

LESSON 04

JSX, Props, and State

Concepts

WEEK 01

Objectives

- ❖ Understanding JSX in React
- ❖ Working with Props
- ❖ Managing State in React

Understanding JSX

Understanding JSX

❖ What is JSX?

- JSX is a syntax extension for JavaScript that allows you to write HTML-like code inside JavaScript, which React can render to the DOM. It simplifies UI development by combining the structure and behavior in one place.

```
const element = <h1>Hello, world!</h1>;
```

❖ Why JSX?

- JSX makes the code more readable and concise, allowing developers to write components in a declarative way.
- React components use JSX to define the structure of the UI.

❖ JSX Expression Syntax

- JSX can be used inside JavaScript functions and expressions. It can include variables, functions, and even conditions.

```
const name = "John";  
const element = <h1>Hello, {name}!</h1>;
```

JSX Syntax Rules

❖ Self-Closing Tags

DESC: All JSX tags must be self-closed, even for elements that don't contain children, such as `` or `<input />`.

❖ Wrapping Tags

JSX elements must have one root element. Multiple elements should be wrapped in a single parent element, like a `<div>` or React Fragment (`<>` and `</>`).

❖ Using Expressions Inside JSX

You can include any JavaScript expression inside curly braces `{}`. For example: `{5 + 2}`, `{variableName}`, or `{myFunction()}`.

❖ Attributes in JSX

Attributes in JSX follow the camelCase convention. For instance, use `className` instead of `class` and `htmlFor` instead of `for`.

❖ Comments in JSX

Comments in JSX are enclosed in curly braces with `/* */`. Example: `{/* This is a comment */}`.

JSX in React Components

❖ **Using JSX in Components**

JSX is commonly used to define the structure of React components, which are the building blocks of React applications.

❖ **Rendering JSX**

The JSX code inside components is rendered by React using ReactDOM.render(), which displays the components in the browser.

❖ **Props in JSX**

Props are passed to components as attributes in JSX. You can access them inside the component using props.

Example: `<MyComponent name="Alice" />`

❖ **State and JSX**

React components can use state to dynamically render JSX content based on the current state of the component.

❖ **Conditional Rendering in JSX**

You can use JavaScript conditional expressions like ternary operators inside JSX for dynamic content rendering.

JSX with Expressions and Loops

❖ JavaScript Expressions Inside JSX

You can use JavaScript expressions inside JSX to dynamically render data.

Example: {user.name}.

❖ Conditionals in JSX

Use JavaScript operators like if or ternary operators within JSX to conditionally render components or elements.

Example: {isLoggedIn ? <LogoutButton /> : <LoginButton />}

❖ Looping in JSX

Loop over arrays to render lists in JSX using JavaScript's map() method.

Example: ['Apple', 'Banana', 'Orange']. map(item => <li key={item}>{item});

Working with Props

PropTypes for Validation

❖ What is PropTypes?

PropTypes is a runtime type checking feature in React that helps validate props passed to components. It ensures that props have the correct data types and helps catch errors early.

❖ Why Use PropTypes?

PropTypes improve code reliability by checking if the correct data types are passed. It acts as a safeguard to avoid bugs caused by incorrect prop types.

❖ How to Use PropTypes

You can define prop types by specifying the expected type for each prop.

Example: `Component.propTypes = { propName: PropTypes.string.isRequired }`

PropTypes for Validation

❖ Installation

- Install via npm: `npm install prop-types` to use PropTypes in your project.

❖ Ref:

- <https://www.npmjs.com/package/prop-types>

Common PropTypes

❖ Primitive Data Types

- PropTypes supports common primitive data types such as:
 - `PropTypes.string`
 - `PropTypes.number`
 - `PropTypes.bool`
 - `PropTypes.array`
 - `PropTypes.object`

❖ Required Props

Props can be marked as required using `.isRequired`. If the prop is not passed, React will show a warning in development mode.

