**Module 2**

**Topic 1: Dynamic events and how to handle them**

**Lecture 1: Types of events**

✅ **Understanding JavaScript Events**

* Events allow JavaScript to interact with HTML.
* Triggered by user or browser actions (e.g., clicks, keypresses).
* Enable enhanced interactivity on web pages.

✅ **How Events Work**

* Events "listen" in the background for interactions.
* HTML elements have associated events accessed via **event listeners**.
* Common example: clicking a button to trigger a function.

✅ **Event Triggering and Handling**

* Triggering: user interaction activates event handler.
* Example: clicking “Add to Cart” updates the cart count dynamically.
* Handlers execute specific code in response to actions.

✅ **Events in the DOM**

* Events are part of the **Document Object Model (DOM)**.
* Understanding events is essential for modern frontend development.

✅ **Introduction to React Events**

* Events in React are handled differently from plain JavaScript.
* Use **JSX event attributes** similar to HTML but with camelCase syntax.

✅ **JSX Event Attributes**

* HTML: onclick
* React JSX: onClick (camelCase format)
* CamelCase means starting with a lowercase letter and capitalizing subsequent words.

✅ **Types of Events in React**

* Events are grouped into categories such as:
  + **Mouse Events**: onClick, onDoubleClick, onContextMenu
  + **Clipboard Events**: onCopy, onCut, onPaste
  + **Keyboard Events** and many others.

✅ **Scope of Event Knowledge**

* Not all events need to be memorized.
* Focus on **commonly used events** relevant to your application needs.
* Many events are tied to specific use cases (e.g., drag-and-drop).

**Lecture 2: Common event handling**

**✅ Creating and Using a New React Component (Btn)**

* A new component named Btn was added to the src folder.
* The App.js file was simplified by:
  + Removing the logo import.
  + Importing the Btn component.
  + Clearing the return content and rendering <Btn />.

**✅ Rendering a Button Element**

* Inside Btn.js, a button element was added with the label: **Click Me**.
* Saved the file using Ctrl + S or Cmd + S, and the button rendered successfully on the screen.

**✅ Handling Click Events with JSX**

* onClick attribute was added to the button with a function reference: {clickHandler}.
* The return statement was reformatted over several lines for better readability.

**✅ Defining the Event Handler Function**

* A clickHandler was created using a **const arrow function**:

const clickHandler = () => {

console.log('clicked');

};

* On button click, the message **"clicked"** appears in the browser console.

**✅ Testing the Click Event**

* Developer tools were opened and the **Console** tab was used.
* Clicking the button printed **"clicked"** to the console each time.

**Lecture 3:** **Syntax for handlers**

**✅ What Are Events in the Browser?**

* Every user interaction like **clicking**, **scrolling**, or **closing a popup** produces an event.
* Events require **event handlers** to execute corresponding actions.
* Example: Click (event) → onClick (handler) → open menu (action).

**✅ HTML vs React Event Handling: Overview**

* HTML and React differ in **event handler syntax** and **implementation approach**.
* React promotes **declarative** coding; HTML often requires direct DOM manipulation.

**✅ HTML Event Handling Syntax**

* Example HTML button:

html

CopyEdit

<button id="js-btn" onclick="clickHandler()">Click Me</button>

* Uses onclick attribute directly within HTML with a function invocation.

**✅ JavaScript DOM-Based Event Handling**

* HTML is typically controlled using JavaScript via the **DOM**.
* Requires two steps:
  1. Select the element using:

const jsBtn = document.getElementById('js-btn');

* 1. Add event listener using:

jsBtn.addEventListener('click', clickHandler);

**✅ Why Prefer JavaScript Over Inline HTML Events**

* JavaScript provides **separation of concerns**, **cleaner logic**, and **flexibility**.
* Inline HTML event handlers are less maintainable and discouraged in modern apps.

