JavaScript.
- **Example**
  - const element = <h1>Hello World!</h1>
  - As you can see above, JSX is not JavaScript nor HTML.
- JSX is a XML syntax extension to JavaScript that also comes with the full power of ES6 (ECMAScript 2015).
- Just like HTML, JSX tags can have a tag names, attributes, and children. If an attribute is wrapped in curly braces, the value is a JavaScript expression.
- JSX does not use quotes around the HTML text string

JSX Element

A JSX element is a JavaScript expression that represents a React element. It looks like HTML but is actually a syntax extension of JavaScript. JSX elements are used to define the structure of UI components in React. Here's an example of a JSX element

const element = <h1>Hello, JSX!</h1>;

Commenting a JSX Element : You can comment within JSX using curly braces and JavaScript-style comments.

```
const element = (
  <div>
    {/* This is a comment */}
    <h1>Hello, JSX!</h1>
  </div>
);
```

Rendering a JSX Element

To render a JSX element in the DOM, you use the ReactDOM.render() function provided by React. You pass the JSX element as the first argument and specify the target DOM container as the second argument.

```
import React from 'react';
import ReactDOM from 'react-dom';
const element = <h1>Hello, JSX!</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

Style and className in JSX

You can apply inline styles to JSX elements using the style attribute. The style attribute should be an object with camel-cased CSS property names and their values. Additionally, you can use the className attribute to assign CSS classes to JSX elements.

```
const styles = {
  color: 'blue',
  fontSize: '20px'
};
const element = (
  <p style={styles} className="custom-class">
    This is a styled JSX element.
  </p>
);
```

Injecting Data into a JSX Element

You can inject data, variables, and expressions into JSX elements using curly braces {}. This allows you to dynamically generate content within your JSX elements.

```
const userName = ‘Rajath’;
const element = (
  <div>
    <h1>Hello, {userName}!</h1>
    <p>Today is {new Date().toLocaleDateString()}.</p>
  </div>
);
```

React's component-based architecture

- React's component-based architecture is a fundamental concept that underpins the way React applications are structured and developed.
- It's a design approach where the UI is broken down into smaller, reusable, and modular pieces called components.
- Each component encapsulates a specific piece of functionality or user interface and can be composed together to create complex UIs.

An overview of the key aspects of React's component-based architecture

Components:

Components are the building blocks of a React application. They can be thought of as self-contained units that handle rendering and behavior for a specific part of the UI. Components can be function components or class components (prior to the introduction of hooks). With the advent of hooks, function components have become the more commonly used option due to their simplicity and reusability.

Reusability:

One of the main advantages of a component-based architecture is reusability. Components can be reused across different parts of an application or even across different projects. This helps in maintaining a consistent look and behavior while reducing code duplication.

Composition:

React encourages the composition of components. This means that you build complex UIs by combining smaller components together. This approach enables you to manage and understand the application's structure more easily.

Hierarchical Structure:

Components can be nested within each other to create a hierarchical structure. This mirrors the way user interfaces are often organized. The parent component can manage the state and pass down data to its child components through props.

Separation of Concerns:

Components promote the separation of concerns by encapsulating logic and rendering related to a specific UI element. This makes the codebase more maintainable and easier to debug.

Data Flow:

In React, data flows in one direction. This means that data typically travels from parent components to child components through props. If a child component needs to communicate changes back to the parent, it can do so by invoking callback functions passed down as props.

React's Virtual DOM:

React uses a Virtual DOM to optimize rendering performance. When a component's state or props change, React creates a virtual representation of the UI changes, compares it with the previous virtual representation, and then updates only the necessary parts of the actual DOM. This helps minimize DOM manipulation and improves performance.

Component Lifecycle (in class components):

Before hooks were introduced, class components had a lifecycle that included methods like componentDidMount, componentDidUpdate, and componentWillUnmount. These methods allowed developers to perform actions at specific stages of a component's life.

Hooks (Modern Approach):

Hooks were introduced as a way to manage state and side effects in functional components. Hooks like useState, useEffect, useContext, and others allow you to manage state and perform side effects without the need for class components.

React's component-based architecture provides a structured and modular way to develop user interfaces. By breaking down UIs into reusable components, developers can create maintainable, scalable, and easy-to-understand applications.

