ReactJS

TECHNOLOGY

ReactHook, useReducer, useEffect, and Custom Hooks Management

simplilearn

Learning Objectives

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

- Work with useContext Hook
- Compare the differences between useState and useReducer
- Demonstrate how to create a custom Hooks in React
- Create custom Hooks for managing form state

A Day in the Life of a ReactJS Developer

An online sports news channel published match details every half an hour. However, during peak hours, the site failed to operate correctly.

Sarah, a budding React developer, received a call from the manager of this news channel asking her to manage the backend programming so that the customers could get what they were paying for.

To achieve this, Sarah needs to manage data fetching activities, provide details for subscriptions, update data, set up timers, and update prices and quantities of the products.

Sarah can complete the tasks and deliver a solution for the given scenario by learning the key concepts of React Hooks and their relationships with one another.

useReducer Hook

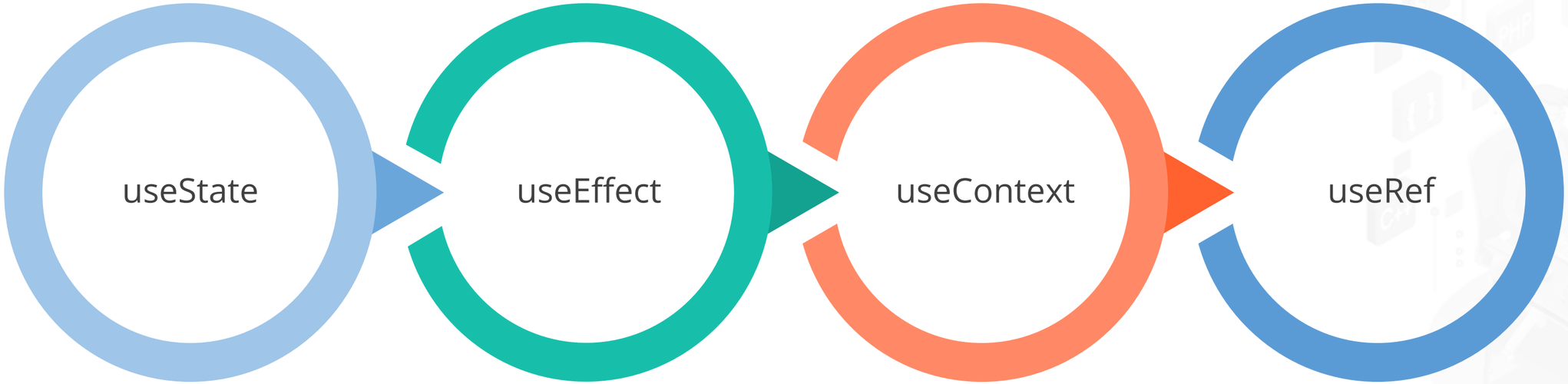
Overview of Hooks: A Recap

Hooks soon replaced class components. These JavaScript functions revolutionized the method of writing React components by:

- Allowing users to use features of stateful components
- Permitting the use of React state and lifecycle features
- Arranging the logic of components into units

Types of Hooks: A Recap

Here are some popular Hooks used in React:



useReducer Hook: An Overview

A **useReducer** Hook is a powerful tool for managing complex states and actions.

It is an alternative to using useState, where the state logic becomes more advanced.

It is a built-in Hook that provides a way to manage complex states and actions.

It is the best solution to handle such complicated and dynamic conditions.

useReducer Hook: Syntax

The syntax of the **useReducer** Hook in React is as follows:

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

state	This is the current state value.
dispatch	This is a function that is used to dispatch actions.
reducer	This is a function responsible for updating the state.
initialState	This is the initial value of the state.

useReducer Hook: Benefits and Limitations

Although a **useReducer** Hook is flexible in managing a complex state, coders often find it challenging to use. Here are some of the benefits and limitations of using useReducer.

Advantages of useReducer:

- Helps complex state logic easily
- Facilitates testing of state management and state transitions
- Works well with the **Context API**

Disadvantages of useReducer:

- Hard to maintain **useReducer** code due to boilerplate code and intricate logic
- Difficult to debug state updates

useReducer Hook: Arguments

useReducer takes two arguments:

A reducer function

An initial state

The dispatch function is used to send an action to the reducer function, which returns a new state.

The arguments in useReducer and dispatch help define the initial state, specify the action type, and pass any necessary data to update the state correctly.

