**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**.

**Module 2**

**Topic 1:** **Getting started with hooks**

**Lecture 1:** **Working with React hooks**

**✅ Benefits of Using Hooks**

* **Simplifies applications**.
* Improves **performance**.
* Makes code more **readable**, **manageable**, and **reusable**.
* Helps **modularize logic** previously tied up in **class components**.

**✅ Problem with Class Components**

* Became **bloated** and **hard to manage** over time.
* Difficult to **break into smaller, reusable pieces**.

**Lecture 2:** **Revising useState hook**

**✅ What is useState Used For?**

* Manages **state** in **React components**.
* State is **dynamic data** that can **change over time**.
* Example: Tracking the **restaurant name** or **food inventory**.

**✅ Understanding Array Destructuring**

* Used to **extract values** from an array and assign them to variables.
* Syntax:

const [v1, v2] = ['carrots', 'potatoes'];

* In contrast, **object destructuring** requires **exact property names**.

**✅ Why useState Returns an Array**

* To **leverage array destructuring** freely with any variable names.
* It returns a **2-member array**:

const [stateVariable, setStateFunction] = useState(initialValue);

**✅ Example: Restaurant Name**

const [restaurantName, setRestaurantName] = useState("Lemon");

* restaurantName: Holds current state.
* setRestaurantName: Function to **update state**.

**✅ Correct Way to Update State**

* **Only use the updater function** returned from useState.
* ❌ **Do not** assign directly like restaurantName = "Little Lemon".
* ✅ Use:

setRestaurantName("Little Lemon");

**✅ State Changes via Events**

* Typically triggered by **user actions** (e.g., button clicks).
* Example:

<button onClick={updateRestaurantName}>Update Name</button>

**✅ Conventions**

* State updater functions follow camelCase with a **set prefix**:
  + restaurantName → setRestaurantName
  + count → setCount

**✅ Visual Example Flow**

1. Initial state: "Lemon" is displayed.
2. User clicks the button.
3. setRestaurantName("Little Lemon") is called.
4. React re-renders the component showing "Little Lemon".

**Lecture 3:** **Using the useState hook**

**✅ App Overview**

* The app tracks restaurant goals with two key inputs: **Goal** and **By** (the timeframe).
* The app is built using three components:
  1. **GoalForm**: Captures new goals via a form.
  2. **ListOfGoals**: Displays all previously added goals.
  3. **App**: Combines the above components and handles state updates.

**✅ State Management with useState**

* **GoalForm** component uses useState to manage form data (goal and timeframe).

const [formData, setFormData] = useState({ goal: "", by: "" });

**✅ Handling Form Data**

* **Change Handler**:
  + Updates the formData state when user inputs data.
  + Uses the **spread operator** to copy the existing state, then updates the relevant property dynamically using **bracket notation**.

const changeHandler = (e) => {

setFormData({

...formData,

[e.target.name]: e.target.value

});

};

* **Submit Handler**:
  + Sends the form data to the **App** component through the onAdd prop.
  + After submission, resets the form fields.

const submitHandler = (e) => {

e.preventDefault();

onAdd(formData); // Sends form data to App

setFormData({ goal: "", by: "" }); // Resets form

};

**✅ Managing Goals in the App Component**

* **App State**: The App component maintains the allGoals state, which tracks all goals.

const [allGoals, setAllGoals] = useState([]);

* **Add Goal Function**:
  + Adds a new goal to the allGoals state by invoking the **state updating function** (setAllGoals).

const addGoal = (goalEntry) => {

setAllGoals([...allGoals, goalEntry]);

};

* **Passing addGoal Function**: The addGoal function is passed down to the **GoalForm** component via props (onAdd), allowing the goal to be added from the form.

**✅ ListOfGoals Component**

* Maps over the allGoals array and displays each goal in an unordered list.

**✅ Key Takeaways**

* **State Immutability**: React requires state to be updated immutably. You cannot mutate the state directly.
* **Using useState**: Initialize state using useState, and always use the setter function (e.g., setFormData) to update the state.
* **State in Forms**: Track form input values using state, and update it using change handlers.
* **Props for State Management**: Pass functions down as props for child components to update parent state.

