Design a Dynamic Frontend with React

Building Scalable React Applications with Hooks

Engage and Think

An online sports news platform must update live match data every 30 minutes. During peak hours, delays and performance issues affect user experience. You are responsible for ensuring real-time updates, managing API calls efficiently, and keeping the data synchronized while maintaining smooth app performance.

How would you design a solution that ensures match data is updated regularly without slowing down the site or overwhelming the server?

Learning Objectives

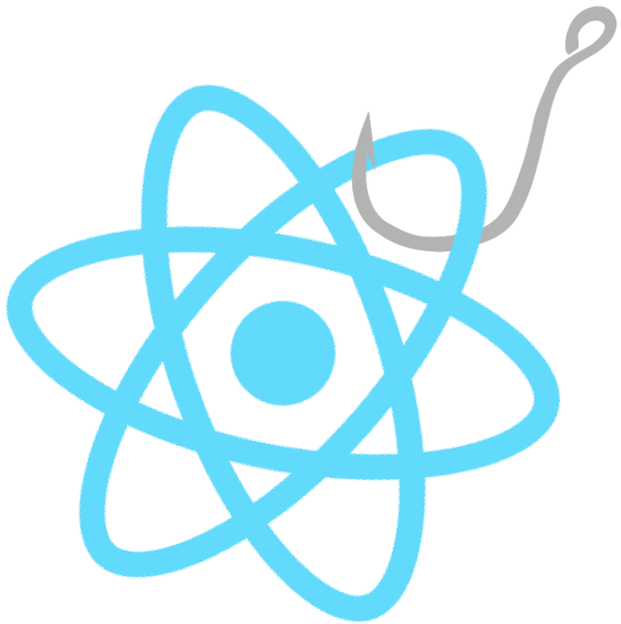
By the end of this lesson, you will be able to:

- Compare the differences between useState and useReducer
- Explain how useEffect is used for handling side effects in React components
- Work with useContext Hook to simplify state management in React applications
- Use useReducer for managing complex state transitions more efficiently
- Create custom Hooks in React to enhance modularity, reusability, testing, and overall scalability

Understanding Hooks in React

What Is a Hook in React?

Hooks are functions that let you use React features, such as state and lifecycle methods, inside functional components. Previously, these features were only available in class components.



Hooks allow you to write cleaner, reusable, and more readable code using function components.

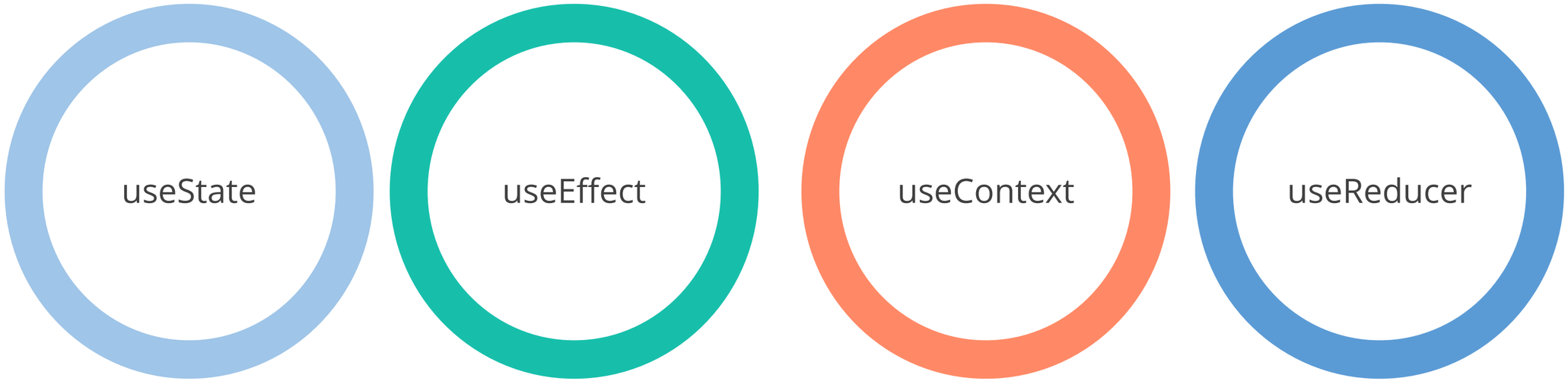
Benefits of React Hooks

They were designed to:

- Reduce dependency on class components by enabling functional components to manage state and lifecycle
- Simplify logic handling in components to enhance code readability and maintainability
- Improve code reuse and modularity by creating reusable custom hooks

Types of Hooks

The following are some commonly used Hooks in React:



Each of these Hooks enables specific capabilities, ranging from managing state and side effects to sharing global data and handling complex logic. This makes functional components more powerful and efficient.

Exploring useState Hook

What Is useState Hook?

The useState Hook is a built-in function in React that allows you to add state to functional components.

Syntax:

```
const [state, setState] = useState(initialValue);
```

state: The current state value

setState: A function used to update that value

initialValue: The starting value for the state

It enables a component to track and update values over time, such as user input, toggles, counters, or any data that changes during the lifecycle of a component.

useState Hook: Example

This example demonstrates how the useState Hook updates state in a functional component.

