Design a Dynamic Frontend with React

Rendering, LifeCycle, and React Hook Concepts

Engage and Think

You are developing a product listing page for an e-commerce website. The page needs to display a list of products dynamically, but you notice that the page refreshes slowly every time new products are added. Additionally, when filtering the product list based on price, some items do not update correctly.

What challenges do you think developers face when rendering and updating dynamic lists in React?

Learning Objectives

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

- Implement list rendering using the map() method to dynamically display data in React applications
- Apply CSS styling techniques such as inline styles, CSS modules, and CSS-in-JS to enhance UI design
- Analyze the React component lifecycle by identifying key lifecycle methods and their impact on component behavior
- Evaluate component performance by using React fragments and pure components to optimize re-rendering

List Rendering in React

What Is List Rendering?

It refers to the process of dynamically generating and displaying a list of elements based on an array of data. The following aspects represent the process of list rendering in React:

Mapping to components
map() dynamically converts array items into React components

Handling various data
Supports different data structures such as arrays of products, users, or books

Rendering data
Visually presents the data as UI elements like lists, tables, or cards

List Rendering: Example

React efficiently renders lists using the map() method, dynamically generating UI elements. The code below demonstrates how an object array represents a book list:

Step 1: Define the data array

```
const books = [
  { title: "Time Machine", author: "H.G. Wells" },
  { title: "Origin of Species", author: "Charles Darwin" },
  { title: "The Odyssey", author: "Homer" }
];
```

Note:

This array will be used for rendering with .map() in the next step.

List Rendering: Example

To iterate through an array in React, use the map() method to dynamically generate UI components, as shown below:

Step 2: Map array to React components

```
const bookList =
books.map((book, index) => (
  <li key={index}>
    <h3>{book.title}</h3>
    <p>by {book.author}</p>
  </li>
));
```

This code generates a new array of React components, with each component representing an item in the book list.

List Rendering: Example

The components can be displayed as an unordered list on the screen using the following code:

Step 3: Render the list in JSX

```
return (
  <ul>
    {bookList}
  </ul>
);
```

This code displays the list of book titles and authors as list items on the web page.

List Rendering with Keys

A key is a special attribute in React used when rendering lists. It uniquely identifies each list item, helping React efficiently update and re-render components when the list changes.

Why are keys important?

Optimize performance
Reduce unnecessary re-renders

Ensure stability
Maintain correct element order

Prevent warnings
Avoid rendering issues

List Rendering with Keys: Example

To render a dynamic list in React, an array of objects with unique id values enables efficient iteration using the map() method. The following code demonstrates the creation of items with the key attribute:

Example:

```
const items = [
  { id: 1, name: "Item 1" },
  { id: 2, name: "Item 2" },
  { id: 3, name: "Item 3" }
];
```

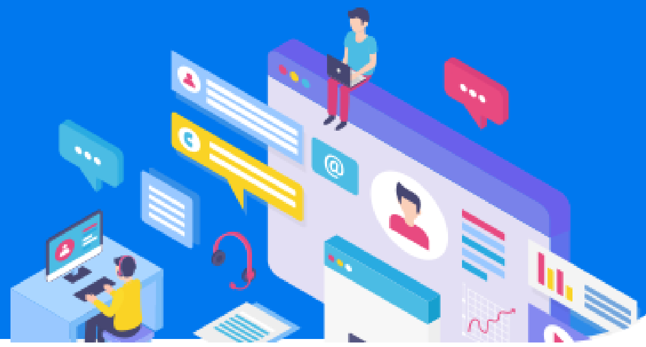
List Rendering with Keys: Example

React efficiently updates the DOM by using keys to track and manage list items. The following code demonstrates how to render a list dynamically with unique keys.

Example:

```
function ItemList() {
  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
}

export default ItemList;
```

Assisted Practice



Rendering an Unordered List in React **Duration: 15 Min.**

Problem statement:

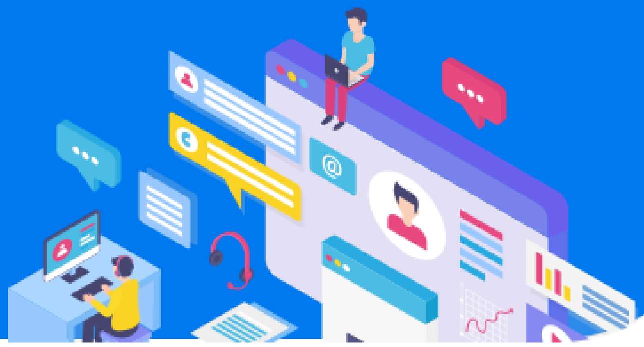
You have been asked to set up a React project using Vite and implement a React component that displays an unordered list dynamically. This task involves creating a new React component, implementing state management, and testing the application.

Outcome:

By the end of this task, you will have successfully set up a React project using Vite, created a functional component, dynamically rendered an unordered list, and validated its execution within a web browser.

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

Assisted Practice: Guidelines



Steps to be followed:

1. Set up a new React project using Vite
2. Create a React component
3. Implement and use state
4. Run and test the application

Quick Check

You are developing a task management app where users can add and remove tasks dynamically. You need to display the list of tasks efficiently. Which approach ensures that React dynamically renders the list of tasks while minimizing performance issues?