**✅ React Event Handling: Key Differences**

* React discourages direct DOM manipulation.
* Uses **JSX event attributes** to handle events declaratively.
* React and HTML event names differ slightly:
  + HTML: onclick
  + React: onClick

**✅ JSX Syntax for Event Handling**

* Example in React:

<button onClick={clickHandler}>Click Me</button>

* Use **camelCase** for event names.
* Enclose function name in **curly braces {}**.
* Do **not invoke** the function directly; just **pass the reference**.

**✅ HTML vs React: Event Syntax Comparison**

|  |  |  |
| --- | --- | --- |
| Feature | HTML | React (JSX) |
| Event Name | onclick | onClick |
| Function Call | "clickHandler()" | {clickHandler} |
| Invocation | Function is called in quotes | Function reference only |

**✅ Passing Event Handlers as Props in React**

* You can pass event handler functions to child components using **props**.
* Example:

<Counter onClick={clickHandler} />

* Enables **component-level control** and **reusability**.

**✅ Key Takeaways**

* React and HTML have **similar concepts** but **different syntaxes** for events.
* React emphasizes:
  + Declarative syntax
  + JSX expressions
  + Avoiding direct DOM manipulation

**Lecture 4:** **Event handling and embedded expressions (Reading)**

**✅ Different Ways to Handle Events in React**

* You can handle events in React using **different types of functions**.
* All approaches produce the same result (e.g., logging to console), but differ in **syntax and readability**.

**✅ 1. Inline Anonymous ES5 Function**

* **Syntax:**

<button onClick={function() { console.log('first example') }}>

An inline anonymous ES5 function event handler

</button>

* ✅ Rarely used in React.
* ❌ Verbose and not idiomatic for modern React code.

**✅ 2. Inline Anonymous ES6 Arrow Function**

* **Syntax:**

<button onClick={() => console.log('second example')}>

An inline anonymous ES6 function event handler

</button>

* ✅ Very common and concise.
* ✅ Good for short, simple logic.
* ❌ Can reduce readability if logic gets too long.

**✅ 3. Separate Function Declaration**

* **Syntax:**

function App() {

function thirdExample() {

console.log('third example');

}

return (

<button onClick={thirdExample}>

using a separate function declaration

</button>

);

}

* ✅ Best for **complex logic** or **multi-line functions**.
* ✅ Improves **readability** and **maintainability**.
* ✅ Suitable when following a **clean coding style**.

**✅ 4. Separate Function Expression (Arrow Function Assigned to Const)**

* **Syntax:**

function App() {

const fourthExample = () => console.log('fourth example');

return (

<button onClick={fourthExample}>

using a separate function expression

</button>

);

}

* ✅ Most **modern** and **preferred** pattern in many React apps.
* ✅ Combines arrow function syntax with cleaner modular structure.
* ✅ Ideal for functions that grow in complexity.

**✅ Quick Tip: Declaration vs Expression**

* **Declaration**: Starts with function keyword.
* **Expression**: Assigned to a variable (const, let), often an arrow function.

**✅ Key Takeaways**

* **Use inline arrow functions** for short actions.
* **Use separate functions** for reusable or multi-line logic.
* **Understand all styles** to read and adapt code across different teams or companies.
* Pick the **most appropriate style** based on readability, complexity, and code guidelines.

**Topic 2: Data and Events**

**Lecture 1:** **Parent-child data flow**

✅ **Understanding One-Way Data Flow in React**

* React enforces **unidirectional (top-to-bottom)** data flow.
* Data moves **from parent to child** through the component hierarchy.
* Ensures predictable behavior and controlled data transmission.

✅ **Why One-Way Data Flow Matters**

* Prevents unexpected side effects and makes debugging easier.
* Centralizes data management in parent components.
* Supports a clean, scalable component structure.

✅ **Props as a Data Carrier (Stateless)**

* Props are **read-only** data passed from **parent ➝ child**.
* Child components **cannot modify** props — only consume and render them.
* Analogy: *Props = money given to you by employer (parent); you can use, not change the source*.

