**Module 1**

**Topic 1: Course Introduction**

**Lecture 1:** **Introduction to the course**

✅ **Course Overview & Progression**

* Continues from the React Basics course in the specialization.
* Covers **advanced concepts and practical applications** of React.
* Focus on applying concepts in real-world projects like the **Little Lemon** restaurant app and a final **portfolio project**.

✅ **Module 1: Lists, Forms & Context**

* Learn to **render and transform lists** using the map() method.
* Understand the importance of **keys** in list rendering.
* Dive into **controlled components** and build a **feedback form**.
* Review **props and state**, and explore **React Context API** for managing global state.

✅ **Module 2: Deep Dive into Hooks**

* Understand the **purpose and rules** of React Hooks.
* Learn key hooks:
  + useState
  + useEffect
  + useReducer
* Build and apply **custom hooks** in real scenarios.

✅ **Module 3: Advanced JSX & Patterns**

* Explore **components vs. elements** and different types of JSX children.
* Study **component composition** and use of the **children prop**.
* Use the **spread operator** in JSX props.
* Learn advanced patterns like:
  + **Higher-Order Components (HOCs)**
  + **Render Props**
  + Handle **cross-cutting concerns** efficiently.

✅ **Module 4: Performance, Testing & Debugging**

* Improve **performance** of React applications.
* Learn **testing and debugging techniques**.
* Write **integration tests** using **React Testing Library**.
* Focus on testing **form behavior** and applying tests in real-world scenarios.

✅ **Final Module: Capstone Project**

* Build a **portfolio app** as a lab project to apply advanced skills.
* Complete a **graded assessment** to demonstrate your understanding.

**Topic 2:** **Rendering Lists in React**

**Lecture 1:** **Transforming lists in JavaScript**

✅ **Why *list manipulation* matters in apps**

* Lists are common in apps—e.g., **menus in food ordering apps**.
* Displaying user-friendly lists often requires **transforming raw data** from third-party sources.

✅ **Introducing the JavaScript *map method***

* The map() method creates a **new array** by transforming elements from an existing array.
* It's ideal for shaping and **filtering data** before rendering in the UI.

✅ **Practical example: *Little Lemon's dessert list***

* Suppose the data array includes dessert objects with:
  + id, title, image, description, price.
* Goal: Display simplified objects with just two properties:
  + content: combines title and description using a dash -.
  + price: passed directly from the original object.

✅ **Steps to *transform* the list**

1. Define a new variable: topDesserts = data.map(...).
2. Use map() to return a **new object** with only needed properties.
3. Log the result with console.log() to verify structure.

✅ **Why the map method is *powerful***

* Simplifies messy or excessive external data.
* Makes apps easier to use by presenting **clear and focused information**.
* Essential when working with arrays, especially when consuming APIs or services

**✅ Example**

// Original data from third-party (e.g., API)

const data = [

  {

    id: 1,

    title: "Chocolate Cake",

    image: "chocolate.jpg",

    description: "Rich and creamy",

    price: "$5"

  },

  {

    id: 2,

    title: "Lemon Tart",

    image: "lemon.jpg",

    description: "Tangy and sweet",

    price: "$4"

  },

  {

    id: 3,

    title: "Vanilla Ice Cream",

    image: "vanilla.jpg",

    description: "Classic delight",

    price: "$3"

  }

];

// Transform the data using map()

const topDesserts = data.map(dessert => {

  return {

    content: `${dessert.title} - ${dessert.description}`,

    price: dessert.price

  };

});

// Output the transformed list

console.log(topDesserts);

✅ Output:

[

  { content: "Chocolate Cake - Rich and creamy", price: "$5" },

  { content: "Lemon Tart - Tangy and sweet", price: "$4" },

**Lecture 2: Render a simple list component**

✅ **Transforming *lists into components* with React**

* In React, you can convert a list of items into a **collection of components** using the map() method.
* Useful for dynamically rendering UI based on structured data (e.g., **Little Lemon's dessert list**).

✅ **JSX and *component rendering***

* JSX is a **syntax extension** to JavaScript used to **describe UIs** in React.
* A component in React is a function that **returns JSX**.
* You can return **semantic HTML tags** like <li> directly in JSX.