Virtual DOM and its advantages

The Virtual DOM (Virtual Document Object Model) is a concept and optimization technique used by React to improve the efficiency and performance of updating the actual DOM (Document Object Model) in web browsers. It's one of the core features that make React an efficient and powerful library for building user interfaces.

An explanation of the Virtual DOM and its advantages

Traditional DOM Manipulation:
In web development, when the state of a web page changes, the browser's DOM needs to be updated to reflect those changes. However, directly manipulating the DOM can be slow and resource-intensive, especially when dealing with complex and dynamic user interfaces.

Virtual DOM and its advantages contd…

The Virtual DOM Concept:
The Virtual DOM is an abstract representation of the actual DOM. It's a lightweight copy of the real DOM tree maintained by React. Whenever the state of a component changes, React doesn't immediately update the actual DOM. Instead, it creates a new Virtual DOM tree that reflects the updated state.

Diffing Algorithm:
React then performs a process called "reconciliation" or "diffing" where it compares the previous Virtual DOM tree with the new one. It identifies the differences (or "diffs") between the two trees. This process allows React to determine the minimum number of updates required to bring the actual DOM in line with the new state.

Batched Updates:
React batches these updates and applies them to the actual DOM in a single operation, minimizing the number of times the browser has to reflow and repaint the page. This approach significantly improves performance compared to directly manipulating the DOM for every change.

Virtual DOM and its advantages contd…

Performance: The primary advantage of the Virtual DOM is improved performance. By minimizing direct DOM manipulation and applying updates in an optimized manner, React reduces the computational and rendering overhead, resulting in a smoother user experience.

Efficiency: The Virtual DOM allows React to perform updates more efficiently. Instead of modifying the actual DOM for each change, React only updates the necessary parts of the DOM, reducing redundant operations and making the application faster.

Developer Productivity: The Virtual DOM simplifies the process of building and maintaining complex user interfaces. Developers can focus on writing component logic and let React handle the rendering optimizations.

Virtual DOM and its advantages contd…

Cross-Platform Consistency: The Virtual DOM abstracts away browser-specific rendering details. This helps ensure that React applications behave consistently across different browsers and platforms.

Optimized Reconciliation: React's diffing algorithm optimizes the process of updating the DOM by minimizing unnecessary changes. This results in efficient updates and faster rendering times.

Easier Testing: Since the Virtual DOM is a JavaScript object, you can easily test and manipulate it within your tests, allowing for better unit and integration testing.

In summary, the Virtual DOM is a crucial concept that contributes significantly to React's efficiency and performance. It allows React applications to efficiently manage updates and changes to the UI, resulting in smoother user experiences and more responsive web applications.

Applications

Few popular websites powered by *React library* are listed below

- *Facebook*, popular social media application
- *Instagram*, popular photo sharing application
- *Netflix*, popular media streaming application
- *Code Academy*, popular online training application
- *Reddit*, popular content sharing application

Components and Props

- Creating functional components
- Creating class components
- Using props to pass data to components
- Default props and prop types
- Props validation with PropTypes
- Component composition and reusability

Functional Components

Functional components are JavaScript functions that return JSX elements, defining the UI and behavior of a specific part of your application.

```
import React from 'react';
const TestComponent = () => {
    return (
        <div>
            <h1>Hello from Component!</h1>
            <p>This is a functional component in React.</p>
        </div>
    );
};
export default TestComponent;
```

Class Components

Class components are another way to create components in React, alongside functional components. They were the primary method of creating components before the introduction of React Hooks, and they still have their use cases.

```
import React, { Component } from 'react';
class ClassComponent extends Component {
    render() {
        return (
            <div>
                <h1>Hello from Class Component!</h1>
                <p>This is a class component in React.</p>
            </div>
        );
    }
}
export default ClassComponent;
```

Props

- In React, props (short for "properties") are a way to pass data from a parent component to its child components. They are a fundamental concept that allows you to create dynamic and reusable components by customizing their behavior and content based on the data they receive.

Passing Props:

- In the parent component, you can pass data to a child component by including attributes with values. These attributes represent the data you want to send, and their values can be strings, numbers, booleans, objects, or even functions.
- In summary, props are a core mechanism in React that enable the composition of components by allowing data to flow from parent to child components. This data-driven approach helps create dynamic and reusable UI components in your applications.

Default props and prop types

- Using default props is a way to provide default values for props in your components. This can be useful when you want to ensure that a prop has a value even if it's not explicitly provided by the parent component.