**Lecture 4:** **Side Effects in React**

✅ **Understanding *Side Effects*** in React

* A **side effect** is anything a function does that affects something **outside its scope**, like:
  + Calling console.log()
  + Fetching data from an API
  + Using browser APIs (e.g. geolocation)

✅ **Pure vs Impure Functions**

* **Pure Function**:
  + Returns the **same output** for the **same input**.
  + Doesn’t interact with external systems.
  + Example:

function EstablishedYear({ year }) {

return <h2>Established Year: {year}</h2>;

}

* **Impure Function**:
  + Has **side effects** (e.g., API calls, logging).
  + Example:

function ShoppingCart({ total }) {

console.log(total); // Side effect

return <h2>Total: {total}</h2>;

}

✅ **Dealing with Side Effects Using useEffect**

* **useEffect()** helps isolate side effects within React components.
* Syntax:

useEffect(() => {

// Side effect logic (e.g., API call, logging)

}, [dependencies]);

* + First argument: a **callback function** where you write the side effect.
  + Second argument: a **dependency array** (can be empty for one-time execution).
* Example (fixing the impure ShoppingCart):

useEffect(() => {

console.log(total);

}, [total]);

✅ **Key Concepts Recap**

* **Pure functions** → predictable and side-effect free.
* **Impure functions** → perform actions outside their scope.
* Use **useEffect** to handle impure actions safely in React.

**Lecture 5: Using the useEffect hook**

✅ **Initial Component Setup**

* A React component is built with:
  + A **toggle state** (useState) to track the welcome message visibility.
  + A **button** that toggles this state via clickHandler.
  + A conditional <h2> tag that displays "Welcome to Little Lemon" when toggle is true.

✅ **Side Effect Requirement**

* The restaurant owner wants the **browser tab title** to:
  + Display “Welcome to Little Lemon” when the message is shown.
  + Show “Using the useEffect hook” when it is hidden.
* This requires a **side effect**, which is done using the useEffect hook.

✅ **Adding the useEffect Hook**

useEffect(() => {

document.title = toggle ? 'Welcome to Little Lemon' : 'Using the useEffect hook';

});

* This **runs after every render**, because no dependency array is provided.
* It updates the **browser tab title** based on the value of toggle.

✅ **Optimizing with Dependency Array**

1. **Empty array []**:

useEffect(() => {

document.title = 'Using the useEffect hook';

}, []);

* + Runs **only once** on **initial render**.
  + Tab title won’t change after that, regardless of state updates.

1. **Tracking toggle**:

useEffect(() => {

document.title = toggle ? 'Welcome to Little Lemon' : 'Using the useEffect hook';

}, [toggle]);

* + Now useEffect **only runs when toggle changes**.
  + This is the correct behavior for updating the tab title conditionally.

✅ **Key Learnings**

* Use useEffect to handle **DOM updates, API calls, subscriptions**, etc.
* Control when useEffect runs using the **dependency array**:
  + []: Run only once.
  + [stateVar]: Run only when stateVar changes.
* Helps improve **performance** and **logic separation** in components.

**Topic 2: Rules of Hooks and Fetching Data with Hooks**

**Lecture 1:** **What are the rules of hooks?**

✅ **Rules of Hooks in React**

**4 main rules** of React hooks and why they matter:

✅ **1. Call hooks only from React functions**

* ❌ **Don’t use hooks** (like useState, useEffect) in regular JavaScript functions.
* ✅ **Only use hooks inside:**
  + Functional React components
  + Custom hooks
* ⚠️ *You can still use the state-setting function (e.g., setState) anywhere inside the component logic.*

✅ **2. Call hooks at the top level only**

* ❌ Don’t place hooks inside:
  + Loops
  + Conditionals
  + Nested functions
* ✅ Always place hooks **before return**, and at the **top level** of your component.
* ✅ To handle conditions, wrap logic **inside** the hook:

useEffect(() => {

if (condition) {

// Do something

}

}, []);

✅ **3. You can call multiple hooks**

* ✅ It's valid to use **multiple useState or useEffect calls** in one component.
* ⚠️ Just ensure they're always called in the **same order**, every render.

