Design a Dynamic Frontend with React

Introduction to React and Project Setup

Engage and Think

You have joined a new company as a MERN Stack Developer, and your first project involves building a dynamic web application. The team has chosen a specific technology to develop the frontend, but you have never worked with it before. As you start exploring it, you realize that it follows a different approach compared to traditional JavaScript or other frontend frameworks.

Why do you think this technology has become so popular for building modern web applications? What advantages might it offer compared to traditional JavaScript-based development?

Learning Objectives

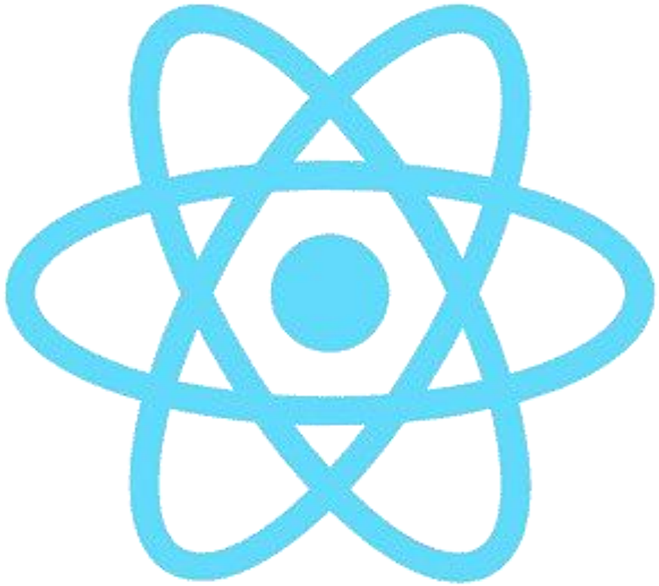
By the end of this lesson, you will be able to:

- Construct the default structure of a ReactJS project to establish a solid foundation for the development workflow
- Develop an efficient user interface by applying the concepts of React components
- Implement interactive web applications using event handlers to dynamically enhance the user interface
- Optimize rendering concepts to deliver a smooth and responsive user experience

Getting Started with React

What Is React?

It is a JavaScript-enabled, open-source library that is used for creating the frontend of an application.



React: Benefits

React is a powerful JavaScript library known for its efficiency and scalability. Below are some advantages of using React:

Prevents unintended side effects

Pre-renders for better performance

Converts code for faster startup

Updates efficiently by representing the page in memory

Breaks UIs into reusable components

Passes data from parent to child

Applications Using React

Here are examples of major companies utilizing React for their applications and the reasons behind their choice:

Facebook

Used ReactJS for its web platform and React Native for its mobile app, leveraging its component-based architecture for scalability

Instagram

Integrated ReactJS for features such as geolocation, Google Maps APIs, and dynamic tags, ensuring a seamless user experience

Applications Using React

Here are examples of major companies utilizing React for their applications and the reasons behind their choice:

Netflix

Implemented React in its Gibbon platform for TV devices, benefiting from its efficient rendering and performance optimization

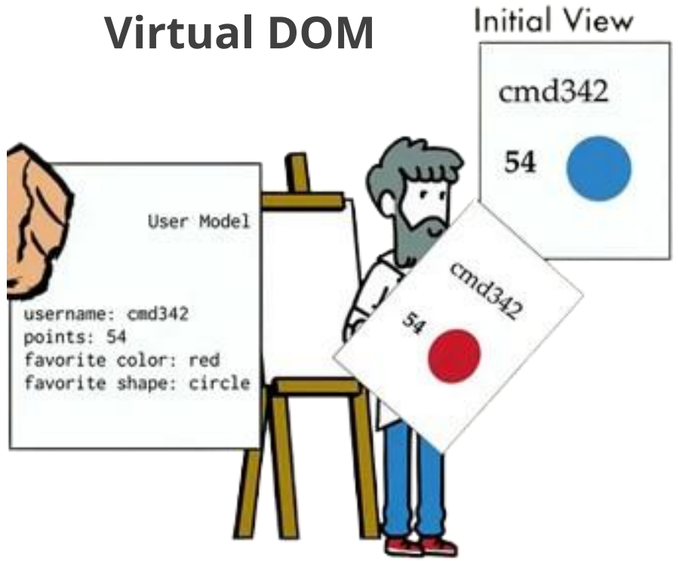
BBC

Developed BBC iPlayer with React and migrated the BBC News website to Next.js, improving performance and maintainability

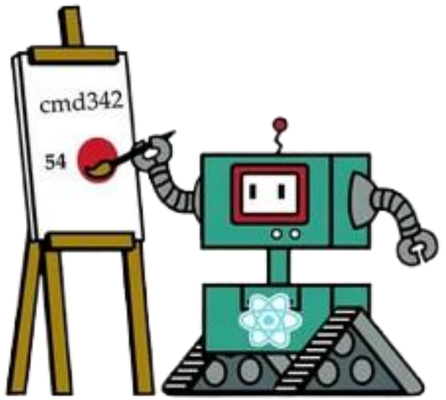
Virtual DOM in React

It is a lightweight, in-memory version of the real DOM, used by frameworks like ReactJS to efficiently update and render UIs without direct DOM manipulation.

Virtual DOM



Real DOM



Vite Integration in React Projects

What Is Vite?

It is a JavaScript build tool that is used alongside React to develop web applications more efficiently and quickly.

Vite leverages native ES modules and a lightning-fast Hot Module Replacement (HMR) system, ensuring a seamless and responsive development experience.

Vite: Importance

The following are some reasons for using Vite:

Faster builds
Speeds up build times, especially for large projects

Instant updates
Provides real-time browser updates

Optimized production
Uses Rollup for efficient code splitting

TypeScript support
Enhances maintainability and error detection

Better performance
Loads only necessary modules

Extensible plugins
Supports CSS preprocessing, state management, and more

Setting Up a React Project with Vite

Create a new React project using Vite by running the following command in the terminal:

Syntax:

```
npm create vite@latest my-react-app
```

This approach offers a faster and more efficient development experience compared to traditional methods.