- Prop types are a way to define the types of props that your components should receive. They help catch bugs and provide better documentation for your components.

Component composition and reusability

- Component composition and reusability are crucial aspects of building a well-structured and maintainable application.
- These concepts allow us to create modular and versatile components that can be combined to build complex UIs while keeping the codebase organized and easy to manage.

Component Composition:

- Component composition involves combining smaller, reusable components to create more complex components or UI structures. This approach makes it easier to manage and understand the structure of your application.

Reusability:

- Reusability is achieved by designing components in a way that allows them to be used across different parts of your application. This reduces code duplication and promotes consistency.

Component composition and reusability contd…

Prop Customization

- Utilize props to customize component behavior and content based on the data passed.

Higher-Order Components (HOCs)

- HOCs are components that take another component as input and return an enhanced version of that component. They are a powerful way to share behavior and logic across multiple components.

Custom Hooks

- Custom Hooks are a way to share logic between components, allowing you to encapsulate complex state management or API calls into reusable functions.

Atomic Design Principles

- Consider adopting Atomic Design principles, which suggest breaking down UI components into smaller building blocks like atoms (simple components), molecules (combinations of atoms), organisms (complex UI components), and templates (page layouts). This methodology promotes modularity and reusability.

Component composition and reusability contd…

Folder Structure:

- Organize your components into a clear folder structure. Group related components together and use meaningful names for folders and files. This makes it easier to locate and maintain components.

By applying these principles, we can create a highly modular, maintainable, and reusable codebase for your online consultation and content sharing platform. This approach not only streamlines development but also improves the scalability and robustness of your application.

State and Lifecycle

- Understanding component state
- Setting initial state
- Updating state using setState()
- Component lifecycle methods: componentDidMount, componentDidUpdate, componentWillUnmount
- Hooks
- Using state effectively to manage dynamic UI

State

- State is a built-in object in React for managing and storing data that can change over time within a component.
- It's essential for creating dynamic and interactive user interfaces.
- State enables components to respond to user interactions, API responses, and other events by updating and re-rendering parts of the UI affected by data changes.
- State can be initialized using the useState hook in functional components or by extending the Component class in class components.
- Updating state is done using setter functions provided by React, such as setCount in functional components using the useState hook or this.setState({}) in class components.

State contd…

- State updates are asynchronous, and React batches multiple setState calls together for performance reasons. Therefore, you should avoid relying on the current state value when updating state.
- You can use state values within JSX to dynamically render content based on the current state.
- State is local to a component, and it's not automatically shared between different components. If you need to pass state between components, you can do so using props.
- State updates trigger re-renders of the component to reflect the latest data.
- For more complex state management needs, like managing global state or sharing state between components far apart in the component tree, you might consider using external state management libraries like Redux or the React Context API.

Understanding how to use and manage state is fundamental to building dynamic and interactive React applications.

Setting initial state

Setting initial state in React components is crucial for defining the initial values that your component will use. Depending on whether you're using functional components with hooks or class components, the approaches to setting initial state slightly differ.

Setting Initial State in Functional Components with Hooks:

In functional components with hooks, we can use the useState hook to set the initial state.
The useState hook returns an array with two elements: the current state value and a function to update the state.
We pass the initial value as an argument to the useState function.

Setting initial state contd…

Setting Initial State in Class Components:

In class components, we set the initial state in the constructor.

The constructor is called when an instance of the class is created.

Within the constructor, we can use this.state to set the initial values.

Component lifecycle methods:

- **componentDidMount:**

This method is called after a component is rendered for the first time (mounted). It's a good place to perform initialization tasks or fetch data from APIs.

- **componentDidUpdate:**

This method is called when the component updates after a change in props or state. It's useful for reacting to changes and performing actions like making API calls based on updated data.

- **componentWillUnmount:**

This method is called just before a component is unmounted and removed from the DOM. It's a good place to perform cleanup tasks, such as canceling timers or subscriptions.

Hooks

- In functional components with hooks, the concept of lifecycle methods is replaced by the useEffect hook.
- useEffect allows you to perform side effects in a functional component, including actions that correspond to the lifecycle stages of class components.

Lifecycle Equivalents:

- componentDidMount: The code inside the useEffect function runs after the component renders for the first time, simulating the behavior of componentDidMount.