✅ **4. Always call hooks in the same sequence**

* ❌ Never put hook calls inside conditions or blocks that may **skip execution**.
* ⚠️ Violating this rule leads to **“Invalid hook call”** errors.
* ✅ Keep hook calls **deterministic and static** (same sequence on every render).

✅ **Recap: The Four React Hook Rules**

1. ✅ **Only in React functions** (component or custom hook)
2. ✅ **Only at top level** (not in conditionals or loops)
3. ✅ **Multiple hooks allowed**
4. ✅ **Call hooks in the same order every time**

Following these ensures **stable, predictable** behavior in your React components using hooks.

**Lecture 2:** **What you need to know before fetching data**

✅ **JavaScript is single-threaded**

* JavaScript can **only execute one task at a time**.
* Tasks are performed **in order**, and one must finish before the next begins.

✅ **The Fetch API enables asynchronous operations**

* **fetch()** allows JavaScript to request data **without blocking** the rest of the code.
* It **delegates the task** to the browser (like calling another post office clerk).
* This delegation is part of **asynchronous JavaScript**.

✅ **Browser APIs as “extra clerks”**

* Metaphor:
  + JavaScript = a **clerk** at the post office
  + Browser APIs (like Fetch) = **additional clerks**
  + Browser = the **post office building**
* JavaScript assigns tasks to browser APIs and **continues processing** the next task.

✅ **fetch() is a *facade function***

* Appears to be part of JavaScript, but is actually a **bridge to a browser API**.
* **Not natively handled by JS** — JavaScript uses fetch() to **call the browser's fetch capability**.

✅ **Code execution sequence example**

console.log("Another customer approaching");

fetch("https://randomuser.me/api/")

.then(response => response.json())

.then(data => console.log(data));

console.log("Our valued customer, please wait...");

🟢 **Output order**:

1. "Another customer approaching"
2. "Our valued customer, please wait..."
3. (After async fetch completes) user data is logged

✅ **Key takeaways**

* **JavaScript does not wait** for fetch() to complete — it moves on.
* **Async behavior** allows JavaScript to stay responsive.
* You must understand this **delegation concept** before fetching data in **React**.

**Lecture 3: Data Fetching Using Hooks in React (Reading)**

✅ **Fetching data is a *side-effect***

* In React, **side-effects** like API calls are handled using the useEffect hook.
* This ensures that data fetching occurs **after the initial render**, not during it.

✅ **Basic fetch with useEffect**

import { useState, useEffect } from "react";

export default function App() {

const [btcData, setBtcData] = useState({});

useEffect(() => {

fetch("https://api.coindesk.com/v1/bpi/currentprice.json")

.then((response) => response.json())

.then((jsonData) => setBtcData(jsonData.bpi.USD))

.catch((error) => console.log(error));

}, []);

}

* [] means this effect runs **once on mount**.
* .then() handles the promise returned by fetch().

✅ **Clean structure with external function**

const fetchData = () => {

fetch("https://api.coindesk.com/v1/bpi/currentprice.json")

.then((res) => res.json())

.then((data) => setBtcData(data.bpi.USD))

.catch((err) => console.log(err));

};

useEffect(() => {

fetchData();

}, []);

* Improves readability by **declaring the fetch logic separately**.

✅ **Why conditional rendering is important**

* API calls may **fail** or **take time**, so rendering must adapt accordingly.
* Use conditional rendering to show a loading state or fallback UI.

✅ **Simple conditional rendering example**

return someStateVariable.length > 0 ? (

<div>

<h1>Data returned:</h1>

<h2>{someStateVariable.results[0].price}</h2>

</div>

) : (

<h1>Data pending...</h1>

);

* If someStateVariable.length > 0, data is shown.
* Otherwise, shows a **“Data pending…”** loading state.