Coding with useReducer: An Example

This code demonstrates a simple counter implemented using React's **useReducer** Hook.

```
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();
  }
```

Coding with useReducer: An 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>
  );
}
```

useReducer Hook: Simple State and Action

The **useReducer** Hook manages the state using a simple state and action approach with the help of the following steps:

Define the initial state of the component

↓

Define a reducer function

↓

Use the useReducer Hook to manage state

↓

Update the state based on the action

useReducer Hook: Simple State and Action

Here are the steps where the key properties of the state are managed with **useReducer**:

Define an initial state object

↓

Define a reducer function

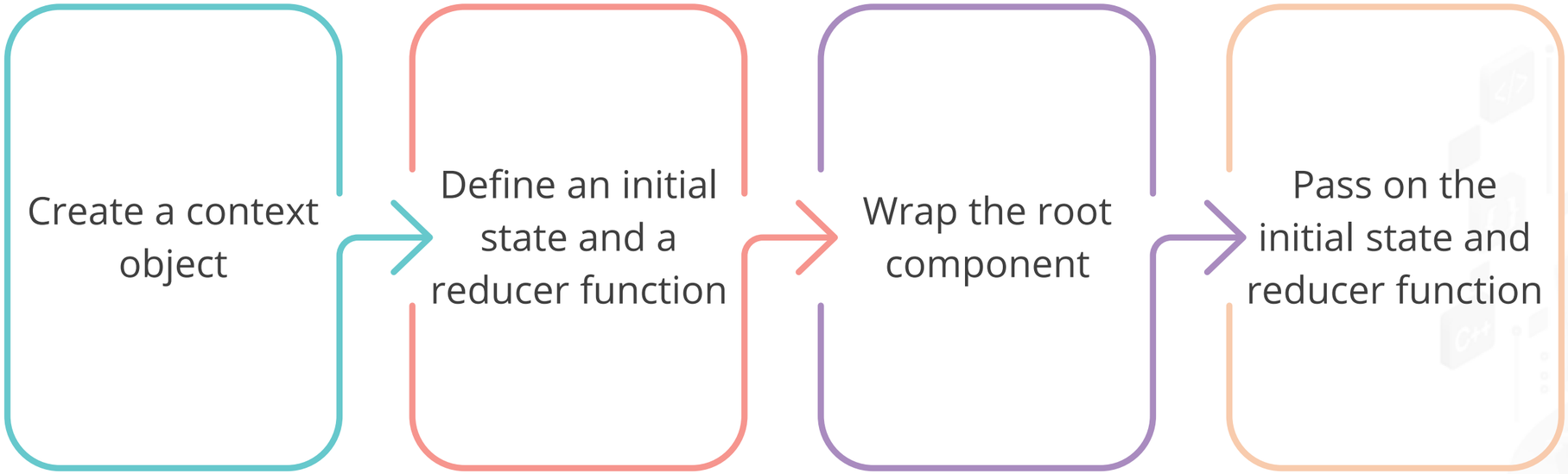
↓

Dispatch actions with different types and payload values

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

useReducer with useContext

The **useReducer** Hook is often used together with the useContext Hook to improve state management.



Combining the **useReducer** and **useContext** Hook allows for easy access and sharing of state across different components in an application.

Demo with useReducer Hook

Duration: 10 Min.

Problem Statement:

You are given a task to build a counter app employing the **useReducer** Hook.

ASSISTED PRACTICE

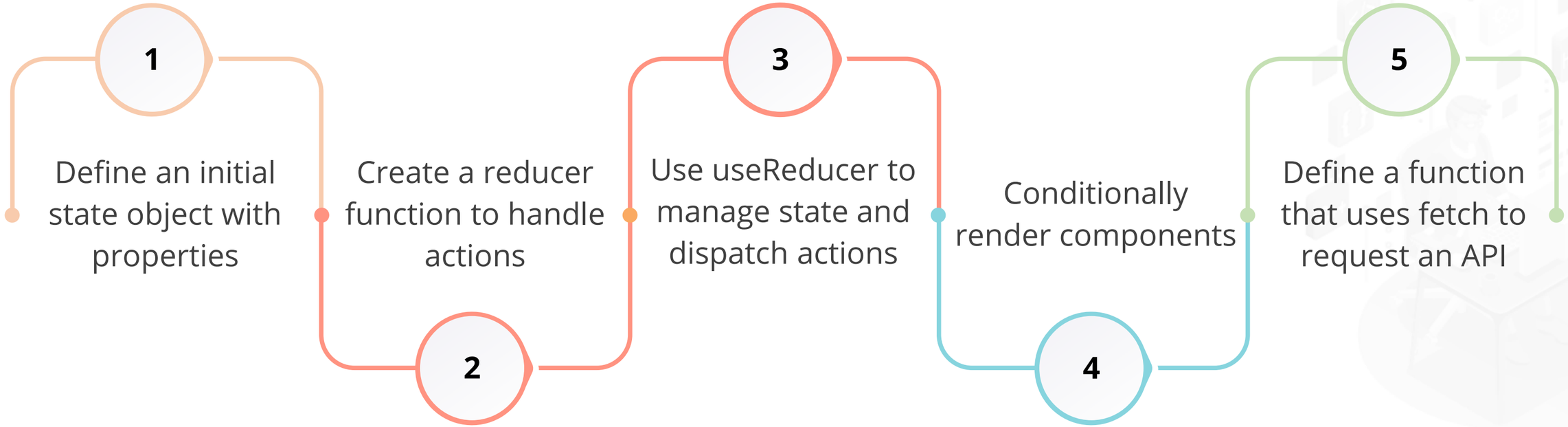
Assisted Practice: Guidelines

The steps to be followed are:

1. Create a new React app using **create-react-app**