A. Using forEach() to iterate over the array

B. Using map() to generate React components

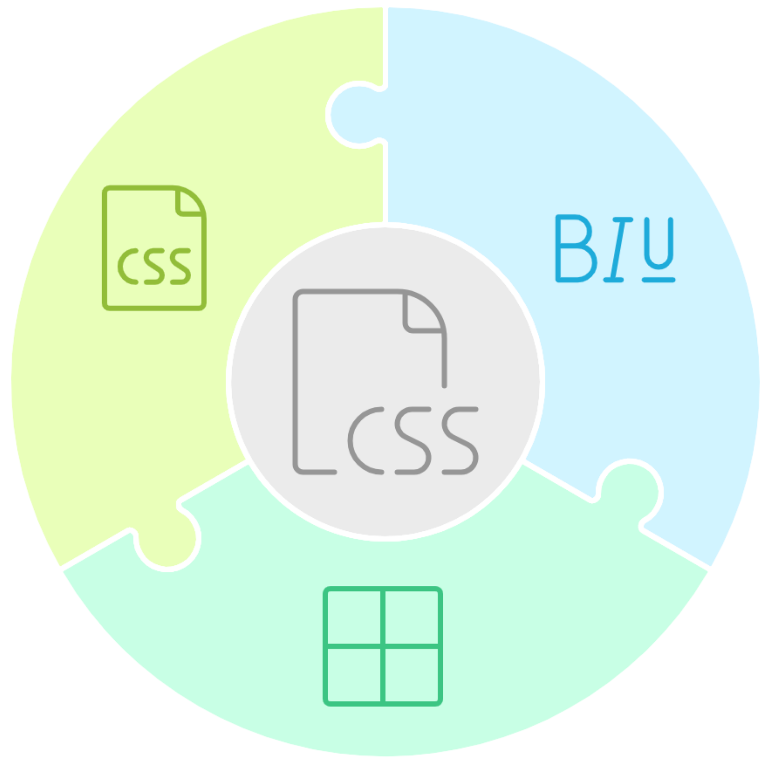
C. Wrapping tasks in a <div> without iterating

D. Using filter() to remove completed tasks

Styling in React

Fundamentals of CSS Styling

Cascading Style Sheets (CSS) enhance the visual presentation and structure of components in a web page. CSS can be applied in the following ways:



CSS-in-JS
Styling managed within JavaScript using libraries like Styled Components

Inline styles
Applied directly using the style attribute

CSS modules
Scoped styles to avoid conflicts

Inline Styles

Inline styles in React allow developers to apply styles directly to components using JavaScript objects. Instead of defining styles in an external CSS file, properties are written as key-value pairs within an object and passed to the style attribute of an element.

Example:

```
const styles = {
  backgroundColor: "blue",
  color: "white",
  padding: "10px",
  borderRadius: "5px",
};

function MyComponent() {
  return <div style={styles}>Hello, world!</div>;
}
```

CSS Modules

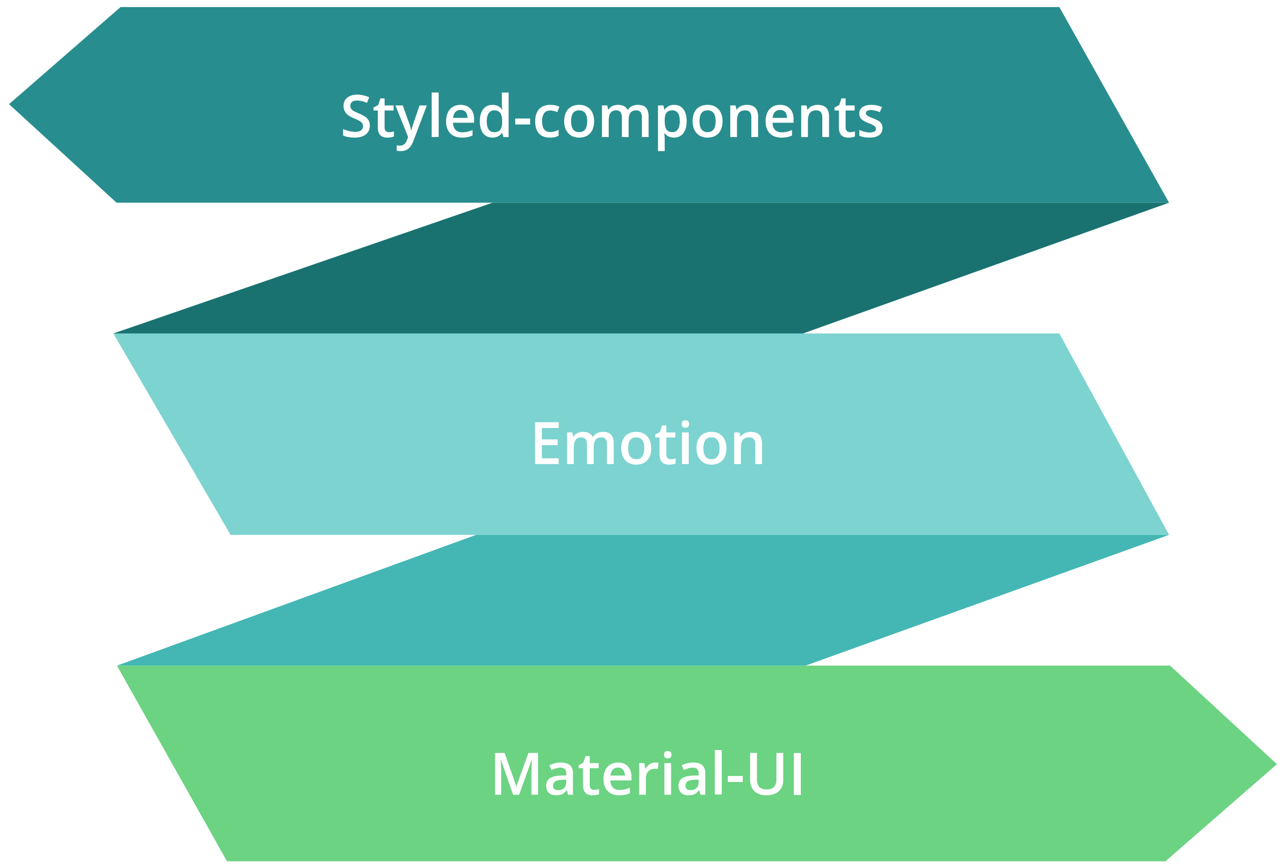
CSS modules scope styles locally to a specific component in React or JavaScript, preventing global overrides. Below is an example where **MyComponent.module.css** is imported with styles for the container class:

Example:

```
import styles from "./MyComponent.module.css";

function MyComponent() {
  return <div className={styles.container}>Hello,
world!</div>;
}

export default MyComponent;
```

CSS-in-JS

It allows styling to be written directly in JavaScript, enabling dynamic, component-scoped styles with improved flexibility and performance. The following are some popular CSS-in-JS libraries:



Allows defining styled elements with template literals

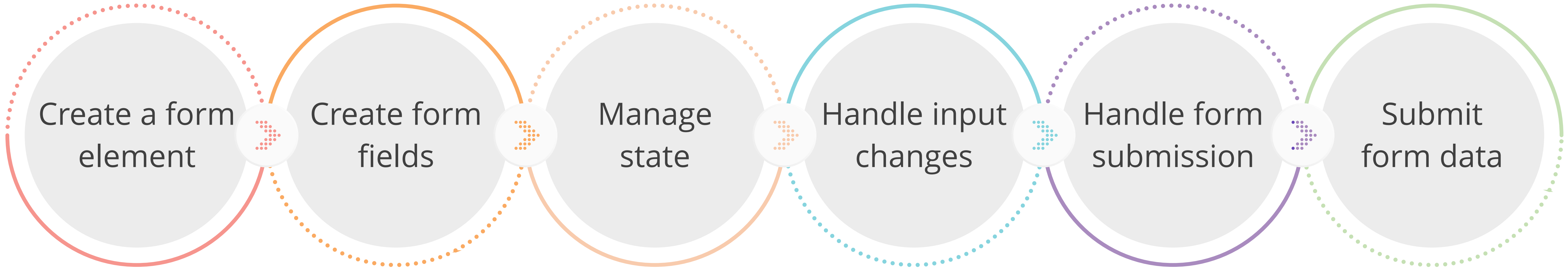
Provides high-performance styling with powerful composition features

Offers pre-designed components with customizable themes for building modern UIs

Handling Forms in React

Basics of Form Handling

It includes managing user input, state updates, and form submission using controlled components tied to the state. The basic steps involved in handling a form in React are as follows:



Form Handling: Example

In React, the useState hook is used to manage the state of form fields, such as name and email. Below is an example of basic form handling in React:

Example:

```
import { useState } from 'react';
function MyForm() {
  const [fullName, setFullName] = useState('');
  const [userEmail, setUserEmail] = useState('');
  const handleSubmit = (event) => {
    event.preventDefault();
    console.log(`Full Name: ${fullName}, Email: ${userEmail}`);
  };
  // Rest of the component code...
  return (
    // JSX code for your form...
  );
}
```

Form Handling: Example

This code represents a form with input fields for name and email, allowing users to submit their information.

Example:

```
return (
  <form onSubmit={handleSubmit}>
    <label>
      Name:
      <input
        type="text"
        name="name"
        value={name}
        onChange={(e) =>
setName(e.target.value)}
      />
    </label>
    <label>
```

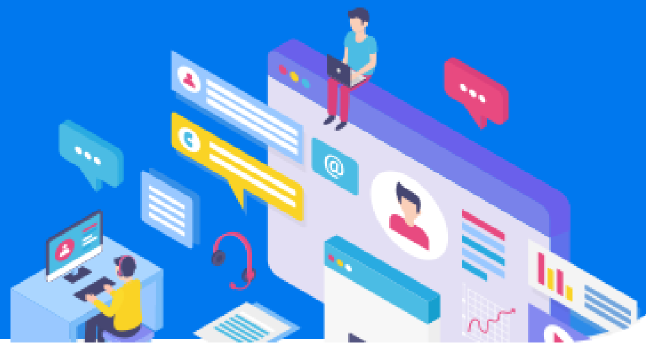
```
Email:
        <input type="email"
name="email" value={email}
onChange={(e) =>
setEmail(e.target.value)} />
      </label>
      <button
type="submit">Submit</button>
    </form>
  ); }
```

Form Handling: Example

This set of steps involves creating an event handler and updating and rendering form fields for submitting a form.