Example:

```
1 User.propTypes = {  
2   name: PropTypes.string.isRequired,  
3 };
```

Validating Arrays and Objects

❖ Arrays

You can validate arrays using `PropTypes.arrayOf()` to ensure all items in the array are of a specific type

```
1 Component.propTypes = {  
2   items: PropTypes.arrayOf(PropTypes.string).isRequired,  
3 };
```

Validating Arrays and Objects

❖ Objects

You can validate objects using `PropTypes.shape()` to specify the structure of the object.

```
1  Component.propTypes = {  
2    user: PropTypes.shape({  
3      name: PropTypes.string,  
4      age: PropTypes.number,  
5    }).isRequired,  
6  };
```

Advanced PropTypes Usage

❖ Functions as Props

PropTypes can validate functions passed as props using `PropTypes.func`.

```
1 Component.propTypes = {  
2   onClick: PropTypes.func.isRequired,  
3 };
```

Advanced PropTypes Usage

❖ Custom Validation

You can create custom prop validations using a function that checks if the prop meets specific criteria.

```
1  Component.propTypes = {  
2    isValid: (props, propName, componentName) => {  
3      if (!props[propName]) {  
4        return new Error(`${propName} is required in ${componentName}`);  
5      }  
6    },  
7  };
```

Component tutorials

1. Create SuperButton component: *Include props: Text, Icon, Color, Style, TextStyle, Disabled, Loading.*
2. Importing and Exporting Components.
3. Writing Markup with JSX.
4. Passing Props to a Component.
5. Conditional Rendering.
6. Apply PropTypes for component.

Managing State in React

Introduction to State in React

❖ **What is State?**

State is a built-in object used to hold data that may change over time.

❖ **Why is State Important?**

It helps components respond to user input and behave dynamically.

❖ **Where to Use State?**

Inside function components using the useState hook.

❖ **State vs Props**

Props are passed in; state is managed internally.

❖ **Triggering Rerenders**

Updating state causes React to re-render the component.

❖ **State Lifecycle**

From initialization to updates and unmounting.

Managing State in React

❖ What is State?

- State is a built-in object that stores dynamic data for a component, triggering re-renders when updated.

```
1 import { useState } from 'react';
2
3 export default function Counter() {
4   const [count, setCount] = useState(0);
5
6   return (
7     <div>
8       <p>Count: {count}</p>
9       <button onClick={() => setCount(count + 1)}>Increment</button>
10    </div>
11  );
12 }
```

Why State Matters

❖ Importance of State

- Controls re-rendering.
- Enables interactivity.
- Drives conditional rendering.
- Influences component lifecycle.
- Shared state can connect components.

useState Hook Basics

❖ useState Hook

- Syntax: `const [value, setValue] = useState(initialValue)`
- Used for primitive/local state.
- Triggers re-render on update.

```
1 function Toggle() {  
2   const [isOn, setIsOn] = useState(false);  
3   return <button onClick={() => setIsOn(!isOn)}>{isOn ? 'On' : 'Off'}</button>;  
4 }
```

Updating State Correctly

❖ **Avoid direct mutation:**

- Always use setState functions.
- State updates are asynchronous.
- Use previous state:
`setCount(prev => prev + 1)`

Initializing State

❖ Lazy Initialization

- Use a function for expensive initial state to compute it only once.

```
1  function App() {  
2    const [data, setData] = useState(() => expensiveComputation());  
3    return <div>{data}</div>;  
4  }  
5  
6  function expensiveComputation() {  
7    return Array(1000).fill('data');  
8  }
```

Updating Objects in State

❖ Copy and Update

- Spread objects to create a new copy when updating state properties.

```
1  function Form() {  
2    const [form, setForm] = useState({ name: '', age: 0 });  
3    const updateName = (e) => {  
4      setForm({ ...form, name: e.target.value });  
5    };  
6    return <input value={form.name} onChange={updateName} />;  
7  }
```


Updating Arrays in State

❖ Non-Mutative Updates

- Use array methods like map, filter, or spread to update arrays.

```
1  function Todos() {  
2    const [todos, setTodos] = useState(['Task 1']);  
3    const removeTodo = (index) => {  
4      setTodos(todos.filter((_, i) => i !== index));  
5    };  
6  
7    return <button onClick={() => removeTodo(0)}>Remove</button>;  
8  }
```

Batching State Updates

❖ Automatic Batching

- React groups multiple state updates in one render for performance.

```
1  function Counter() {  
2    const [count, setCount] = useState(0);  
3    const increment = () => {  
4      setCount(count + 1);  
5      setCount(count + 1); // Only increments once  
6    };  
7    return <button onClick={increment}>{count}</button>;  
8  }
```