✅ **Rendering list items: *step-by-step guide***

1. **Start with a data array** (e.g., desserts with id, title, image, description, price).
2. Create a variable listItems using map() to loop through each dessert item.
3. Inside map(), define a new variable like itemText = title + " - " + price.
4. Return an <li> element with itemText wrapped in **curly braces {}** to render dynamic content.
5. In the return statement of your component, wrap listItems inside a <ul> to create an unordered list.

✅ **Important JSX considerations**

* In JSX, **curly braces {}** are used to evaluate expressions (like variables or function calls).
* Each element returned in a list **should have a unique key prop** (not shown here, but important for React performance).

✅ **Why this matters in *React development***

* Lists are a core part of most apps—learning to **render them dynamically** helps in building scalable UIs.
* Using map() with JSX ensures your UI is **data-driven, declarative**, and easy to maintain.

**Lecture 3:** **What are Keys in React?**

✅ **React’s *Autopilot and Manual Controls***

* React optimizes UI updates **automatically** using a diffing algorithm.
* In certain situations, **manual guidance** is needed for better performance—like adding keys to lists.

✅ **What are *Keys* in React?**

* **Keys** are unique identifiers used to help React **track and manage** elements in a list.
* They enable React to identify which items were **added, removed, or moved** during updates.
* Proper use of keys helps **preserve internal state** and improves performance

✅ **Why Keys Matter for *Performance***

* Without keys: React may **re-render or mutate every child** unnecessarily.
* With proper keys: React understands **which elements have changed** and can reuse DOM efficiently.

✅ **Choosing the *Right Key***

* Always use a **stable, unique ID** from your data (like a database ID).
* IDs help React **avoid unnecessary recreation** of unchanged elements.
* Example: When adding "Cider" to a list with "Beer" and "Wine," keys let React identify the **new item vs. moved items**.

✅ **What *Not* to Use as Keys**

* ❌ Avoid using Math.random() or libraries that generate new keys on each render.
  + These cause **new keys every time**, breaking state retention and reusability.
* ❌ Avoid using **array index** as a key if the list can change in order.
  + React may misidentify items, leading to **UI bugs** and inefficient rendering.

✅ **When Index is *Acceptable***

* Only use index as a key **when:**
  + The list is **static**
  + Items are **not reordered**, added, or removed
  + No **dynamic operations** are applied on the list

✅ **Summary: Best Practices for Keys**

* ✔️ Use **unique, stable IDs** from your data when available.
* ⚠️ Use **indexes only as a last resort**, especially when list order doesn't change.
* ❗ Incorrect key usage can lead to **glitches** and **performance issues**.

**Topic 3: Forms in React**

**Lecture 1:** **What are controlled components?**

✅ **Forms Are *Everywhere***

* Forms appear **frequently online**: newsletter subscriptions, sign-ups, reservations, etc.
* React devs will **regularly implement** forms in their apps.

✅ **React vs *Traditional HTML Forms***

* **HTML forms** manage their own internal state and use action for form submission.
* React takes a different approach: it **controls form inputs via state**.
* DOM structure is still tree-like, but **React controls updates** through re-renders.

✅ **What Are *Controlled Components***

* React form elements where the **component state controls input values**.
* The **state becomes the source of truth** instead of the DOM.
* Uses:
  + value prop → sets displayed input value
  + onChange prop → updates the state from user input

✅ **How to *Implement Controlled Components***

* **Step 1**: Declare local state with useState
* **Step 2**: Use value={state} on input elements
* **Step 3**: Add onChange={handleChange} to update state
* **Step 4**: Use onSubmit={handleSubmit} on <form> for full control
* In handleSubmit, use event.preventDefault() to stop native submission

✅ **Example Use Case: *Table Reservation Form***

* Little Lemon app uses a controlled form for table reservations
* React **handles the logic**, validates data, and submits it manually
* All input data stays inside React’s **controlled state system**

✅ **Why Controlled Components Matter**

* Provide **predictability** and **real-time updates**
* Enable advanced features like:
  + Form validation
  + Conditional rendering
  + Live feedback to users
* Most input types have controlled versions in React

✅ **Uncontrolled Elements Still Exist**

* Some inputs remain **uncontrolled by default** (e.g., file inputs)
* Controlled vs. Uncontrolled depends on the **use case and required control**

✅ **Key Takeaway**

* Use **controlled components** for consistent, reliable forms
* They give React **full control over user input and submission logic**

**Lecture 2:** **Controlled components vs. Uncontrolled components**

✅ **Introduction: *Controlled vs. Uncontrolled Inputs***

* React recommends **controlled components**, but **uncontrolled** fields are valid in simple use cases.
* Key difference: *where the input state lives*—React state vs. DOM.