- componentDidUpdate: By default, the useEffect code runs after every render. To control when it runs, you can specify dependencies in the dependency array. When the values in the dependency array change, the effect will re-run, simulating the behavior of componentDidUpdate.
- componentWillUnmount: If you return a cleanup function inside the useEffect function, it will be executed when the component unmounts, simulating the behavior of componentWillUnmount.

Using state effectively to manage dynamic UI

Using state effectively is crucial for managing dynamic user interfaces in our application.
State allows us to keep track of changing data and trigger UI updates accordingly.

Here are some strategies for using state effectively to manage dynamic UI

Define Relevant State:
Identify the data that needs to change and drive the dynamic aspects of your UI. For example, in your consultation platform, you might have state for consultation data, user preferences, or interaction states like likes and comments.

Use Local State:
Use local component state for data that's specific to a particular component and doesn't need to be shared globally across your application. This keeps your components self-contained and avoids unnecessary complexity.

Avoid Redundant State:
Keep your state minimal and avoid duplicating data that can be derived from other state or props. Redundant state can lead to confusion and make it harder to maintain your application.

Update State Responsibly:
Use the appropriate methods to update state, such as setState in class components or state updater functions in functional components. Avoid modifying state directly, as it can lead to unexpected behavior.

Combine Related State:
If multiple pieces of state are closely related and often change together, consider grouping them together into an object. This can help organize your code and make updates more predictable.

Use State for UI Interactions:
Use state to manage UI interactions, such as toggling visibility, showing/hiding content, enabling/disabling buttons, and more.

Leverage State for Conditional Rendering:

Use state to conditionally render different parts of your UI based on the current state. For instance, show different views based on whether a user is logged in or not.

Handle Dynamic Data Updates:

If your consultation or content data can change over time (likes, comments, timestamps), update the relevant state when this data changes. This will automatically trigger UI updates.

Use Effect for Side Effects:

Use the useEffect hook to handle side effects related to state changes, such as fetching data from APIs, subscribing to events, or updating external resources.

Clear Unused State:

When a component is unmounted or no longer needs a certain piece of state, make sure to clear that state to prevent memory leaks and unnecessary resource usage.

Think About Component Composition:

Break down your UI into smaller, reusable components. Each component can manage its own specific state, making it easier to reason about and maintain.

Plan for Scalability:

Consider how your state management approach will scale as your application grows. For more complex scenarios, you might need to explore state management libraries like Redux or the React Context API.

By following these strategies, we will be able to effectively use state to manage dynamic UI in your application creating a smooth and interactive user experience.

Handling Events and Forms

- Handling user events in React
- Event handling in functional components
- Event handling in class components
- Controlled vs. uncontrolled components
- Form handling and validation
- Handling multiple input fields

Handling user events in React

Handling user events is a fundamental aspect of building interactive user interfaces in React.
In our application, we will likely need to respond to various user interactions, such as clicks, input changes, and form submissions.

Handling Click Events: onClick

Handling Input Changes: onChange

Handling Form Submissions: onSubmit

Handling Conditional Rendering:

Handling user events in React contd…

- **Event Handling in Functional Components:**
  - Functional components in React can handle user interactions using event handling.
  - Events are actions triggered by users, such as clicks, input changes, and submissions.
- **Controlled Components:**
  - In controlled components, the value of input elements is controlled by React state.
  - Changes to input values are managed through state manipulation.
  - Controlled components provide better control and validation over user inputs.
- **Uncontrolled Components:**
  - Uncontrolled components allow the DOM to manage the input value directly.
  - Input values are accessed using ref attributes.
  - Useful for integrating React with existing HTML forms or for simpler interactions.

Handling user events in React contd…

- **Event Prop and Event Handler:**
    - To handle events, JSX elements have special props like onClick, onChange, etc.
    - An event handler function is defined to respond to specific user actions.
- **Defining Event Handlers:**
    - Event handler functions are defined within the component function.
    - They accept an event parameter that provides information about the event.
    - Event handlers are used to define what should happen when the event occurs.
- **Updating State with Event Handling:**
    - Controlled components use event handlers to update the state based on user actions.
    - State updates trigger component re-rendering with the new values.

Handling user events in React contd…

- **Using** preventDefault()**:**
  - For form submissions, use event.preventDefault() to prevent the default browser behavior.
  - This allows you to handle the form submission within your event handler.