2. Run **cd-use-reducer-demo** and change to the new directory
3. Launch the project in the preferred code editor
4. Create a new reducer function that returns a new state
5. Create a simple counter that can be incremented or decremented for this example
6. Create a new state object
7. Run **npm start** in the terminal
8. Open **http://localhost:3000** in the browser

Fetching Data with useReducer

Fetching data with **useReducer** provides a structured and centralized approach for managing state updates with the help of the following processes:



useState vs. useReducer

useState and **useReducer** are both Hooks in React that serve different purposes for managing state.

	useState	useReducer
Complexity	For simple state updates	For complex state and its transitions
State Update	Updates the state directly	Updates state via action dispatch and reducer function
State Shape	Manages simple states	Handles complex state with many properties
Performance	Faster for simple state updates	Less performance for simple updates
Code Organization	May cause code duplication	Aids maintainable and organized codes

Demo with Fetching Data Using the useReducer Hook

Duration: 15 Min.

Problem Statement:

You are given a task to develop a React application that demonstrates fetching data using the **useReducer** Hook.

ASSISTED PRACTICE

Assisted Practice: Guidelines

The steps to be followed are:

1. Create a new React app using **create-react-app**
2. Change in the newly created directory
3. Open the project in a code editor
4. Import **useReducer** and **useEffect** from React