✅ **What Are *Uncontrolled Components***

* DOM **maintains its own state** (just like plain HTML).
* You use **useRef()** to **pull input values manually** on actions like submit.
* Example:

jsx

CopyEdit

const inputRef = useRef();

const value = inputRef.current.value;

* Best for **simple forms** with **no dynamic feedback or validation**.

✅ **What Are *Controlled Components***

* Input values are **tied to React state** using the value and onChange props.
* Example:

const [value, setValue] = useState("");

<input value={value} onChange={(e) => setValue(e.target.value)} />

* React **pushes changes to inputs**, so the UI and data are **always in sync**.

✅ **Benefits of Controlled Components**

* Enables **real-time features**:
  + ✅ **Instant validation** per field
  + ✅ **Disable submit** if inputs are invalid
  + ✅ **Input masking** (e.g., phone, credit card)
  + ✅ **Dynamic input control** (e.g., show/hide fields)
  + ✅ Multiple inputs for **one piece of state**

✅ **Special Case: *File Inputs Are Always Uncontrolled***

* <input type="file" /> cannot be controlled in React.
* File values are **read-only** and must be accessed via ref.
* Example:

const fileInput = useRef();

const files = fileInput.current.files;

✅ **When to Use Each Type**

* Use **uncontrolled components** when:
  + You only need to get values **once (on submit)**
  + **No validation or feedback** is required
  + Simpler, less overhead
* Use **controlled components** when:
  + You need **real-time validation**
  + **Dynamic behavior** (enable/disable buttons, live formatting, etc.)
  + **Better control and testability**

✅ **Feature Comparison Table**

|  |  |  |
| --- | --- | --- |
| Feature | Uncontrolled | Controlled |
| One-time value retrieval (e.g., on submit) | ✅ Yes | ✅ Yes |
| Validating on submit | ✅ Yes | ✅ Yes |
| Instant field validation | ❌ No | ✅ Yes |
| Conditionally disabling submit button | ❌ No | ✅ Yes |
| Enforcing specific input format | ❌ No | ✅ Yes |
| Several inputs for one piece of data | ❌ No | ✅ Yes |
| Dynamic inputs (e.g., add/remove fields) | ❌ No | ✅ Yes |

✅ **Conclusion**

* **Controlled components** are ideal for **interactive, complex forms**.
* **Uncontrolled components** are simpler and work for **basic use cases**.
* Choose based on **your project’s requirements**.

**Topic 4: React Context**

**Lecture 1:** **What you know about Props and State**

✅ **What Are Props and State?**

* Both are **plain JavaScript objects**
* Both hold data that determines **how components render**
* Both changes **trigger a re-render**
* Both are **deterministic**: same inputs = same output

✅ **Key Difference**

|  |  |  |
| --- | --- | --- |
| Feature | Props | State |
| Source | Passed **from parent** | **Managed internally** by component |
| Mutability | **Immutable** (cannot be changed) | **Mutable** (via setState) |
| Analogy | Like **function parameters** | Like **function variables** |

✅ **When to Use Props vs State**

* Use **props** when:
  + Data is **passed in from parent**
  + Component doesn't need to **modify** the data
* Use **state** when:
  + Component needs to **track changes** over time
  + Data is influenced by **user interaction**, timers, etc.

📌 **Rule of Thumb**:

“If the component **needs to change** the data, use **state**. If not, use **props**.”

✅ **Props (Properties)**

* **Configuration** of a component
* **Immutable** within the component
* Passed **top-down** from parent
* Component can define **props for its children**

✅ **State**

* **Private** data managed within the component
* Initialized when the component **mounts**
* **Updated over time** via user interaction or app logic
* **Triggers re-render** on change

🕰️ Think of state as a **snapshot** of data at one moment in time

✅ **Stateless vs Stateful Components**

|  |  |
| --- | --- |
| Type | Description |
| Stateless | Only use **props**, no internal state. Simple and easy to test. |
| Stateful | Use **both props and state**. Handle logic, events, data fetching, etc. |

📌 **Best Practice**:

Keep most components **stateless**. Only a few should manage **state** (to reduce complexity).