✅ **Tips for correct setup**

* Initialize your state properly (e.g., useState([]) for arrays).
* Always handle **errors** to prevent UI from crashing.
* Render appropriate UI **based on state** (empty, error, or loaded)

✅ **Conclusion Recap**

* useEffect is required to fetch data in React because it's a **side-effect**.
* Data fetching should be done **cleanly and conditionally**.
* Providing **fallback rendering** improves user experience during loading or failure.

**Lecture 4:** **Fetching data – Putting it all together**

✅ **Use Case: Restaurant Giveaway**

* Little Lemon restaurant wants to run a **giveaway** for app users.
* A **random user** is selected using **randomuser.me API**.
* The app uses **React** to fetch and display this random user data.

✅ **App Logic Flow**

1. **Initialize state**

const [user, setUser] = useState([]);

* + State starts as an **empty array**.

1. **Fetch data function**

const fetchData = () => {

fetch("https://randomuser.me/api/")

.then((res) => res.json())

.then((data) => setUser(data.results[0]))

.catch((err) => console.error(err));

};

* + Fetches a **random user**.
  + Updates state with **data.results[0]** (the user object).
  + **No hooks** are used inside this function — complies with rules of hooks.

1. **useEffect hook**

useEffect(() => {

fetchData();

}, []);

* + Calls fetchData on **initial render**.

✅ **Conditional Rendering Logic**

return Object.keys(user).length > 0 ? (

<div>

<h1>Data returned:</h1>

<h2>{user.name.first}</h2>

<h2>{user.name.last}</h2>

</div>

) : (

<h1>Data pending...</h1>

);

* Uses Object.keys(user).length > 0 to **detect when data is loaded**.
* Initially shows **"Data pending…"**.
* On successful fetch, shows user’s **first and last name**.

✅ **Testing with Slow Network**

* In **DevTools → Network → Slow 3G**, you can test how the app behaves before data is fetched.
* This demonstrates why the **loading state** is important.

✅ **React Concepts Applied**

* **useState()** for holding fetched data.
* **useEffect()** for running fetch logic on mount.
* **Conditional rendering** to show different UI before/after data load.

✅ **Real-World Application for Little Lemon**

* Replace the randomuser.me URL with **Little Lemon’s customer list API**.
* Fetch user data and select/display **a random winner**.

**Lecture 5:** **APIs**

✅ **Why APIs Matter**

* Without APIs, platforms like **Facebook or Instagram would show nothing** — no photos, likes, or comments.
* APIs are **critical** for **data delivery** to front-end apps.
* Building APIs with **integrity and efficiency** is essential to application success.

✅ **Meta’s API Design Process**

* Starts with an **API design proposal document**:
  + Defines **structure** and **agreements** between system parts.
* **Peer review** of the document:
  + Engineers provide **feedback and comments**.
  + The proposal goes through **multiple iterations**.
* **Code implementation** follows after design finalization:
  + **Code reviews** happen again for quality and correctness.
* Collaboration involves:
  + **Frontend**, **backend**, and **middleware engineers**.
  + Ensures **consistency across the stack**.

✅ **Core API Design Principles**

* **Type Safety**
  + Prevents mismatches (e.g., expecting a photo but getting a video).
  + Increases **app stability** and reduces failures.
* **Communication Clarity**
  + APIs are essentially **contracts between systems**.
  + Teams must **agree** on the expectations and data formats.

✅ **Designing for the Future**

* Aim for a **balance**:
  + Consider **future needs** without **overengineering**.
  + Avoid paralysis by perfection — solve **today's problems first**.

✅ **Advice for Learners**

* APIs can feel **intimidating**, especially if you’re new to back-end.
* But learning them:
  + **Expands your skillset**.
  + Makes you **more employable**.
  + Helps you work across **full-stack**.
* Embrace the complexity — it's **worth it**.