✅ **State for Internal Control (Stateful)**

* **State** is internal, **mutable** data managed by the component itself.
* Unlike props, components can **change their own state** using setState.
* Helps manage dynamic, interactive behavior in apps.

✅ **Difference Between Props and State**

|  |  |
| --- | --- |
| Props | State |
| Passed from parent | Managed internally |
| Read-only | Can be updated |
| Used to render content | Used to render and change |

✅ **Code Example Summary**

* **App.js**: A **class component** with internal **state** holding current date/time.
* **Child.js**: A **function component** that receives message as **prop** and displays it in <h1>.
* App's state ➝ passed as props ➝ consumed in child ➝ displayed in UI.

✅ **Stateless vs. Stateful Components in Practice**

* Stateless: Only **render data** passed to them (Child.js).
* Stateful: **Manage and update data** (App.js) and pass to children.

✅ **React Data Flow Recap**

* **Props**: Passed from **parent ➝ child**, immutable in child.
* **State**: Exists within a component, mutable by the same component.
* Together, props and state support **structured and interactive** UIs.

**Lecture 2:** **What Are Hooks?**

✅ **Introduction to React Hooks**

* Hooks add **interactivity** and **state management** to functional components.
* Introduced in **React 16.8**.
* Help manage **complex component logic** and reduce **code duplication**.

✅ **What Are Hooks?**

* Hooks are **functions** that let you "hook into" React features like state and lifecycle methods.
* Can only be used in **functional components** (not class components).
* Eliminate the need for HOCs and render props for logic reuse.

✅ **Using the useState Hook**

* useState is the **most common** hook.
* Allows you to **add state** to functional components.
* Must be **imported** from 'react':

import { useState } from 'react';

✅ **Declaring State with useState**

* Syntax (with array destructuring):

const [showMenu, setShowMenu] = useState(false);

* showMenu: state variable
* setShowMenu: function to update the state
* false: initial value

✅ **Why Array Destructuring?**

* useState returns an array: [currentState, setStateFunction]
* Destructuring improves **code readability** and **simplicity**
* Without it, you'd need to access items via index (e.g. state[0]), which is less clear.

✅ **Types of State You Can Track**

* Works with **Booleans, Strings, Numbers, Arrays, Objects**, etc.
* Example: Count button clicks with useState(0).

✅ **Best Practices with useState**

* Call at the **top level** of the component.
* **Don't call** hooks inside loops, conditions, or nested functions.
* You can use **multiple useState** calls in a single component.

✅ **Custom Hooks**

* You can create **your own hooks** to reuse logic across components.
* Improves **code modularity and reusability**.

✅ **Benefits of Hooks**

* Enhance **readability** and **simplicity**.
* Reduce need for **boilerplate code**.
* Enable better **logic organization** in functional components.

**Lecture 3:** **Using Hooks**

**✅** **Using Hooks in React**

* Hooks let you manage **state and side effects** in functional components.
* useState is used to **store and update local state** (like text input).
* State is **local to the component** – not accessible from outside.

**✅ Example: InputComponent with useState**

* Uses useState to track input value:

const [inputText, setText] = useState('hello');

* **handleChange function** updates state on user input:

function handleChange(e) {

setText(e.target.value);

}

* A **reset button** sets the state back to 'hello'.

**✅ State is Local**

* The variable inputText exists **only within** InputComponent.
* State in React is **component-scoped**, not global.

**✅ Rules of Using Hooks**

1. ✅ Call hooks **at the top level** of your component or custom hook.
2. ❌ Don't call hooks inside **loops, conditions, or nested functions**.
3. ✅ Hooks can only be called from **React function components** or **custom hooks**.

**✅ Multiple Fields: RegisterForm Example**

* Use **one useState call** for multiple form fields:

const [form, setForm] = useState({

firstName: 'Luke',

lastName: 'Jones',

email: 'lukeJones@sculpture.com'

});

* **Better readability** and easier updates than having multiple states.