- **Using** ref **for Uncontrolled Components:**
  - ref is used to access the DOM node directly in uncontrolled components.
  - Useful when you need to interact with the DOM, but state management isn't necessary.
- **Passing Event Handlers:**
  - Event handler functions are passed as props to child components.
  - This allows child components to trigger actions in parent components.
- **Integration in Functional Components:**
  - Define event handlers within functional components where the interaction occurs.
  - Attach event handlers to JSX elements using the appropriate event props.

Event handling in class components

- In class components, event handling involves defining methods that respond to user interactions.
- React provides a set of predefined event handler methods that you can override.

Event Handler Methods:

- In class components, you can use methods like render, componentDidMount, componentDidUpdate, componentWillUnmount, etc.
- To handle user events, you can define custom methods like handleClick, handleChange, etc.

Binding Event Handlers:

- When using custom event handler methods, you need to bind them to the instance of the class using .bind(this) in the constructor or by using arrow functions.

Rendering Event-Triggering Elements:

- JSX elements within the render method can have event handler methods attached to them using special props like onClick, onChange, etc.

Updating State with Event Handling:

- Event handlers in class components often update the component's state.
- Using setState within an event handler triggers a re-render with the updated state.

Preventing Default Behavior:

- For form submissions or anchor clicks, use event.preventDefault() to prevent the default browser behavior and handle the action within your event handler.

Passing Data to Event Handlers:

- You can pass additional data to event handler methods by using arrow functions or the bind method.
- This is useful when you need to access specific data associated with the event.

Controlled vs. uncontrolled components

Controlled Components

In a controlled component, the React state is the "single source of truth" for the input elements. This means that the value of the input element is controlled by React state, and any changes to the input are controlled through state manipulation.
Benefits of Controlled Components:

- React has full control over the input's value, enabling validation and custom logic.
- Easier to perform actions like resetting or clearing the input value.
- Makes it straightforward to synchronize multiple inputs or manage their interactions.

Uncontrolled Components

In an uncontrolled component, the input value is managed by the DOM itself, and React doesn't control or track the input's value through state. You typically use ref to access the input's value when needed.
Benefits of Uncontrolled Components:

- Useful when integrating React into projects with existing DOM-based forms.
- Simplifies code in cases where React state management isn't necessary.

Form handling and validation

Form handling and validation are crucial aspects of building interactive and user-friendly applications. In React, we can achieve form handling and validation using both controlled components and state management.

Form Handling:

- Form handling involves capturing user input from form elements like input fields, text areas, checkboxes, and radio buttons.
- In React, we can use controlled components to manage form input elements and their values through state.

Form Validation:

- Form validation ensures that the data submitted by users meets certain criteria or constraints before it's processed.
- We can implement client-side validation to provide immediate feedback to users and prevent invalid data from being submitted.

Conditional Rendering and Lists

- Using conditional rendering to show/hide components
- Ternary operators and conditional statements
- Rendering lists of data using .map()
- Keys and their importance in list rendering
- Fragments for cleaner JSX structure

Styling in React

- Styling options: Inline styles, CSS classes, CSS modules
- CSS-in-JS libraries like styled-components
- Managing dynamic styles with conditional classes
- Theming and global styles
- Using third-party UI libraries with React

Component Communication

- Passing data between parent and child components
- Callback functions as props
- Using the Context API for global state management
- Props drilling and its drawbacks
- Introduction to state management libraries (Redux, MobX)

Context API

The Context API is a feature in React that allows you to manage global state and share data across your component tree without having to pass props through multiple levels of components (props drilling). It's designed to solve the problem of sharing state between components that are not directly connected in the component hierarchy.

The Context API consists of two main parts: the Context object and the Provider component.

Props drilling and its drawbacks

Props drilling, also known as prop threading, occurs when you need to pass data through multiple layers of components in a React application. While it's a common and straightforward way to share data between components, it can have some drawbacks as your application becomes larger and more complex.

1. Complexity and Boilerplate: As your component tree deepens, passing props down multiple levels can lead to increased complexity and boilerplate code. Components at intermediate levels might not actually need the data being passed, but they are required to accept and pass the data along.
2. Coupling: Components in the middle of the tree can become tightly coupled to the data flow because they need to accept and pass props without using the data themselves. This makes your components less modular and harder to refactor.