Running a React Project with Vite

After creating the React project, navigate to its directory and start the Vite development server using the command below for an optimized development experience:

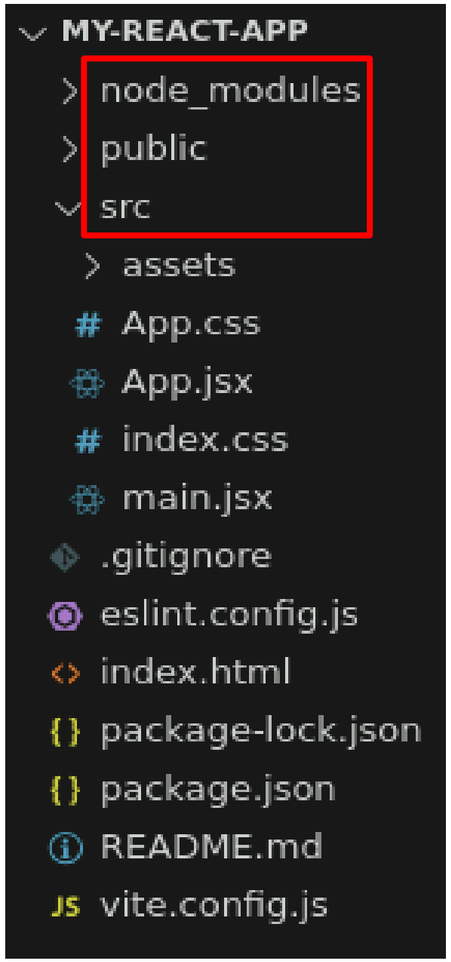
Example:

```
cd my-react-app

npm run dev
```

React Folder Structure

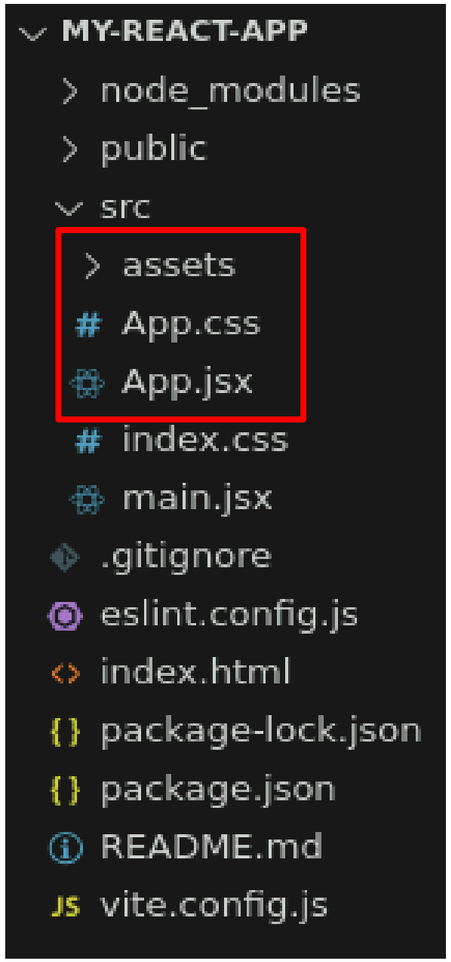
The following is the folder structure of a React app created using the Vite framework:



- **node_modules**: Directory where all project dependencies are installed; it is managed automatically by npm.
- **public:** Folder for storing static assets like images and fonts that are not processed by Vite
- **src**: Main working directory containing core files and components of the React project

React Folder Structure

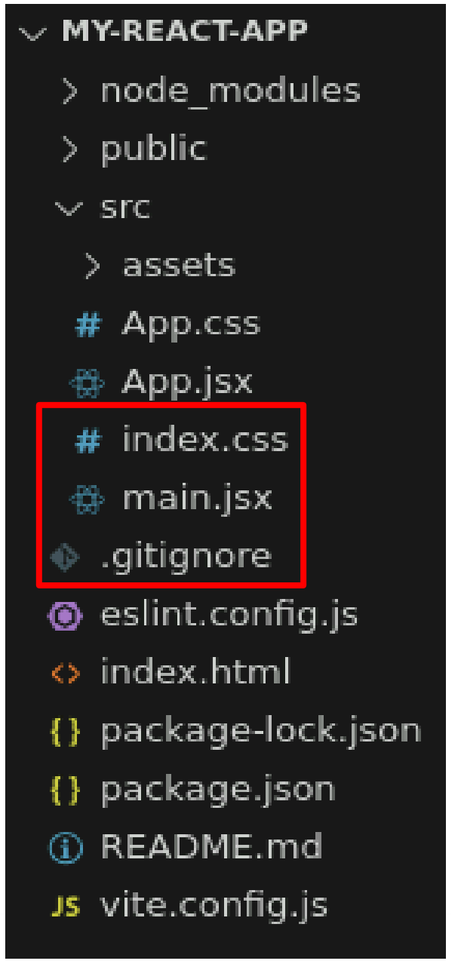
The following is the folder structure of a React app created using the Vite framework:



- **assets:** A subfolder inside the src directory that stores images, audio, and video files optimized by Vite
- **App.css:** Stylesheet for the App.jsx component
- **App.jsx:** Root component where the main application structure and logic reside

React Folder Structure

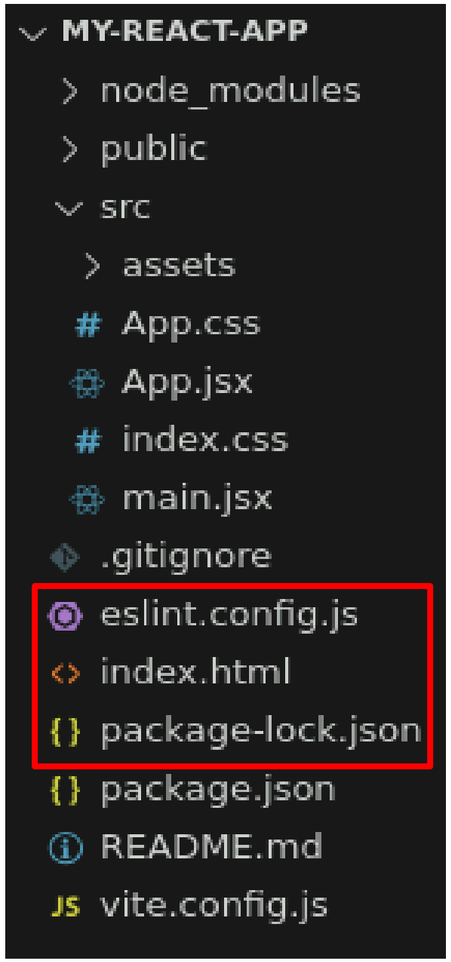
The following is the folder structure of a React app created using the Vite framework:



- **index.css:** Global stylesheet for applying consistent styles across the entire application
- **main.jsx:** Entry point file that injects the root component into index.html and sets up global configurations
- **.gitignore:** Configuration file specifying which files and directories should be ignored by Git for version control