✅ **Final Design Tip**

* **Keep APIs simple**:
  + Avoid **over-complication** and unnecessary patterns.
  + Focus on **readability**, **stability**, and being **bug-free**.

**Topic 3:** **Advanced hooks**

**Lecture 1:** **What is useReducer and how it differs from useState**

✅ **Why useReducer?**

* useState becomes **cumbersome** with:
  + Complex logic
  + Multiple related sub-states
  + State updates that depend on **previous state**

✅ **What is useReducer?**

* It's like a **"supercharged"** version of useState
* Accepts:
  + A **reducer function**
  + An **initial state**
* Returns:
  + **Current state**
  + A **dispatch** function to trigger state changes

✅ **Reducer Function Basics**

* **Takes** two arguments:
  + state (previous state)
  + action (object with at least a type property)
* **Returns** the new state based on action.type

✅ **useReducer vs. useState**

|  |  |  |
| --- | --- | --- |
| Feature | useState | useReducer |
| Simplicity | Great for simple states | Better for complex logic |
| Update method | setState() | dispatch(action) |
| State logic | Inline updates | Centralized in reducer function |
| Use cases | Local UI states | State machines, transactions, complex flows |

✅ **Little Lemon Restaurant Example**

* **Goal**: Track wallet balance for expenses/income
* Actions:
  + "buy\_ingredients": decreases wallet by $10
  + "sell\_meal": increases wallet by $10
  + "celebrity\_visit": increases wallet by $5,000

const reducer = (state, action) => {

switch (action.type) {

case "buy\_ingredients":

return state - 10;

case "sell\_meal":

return state + 10;

case "celebrity\_visit":

return state + 5000;

default:

return state;

}

};

const [wallet, dispatch] = useReducer(reducer, 0);

✅ **Triggering Actions with Buttons**

<button onClick={() => dispatch({ type: "buy\_ingredients" })}>

Shopping for Veggies

</button>

<button onClick={() => dispatch({ type: "sell\_meal" })}>

Serve Meal to Customer

</button>

<button onClick={() => dispatch({ type: "celebrity\_visit" })}>

Celebrity Visit

</button>

✅ **Benefits of useReducer**

* **Centralized logic** in one place (reducer function)
* Easier to **track, test, and debug**
* Scales well with **complex state transitions**
* Makes **state predictable and traceable**

✅ **Final Tips**

* Keep action objects **minimal but meaningful**
* Don’t overengineer — use useReducer only when needed
* It’s a great tool for **financial tracking**, **form management**, and **state machines**

**Code Example:**

import { useReducer } from 'react';

import './App.css';

const reducer = (state, action) => {

  if (action.type === 'buy\_ingredients') {

    return { money: state.money - 10 };

  }

  if (action.type === 'sell\_a\_meal') {

    return { money: state.money + 10 };

  }

  return state;

};

function App() {

  const initialState = { money: 100 };

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

  return (

    <div className="App">

      <h1>Wallet: {state.money}</h1>

      <div>

        <button onClick={() => dispatch({ type: 'buy\_ingredients' })}>

          Shopping for veggies!

        </button>

        <button onClick={() => dispatch({ type: 'sell\_a\_meal' })}>

          Serve a meal to the customer

        </button>

      </div>

    </div>

  );

}

export default App;

**Lecture 2: When to choose useReducer vs useState**

✅ **When to Use useState**

* Ideal for **primitive data types**:
  + string, number, boolean
* Works best with **simple state updates**:
  + Toggling a modal (true/false)
  + Storing a user input (string)
  + Counting clicks (number)
* **Quick and intuitive** to implement
* **Minimal boilerplate**

✅ **When to Use useReducer**

* Suitable for **complex state structures**:
  + Objects or arrays with multiple nested values
* Best when:
  + State **depends on previous state**
  + You need to handle **multiple state transitions**
  + You want a **centralized state management logic**
* Ideal for:
  + Forms with many inputs
  + Apps with financial or game logic
  + Managing lists with CRUD actions