Example:

```
import { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);  // count
starts at 0

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

This code uses the useState Hook to track how many times a button is clicked and updates the displayed count each time the button is pressed.

Benefits of useState Hook

The useState Hook provides several advantages that enhance the functionality, readability, and maintainability of React components, such as:

- Enabling state management in functional components
- Simplifying component logic and structure
- Providing a clear and consistent way to update state
- Supporting multiple state variables within a single component
- Triggering automatic re-rendering when state changes
- Improving modularity and reusability of stateful logic

Introduction to the useEffect Hook

What Is useEffect Hook?

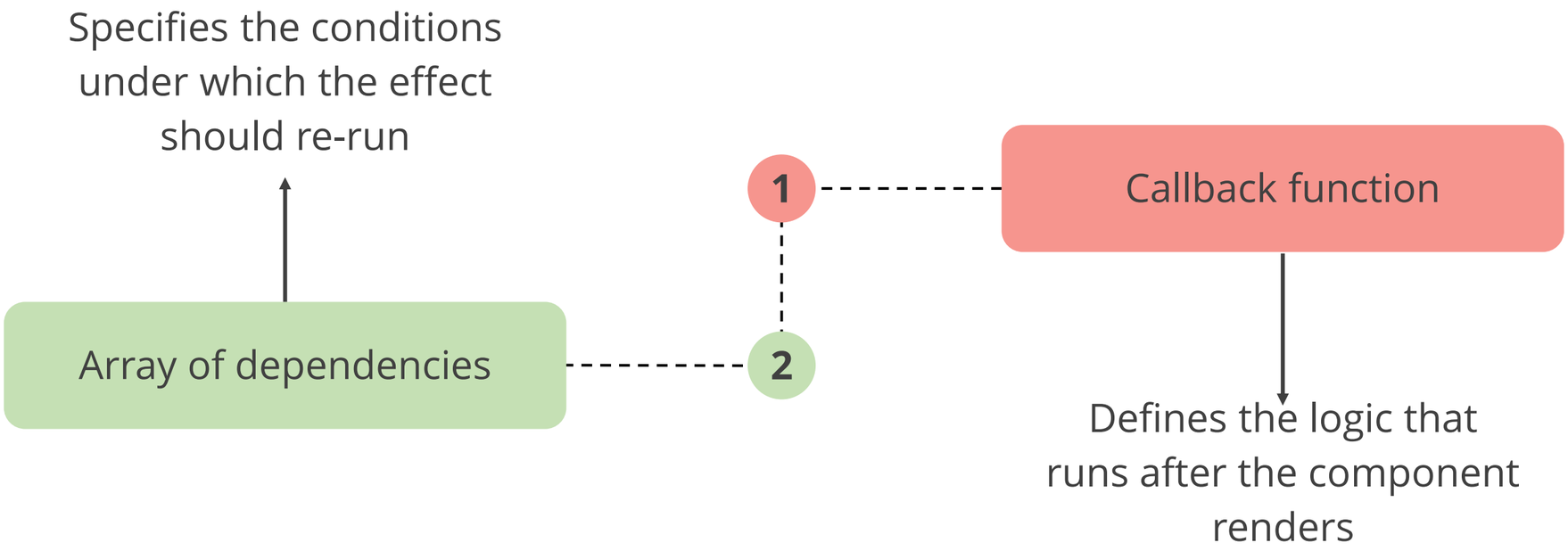
The useEffect Hook is a tool for managing side effects in functional components. A side effect is any operation that affects the application's state or interacts with the outside world.

useEffects can be used to:

- Manage subscriptions
- Define side effects in functional components
- Run logic after the component renderes
- Re-run logic when dependencies change

Key Parameters of the useEffect Hook

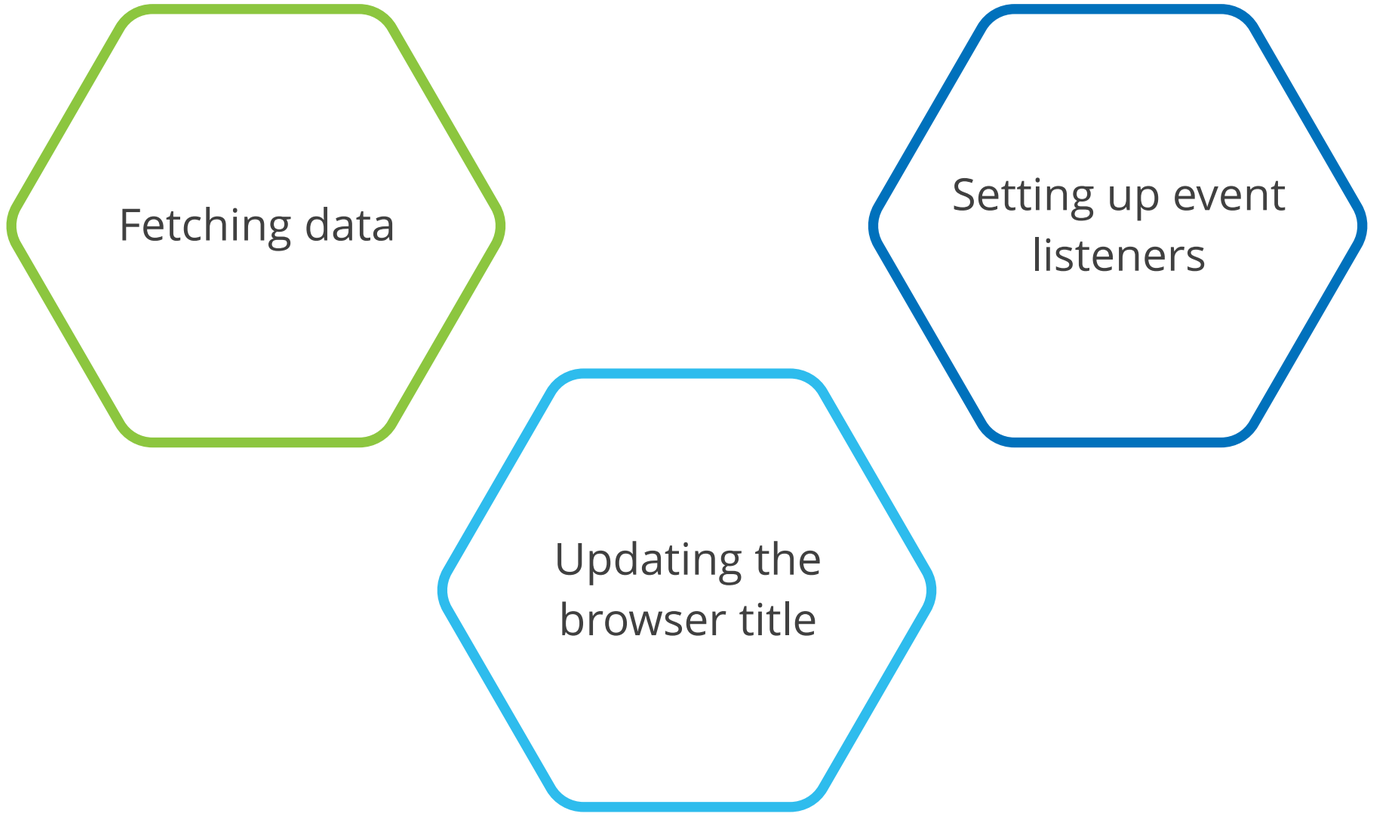
The useEffect Hook relies on two key arguments to manage side effects and control re-execution as shown below:



useEffect manages the effect’s logic and lists dependencies that control when it runs again.

Essential Tasks Handled by the useEffect Hook

The **useEffect Hook** performs various tasks, such as:



Understanding useEffect Execution Flow

The useEffect Hook runs after the component is rendered and helps manage side effects caused by updates. It works as follows:

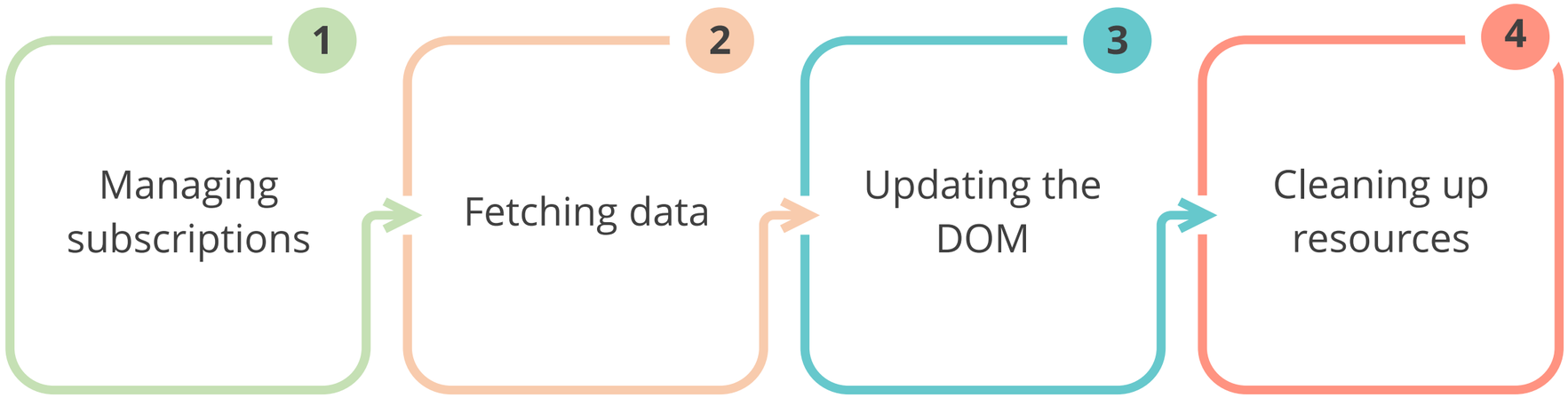
The useEffect Hook in React allows you to perform side effects such as data fetching, logging, or DOM updates.

An array of dependencies is passed as the second argument to control when the effect should be re-run.

The callback function runs after the component is rendered and can optionally return a cleanup function.

Side Effects Managed by useEffect

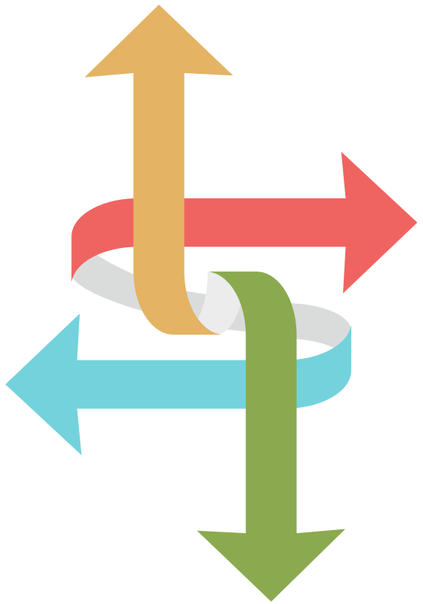
The useEffect Hook executes after rendering and enables critical side effects for dynamic behavior and resource control, such as:



Fetching Data with useEffect

The useEffect Hook enables data fetching from an API after the component renders, making it an effective strategy for dynamic applications because it:

Ensures asynchronous data retrieval



Maintains a smooth user experience without blocking UI updates

Supports cleanup of outdated requests or responses

Gives control over when and how requests are triggered

Fetching Data with useEffect: Example

This example demonstrates how to fetch data from an API using useEffect after a component mounts.

Example:

```
import React, { useState, useEffect } from 'react’;

function App() {

  const [data, setData] = useState(null);

  useEffect(() => {

    fetch('https://api.example.com/data')

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

      .then(data => setData(data))

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

}, []);
```

Fetching Data with useEffect: Example

Example:

```
return (
    <div>
      {data ? (
        <ul>
          {data.map(item => (
            <li
key={item.id}>{item.name}</li>
          ))}
        </ul>
      ) : (
        <p>Loading...</p>
      )}
    </div>
  );
}
```

Display a list of data items when data is available; otherwise, show a “Loading...” message

Quick Check

You are building a weather app that fetches the latest forecast from an external API after the component renders. You notice that switching between cities quickly causes multiple overlapping API requests, sometimes showing outdated data. Which feature of useEffect is most helpful in solving this issue?

A. It restarts the entire component tree on each request.

B. It supports cleanup of outdated requests to prevent memory leaks and stale data.

C. It blocks UI updates while fetching new data.

D. It delays rendering until the fetch operation is complete.

Assisted Practice

Implementing a React Application Using useEffect Hook **Duration: 20 Min.**

Problem statement:

You are required to build a React application that dynamically fetches and displays user data when the component is mounted. The goal is to use the useEffect Hook to trigger the API call and update the component state efficiently.

Outcome:
By the end of this demo, you will be able to implement the useEffect Hook to manage side effects in a React application and ensure that data is fetched and rendered when the component loads.

Note: Refer to the demo document for detailed steps:
01_Implementing_a_React_Application_Using_the_useEffect_Hook

Assisted Practice: Guidelines

Steps to be followed:

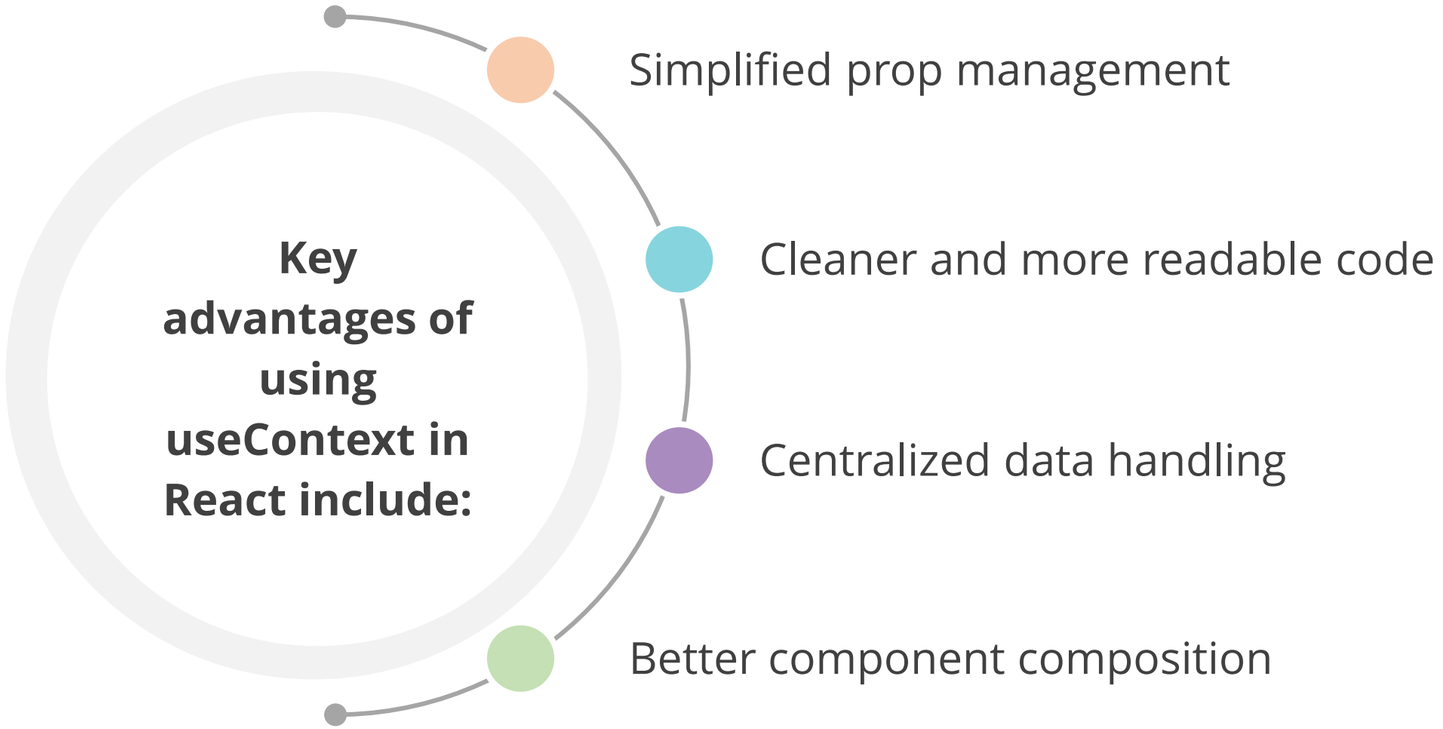
1. Set up a new React project using Vite
2. Modify the App.jsx file
3. Run and verify the application

ASSISTED PRACTICE

Working with the useContext Hook in React

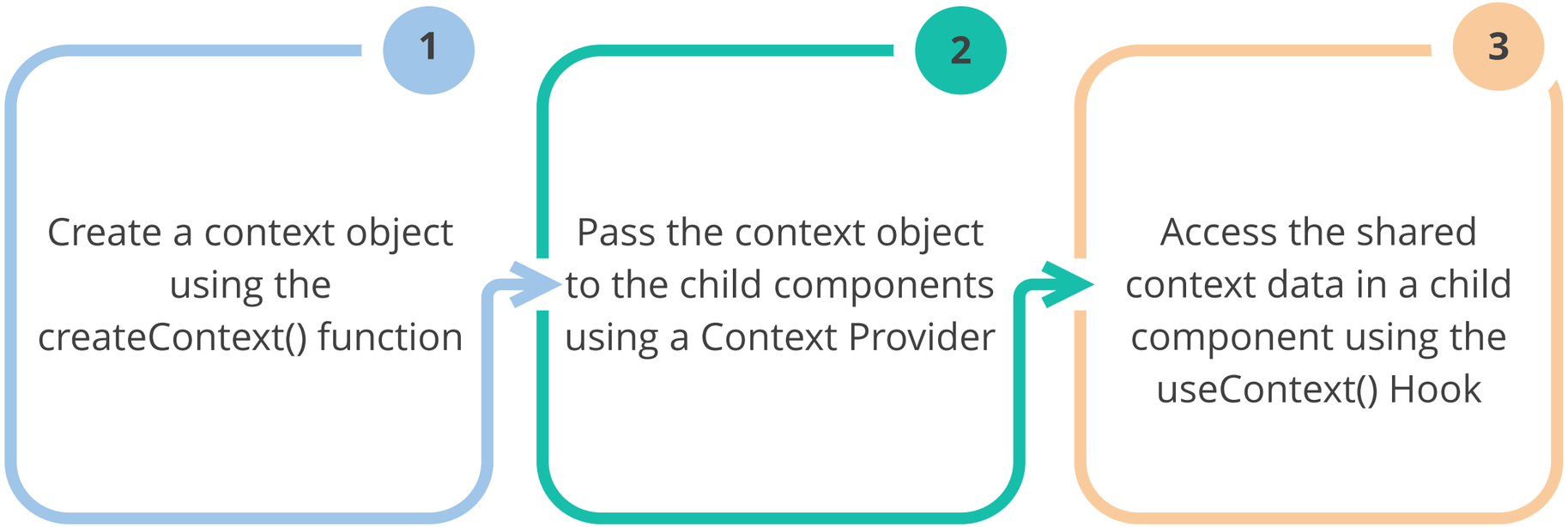
What Is useContext Hook?

The useContext Hook allows components to access shared state or data directly, avoiding the need to pass props through multiple layers of the component tree.



Steps to Use the useContext Hook

The useContext Hook allows components to access shared data without prop drilling. The steps to using:



Quick Check

You want to avoid prop drilling in a React application when sharing data like user preferences across deeply nested components. Which step is essential for making the shared data available in child components?

A. Use useContext() to consume the shared data inside a child component

B. Use useEffect() to distribute the data to nested componentsuseState because it updates state directly

C. Pass props manually from parent to each nested child

D. Use useState() and useReducer() together to manage and share global state

Assisted Practice

Creating a React Application Using useContext Hook **Duration: 10 Min.**

Problem statement:

You need to implement a theme toggle feature in a React application. The task is to use the useContext Hook to manage and share the theme state across components, allowing users to switch between light and dark modes.

Outcome:

By the end of this demo, you will be able to use the useContext Hook in a React app to provide and consume shared state and implement dynamic styling based on user interaction.

Note: Refer to the demo document for detailed steps:
02_Creating_a_React_Application_Using_the_useContext_Hook

Assisted Practice: Guidelines

Steps to be followed:

1. Set up a new React project using Vite
2. Create react components
3. Import and use the components
4. Run and verify the application

Overview of useReducer Hook

What Is useReducer Hook?

The useReducer Hook is a React Hook used for managing more complex state logic in functional components. It is an alternative to useState and is especially useful when:

- The state logic involves multiple sub-values.
- The next state depends on the previous state.
- You want to centralize state updates in a single function—a reducer—similar to Redux.

useReducer Hook: Syntax

The syntax of the useReducer Hook in React is given below:

Syntax:

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

state	Checks the current value of the state
dispatch	Sends actions to the reducer
reducer	Updates the state
initialState	Sets the initial value of the state

Advantages of useReducer Hook

The useReducer Hook offers several advantages that make it suitable for managing complex application states. The key advantages include:

- Simplification of complex state logic
- Ease of testing for state management and transitions
- Compatibility with the Context API for global state management

Arguments in useReducer Hook

useReducer takes two arguments:

A reducer function

An initial state

The dispatch function sends an action to the reducer, which calculates and returns a new state.

The useReducer arguments and dispatched actions help define the initial state, specify what should happen, and update the state accordingly.

useReducer: Example

This example demonstrates how to implement a simple counter using React's useReducer Hook for state management.

Example:

```
import React, { useReducer } from 'react';
const initialState = { count: 0 };
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
 }
```

useReducer: Example

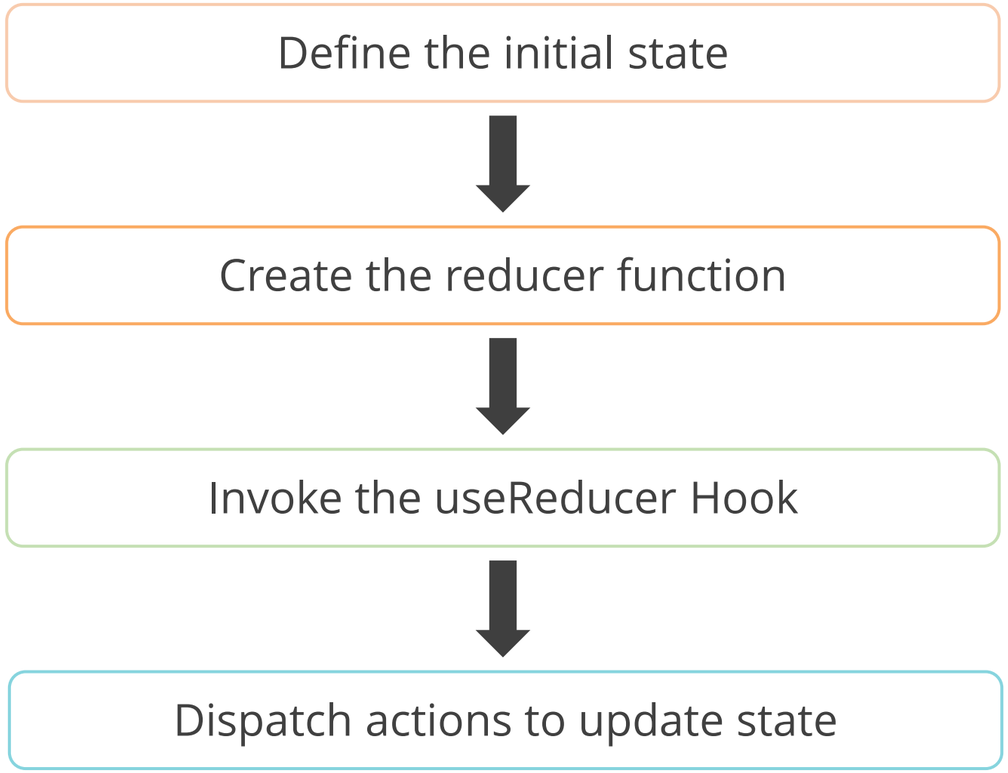
This example demonstrates how to use the useReducer Hook to manage state in a simple counter application, enabling increment and decrement actions using dispatched events.

Example:

```
}
function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    </div>
  );
}
```

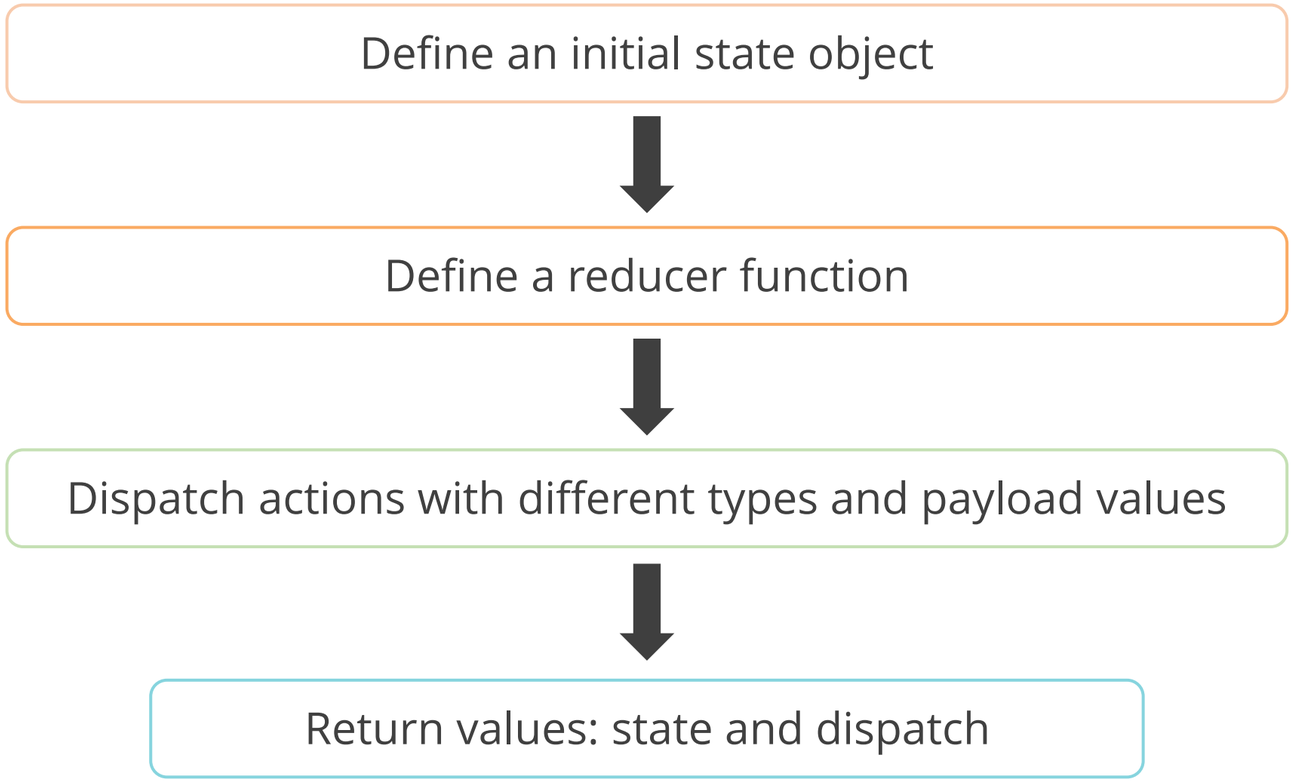
Step-by-Step Set Up of useReducer

The useReducer Hook manages state using a reducer pattern. Follow these key steps to set it up:



Setting Up useReducer: A Step-by-Step Guide

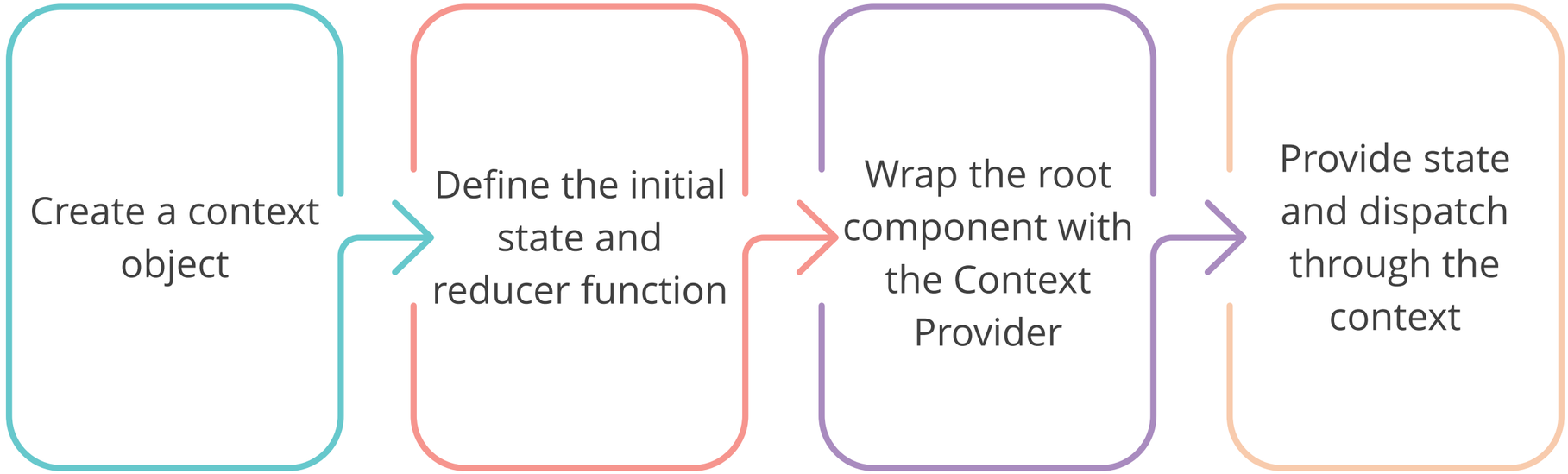
Here is the flow for updating state with useReducer:



The **useReducer** Hook returns the current state object and the dispatch function.

Combining useReducer with useContext

The useReducer Hook is often combined with the useContext Hook to manage and share state efficiently across components. Follow these steps to implement them together:



Combining the useReducer and useContext Hooks enables efficient state sharing across components in an application.

Assisted Practice

Creating a React App with the useReducer Hook **Duration: 10 Min.**

Problem statement:

You are assigned a task to build a React application using Vite and implement the useReducer Hook to manage and update a simple counter state.

Outcome:

By the end of this demo, you will be able to create a functional counter app using React's useReducer Hook to handle state transitions for incrementing and decrementing values.

Note: Refer to the demo document for detailed steps:
03_Creating_a_React_App_with_the_useReducer_Hook

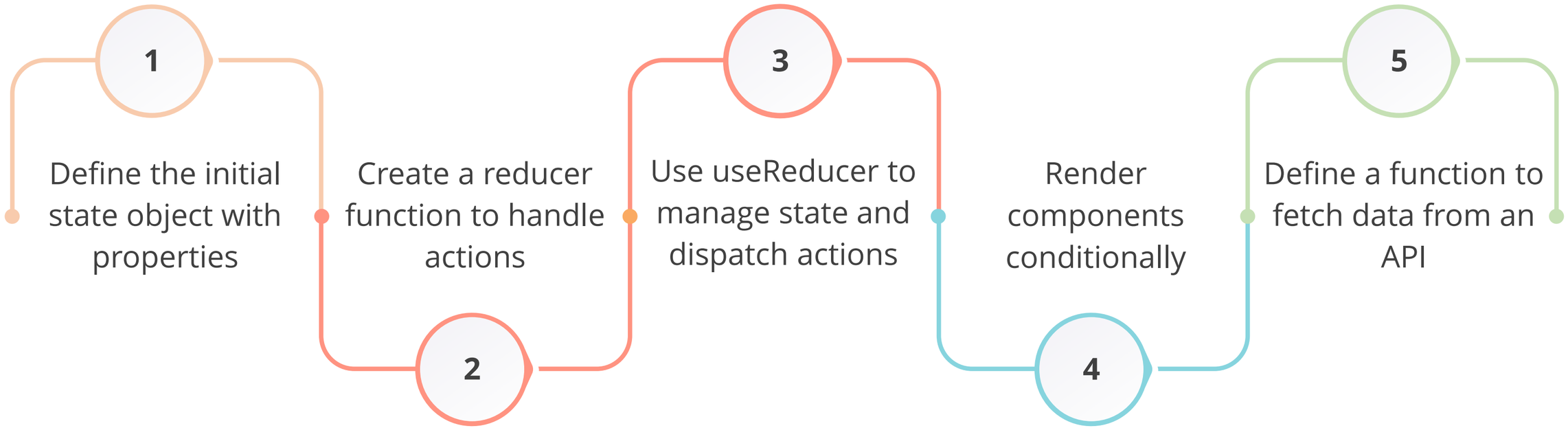
Assisted Practice: Guidelines

Steps to be followed:

1. Set up a new React project using Vite
2. Modify the App.jsx file
3. Run and verify the application

How to Fetch Data Using useReducer?

Fetching data with useReducer provides a structured and centralized approach to managing state updates. Follow these key steps to implement it effectively:



useState vs. useReducer

The useState and useReducer Hooks in React are used for managing component state in different ways. The table below highlights their key differences:

Aspect	useState	useReducer
Complexity	Handling simple state updates	Handling complex state logic and transitions
State Update	Direct state updating	State updating via action dispatch and reducer
State Shape	Managing simple state structures	Managing complex state structures
Performance	Higher efficiency for simple updates	Lower efficiency for simple updates
Code Organization	Risk of code duplication	Support for maintainable and organized code

Quick Check

You are building a to-do list application. Each task has multiple properties like ID, title, completed, and priority. You also need to handle multiple actions such as adding, deleting, toggling completion, and updating priority. The state will grow more complex as features are added. Which state management Hook is more suitable for this scenario?

A. useState because it is simpler and more efficient

B. useReducer because it handles multiple related state properties and actions

C. useState because it updates state directly

D. useReducer because it avoids re-rendering the entire component