React Folder Structure

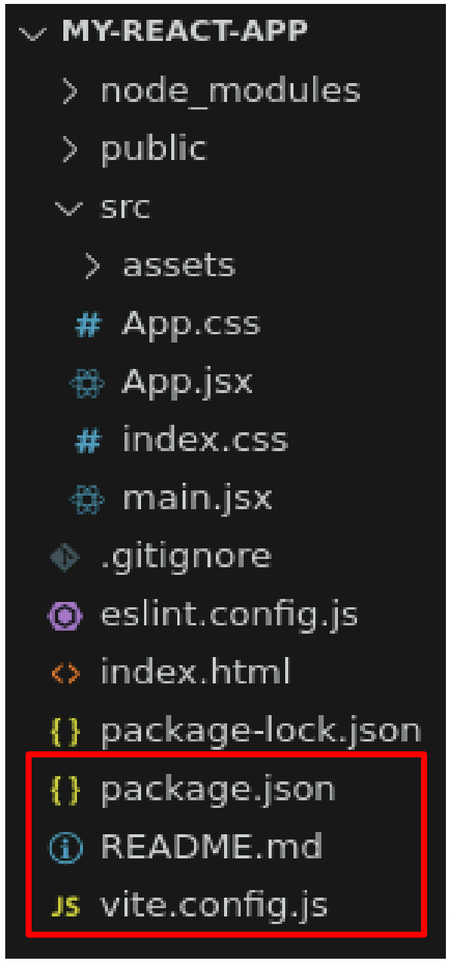
The following is the folder structure of a React app created using the Vite framework:



- **eslint.config.js:** Configuration file for ESLint to enforce coding standards and catch potential issues in JavaScript and JSX
- **index.html:** Main HTML file serving as the entry point for the React application
- **package-lock.json:** Auto-generated file that locks dependency versions to ensure consistency across environments

React Folder Structure

The following is the folder structure of a React app created using the Vite framework:



- **package.json:** Metadata file containing project details, dependencies, scripts, and configurations
- **README.md:** Project documentation file containing essential details about the project, including setup instructions and contributions
- **vite.config.js:** Configuration file for Vite, used to customize the development environment, add plugins, and configure loaders

Quick Check

A developer is setting up a React project with Vite. They need to find the main entry point for the application. Which file should they look for?

A. eslint.config.js

B. index.html

C. package-lock.json

D. vite.config.js

Basic React Application

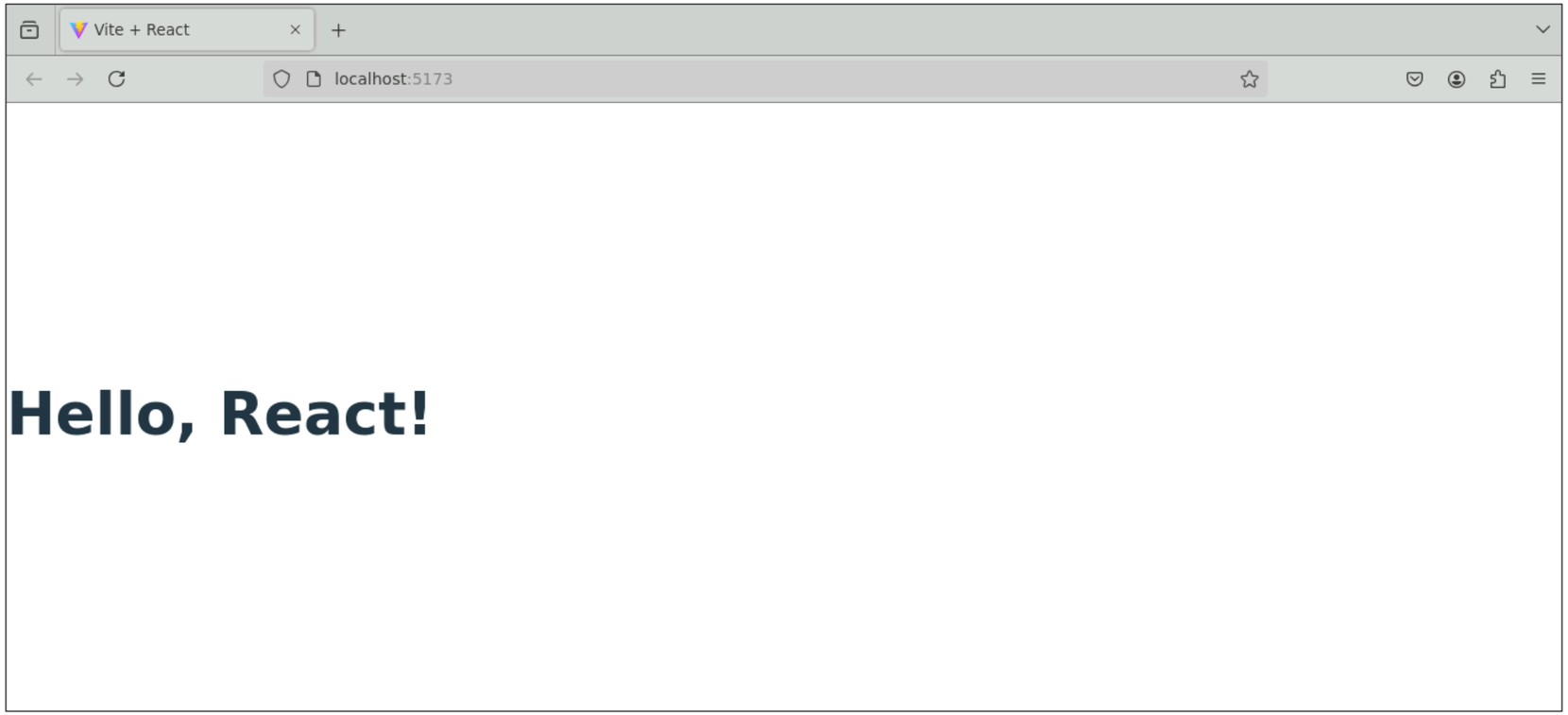
The following is the basic React application code to display **Hello, React!** on the screen:

```
function App() {
  return (
    <div>
      <h1>Hello, React!</h1>
    </div>
  );
}

export default App;
```

Basic React Application

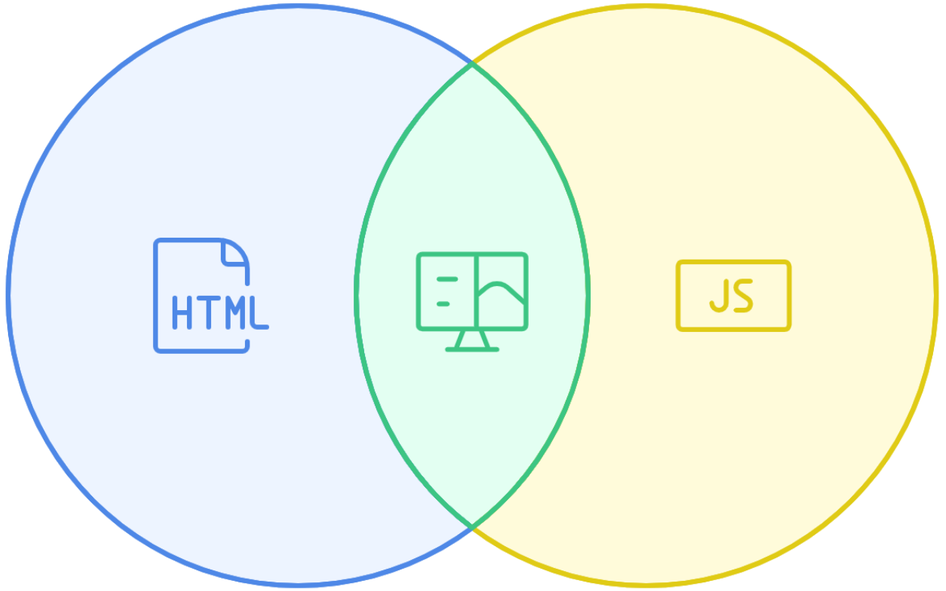
The following is the output of the basic React application