3. Maintainability: As your application grows, maintaining and tracking the flow of props becomes challenging. Adding, removing, or modifying props in one part of the tree might require changes to multiple components throughout the tree.

Props drilling and its drawbacks contd…

4. Performance Implications: Passing props through multiple levels can have performance implications, especially if you're dealing with deeply nested components. Each time a prop is updated, all the components in the chain that use that prop will re-render.
5. Readability and Understandability: A deeply nested component structure with props drilling can make the code less readable and harder to understand, especially for new developers joining the project.
6. Debugging: Debugging can become more complex when you need to trace how data flows through multiple components. Identifying where a certain value originates or changes can be time-consuming.

To mitigate the drawbacks of props drilling, consider using state management solutions like Redux, MobX, or the Context API for centralized state management. These solutions provide a global state store that can be accessed by any component without the need to pass props through intermediate levels. While they might introduce additional complexity, they can greatly simplify the data flow in larger applications and improve maintainability.

Introduction to state management libraries (Redux, MobX)

State management libraries like Redux and MobX provide solutions for managing application state in more complex React applications. They offer centralized state management, enabling components to access and modify data without passing props through multiple layers of the component tree.

Redux and MobX offer different approaches to state management in React applications. Redux provides a more structured and explicit way of managing state, while MobX emphasizes simplicity and automatic updates. The choice between them depends on the complexity of your application, your team's familiarity, and your preference for how state management is handled.

Redux

- **What is Redux:**
  - Redux is a predictable state container for JavaScript applications.
  - It's commonly used with React but can work with any JavaScript framework.
  - Redux follows the Flux architecture pattern.
- **Core Concepts:**
  - **Store:** A centralized data store that holds the application state.
  - **Actions:** Plain JavaScript objects that describe state changes.
  - **Reducers:** Functions that specify how state changes in response to actions.
  - **Selectors:** Functions used to extract specific parts of the state from the store.

Redux contd…

- **Workflow:**
    - Components dispatch actions to change the state.
    - Actions are processed by reducers, which update the state.
    - Updated state triggers re-rendering of components.
- **Advantages:**
    - Predictable state management: State changes are explicit and easy to understand.
    - Centralized state: Simplifies data sharing between components.
    - Time-travel debugging: Ability to replay actions to reproduce bugs.
    - Large ecosystem: Middleware, tools, and extensions available.
- **Drawbacks:**
    - Boilerplate: Writing actions, reducers, and action creators can be verbose.
    - Learning curve: May be challenging for newcomers to React.

MobX

- **What is MobX:**
  - MobX is a reactive state management library.
  - It aims to make state management simple and scalable.
- **Core Concepts:**
  - **Observables:** Data that can be observed and reacts to changes.
  - **Actions:** Functions that modify the observables.
  - **Reactions:** Functions that automatically update when observables change.
- **Workflow:**
  - Define observables (state).
  - Create actions to modify observables.
  - Reactions automatically update based on observable changes.

MobX contd…

- **Advantages:**
  - Simplicity: Fewer concepts compared to Redux.
  - Automatic updates: Components re-render automatically on data changes.
  - Flexibility: Can be used in various ways, from minimal setup to complex structures.
- **Drawbacks:**
  - Lack of strict structure: Less opinionated than Redux, which might lead to inconsistency.
  - Limited ecosystem: Smaller compared to Redux.

Use Cases of Both:

- Both Redux and MobX are suitable for medium to large-scale applications where managing state becomes more complex.
- Use when you have a lot of shared data across components or need to handle asynchronous data flows.

Routing with React Router

- Introduction to client-side routing
- Setting up React Router
- Creating routes and navigation
- Route parameters and query parameters
- Nested routes and route organization

Hooks

- Introduction to React Hooks
- useState for managing state in functional components
- useEffect for handling side effects
- Custom hooks for reusable logic
- useContext for accessing context in functional components

Introduction to React Hooks

React Hooks are a set of functions introduced in React 16.8 that allow developers to use state and other React features in functional components without writing class-based components. Hooks aim to simplify and enhance the development experience by offering a more concise and composable way to manage state, side effects, and other component logic.

Why Hooks?

- Prior to Hooks, state and lifecycle management in React components was primarily done through class components.
- Class components introduced complexity due to lifecycle methods, making code harder to reuse and test.

Core Principles of Hooks:

- Hooks are functions provided by React.
- Hooks don't work in class components; they are designed for functional components.