5. Run the app by running **npm start** in the terminal
6. Open **http://localhost:3000** in the browser

useEffect Hook

useEffect Hook: Introduction

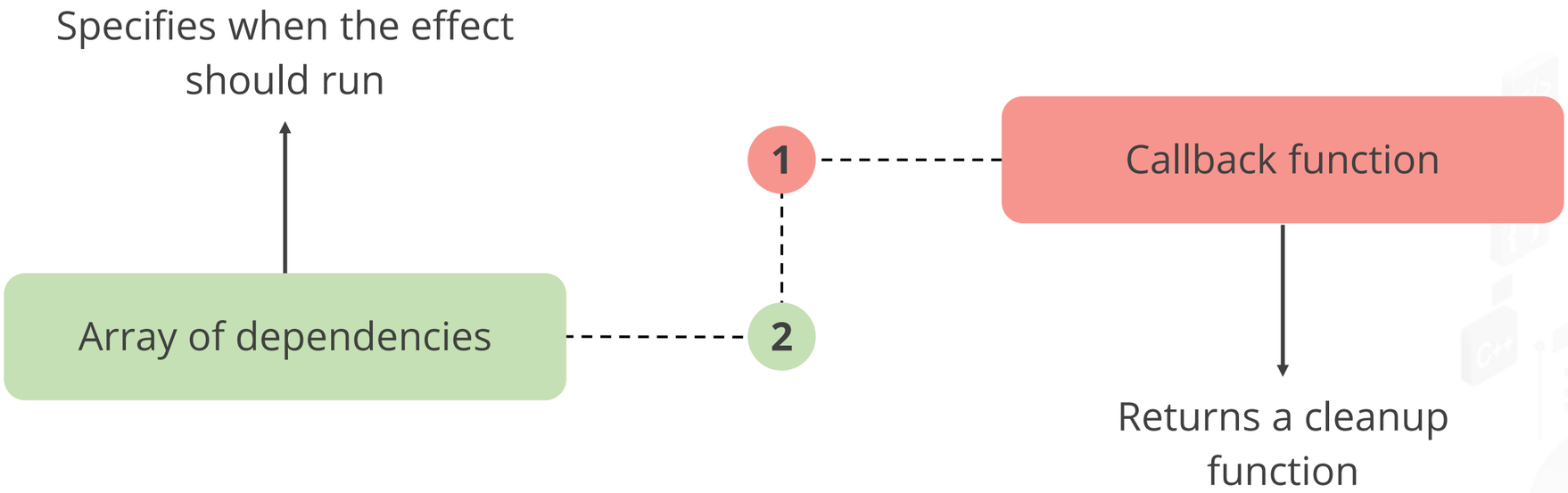
The **useEffect Hook** is a tool for managing side effects in function components. A side effect is any operation that modifies the application's state or interacts with the external world.

useEffects can be used to:

- Manage subscriptions
- Define effects in functional components
- Run after the component has rendered
- Work on a single component many times

useEffect Hook: Arguments

The **useEffect** Hook takes two arguments:



useEffect defines the effect's logic and specifies dependencies that trigger the effect's re-run.

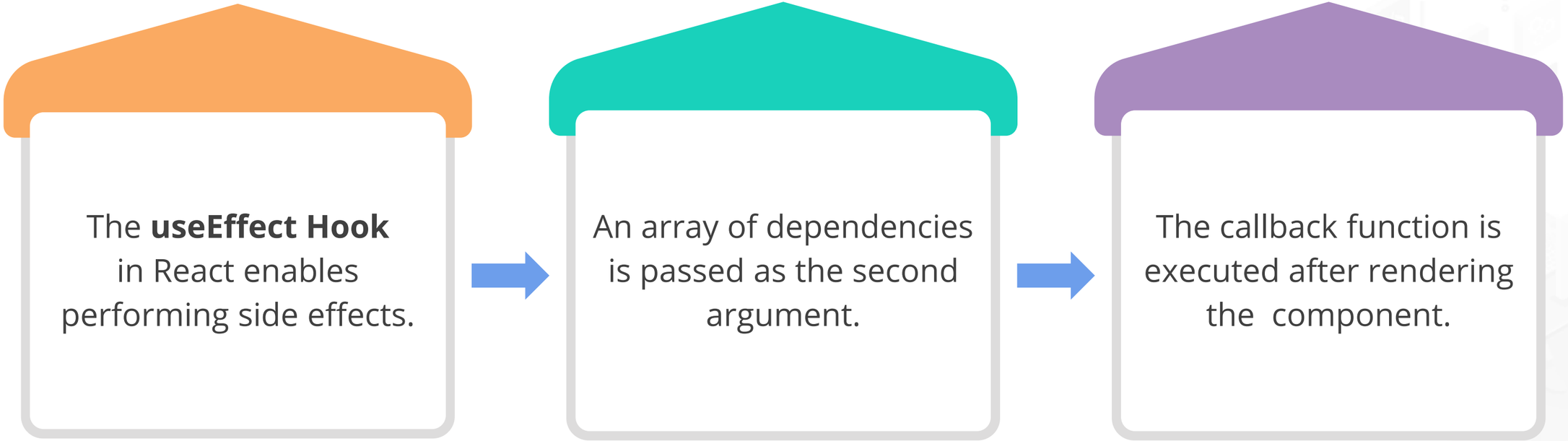
useEffect Hook: Tasks

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

- Fetching data
- Setting up event listeners
- Updating the browser title

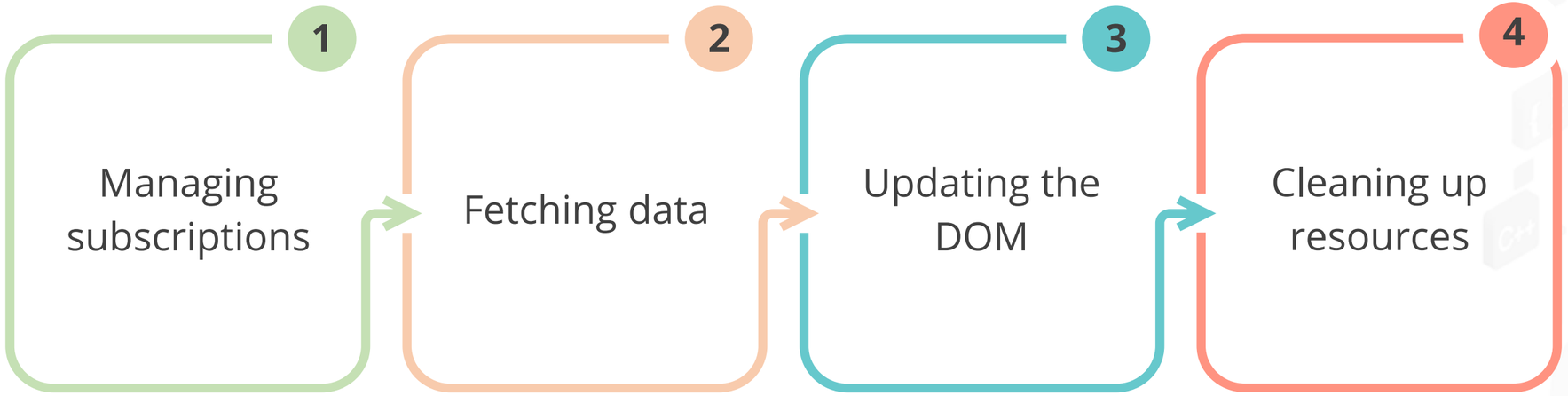
useEffect After Render

The **useEffect Hook** after rendering can handle side effects resulting from component updates in the following way:



useEffect After Render

Tasks that can be achieved using the **useEffect Hook** are:



Fetching Data with useEffect

It is essential to fetch data with useEffect to asynchronously retrieve data from an API or server after the component has been rendered as it:

- Ensures data retrieval happens asynchronously
- Helps maintain a smooth user experience
- Allows easy control when a request is made

Fetching Data with useEffect

Here is the code to fetch data:

```
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

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

Render a list of data items if the data is not null, or a **Loading..** message if the data is null.

Demo with useEffect Hook

Duration: 20 Min.

Problem Statement:

You are given a task to demonstrate how to use the **useEffect Hook** to fetch data from the **JSONPlaceholder API** and manage the state in the app.

ASSISTED PRACTICE

Assisted Practice: Guidelines

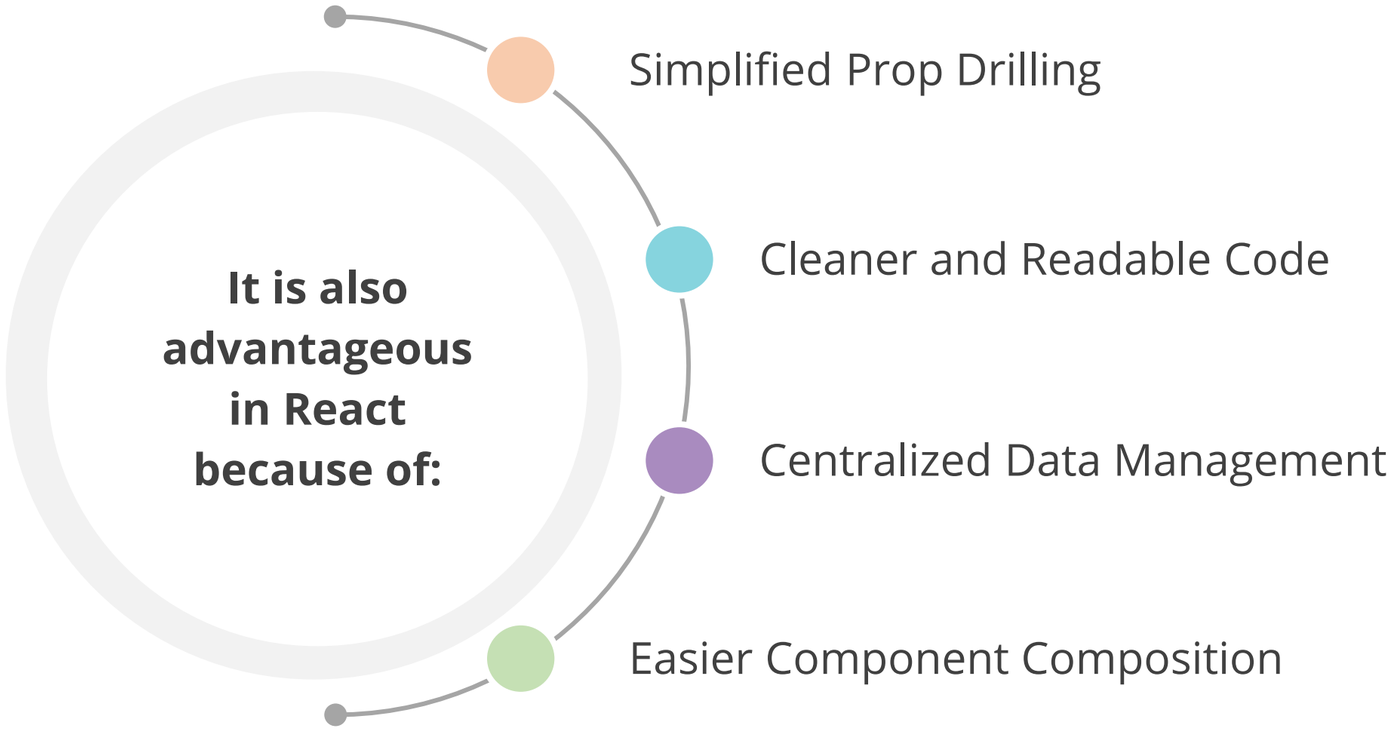
The steps to be followed are:

1. Create a new React app using **create-react-app**
2. Run **cd-use-reducer-demo**
3. Launch the project in the preferred code editor
4. Import **useReducer** and **useEffect** from React app
5. Run **npm start** in the terminal
6. Open **http://localhost:3000** in the browser