Rendering UI with JSX and Component-Based Architecture

JSX in React

It is a syntax extension of JavaScript used in React. It allows developers to write HTML-like code within JavaScript, making it easier to create and manage UI components.



JSX

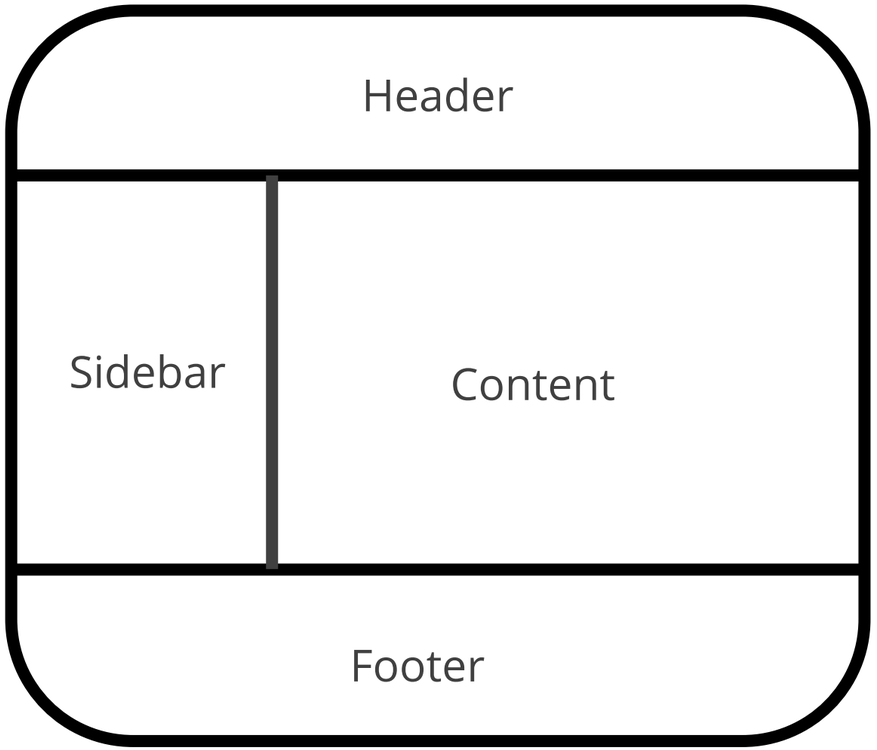
Benefits of Using JSX

Developers use the JSX component in React applications for the following reasons:

- Allows HTML-like code in JavaScript files
- Efficiently creates interactive UI components
- Allows declarative and component-based code
- Allows dynamic content and conditional rendering
- Improves code organization and reusability
- Provides intuitive syntax for defining UI elements

Component Architecture in React

It is a design pattern where the UI is broken down into reusable, independent pieces called components.



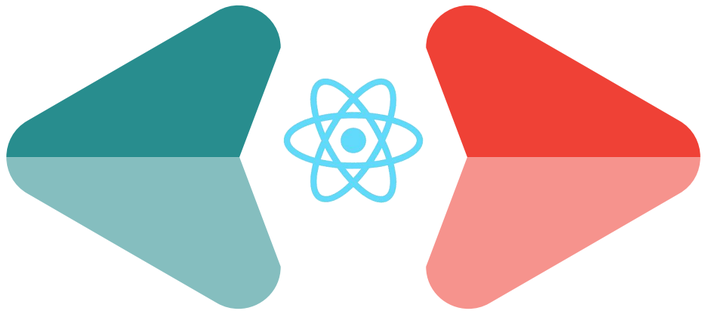
Each component is responsible for:

- Rendering a specific part of the UI
- Handling its own logic
- Creating a hierarchy of components

Types of Components in React

The following are the two main types of components in React:

Functional components



Class components

Functional Components

They are simple, reusable JavaScript functions that return UI elements and perform better without using classes. The following are key features of functional components:

Testing
Provides easier testing due to predictable behavior

Hooks
Enables state management and lifecycle handling

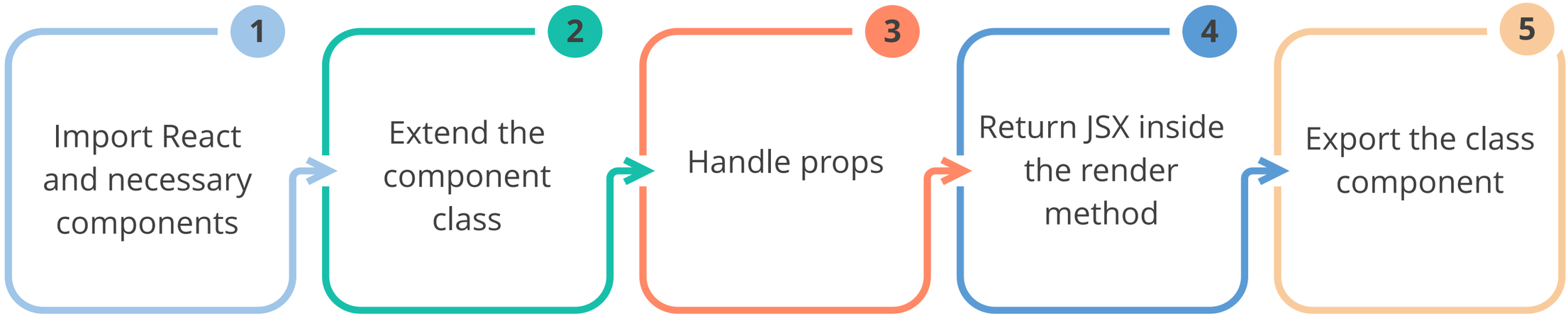
Simplicity and readability
Makes code easier to write and understand

Reusability
Allows reuse across different parts of an application

Good performance
Ensures faster execution without class overhead

Class Components

They are React components written as classes that manage data and handle updates using lifecycle methods. The following are the steps to create a class component in React:



Class Components: Example

The following example shows how a class component can be developed:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, John</h1>;
  }
}
```

Example:
The Welcome class extends React.Component and returns an <h1> tag with **Hello, John**.

Functional Components: Example

The following example shows how a functional component can be developed:

```
function Welcome() {
  return <h1>Hello, John</h1>;
}
```

Example:
The Welcome function is a functional component that returns an <h1> tag with **Hello, John**.

Functional vs. Class Components

Developers often choose functional components over class components due to their simplicity and efficiency. Here are some of the reasons why they do so:

Simplicity	Accept props and return JSX, making them easy to read
No *this* Keyword	Eliminates complexity, making code beginner-friendly
Stateless by default	Lightweight, unless using hooks for state
Simplified logic	Focuses on UI rendering with minimal complexity

Assisted Practice

Creating a React Application Using React Components **Duration: 15 Min.**

Problem statement:

You have been tasked with creating a React application using React components. The goal is to set up a new React project using the Vite framework, implement a reusable component, and successfully render it within the application.

Outcome:

By the end of this task, you will be able to set up and configure a React application using Vite, create and integrate React components, and successfully render them within the application, enhancing modular development and improving code maintainability.

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

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

Quick Check

A front-end developer is frustrated with slow updates in his React project. A teammate suggests using Vite. How does Vite improve development speed?

A. It rebuilds the entire project faster.

B. It uses ES modules and Hot Module Replacement.

C. It converts JavaScript to a different language.

D. It removes the need for a development server.

React Hooks and Lifecycle Management

Hooks in React

It allows functional components to use state and lifecycle features in React, enhancing flexibility and reusability. Key features of Hooks include:

Encapsulated logic
Organizing reusable component logic for better maintainability

Top-level placement
Ensuring hooks are called only at the top level of a component

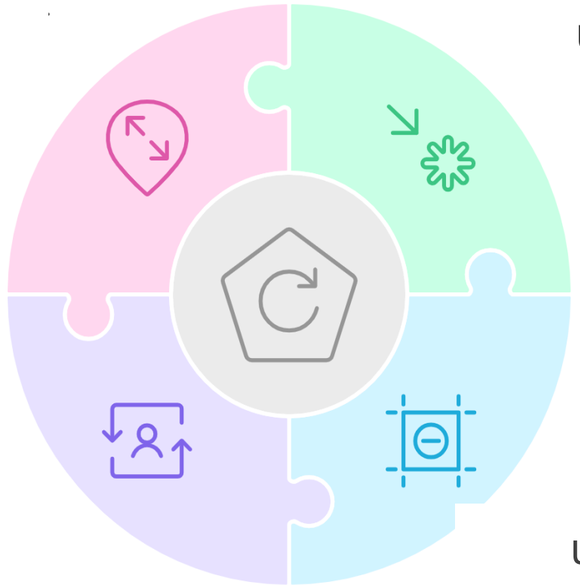
Using a consistent naming convention
Starting hook names with *use*, like *useState* and *useEffect*

Types of Hooks

React provides various hooks to manage state, handle side effects, and share data between components. Some key hooks include:

useRef

useState



useContext

useEffect

The useState Hook

It is a React Hook that manages state in functional components, allowing value updates that trigger re-renders. The main features of the useState Hook are:

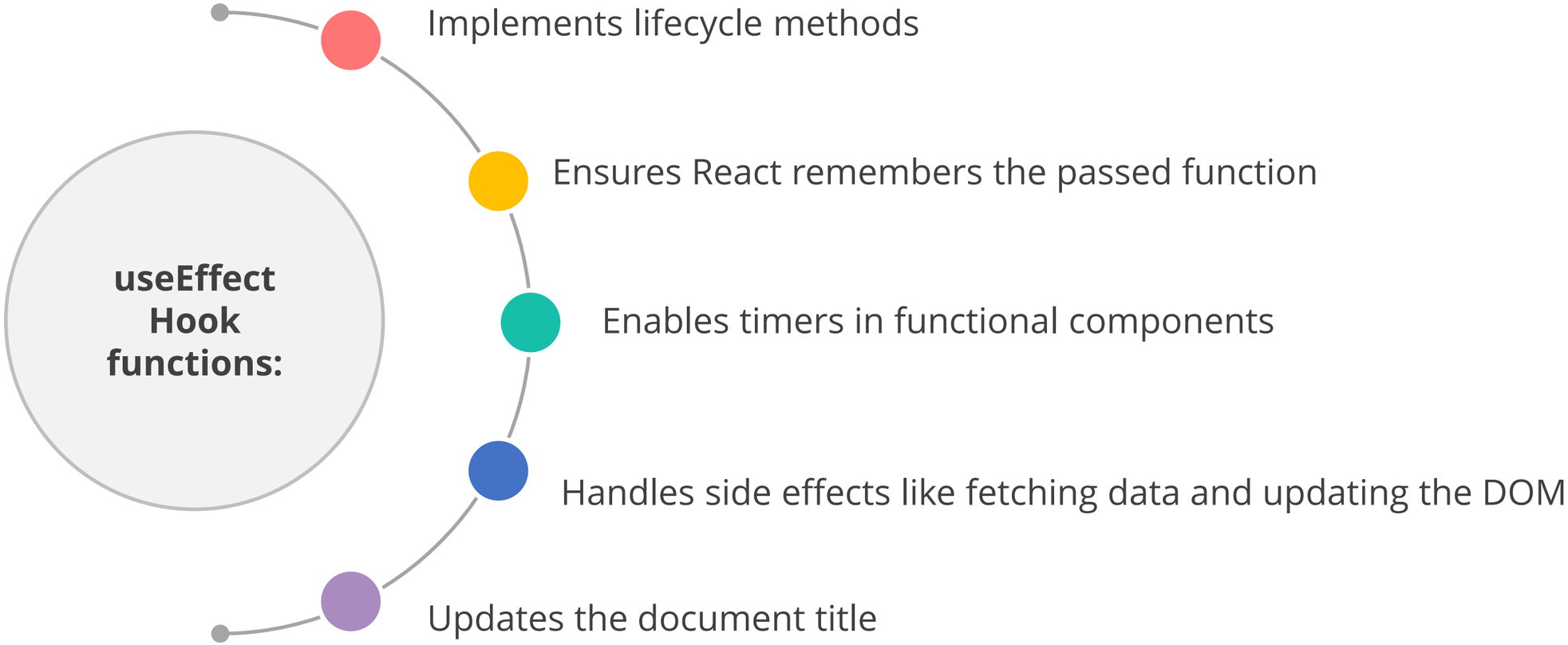
Stores state as any data type

Takes an initial value as an argument

Returns the current state and an update function

The useEffect Hook

It is a React Hook that handles side effects in functional components, such as fetching data, updating the DOM, and managing subscriptions. The main features of this hook are:

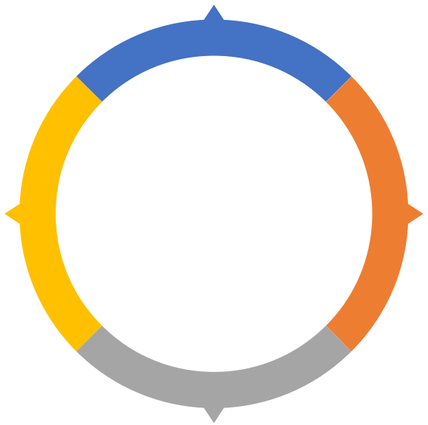


The useRef Hook

It is a React Hook that retains a reference across renders without causing re-renders, mainly for accessing DOM elements and storing mutable values. Key features include:

Preserving values
Maintains data across renders without resets

Storing mutable values
Retains data without affecting renders



Interacting with the DOM
Handles focus, animations, and element modifications

Optimizing performance
Reduces unnecessary re-renders

The useContext Hook

It is a React Hook that provides access to context values without prop drilling, enabling state and data sharing across components efficiently. Key features include:

Simplify state management
Shares global data without passing props manually

Working with Context API
Accesses values from React.createContext()

Enhancing readability
Reduces nested prop passing in deeply nested components

Quick Check

A React project needs to store user input, trigger effects when data changes, and manage global state. Which hook should be used to store component-level state?

A. useEffect

B. useContext

C. useState

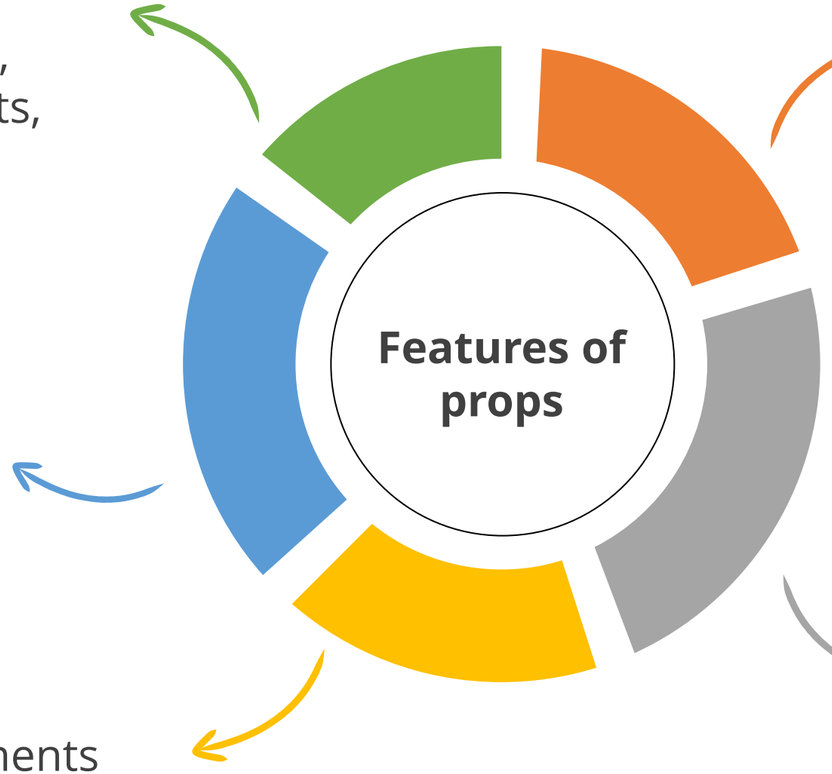
D. useRef

Props in React

They are a mechanism for passing data from a parent component to a child, allowing components to be dynamic and reusable by receiving different inputs.

Versatile
Can be strings, numbers, objects, or functions

Immutable
Cannot be modified by the child component



Customizable
Adapts components to different inputs

One-way flow
Is passed from parent to child

Reusable
Makes components modular and reusable

Props in React

This code demonstrates how props are passed as attributes to a component and how to use the props parameter:

```
// Greet.jsx
import React from 'react';
const Greet = (props) => {
  console.log(props);
  return <h1>Hello, {props.name}!</h1>;
};
export default Greet
```

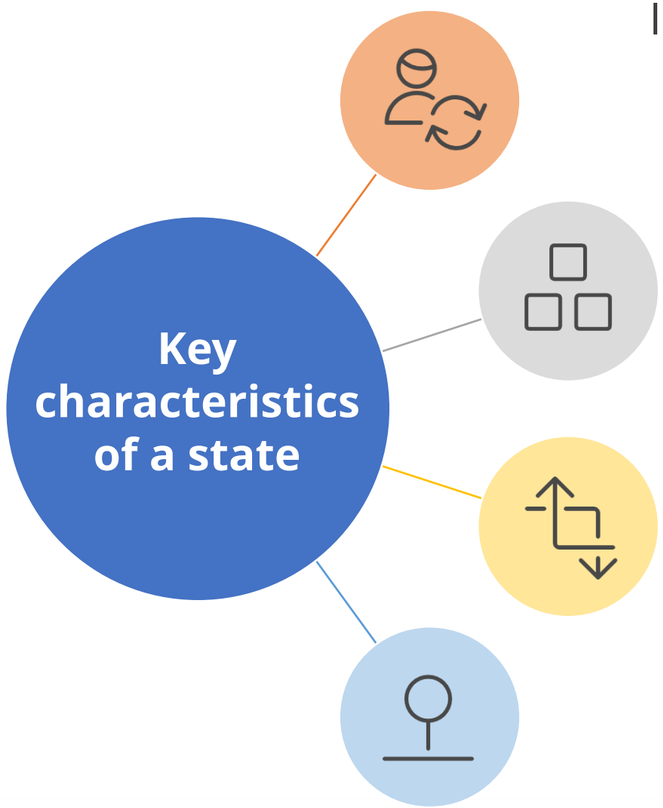
Props in React

Building on how props are passed to a component, the following example demonstrates how props enable rendering dynamic content by passing different values to the Greet component.