✅ **Comparison Spectrum**

* Think of it as a **spectrum, not a rule**
  + 🡸 Simple | useState ←——→ useReducer | Complex 🡺
* No strict threshold like “3+ properties = useReducer”
* Choose the **simplest** and **most maintainable** solution

✅ **Pros and Cons**

|  |  |  |
| --- | --- | --- |
| Hook | Pros | Cons |
| useState | Simple, fast to implement | Becomes harder to manage as state grows |
| useReducer | Cleaner for complex state, easier to extend | Requires more setup, harder for beginners |

✅ **Practical Decision Tips**

* Start with useState — if state becomes **messy or hard to scale**, refactor to useReducer
* Prefer useReducer when:
  + Many **state transitions** with different types
  + Logic can be **extracted and reused**
  + You want to **test state logic** independently

**Lecture 3:** **useRef to access underlying DOM**

✅ **What is useRef?**

* A React hook to **access and persist mutable values**
* Commonly used to **reference DOM elements** directly
* Returns an object with a **.current** property

✅ **When to Use useRef**

* To **focus input fields**
* To **store timers/intervals**
* To **access DOM nodes** without re-rendering
* To **keep mutable variables** across renders

✅ **Step-by-Step: Focusing an Input Field with useRef**

1. **Import the hook:**

import { useRef } from 'react';

1. **Create the ref object:**

const formInputRef = useRef(null);

1. **Attach the ref to the input element:**

<input ref={formInputRef} type="text" placeholder="Search..." />

1. **Create a click handler to focus the input:**

const focusInput = () => {

formInputRef.current.focus();

};

1. **Trigger the handler with a button:**

<button onClick={focusInput}>Focus Input</button>

✅ **Explanation of How It Works**

|  |  |
| --- | --- |
| Step | Description |
| useRef() | Creates a mutable object: { current: null } |
| ref={formInputRef} | React sets formInputRef.current to the actual input DOM node |
| formInputRef.current.focus() | Calls the .focus() method directly on the input DOM node |

✅ **Why This is Useful**

* Improves **UX** by auto-focusing fields
* Avoids extra click/tap for the user
* Keeps the React render cycle **clean and unaffected**

**Lecture 4:** **Custom hooks**

✅ **What Are Custom Hooks?**

* **Reusable functions** in React that encapsulate logic using built-in hooks (e.g., useState, useEffect)
* Help **avoid code duplication** across components
* Allow **cleaner**, **modular**, and **maintainable** code

✅ **Why Use Custom Hooks?**

* DRY principle: **Don't Repeat Yourself**
* Make logic **portable** and **easy to test**
* Promote **separation of concerns** (UI vs. logic)

✅ **Rules for Custom Hooks**

* Name must start with **use**
* Can use **other hooks** inside
* Should follow the **Rules of Hooks** (e.g., no conditional usage)

✅ **Example: Logging State Changes with useConsoleLog**

📁 useConsoleLog.js

import { useEffect } from "react";

function useConsoleLog(varName) {

useEffect(() => {

console.log(varName);

}, [varName]); // Runs whenever varName changes

}

export default useConsoleLog;

✅ **Using the Custom Hook in a Component**

📁 App.js

import { useState } from "react";

import useConsoleLog from "./useConsoleLog";

function App() {

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

useConsoleLog(count); // Logs every time `count` changes

function increment() {

setCount(prev => prev + 1);

}

return (

<>

<h1>{count}</h1>

<button onClick={increment}>Increment</button>

</>

);

}

✅ **How This Works**

* useConsoleLog uses useEffect to watch a value
* It **logs to the console** every time that value updates
* Keeps your **App component cleaner**, with logic extracted

✅ **Benefits of Custom Hooks**

* Easy to **test logic in isolation**
* Clean and **maintainable code**
* **Shared logic** across components/projects

✅ **Conclusion**

* Custom hooks let you **bundle and reuse** React logic
* They follow the same **hook lifecycle rules**
* Useful for handling things like:
  + **Form validation**
  + **Fetching data**
  + **Timers or intervals**
  + **Local storage access**
  + **Event listeners**