useContext Hook

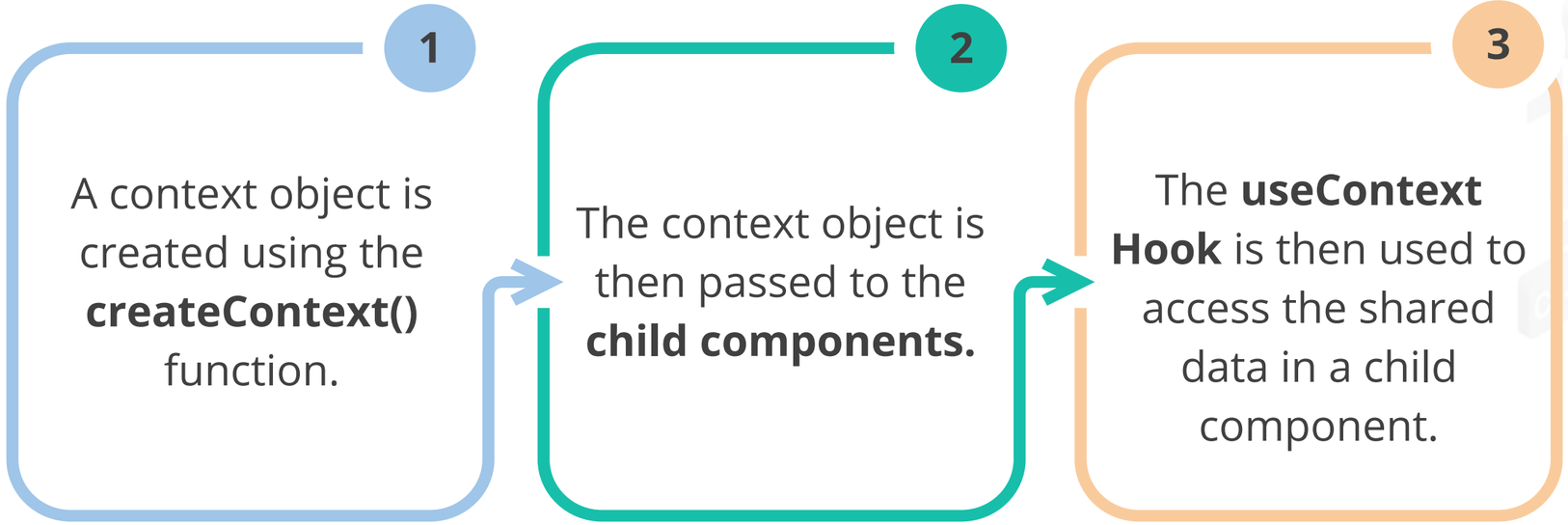
useContext Hook

The **useContext Hook** provides a way to share state and other data between components without passing it down through multiple levels of props.



useContext Hook

The **useContext Hook** is used in the following way:



Demo with useContext Hook

Duration: 10 Min.

Problem Statement:

You are given a task to create a simple app with a button that toggles the theme between light and dark and a child component that displays the current theme using the **useContext Hook**.

ASSISTED PRACTICE

Assisted Practice: Guidelines

The steps to be followed are:

1. Create a new React app using **create-react-app**
2. Change to the newly created directory
3. Open the project in the code editor
4. Create a new file called **ThemeContext.js** in the src directory
5. In **App.js**, import **useState**, **useContext**, and **ThemeContext**
6. Run the app by running **npm start** in the terminal
7. Open **http://localhost:3000** in the browser

TECHNOLOGY

Custom Hooks and State Management with Custom Hooks

What are Custom Hooks in React?

Custom Hooks are functions that use built-in Hooks and/or other custom Hooks to provide reusable functionality.

Here are the following functions of custom Hooks:

- Allow developers to extract logic from components and reuse it
- Help make the code more modular, reusable, and maintainable

What are Custom Hooks in React?

When using custom Hooks, ensure the following conditions are considered:

- 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

Rules for Creating Custom Hooks

Checklist for developing custom Hooks:

- Prefix the Hook name with use
- Use built-in Hooks
- Keep the Hooks focused
- Return an array or object
- Avoid using props
- Avoid using state from other Hooks
- Document the Hook

Creating a Custom Hook

Follow the steps below to create custom Hooks in React:

Create a new file, starting with use

Import the necessary Hooks

Create a function that returns the state

Use built-in Hooks to manage the state

Add the necessary logic to the function

Export the custom Hook function

Best Practices for Naming Custom Hooks

Follow the best practices given below while naming custom Hooks in React:

- Name custom Hooks using the **use** prefix followed by a descriptive name
- Keep the descriptive name in **camelCase** and start with a verb
- Avoid using names that conflict with existing Hooks
- Use a specific and concise name that describes the Hook's functionality

Best Practices for Naming Custom Hooks

Here are a few more tips on the best practices when naming custom Hooks in React:

- Avoid generic names or acronyms
- Use singular nouns instead of plural nouns for the name
- Consider adding a suffix to the Hook name
- Keep the name short while conveying the Hook's purpose

Demo with a Custom Hook

Duration: 10 Min.

Problem Statement:

You are given a task to build a simple app with a count displayed and two buttons that increment and decrement the count using the custom **useCounter Hook**.

ASSISTED PRACTICE

Assisted Practice: Guidelines

The steps to be followed are:

1. Develop a new React app using **create-react-app**
2. Change to the newly created directory
3. Open the project in the selected code editor
4. Create a new file called **useCounter.js** in the src directory
5. In **App.js**, import **useCounter** from the **useCounter.js** file that was just created
6. Run the app by running **npm start** in the terminal
7. Open **http://localhost:3000** in the browser

Sharing Stateful Logic with Custom Hooks

Custom Hooks can be used to share stateful logic between the components, which is a common pattern in React.

Custom Hooks allow coders to:

- Extract reusable logic into a separate function
- Use built-in Hooks or other custom Hooks
- Encapsulate complex logic
- Promote code reusability
- Accept arguments and return values