```
// App.jsx
import React from 'react';
import Greet from './Greet';
const App = () => {
  return (
    <div>
      <Greet name="John" />
      <Greet name="Mary" />
      <Greet name="Rock" />
    </div>
  );
};
export default App;
```

State in React

It is an object that holds data that can change over time and influence a component's behavior and rendering.



Mutable
Is updated using setState or useState

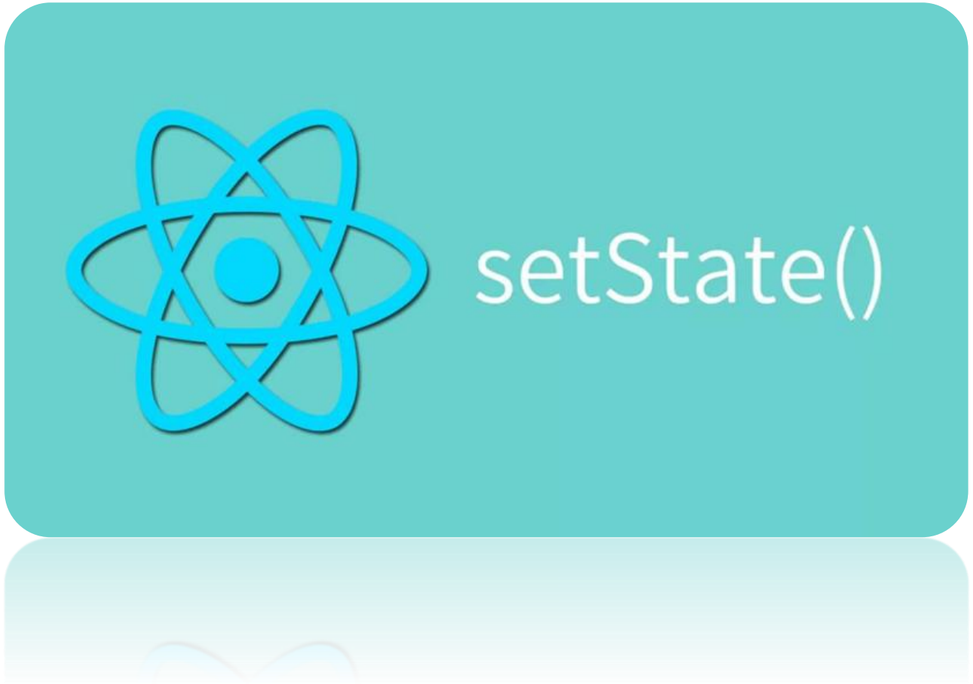
Component-specific
Is managed within its own component

Prone to change
Triggers a re-render to reflect the changes when updated

Local scope
Exists only within the component

setState in React

In class components, setState serves as the primary method for updating state and triggering re-renders. It operates asynchronously and merges the new state with the existing state.



When setState is called in React, it:

- Combines the provided object with the existing state
- Modifies only the specified properties

Source: https://www.poudelanil.com.np/

setState in React: Example

Scenario: Change the truck's color from red to blue by clicking a mouse button
Solution: To achieve this, utilize the onClick event, which triggers when the mouse button is pressed.

```
class Truck extends React.Component {
constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Pickup",
      color: "red",
      year: 1964,
    };
  }
color_change = () => {
this.setState({ color: "blue" });
  };
```

setState in React: Example

In this case, the render() method of the Truck component returns a JSX structure that defines the component's visual output.

```
render() {
  return (
  <div>
     <h1>My {this.state.brand}</h1>
     <p>
        It is a {this.state.color}
        {this.state.model}
        from {this.state.year}
```

```
</p>
      <button
         type="button"
          onClick={this.changeColor}
      >Change color</button>
   </div>
   );
  }

 }
```

Destructure Props and State

In React, destructuring props in the state refers to extracting specific properties from the props object.

Consider an employee object with the given properties:

```
const employee = {
  first_name: “John",
  last_name: “Doe",
  email_Id:
“john@gmail.com"
 }
```

Before ES6, each property had to be accessed individually using dot or bracket notation.

```
console.log(employee.first_name) // John

console.log(employee.last_name) // Doe

console.log(employee.email_Id) // john@gmail.com
```

Destructure Props and State

Destructuring props and state properties efficiently extracts and assigns values concisely. The following example illustrates this concept:

Streamlining code

Destructuring simplifies variable extraction:

```
const { first_name, last_name, email_Id } = employee;
```

Traditional approach

Properties are assigned individually:

```
const first_name = employee.first_name
const last_name = employee.last_name
const email_Id= employee.email_Id
```

Accessing values

Extracted properties can now be used directly:

```
console.log(first_name)
// John
console.log(last_name)
// Doe
console.log(email_Id) // john@gmail.com
```

Destructure Props and State: Example

The provided code defines a functional React component named Employee, which is responsible for displaying employee details:

```
import React from 'react'

export const Employee = props => {

return (

   <div>

         <h1> Employee Details</h1>
         <h2> First Name :
{props.first_name} </h2>
```

```
<h2> Last Name : {props.last_name} </h2>

         <h2> Email Id : {props.email_Id}
</h2>

   </div>

)

 }
//Note that the employee data can be
accessed in the Employee component using
props.
```

Destructure Props and State: Example

This code represents the main entry point of a React application. The Employee component receives props for **firstName**, **lastName**, and **emailId**.

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
import Table from
'./components/functional-
components/Table';
import PropsDemo from
'./components/props/PropsDemo';
import { Employee } from
'./components/Employee';
function App() {
```

```
return (
  <div className="App">
    <header className="App-header">
     <Employee first_name = “John"
last_name = “Doe" email_Id =
“john@gmail.com" />
    </header>
  </div>
  );
}
 export default App;
```

Assisted Practice

Creating a React Application Using Event Handler **Duration: 15 Min.**

Problem statement:

You have been tasked with developing a React application that utilizes event handlers to manage user interactions efficiently. The goal is to create a React component that binds its event handler context and ensures proper event handling using Vite for optimized performance.

Outcome:

By the end of this task, you will be able to set up a React project using Vite, create a component with an event handler, and implement event-driven interactions within a React application.

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

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 the component
4. Run and test the application

Assisted Practice

Creating a React App with Props and State Using Vite **Duration: 15 Min.**

Problem statement:

You have been tasked with building a React application using Vite that demonstrates the concepts of props and state. The goal is to create reusable components, pass dynamic data using props, and manage state effectively to update the UI based on user interactions.

Outcome:

By the end of this task, you will be able to set up a React project using Vite, create reusable components, pass data via props, and manage state using the useState hook to build dynamic applications.

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

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

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

Quick Check

A React developer is working with an object and needs to extract multiple properties without using dot notation repeatedly. What feature of ES6 should be used?

A. Template literals

B. Destructuring

C. Spread operator

D. Arrow functions

Handling Events in React

Event Handling in React

Event handling involves capturing and responding to user actions or system-generated events within a component. The typical steps for handling events in React are:

Attach the event handler to the JSX element

Define an event handler function

Implement the event handler logic

Event Handling vs. HTML

React and HTML event handling share similarities but differ in implementation and behavior. Below are some key differences:

Feature	React event handling	HTML event handling
Event naming	Uses camelCase (for example, onClick, onChange)	Uses lowercase (for example, onclick, onchange)
Event handler syntax	Assigned as a function reference (onClick={buttonClicked})	Assigned as a string (onclick="buttonClicked()")
Event attachment	Passed as props to components, allowing better reuse	Directly defined as an attribute on the HTML element

Event Handling vs. HTML

In React, preventing default behavior differs from traditional HTML. Instead of using return false, React requires calling **event.preventDefault()** explicitly.