Updater Functions

❖ Functional Updates

- Pass a function to the setter to update state based on its previous value.

```
1  import { useState } from 'react';  
2  
3  export default function UpdaterFunctions() {  
4    const [count, setCount] = useState(0);  
5    const increment = () => setCount((prev) => prev + 1);  
6    return <button onClick={increment}>{count}</button>;  
7  }
```

useEffect and State

❖ Side Effects with State

useEffect runs after state updates, useful for syncing state with external systems.

```
1  const [count, setCount] = useState(0);
2  useEffect(() => {
3    document.title = `Count: ${count}`;
4  }, [count]);
5
6  return <button onClick={() => setCount(count + 1)}>{count}</button>;
```

Lifting State Up

❖ Share State Between Components

- Move state to a common parent to share it with child components via props

```
1 export default function LiftingStateUp() {
2   const [count, setCount] = useState(0);
3
4   const handleIncrement = () => setCount(count + 1);
5   const handleDecrement = () => setCount(count - 1);
6
7   return (
8     <div>
9       <h1>Counter App</h1>
10      <Label text={count} />
11      <Counter count={count} onIncrement={handleIncrement} onDecrement={handleDecrement} />
12    </div>
13  );
14 }
```

Prop Drilling

❖ Passing State Down

- State is passed through multiple components, leading to complex prop chains.

```
1  export default function PropDrilling() {  
2    const [theme, setTheme] = useState('light');  
3    return <Middle theme={theme} />;  
4  }  
5  
6  function Middle({ theme }) {  
7    return <Child theme={theme} />;  
8  }  
9  
10 function Child({ theme }) {  
11   return <div>{theme}</div>;  
12 }
```

Sharing State with Context

❖ **Global State**
createContext
and **useContext**
provide state to
components
without prop
drilling

```
1  import React, { createContext, useContext, useState } from 'react';
2  const ThemeContext = createContext();
3
4  export default function ContextAPI() {
5    const [theme, setTheme] = useState('light');
6    return (
7      <div>
8        <h5>Context API Example</h5>
9        <ThemeContext.Provider value={theme}>
10          <button onClick={() => setTheme(theme === 'light' ? 'dark' : 'light')}>Toggle Theme</button>
11          <Child />
12        </ThemeContext.Provider>
13      </div>
14    );
15  }
16
17  function Child() {
18    const theme = useContext(ThemeContext);
19    return <div>{theme}</div>;
20  }
```

Complex State Logic

❖ useReducer Hook

useReducer manages state with a reducer function, ideal for complex updates.

```
1 import { useReducer } from 'react';
2
3 export default function UseReducerHook() {
4   const [state, dispatch] = useReducer(
5     (state, action) => {
6       switch (action.type) {
7         case 'increment':
8           return { count: state.count + 1 };
9         case 'decrement':
10          return { count: state.count - 1 };
11         case 'reset':
12          return { count: 0 };
13         default:
14          return state;
15       }
16     },
17     { count: 0 },
18   );
19   return (
20     <div>
21       <p>{state.count}</p>
22       <button onClick={() => dispatch({ type: 'increment' })}>Increment</button>
23       <button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>
24       <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
25     </div>
26   );
27 }
28
```


Combining useState and useReducer

❖ Flexible State Management

Use useState for simple state, useReducer for complex logic in the same component.

```
1 export default function CombineStateAndReducer() {
2   const [name, setName] = useState('');
3
4   const [todos, dispatch] = useReducer((state, action) => {
5     if (action.type === 'add') {
6       return [...state, action.payload];
7     }
8     if (action.type === 'remove') {
9       return state.filter((todo) => todo !== action.payload);
10    }
11  }, []);
12
13  return (
14    <div>
15      <input value={name} onChange={(e) => setName(e.target.value)} />
16      <button onClick={() => dispatch({ type: 'add', payload: name })}>Add</button>
17    </div>
18    <div>
19      {todos.map((todo, index) => (
20        <div key={index}>
21          {todo}
22          <button onClick={() => dispatch({ type: 'remove', payload: todo })}>Remove</button>
23        </div>
24      ))}
25    </div>
26  </>
27 );
28 }
```

State in Class Components

❖ Legacy State Management

Class components use **this.state** and **this.setState** for state, less common today.

```
1  import React from 'react';
2
3  export default class Counter extends React.Component {
4    state = { count: 0 };
5
6    increment = () => this.setState({ count: this.state.count + 1 });
7
8    render() {
9      return <button onClick={this.increment}>{this.state.count}</button>;
10   }
11 }
```

Component tutorials

1. Create CheckBox component
2. Create LikeButton component
3. Create Picture Viewer App
4. Create ToDo App