**✅ Custom Hooks**

* Extract **reusable logic** into custom hooks.
* Examples: handling forms, timers, API calls, etc.

**✅ useRef Hook**

* Used to **access DOM elements** directly.
* Returns a **ref object** with a current property.
* Example: Focus an input field:

const inputEl = useRef(null);

inputEl.current.focus();

**Lecture 4:** **What Is State?**

**✅ React State vs Props**

* **Props**: Passed **from parent to child** to customize UI.
* **State**: Internal **data of a component**, can change over time.

**✅ Alarm Clock Analogy**

* Modes like **Alarm On**, **Alarm Off**, **Snooze** = internal behavior.
* In React, these modes can be handled using **State**, not just props.

**✅ What Is State?**

* **Internal data** in a component that controls **behavior & rendering**.
* Useful for dynamic content (e.g., toggles, counters, form inputs).
* State changes trigger **re-render** of component.

**✅ Why Use State?**

* Allows components to **track data** over time.
* Keeps **UI in sync** with data.
* Components **update automatically** when state changes.
* Enables **parent-to-child updates** via props.

**✅ Stateless vs Stateful Components**

* **Stateless**: Just render static content, no internal state.
* **Stateful**: Use useState() to manage dynamic behavior.

**✅ useState Hook Explained**

* Syntax:

js

CopyEdit

const [greet, setGreet] = useState("Hello");

* greet: current state value.
* setGreet: function to **update the state**.
* Changing state: setGreet("Hi!") triggers re-render.

**✅ Behind the Syntax – Array Destructuring**

* useState("Hello") returns an array:

["Hello", function to update]

* ES6 destructuring:

const [greet, setGreet] = useState("Hello");

**✅ Best Practices**

* Match state variable & updater function names:  
  Example: count & setCount, user & setUser
* Place useState **at the top** of the function.
* Avoid placing hooks **inside loops or conditions**.

**Lecture 5:** **Management Matters**

**✅ Why State Management Matters**

* As apps grow, managing state **across components** becomes more complex.
* Some components may need access to the **same piece of state**, even if they’re not directly related.

**✅ Example Scenario**

* App that tracks a **daily meal plan**:
  + **App.js** (Root)
  + **MealsList** (Lists today's meals)
  + **Counter** (Shows meals left to eat)
* Problem: Counter needs data from MealsList, but they’re **sibling components**.

**✅ The Problem**

* **Siblings** can’t directly share state.
* State in MealsList cannot be accessed directly by Counter.

**✅ Solution: Lifting State Up**

* Move state **up to the common ancestor**, in this case, App.js.
* Pass state down via **props**:
  + App → MealsList
  + App → Counter
* This enables both child components to share the same state.

**✅ Prop Drilling**

* When props are passed down through **multiple nested components**.
* Example: App → MealsList → MealItem.
* Problems with prop drilling:
  + Becomes **hard to manage** as component tree grows.
  + Components may receive props they **don’t directly use**.
  + Small change can lead to **many re-renders**.

**✅ Global State Problem**

* Some state (like meals) is needed in **many parts of the app**.
* If state is kept in App.js, it becomes **bloated**.
* Not all state belongs in the top-level component.

**✅ Elegant Solution: React Context API**

* Provides a way to **share state globally** without prop drilling.
* Components can **consume context directly**, no middleman.
* Steps:
  1. **Create a Context**
  2. **Wrap** your app with a **Context Provider**
  3. **Use useContext()** in any component to access the state

**✅ Benefits of Context API**

* **No prop drilling**.
* Centralized **global state**.
* Cleaner and more **scalable** code.

**Lecture 6: React Context API**

✅ **Challenges of State Management in Complex React Apps**

* Managing state between parent and child components becomes harder as component levels increase.
* Prop drilling (passing props down multiple levels) becomes inefficient and messy.