✅ **Best Practices for Using State**

* Use state **sparingly** to avoid performance issues
* **Lift state up** if multiple components need to access/modify the same data
* Avoid **deeply nested stateful components**

**Lecture 3:** **What is Context, and why is it used?**

✅ **Why Context Was Introduced**

* In **typical React apps**, data flows **top-down** via **props**
* Some data (e.g., **theme**, **locale**, **authenticated user**) is needed by **many components**
* Passing data through every level = **prop drilling problem**
* **Prop drilling** = passing data through intermediate components that **don’t use it**
* **Context API** solves this by allowing **global state** to be shared **without drilling**

✅ **What Is Context?**

* A **React API** for sharing **global data** across the component tree
* Allows components to **subscribe to data** without manually receiving props
* Ideal for data like:
  + **Theme settings** (light/dark)
  + **Authenticated user info**
  + **Locale or language preference**

✅ **Example Scenario: Little Lemon Blogging Platform**

* Components:
  + **Header**: shows app title + logged-in user
  + **Page**: shows blog posts + author's name
* Both Header and Page need **user info** → a perfect use case for **Context**

✅ **Steps to Create and Use Context**

1. **Create Context**

import { createContext } from "react";

const UserContext = createContext(undefined);

1. **Create Provider Component**

import { useState } from "react";

const UserProvider = ({ children }) => {

const [user, setUser] = useState({ name: "Alex" }); // sample user

return (

<UserContext.Provider value={user}>

{children}

</UserContext.Provider>

);

};

1. **Create a Custom Hook (optional, but recommended)**

import { useContext } from "react";

const useUser = () => useContext(UserContext);

1. **Wrap App with Provider**

import { UserProvider } from "./UserContext";

const App = () => (

<UserProvider>

<Header />

<Page />

</UserProvider>

);

1. **Consume Context in Components**

import { useUser } from "./UserContext";

const Header = () => {

const user = useUser();

return <h3>Welcome, {user.name}</h3>;

};

✅ **Important Notes**

* **Context should be used sparingly**
* For most cases, **props and state** are sufficient and easier to trace
* Use Context **only when multiple components at different levels** need the same data

✅ **Recap: When to Use What**

|  |  |
| --- | --- |
| Use Case | Tool |
| Component-specific data | useState (state) |
| Data from parent only | props |
| Global/shared data | Context API |

**Lecture 4:** **How re-rendering works with Context**

**✅ Default React Re-rendering**

* When a **component re-renders**, **all its children** re-render **recursively**, even if their **props or context didn't change**.
* This behavior can cause **performance issues** in deeply nested or complex component trees.

**✅ Using Context in Component Tree**

* Example tree: App (ContextProvider) > A > B > C
* If App re-renders, all: A -> B -> C re-render, even if **only C uses context**.

**✅ Avoiding Unnecessary Re-renders with React.memo**

* Use React.memo() to **memoize** components that:
  + Render the **same output** given the same props.
  + **Don't need to re-render** every time a parent does.
* Example:

const ComponentA = React.memo(() => <ComponentB />);

**✅ React Context Triggers Re-renders on Value Change**

* All **context consumers** will re-render **if the value prop** of the provider changes.
* Even if the **object's content** is the same, a **new object reference** triggers re-renders.

**✅ Understanding Object Reference in JavaScript**

* {a: 'hi', b: 'bye'} !== {a: 'hi', b: 'bye'} → **Not equal** due to different **object references**.

**✅ Using useMemo to Stabilize Object Reference**

* Wrap the value in useMemo() to prevent it from changing on every re-render:

const value = useMemo(() => ({a, b}), [a, b]);

* This ensures **ComponentC** (a context consumer) **does not re-render** unless a or b change

**✅ Summary**

* **Context triggers re-renders** of all consumers if the **value reference changes**.
* Use React.memo() to skip rendering components if their **output hasn’t changed**.
* Use useMemo() to ensure **stable references** and avoid **unnecessary context updates**.