Sharing Stateful Logic with Custom Hooks: Advantages

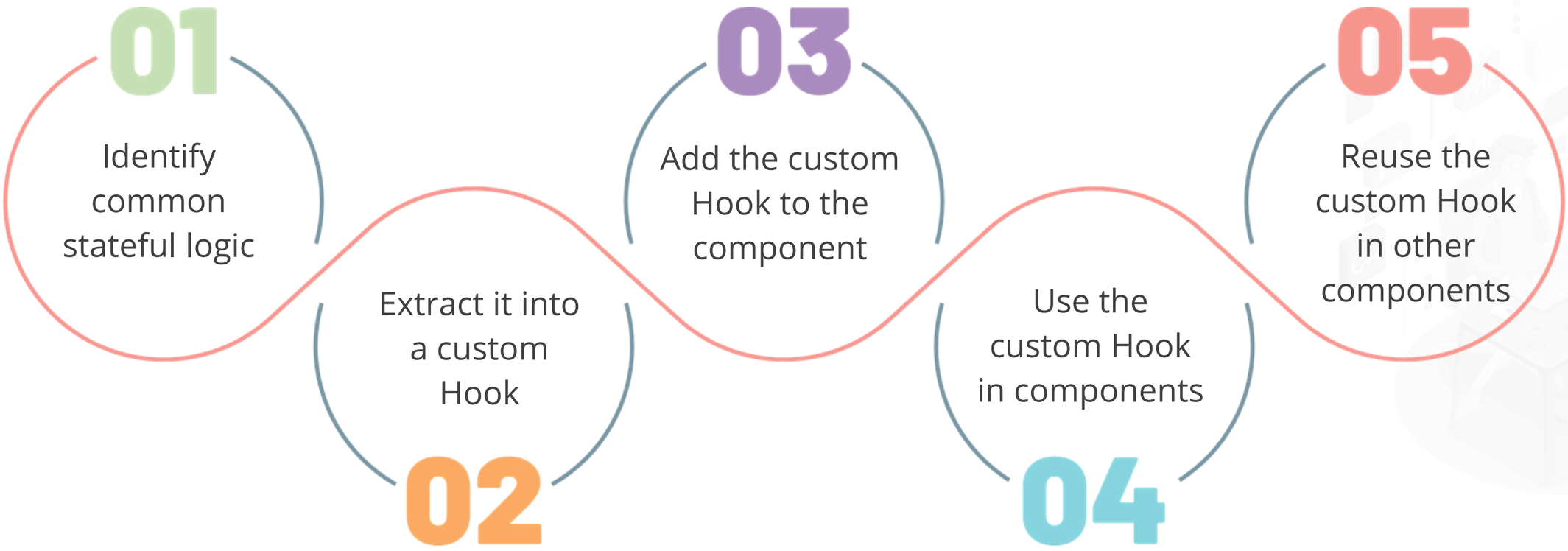
Here are some reasons why coders use Sharing Stateful Logic with custom Hooks when programming in React:

Captures all the stateful logic that multiple components can use

Extracts common logic from components and reuses it in multiple components

Steps to Create Custom Hooks for Sharing Stateful Logic

Steps to create a custom Hook for Sharing Stateful Logic:



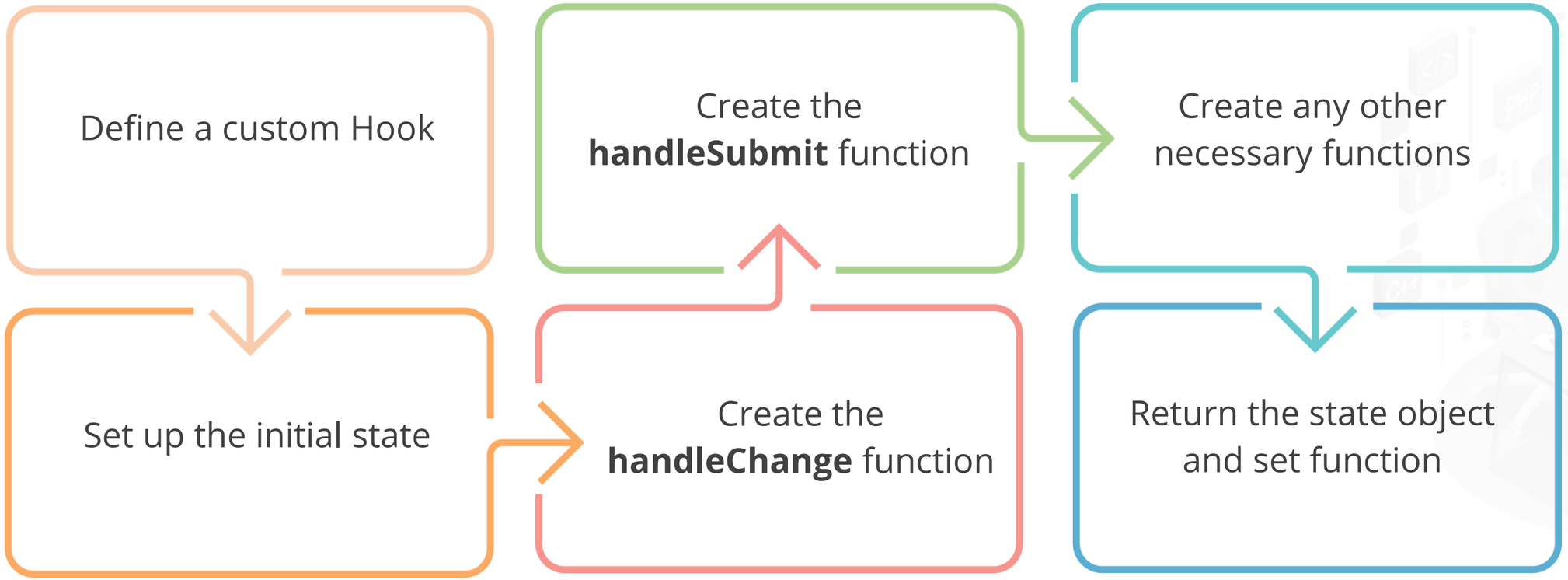
Custom Hooks for Managing Form State: Advantages

Custom Hooks can be used to manage Form States in React, such as useState or useReducer. Here are some advantages of using a custom Hook.