1. Create an onSubmit event handler
2. Use onChange event handlers to update
3. Render the form fields using HTML input elements
4. Use the value prop to set their values
5. Add a submit button element

Assisted Practice



Creating a Form with Three Input Fields in React **Duration: 15 Min.**

Problem statement:

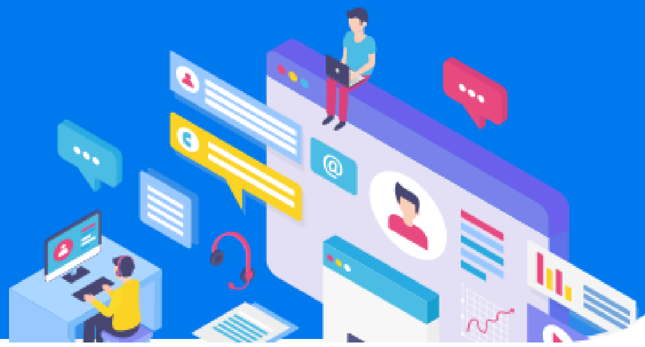
You have been asked to develop a form with three input fields, Name, Email, and Message, using React with Vite for a faster and optimized development experience. This task involves setting up a React project, creating a form component, implementing form validation, and verifying the functionality.

Outcome:

By the end of this task, you will have successfully developed a React form with Name, Email, and Message fields, implemented real-time validation, and verified the submitted data using the browser console.

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

Assisted Practice: Guidelines



Steps to be followed:

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

ASSISTED PRACTICE

React Component Lifecycle

Component Lifecycle Phases: Mounting

It is the first phase of a React component’s lifecycle, where it is created, added to the DOM, and initialized with state, props, and setup tasks. The lifecycle methods invoked sequentially during this phase are:

1 **Initialization**
constructor() is called to set up state and bind methods.

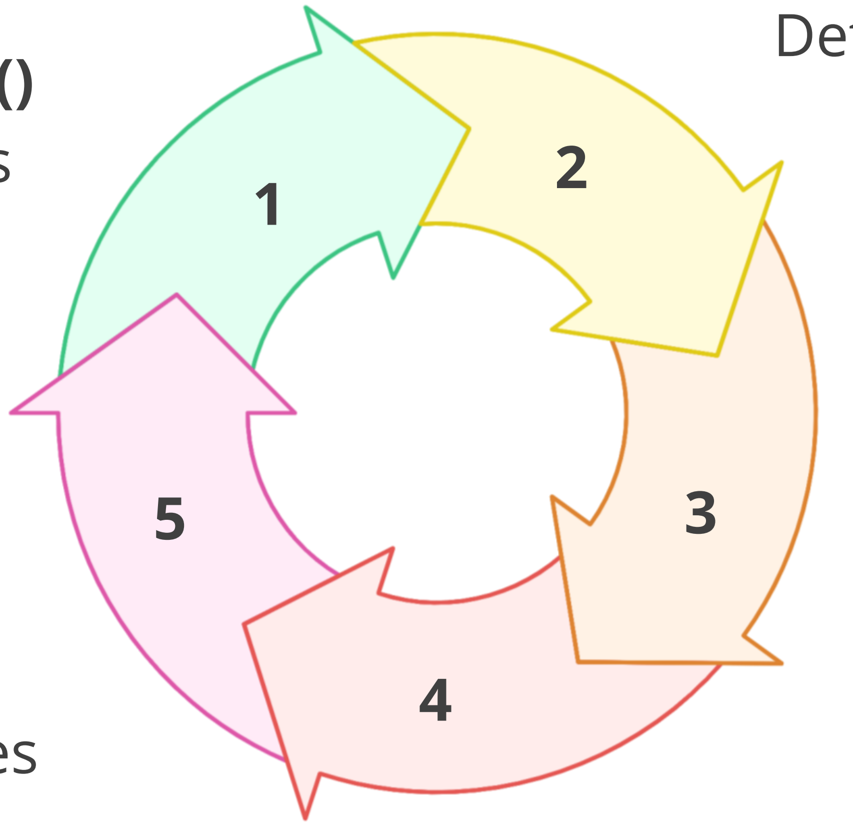
2 **State update**
static getDerivedStateFromProps() updates state based on props.

3 **Rendering**
render() returns the component's UI in JSX.

4 **Post-mounting**
componentDidMount() executes after the component mounts.

Component Lifecycle Phases: Updating

It occurs when a React component re-renders due to state or prop changes, updating the UI dynamically. The lifecycle methods called sequentially during this phase are:



static getDerivedStateFromProps()
Updates state based on new props

shouldComponentUpdate()
Determines if re-rendering is needed

render()
Updates and returns the UI

getSnapshotBeforeUpdate()
Captures information before DOM updates

componentDidUpdate()
Runs after the component updates

Component Lifecycle Phases: Unmounting

This is the last phase of a React component's lifecycle, triggered when the component is removed from the DOM, often due to navigation or conditional rendering.