✅ **Introducing React Context API**

* Context API provides a cleaner way to **share state across multiple levels** without prop drilling.
* It works like "teleporting" state directly to components that need it, skipping intermediate levels.

✅ **Setting Up the Context API**

* Create context using React.createContext().
* Define a **Provider component** to wrap parts of the app that need access to the state.
* The Provider holds the state and shares it via the value prop.
* Components that consume this state use useContext() to access it.

✅ **Using Context API: Example Overview**

* App.js imports:
  + MealsProvider (Context Provider)
  + MealsList and Counter (components that use the context)
* MealsProvider:
  + Defines the context and initial state (todayMeals array).
  + Wraps children in MealsContext.Provider and passes the meals value.
  + Also defines a custom hook useMealsListContext to simplify context access.

✅ **Accessing Context in Components**

* In MealsList, use the useMealsListContext hook to get meals.
* Destructure the array and use .map() to render meals as <h2> elements.
* Counter accesses the same context using the same hook — **no prop drilling needed**.

✅ **Benefits of Using Context API**

* Centralized state management.
* No need for lifting state up or prop drilling.
* Cleaner, more maintainable code structure.

✅ **Enhancing State Logic with useReducer Hook**

* UseReducer is like a **supercharged version of useState**.
* Accepts:
  + An initial state.
  + A **reducer function** to define how state should change.
* dispatch() is used to send actions that the reducer processes.
* Great for **complex state transitions**, such as a rideshare wallet:
  + Add money on customer pickup.
  + Subtract money on refueling.

✅ **Key Takeaways**

* Use Context API to share state **globally** across components.
* Use useContext() for **simple access** to shared state.
* Use useReducer() for **complex state logic**.
* These tools eliminate the need for prop drilling and simplify managing state in large apps.

**Example:**

**✅ 1. MealsContext.js (Context + Provider + Reducer)**

// MealsContext.js

import React, { createContext, useContext, useReducer } from 'react';

// Initial state

const initialState = {

meals: ['Salad', 'Chicken', 'Rice'],

consumed: 0,

};

// Reducer function

function mealsReducer(state, action) {

switch (action.type) {

case 'CONSUME\_MEAL':

return {

...state,

consumed: state.consumed + 1,

};

default:

return state;

}

}

// Create Context

const MealsContext = createContext();

// Provider component

export function MealsProvider({ children }) {

const [state, dispatch] = useReducer(mealsReducer, initialState);

return (

<MealsContext.Provider value={{ state, dispatch }}>

{children}

</MealsContext.Provider>

);

}

// Custom hook for consuming context

export function useMealsContext() {

return useContext(MealsContext);

}

**✅ 2. MealsList.js (List meals and consume them)**

// MealsList.js

import React from 'react';

import { useMealsContext } from './MealsContext';

function MealsList() {

const { state, dispatch } = useMealsContext();

const { meals } = state;

return (

<div>

<h2>Today's Meals</h2>

{meals.map((meal, index) => (

<div key={index}>

<h3>{meal}</h3>

<button onClick={() => dispatch({ type: 'CONSUME\_MEAL' })}>

Mark as Consumed

</button>

</div>

))}

</div>

);

}

export default MealsList;

**✅ 3. Counter.js (Show number of meals consumed)**

// Counter.js

import React from 'react';

import { useMealsContext } from './MealsContext';

function Counter() {

const { state } = useMealsContext();

const total = state.meals.length;

const remaining = total - state.consumed;

return (

<div>

<h2>Meals Left: {remaining} / {total}</h2>

</div>

);

}

export default Counter;

**✅ 4. App.js (Wrap everything with the Provider)**

// App.js

import React from 'react';

import { MealsProvider } from './MealsContext';

import MealsList from './MealsList';

import Counter from './Counter';

function App() {

return (

<MealsProvider>

<div className="App">

<h1>Healthy Meal Tracker</h1>

<MealsList />

<Counter />

</div>

</MealsProvider>

);

}

export default App;