- Helps simplify and organize form management in React
- Handles the most common form of management tasks

Steps to Create Custom Hooks for Managing Form State

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



Custom Hooks for Fetching Data

Custom Hooks can also fetch data from APIs, simplifying data management and reusability across components. The advantages of custom Hook creation for fetching data from APIs are as follows:

Handles network requests, simplifies the code, and improves its reusability

Manages complex data-fetching logic and helps in avoiding repetitive code

Custom Hooks for Fetching Data

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

Step 1

Create a new file for the custom Hook

Step 2

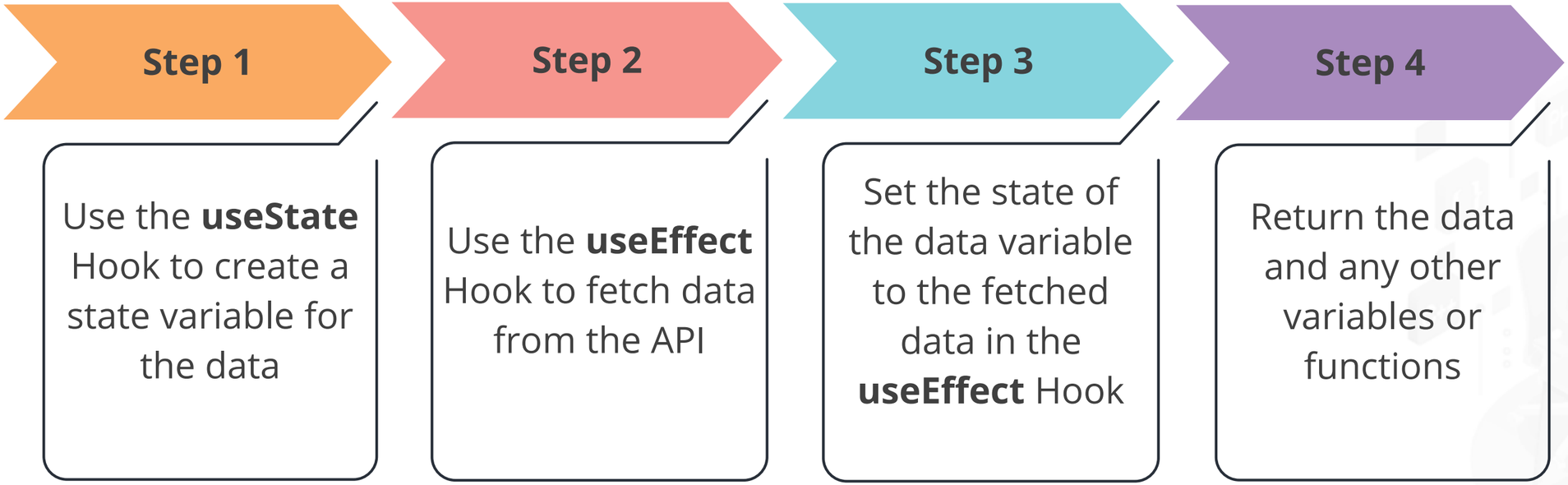
Import the **useState**, **useEffect**, and **useReducer** functions

Step 3

Define the custom Hook function with the parameters

Custom Hooks for Fetching Data

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



Demo with a Custom Hook for Fetching Data

Duration: 15 Min.

Problem Statement:

You are given a task to build a simple app that fetches and displays data from the **JSONPlaceholder API** using the custom **useFetch** Hook.

ASSISTED PRACTICE

Assisted Practice: Guidelines

The steps to be followed are:

1. Create a new React app using **create-react-app**
2. Change to the newly created directory
3. Open the project in the preferred code editor
4. Create a new file called **useFetch.js** in the src directory
5. Import **useFetch** from the **useFetch.js** file that was just created
6. Run the app by running **npm start** in the terminal
7. Open **http://localhost:3000** in the browser

Key Takeaways

- The useReducer Hook helps manage complex states and actions in a more organized and efficient way compared to useState.
- The useContext Hook helps pass data down the component tree without manually passing props at every level.
- The useEffect Hook helps perform side effects such as fetching data, subscribing to events, and updating the DOM in a functional component.
- Custom Hooks can help share logic between components, manage form states, and fetch data in a more reusable and clean way.

Demonstrable Project with All Hooks

Duration: 30 Min.

Project Agenda:

Build an app that displays a random number that changes every second, and employs the **useReducer**, **useEffect,** and **custom Hooks**

Description:

Use different types of Hooks to display data that changes every second from **useRandomNumber.js**. The tools required are Node Terminal, React App, and Visual Studio Code.

Steps to perform the following:

1. Create a new React app and change into the newly created directory
2. Open the project in your preferred code editor and create a new file
3. Import **useRandomNumber** from the **useRandomNumber.js** file
4. Run the app by running **npm start** in your terminal
5. Open **http://localhost:3000** in your browser

LESSON-END PROJECT

TECHNOLOGY

Thank You

simplilearn