Assisted Practice

Fetching Data Using the useReducer Hook **Duration: 15 Min.**

Problem statement:

You are tasked with building a React application that fetches and displays user data from an external API. You must implement the useReducer Hook to manage loading, success, and error states during the fetch process.

Outcome:

By the end of this demo, you will be able to fetch data in a React app and use the useReducer Hook to handle asynchronous state transitions effectively.

Note: Refer to the demo document for detailed steps:
04_Fetching_Data_Using_the_useReducer_Hook

Assisted Practice: Guidelines

Steps to be followed:

1. Set up a new React project using Vite
2. Modify the App.jsx file
3. Run and verify the application

Custom Hooks and Its State Management

What Are Custom Hooks in React?

Custom Hooks are reusable JavaScript functions in React that start with the word use and allow you to extract and share logic between components.

Here are the following functions of custom Hooks:

- Allows developers to extract and reuse logic across components
- Helps make code more modular, maintainable, and easier to test

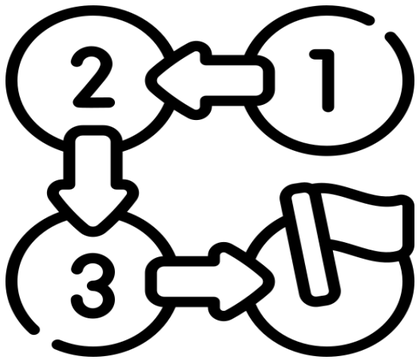
Best Practices for Creating Custom Hooks in React

When creating custom Hooks, ensure the following best practices are followed:

- Build a function that uses one or more built-in Hooks
- Follow the same rules for custom Hooks as for built-in Hooks
- Keep the custom Hooks small and focused on a single functionality
- Use consistent naming to ensure React recognizes it as a Hook

Steps to Build a Custom Hook in React

Follow these steps to build your own reusable custom Hook in React:



01 Create a file with a name starting with use

02 Import built-in Hooks

03 Write a function that returns state or values

04 Use built-in Hooks inside the function

05 Add logic like conditions or effects

06 Export the function for reuse

Best Practices for Naming Custom Hooks

Follow these best practices to create meaningful and consistent names for custom Hooks:

1. Use the *use* prefix followed by a descriptive name
2. Write names in camelCase and start with a verb
3. Keep names specific, short, and purpose-driven
4. Avoid generic, acronym-based, or built-in Hook like names
5. Use singular nouns and add a suffix if it improves clarity

Quick Check

You are creating a custom Hook that fetches user profile data. You are considering naming it useData to keep it simple. Which of the following best aligns with React best practices for naming custom Hooks?

A. useData – It is short and works for any kind of data.

B. useFetch – It sounds like a built-in Hook and might cause confusion.

C. useUserProfile – It is specific, descriptive, and clearly conveys the Hook's purpose.

D. fetchUser – It skips the use prefix, which is optional for Hooks.

Assisted Practice

Creating a React Application Using the Custom Hook **Duration: 10 Min.**

Problem statement:

You need to develop a reusable counter functionality in a React application. The task is to build and use a custom Hook that encapsulates state management logic for incrementing and decrementing a counter value.

Outcome:

By the end of this demo, you will be able to define and implement a custom Hook in React to encapsulate logic and promote reusability across components.

Note: Refer to the demo document for detailed steps:
05_Creating_a_React_Application_Using_the_Custom_Hook

Assisted Practice: Guidelines

Steps to be followed:

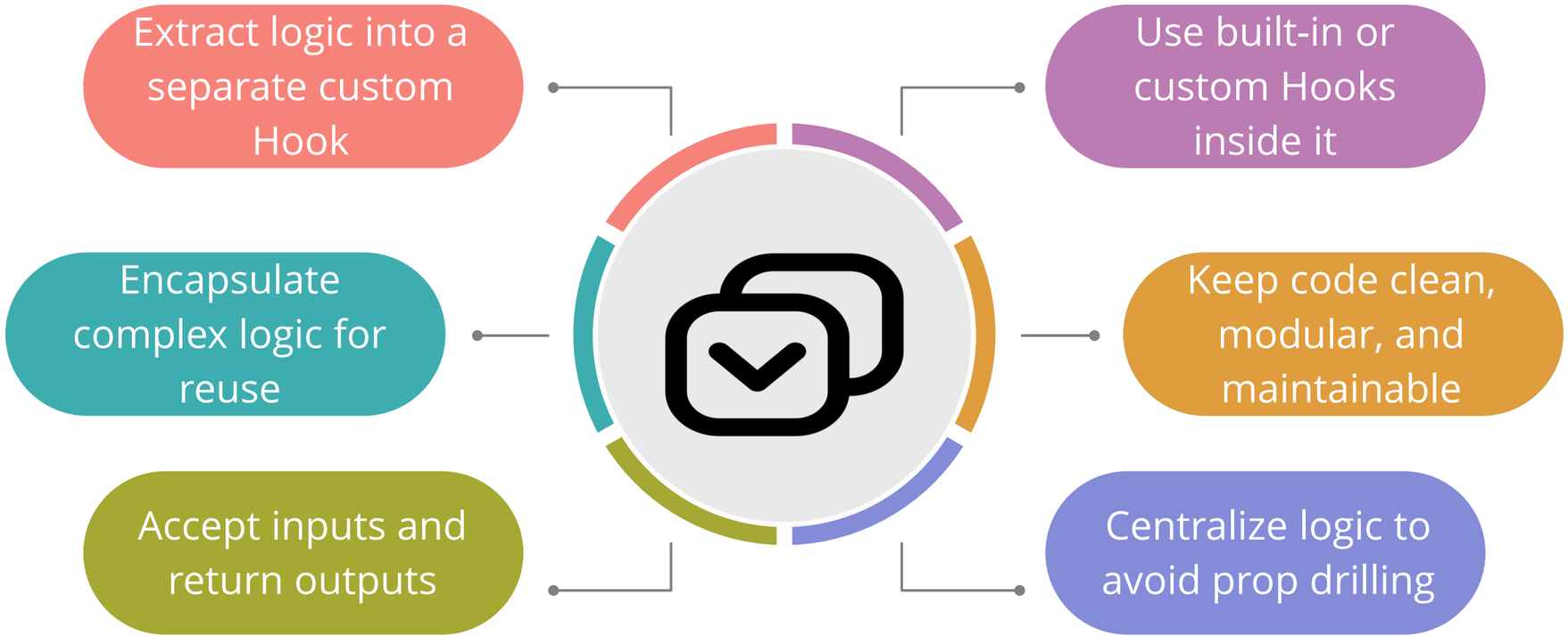
1. Set up a new React project using Vite
2. Create a custom Hook
3. Import and use the custom Hook
4. Run and verify the application

ASSISTED PRACTICE

Benefits of Using Custom Hooks in React

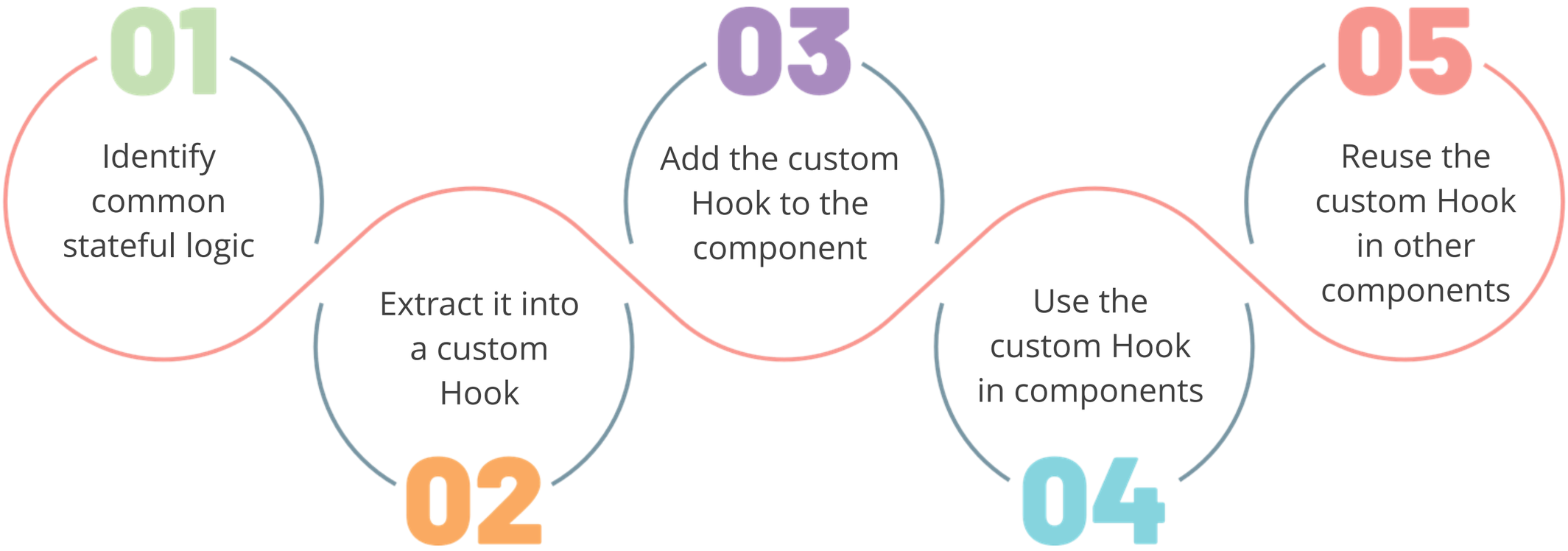
Custom Hooks enable the sharing of stateful logic between components, which is a common and efficient pattern in React.

Custom Hooks allow developers to:



Steps to Create Custom Hooks for Sharing Stateful Logic

Steps to create a custom Hook are as follows:



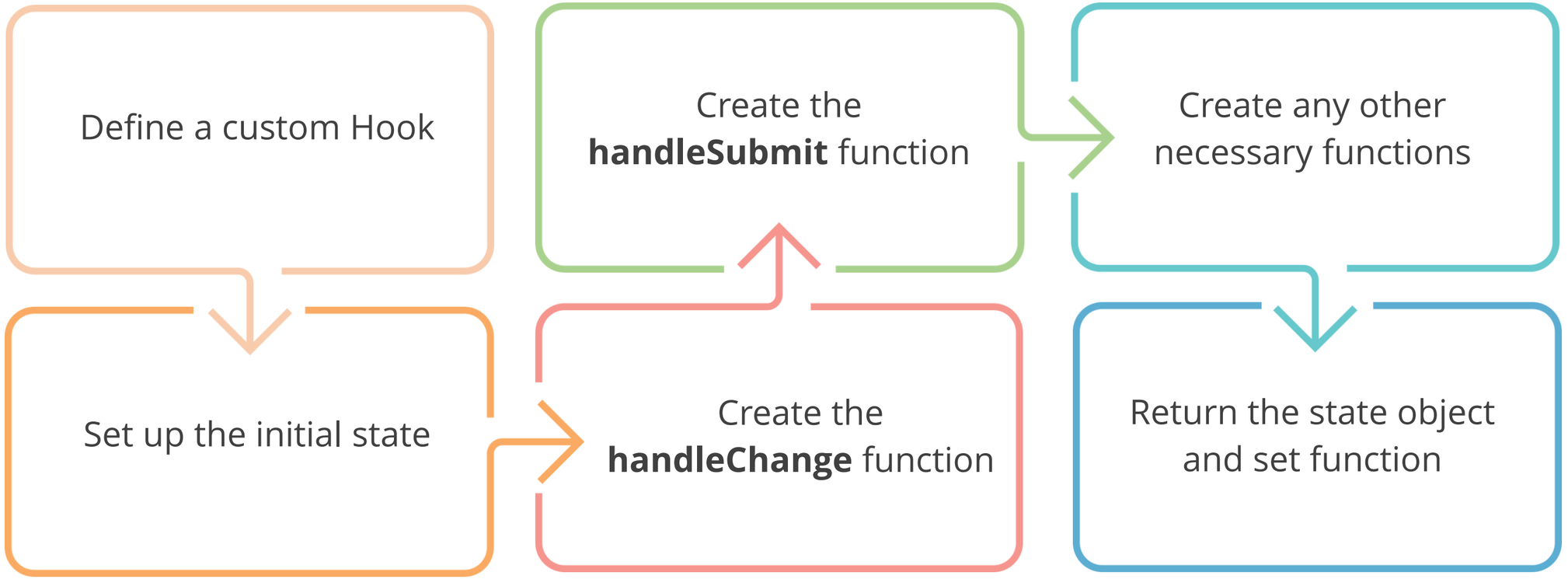
Custom Hooks in Form Handling

Custom Hooks can manage the form state in React using built-in Hooks like useState or useReducer. Here are some advantages of using a custom hook for this purpose:

- Improves code reuse across multiple forms
- Handles common form tasks
- Simplifies and organizes form logic

Steps to Create Custom Hooks for Managing Form State

A custom Hook for managing the form state can be developed with the help of the following steps:



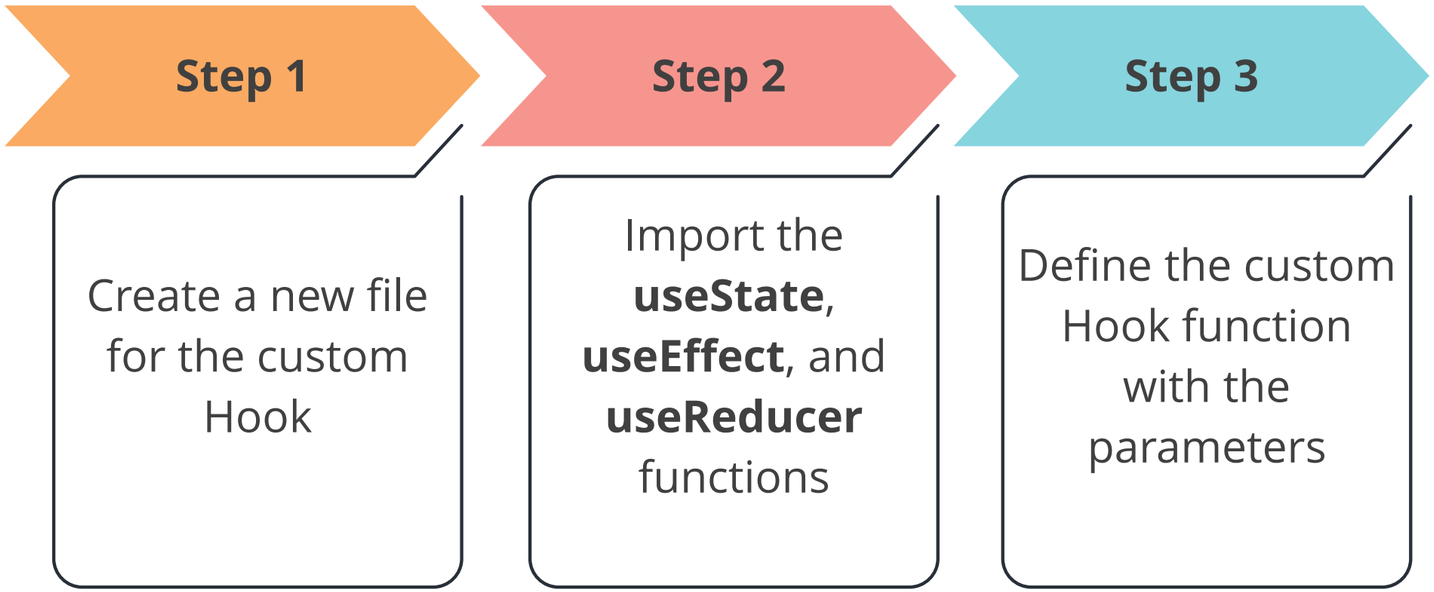
Custom Hooks for Fetching Data

Custom Hooks can be used to fetch data from APIs, making data management efficient and reusable across components.

- Handles network requests and improves code reusability
- Manages complex data-fetching logic
- Supports loading and error state handling

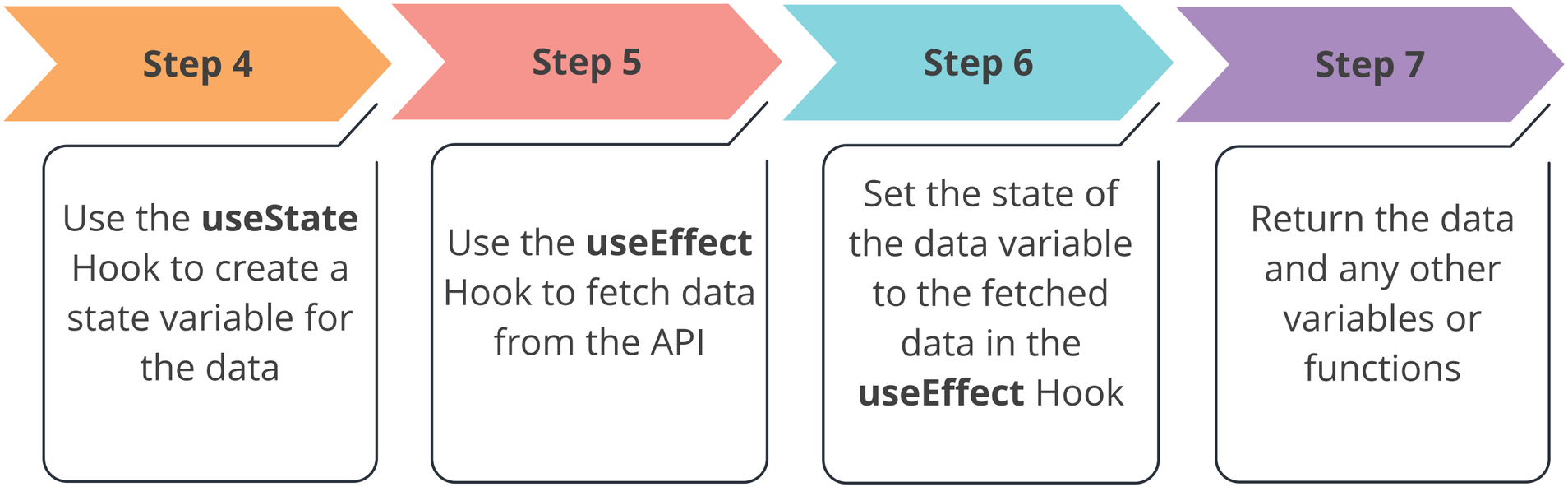
Custom Hooks for Fetching Data

Steps to create a custom Hook for fetching data from APIs:



Custom Hooks for Fetching Data

The remaining steps to create a custom Hook for fetching data from APIs are:



Quick Check

You are building a blog application where multiple components need to fetch posts from the same API endpoint. To simplify your code and avoid repeating fetch logic, you decide to use a custom Hook. Which of the following actions is essential when creating this custom Hook?

A. Use useEffect to declare global variables outside the Hook

B. Fetch the API data directly inside the component's return() block

C. Use useReducer to create HTML elements inside the Hook

D. Create a function that accepts a URL and returns data, loading, and error using useState and useEffect

Assisted Practice

Fetching Data Using a Custom Hook **Duration: 15 Min.**

Problem statement:

You need to build a React application that retrieves and displays data from an API. The objective is to encapsulate the data fetching logic inside a custom Hook for better reusability and separation of concerns.