Here, only one lifecycle method is called:

componentWillUnmount()

- Runs before the component is removed from the DOM
- Handles cleanup tasks like clearing timers and unsubscribing from events

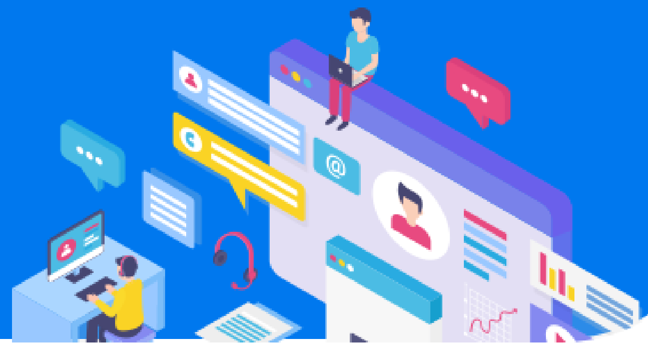
Lifecycle Handling in Functional Components

In functional components, useEffect() hook replaces lifecycle methods, handling mounting, updating, and unmounting in a single function. Below is a code template:

Example:

```
import React, { useEffect } from "react";

const MyComponent = () => {
  useEffect(() => {
    console.log("Component Mounted");

    return () => {
      console.log("Component Unmounted");
    };
  }, []); // Runs once on mount, cleans up on unmount

  return <div>Hello, World!</div>;
};

export default MyComponent;
```

Assisted Practice



Creating a React Timer App Using Vite **Duration: 15 Min.**

Problem statement:

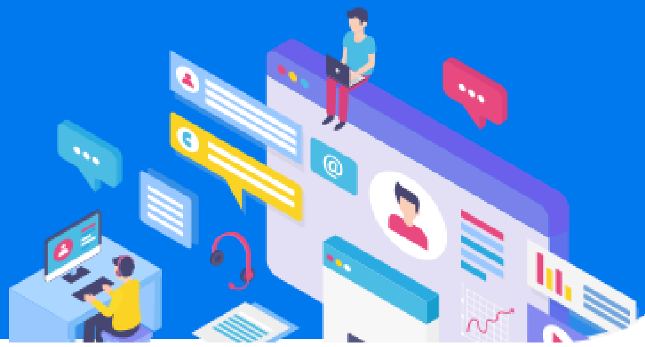
You have been asked to set up a React project using the Vite framework, implement a timer component using functional components and React hooks, and run the application efficiently in a development environment.

Outcome:

By the end of this task, you will have successfully developed a timer component in React that updates every second, along with a reset functionality.

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

Assisted Practice: Guidelines



Steps to be followed:

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

Quick Check

You are developing a weather dashboard that fetches real-time weather data when the component loads. However, you notice that the API call is happening multiple times, even when the component should only fetch data once. Which lifecycle method should you use to ensure the weather data is fetched only once when the component mounts?

A. componentDidUpdate()

B. componentWillUnmount()

C. componentDidMount()

D. render()

React Fragments and Pure Components

What Are Fragments?

Fragments are a React feature that group multiple child elements without adding an extra DOM node. The code below demonstrates this:

Example:

```
render() {
  return (
    <div> {/* Unnecessary
wrapper */}
      <h1>Heading</h1>
      <p>Paragraph</p>
    </div>
  );
}
```

```
render() {
  return (
    <> {/* Fragment
removes extra wrapper */}
      <h1>Heading</h1>
      <p>Paragraph</p>
    </>
  );
}
```

In this code example, the render method returns a JSX expression. Using React Fragments instead of a div can help avoid adding unnecessary DOM nodes.

React Fragments: Example

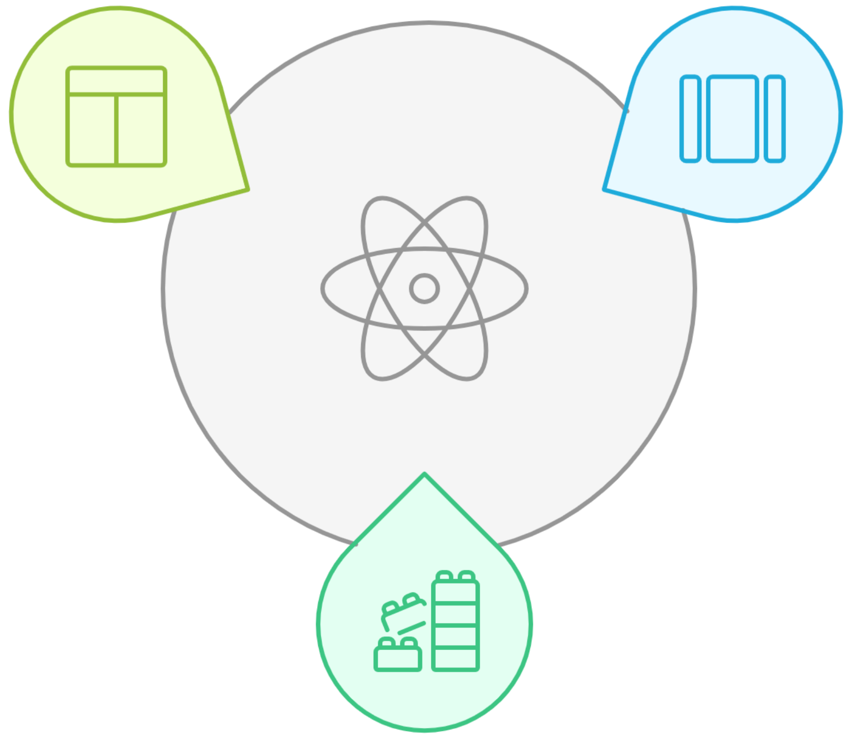
They can be used with keys to provide a stable identity for elements across updates, especially when rendering lists, as demonstrated in the following code:

Example:

```
render() {
  return (
    <>
      {this.props.items.map(item => (
        <React.Fragment key={item.id}>
          <h1>{item.title}</h1>
          <p>{item.content}</p>
        </React.Fragment>
      ))}
    </>
  );
}
```

Fragments: Use Cases

Fragments are useful in situations where multiple elements need to be grouped without unnecessary DOM nodes. Here are key use cases:



Avoiding unnecessary markup
Prevents clutter in the DOM, enhancing performance

Returning multiple elements
Allows components to return multiple elements without extra nodes

Grouping children
Groups children in a component for easier management

Pure Components: Overview

It is a type of component that implements shouldComponentUpdate() with a shallow comparison of props and state to determine whether it should re-render.

Below is an example of a pure component:

Example:

```
import React, { PureComponent } from "react";

class MyComponent extends PureComponent {
  render() {
    return <div>{this.props.title}</div>;
  }
}
```

They only re-render when their props or state change, optimizing performance by preventing unnecessary re-renders.

Key Characteristics of Pure Components

Pure components optimize performance by preventing unnecessary re-renders. Below are some key points about how they work and their benefits:

shouldComponentUpdate() method

Automatically called whenever props or state change

Comparison of Props and State

Checks previous and current values to determine if re-rendering is needed

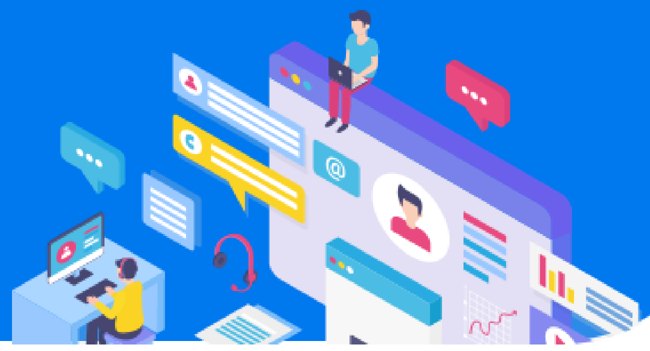
Works with immutable data

Pure components are effective only if props and state are immutable

Performance optimization

Helps improve performance by preventing unnecessary re-renders

Assisted Practice



Creating a Component That Re-Renders in React **Duration: 15 Min.**

Problem Statement:

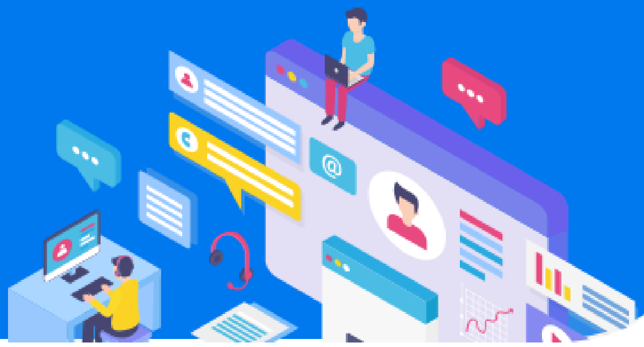
You are building a dynamic UI in React. Your task is to create a component that re-renders when its prop value changes, ensuring that UI updates reflect the latest state.

Outcome:

By the end of this demo, you will be able to build and use a memoized React component that re-renders based on prop changes and displays updated content.

Note: Refer to the demo document for detailed steps:
04_Creating_a_Component_That_Re-renders_in_React

Assisted Practice: Guidelines



Steps to be followed:

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

React Hooks: useState

React Hooks: Introduction

It allows functional components to manage state and lifecycle features without class components.
Below are key rules to follow when using hooks in React:

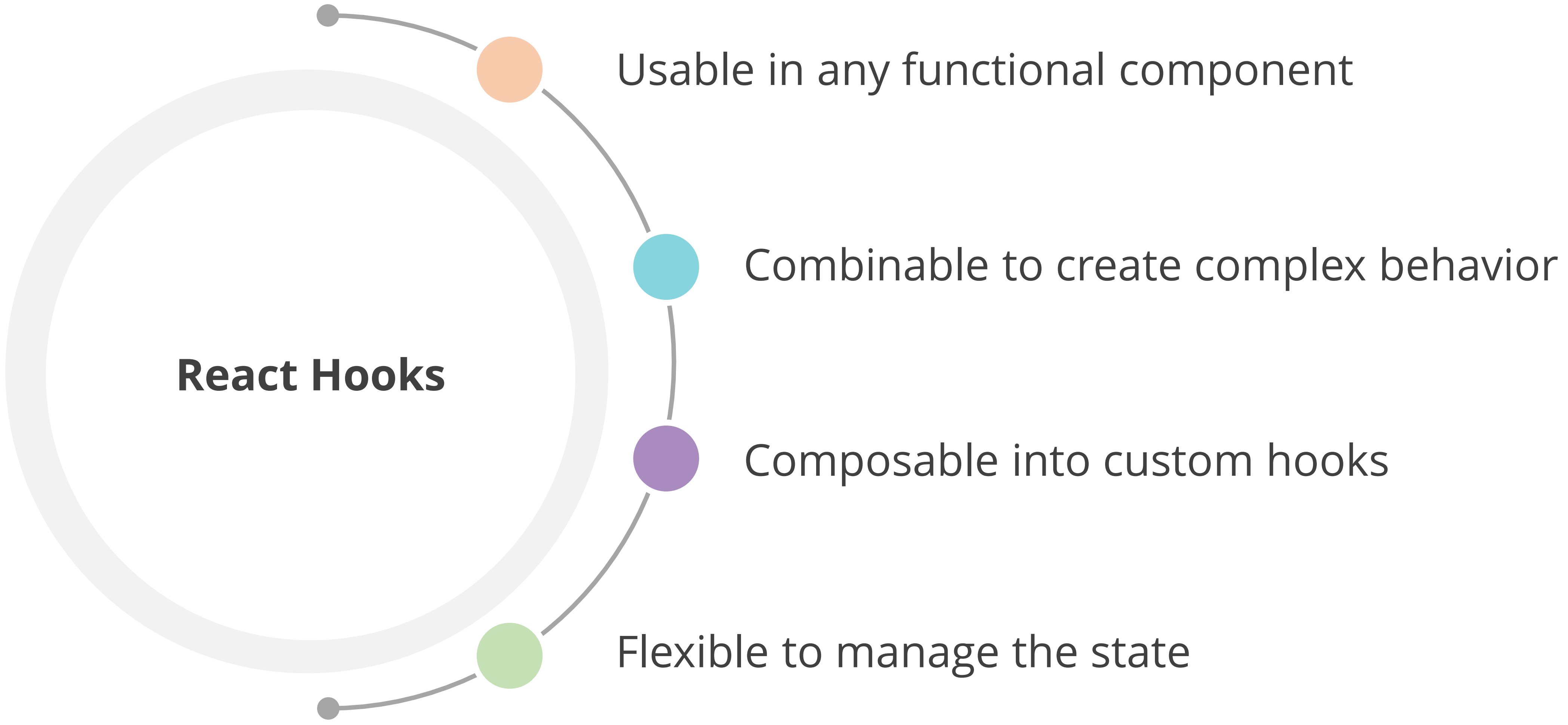
It can be used only at the top level.

It can be called only inside the functional components.

It must always be unconditionally called.

React Hooks: Features

React hooks simplify state management and lifecycle methods in functional components. Below are some of their key features:


React Hooks
Usable in any functional component
Combinable to create complex behavior
Composable into custom hooks
Flexible to manage the state

useState Hook: Example

useState hook manages state in functional components. Below is an example of a counter with increment and decrement functions:

```
import React, { useState } from 'react';
const Counter = () => {
  const [counterValue, setCounterValue] =
useState(0);
  const handleIncrement = () => {
    setCounterValue(counterValue + 1);
  };
  const handleDecrement = () => {
    setCounterValue(counterValue - 1);
  };
```

```
  return (
    <div>
      <p>Count: {counterValue}</p>
      <button
onClick={handleIncrement}>+</button>
      <button onClick={handleDecrement}>-
</button>
    </div>
  );
};
export default Counter;
```

useState Hook: Example

In the previous slide, the Counter component was designed. Now, we import it into the App component and render it to manage the count state. The code below demonstrates this:

Example:

```
import React from 'react';
import Counter from './Counter';

const App = () => {
  return (
    <div>
      <Counter />
    </div>
  );
};

export default App;
```

useState: Updating State

To update the state based on its previous value in React, use the functional form of setState. The code below demonstrates how to use the useState hook efficiently to manage state updates in a counter component:

Example:

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

const Counter = () => {
  const [count, setCount] = useState(0);
```

```
const handleIncrement = () => {
    setCount(prevCount => prevCount + 1);
  };

  const handleDecrement = () => {
    setCount(prevCount => prevCount - 1);
  };
```

In this example, the Counter component uses useState to manage the count state and updates it using event handlers with the functional form of setCount.

useState: Updating State

The code below renders a React counter component, displaying the current count value and providing buttons to increment and decrement the count:

Example:

```
return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleIncrement}>+</button>
      <button onClick={handleDecrement}>-</button>
    </div>
  );
};

export default Counter;
```

useState: Defining Object State Variables

Object state variables in React can be defined using the useState hook by initializing the state with an object. Here is an example:

Example:

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

const Form = () => {
  const [user, setUser] = useState({
    name: '',
    email: '',
    password: '',
  });

  const handleChange = (event) => {
    const { name, value } = event.target;
    setUser((prevUser) => ({
      ...prevUser,
      [name]: value,
    }));
  };
```

useState: Defining Object State Variables

The following code defines a handleSubmit function that prevents the default form submission. It also logs the user object and renders a form with an input field:

Example:

```
 const handleSubmit = event => {
    event.preventDefault();
    console.log(user);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label htmlFor="name">Name</label>
      <input type="text" id="name" name="name" value={user.name} onChange={handleChange} />

;
```

useState: Defining Object State Variables

The code includes input fields for email and password, each with a corresponding label and a submit button, all contained within a form component:

Example:

```
 <label htmlFor="email">Email</label>
      <input type="email" id="email"
name="email" value={user.email}
onChange={handleChange} />

      <label
htmlFor="password">Password</label>
      <input type="password"
id="password" name="password"
```

```
value={user.password}
onChange={handleChange} />

      <button
type="submit">Submit</button>
    </form>
  );
};

export default Form
```

useState: Defining Array State Variables

The following code uses the useState hook to define an array state variable by initializing it with an array:

Example:

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

const List = () => {
  const [items, setItems] =
useState(['Item 1', 'Item 2', 'Item 3']);

  const handleClick = () => {
    setItems([...items, `Item
${items.length + 1}`]);
  };

  return (
    <div>
```

```
<div>
      <ul>
        {items.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
      <button onClick={handleClick}>Add
Item</button>
    </div>
  );
};

export default List;
```

The code defines a List component that maintains a state variable named items, which is an array of strings.

Quick Check

You are working on a real-time stock market dashboard that displays stock prices. However, the entire dashboard re-renders unnecessarily, even when only a few stock prices change. How can you optimize performance and prevent unnecessary re-renders?

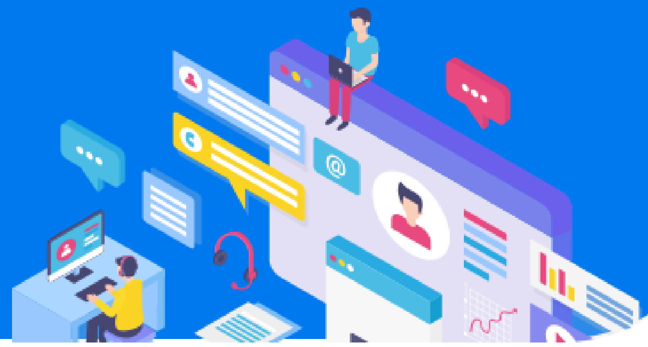
A. Use pure components to optimize rendering

B. Use useEffect to check prop changes

C. Wrap components in a <div>

D. Avoid React state and use only props

Assisted Practice



Creating a New React Component Using useState Hook **Duration: 15 Min.**

Problem Statement:

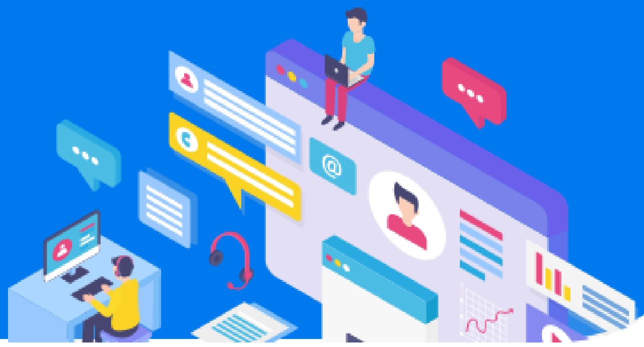
You need to create a React component that tracks and updates a numeric value based on user interaction. The component must maintain internal state using the useState Hook.

Outcome:

By the end of this demo, you will be able to implement the useState Hook within a functional component to handle dynamic changes in application state.

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

Assisted Practice: Guidelines



Steps to be followed:

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

ASSISTED PRACTICE

Creating a Dynamic To-Do List Application

Duration: 25 Mins.

Project agenda: The goal is to develop a dynamic to-do list application using React, focusing on state management, event-driven updates, and rendering optimization. The application will support task addition, completion toggling, and removal, demonstrating React Hooks and efficient rendering techniques.

Description: You are tasked with building a React-based to-do list application to improve task management efficiency. This involves handling user input, updating state dynamically, and optimizing rendering performance. The project ensures smooth user interaction by leveraging React Hooks, event-driven updates, and lifecycle management.

Creating Malware, Exploiting Linux System, and Performing Dynamic Malware Analysis

Duration: 25 Min.

Perform the following:

1. Set up a new React project using Vite
2. Implement core components
3. Integrate components in App.jsx
4. Style the application
5. Run and verify the application

Expected deliverables: A fully functional React-based to-do list application with dynamic task management features. The application will support task addition, completion toggling, and deletion while ensuring optimized performance. It will include structured reusable components, state management with useState, and lifecycle management with useEffect.

Key Takeaways

- The map() method allows the iteration of an array of data to create a new array of React components based on that data.
- There are several ways to add styling and CSS to the React components, such as Inline styles, CSS modules, and CSS-in-JS Libraries.
- A fragment is a feature in React that allows users to group a list of child elements without creating an additional DOM node.
- In React, a Pure Component only re-renders when its props or state changes.
- The useState hook is a built-in React hook that allows users to add a state to the functional components.

Thank You