Introduction to React Hooks contd…

Commonly Used Hooks:

- **useState:**
  - Allows functional components to manage state.
  - Returns a state variable and a function to update that state.
  - Helps in avoiding class-based component's this.state and this.setState.
- **useEffect:**
  - Handles side effects (data fetching, DOM manipulation) in functional components.
  - Replaces lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount.
- **useContext:**
  - Provides access to the Context API in functional components.
  - Avoids prop drilling by allowing components to consume context directly.
- **useReducer:**
  - An alternative to useState for more complex state management.
  - Similar to Redux's reducer pattern.
  - Useful for managing state transitions and actions.

Introduction to React Hooks contd…

Commonly Used Hooks:

- **useMemo:**
  - Memoizes the result of a function to improve performance.
  - Recalculates only if its dependencies change.
- **useCallback:**
  - Memoizes a function to prevent unnecessary re-renders.
  - Useful when passing callbacks to child components.
- **useRef:**
  - Provides a mutable ref object to access DOM elements or store mutable values.
  - Persists across renders without causing re-renders.
- **useLayoutEffect:**
  - Similar to useEffect, but fires synchronously after all DOM mutations.
  - Used for tasks that require synchronous execution, like measuring DOM nodes.

Introduction to React Hooks contd…

Rules of Hooks:

- Hooks must always be called at the top level of a functional component.
- Don't call Hooks inside loops, conditions, or nested functions.

Custom Hooks:

- Custom Hooks are functions that can contain other Hooks.
- Enable you to abstract and reuse component logic.

Benefits of Hooks:

- Improves code readability and maintainability by reducing boilerplate.
- Encourages the use of functional components, which are simpler and easier to understand.
- Makes state management and side effects more intuitive.

Introduction to React Hooks contd…

Drawbacks of Hooks:

- Existing codebases might need refactoring to adopt Hooks.
- Learning curve for developers accustomed to class components.

Use Cases for Hooks:

- Managing component state without classes.
- Handling side effects, such as data fetching.
- Sharing state and logic across components.
- Working with contexts, refs, and memoization.

Summary:
React Hooks revolutionize the way developers work with React by providing a more streamlined and functional approach to managing state and behavior in components. They enable developers to write cleaner, more reusable code and bring React more in line with modern functional programming concepts.

Some hooks list

- **useState**
- **useEffect**
- **useContext**
- **useReducer**
- **useMemo**
- **useCallback**
- **useRef**
- **useLayoutEffect**
- **useImperativeHandle**
- **useDebugValue**

Some hooks list contd…

- **useHistory** (from React Router)
- **useLocation** (from React Router)
- **useParams** (from React Router)
- **useRouteMatch** (from React Router)
- **useForm** (custom hook for managing form state)
- **useWindowSize** (custom hook for tracking window size)
- **useScrollPosition** (custom hook for tracking scroll position)
- **useInterval** (custom hook for managing intervals)
- **useTimeout** (custom hook for managing timeouts)
- **useAsync** (custom hook for managing asynchronous operations)

Some hooks list contd…

- **useToggle** (custom hook for managing boolean toggles)
- **useLocalStorage** (custom hook for working with local storage)
- **useSessionStorage** (custom hook for working with session storage)
- **useSwipeable** (from react-swipeable library)
- **useDrag** (from react-dnd library)
- **useDrop** (from react-dnd library)
- **useMediaQuery** (from @material-ui/core library)
- **useInView** (from react-intersection-observer library)
- **useGesture** (from react-use-gesture library)
- **useAsyncFn** (from react-use library)
- **useMutationObserver** (from react-use library)

Some hooks list contd…

- **useRequest** (from swr library)
- **useFormik** (from formik library, for form handling)
- **useSWR** (from swr library, for data fetching)
- **useQuery** (from react-query library, for data fetching)
- **useMutation** (from react-query library, for data mutation)
- **useSubscription** (from react-query library, for real-time data)
- **useMachine** (from xstate library, for finite state machines)
- **useFormState** (from uniforms library, for form state)
- **useWeb3React** (from web3-react library, for Ethereum Web3 integration)

Some Advanced Topics

- Error handling and debugging in React applications
- Performance optimization techniques (memoization, useMemo, useCallback)
- Server-side rendering (SSR) and static site generation (SSG)
- Testing React components with Jest and React Testing Library
- Best practices and code organization in larger React applications