Outcome:

By the end of this demo, you will be able to implement a custom Hook to fetch data from an API and use it within a React component to render dynamic content efficiently.

Note: Refer to the demo document for detailed steps:
06_Fetching_Data_Using_Custom_Hook

Assisted Practice: Guidelines

Steps to be followed:

1. Set up a new React project using Vite
2. Create a custom Hook
3. Import and use the custom Hook
4. Run and verify the application

ASSISTED PRACTICE

Creating a React App with All Hooks

Duration: 25 Mins.

Project agenda: To develop a React application using Vite that demonstrates the use of all major React hooks, including state, effect, reducer, context, and custom hooks. The project will focus on modular architecture, global state management, asynchronous data flows, and real-time updates, ensuring reusable logic, maintainability, and an interactive user experience.

Description: You are tasked with building a React application that integrates all major React hooks to showcase their practical use in a modular and scalable project. This includes managing state with useState, side effects with useEffect, complex logic with useReducer, shared state using useContext, and reusable logic through custom hooks. The application will also feature real-time data updates, asynchronous data fetching, and a global task manager to demonstrate advanced state and data handling techniques.

LESSON-END PROJECT

Creating a React App with All Hooks

Duration: 25 Min.

Perform the following:

1. Set up a new React project using Vite
2. Structure the application
3. Implement core hooks and custom hooks
4. Build reusable UI components
5. Set up App component
6. Update main.jsx
7. Run and verify the application

Expected deliverables: A fully functional React application built with Vite, demonstrating the use of all major React hooks. The application will include modular components, real-time number generation, a global task manager using context and reducer, a reusable counter hook, and asynchronous data fetching. It will showcase proper project structure, maintainable code, and interactive UI elements, serving as a comprehensive reference for applying React hooks in real-world scenarios.

LESSON-END PROJECT

Key Takeaways

- The useState Hook helps manage simple, local component state directly and is ideal for basic data handling like form inputs and toggles.
- The useEffect Hook enables the execution of side effects such as fetching data, subscribing to events, and updating the DOM in functional components.
- The useContext Hook allows data to be passed through the component tree without the need to manually pass props at every level, simplifying state sharing.
- The useReducer Hook helps manage complex states and transitions in a more organized and scalable way, especially when multiple related state values are involved.

Thank You