The following is an example demonstrating this:

HTML

```
//HTML
<button onclick="console.log('Button
Clicked.'); return false">
 Click Me
 </button>
```

React

```
// React
Function handleClick (e) {
    e.preventDefault ();
    console.log('Button clicked.');
}
Return (
     <button onClick={handleClick}>
         Click me
     </a>
   );
```

Event Handling in Functional Component

It involves defining a function that executes when a specific event, such as a button click, occurs.

The following example illustrates this behavior:

```
import React from 'react'
function FunctionClick() {
  function clickHandler() {
     console.log('Button clicked')
     }
```

```
return (
     <div>
        <button
onClick={clickHandler}>Click</button>
     </div>
     )
}
export default FunctionClick;
```

Event Handling in Class Component

It involves defining a method that executes when a specific event, such as a button click, occurs.

The following example illustrates this behavior:

```
import React, { Component } from 'react'
class ClassClick extends Component {
  clickHandler() {
  console.log('Clicked the button')
  }
 render() {
return (
     <div>
     <button onClick={this.clickHandler}>Click Me</button>
     </div>
  )
  }
}
export default ClassClick
```

Binding Event Handlers

Event binding in React ensures that class methods retain access to the component instance (this) when handling events. The following example illustrates this behavior:

```
import React, { Component } from 'react'
class EventBind extends Component {
  constructor() {
   super()
   this.state = {
   message: 'Hello'
   }
  }
```

When handling events in class components, proper binding of this is necessary to access the component's instance and its properties

Binding Event Handlers

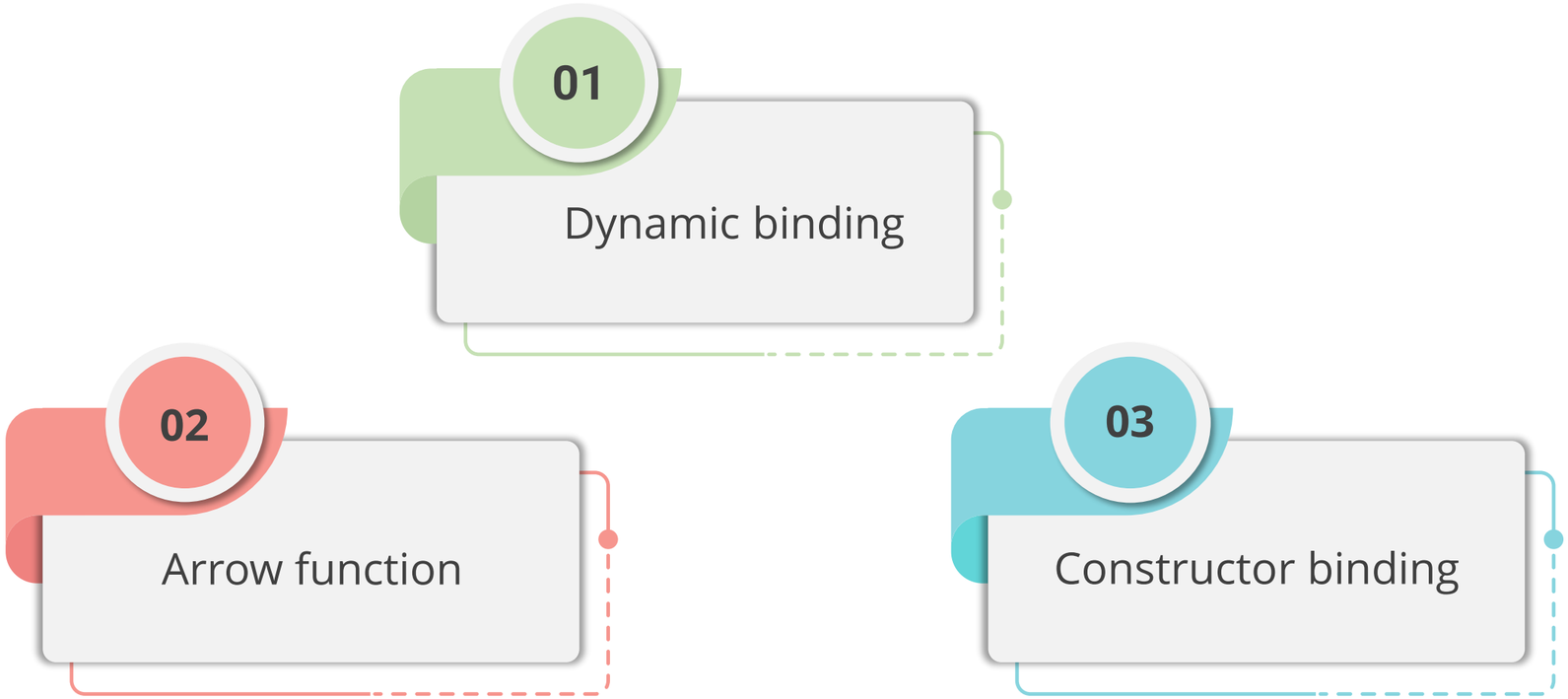
In class components, failing to bind event handler methods can result in this being undefined, leading to a TypeError. The following example demonstrates this issue:

```
 clickHandler() {
   console.log(this)
   this.setState({message:
'Goodbye'})
  }
  render() {
   return (
   <div>
```

```
<div>{this.state.message}</div>
      <button
onClick={this.clickHandler}>Click</b
utton>
   </div>
   )
  }
}
 export default EventBind
```

Methods for Binding Event Handlers

React provides three methods for binding event handlers in class components, which are as follows:



Dynamic Binding

It connects a function to its execution context (this) at runtime, enabling flexible and late binding.
The following example demonstrates this issue:

```
import React, { Component } from
'react'
class EventBind extends Component {
  constructor() {
   super()
   this.state = {
   message: 'Hello'
   }
  }
```

The code in the next slide demonstrates how dynamic binding affects event handlers and state updates in React.

Dynamic Binding

```
clickHandler = () => {
    this.setState({message:'Goodbye'})
  }
  render() {
   return (
   <div>
        <div>{this.state.message}</div>
      <button onClick={this.clickHandler}>Click</button>
   </div>
   )
  }
}
```

Updating this.state.message triggers a re-render, creating a new event handler that differs from the one used in the initial render() call.

Arrow Functions

It automatically captures the **this** context of a component, eliminating the need for explicit binding in event handlers. The following example demonstrates this behavior:

```
import React, { Component } from 'react'
class EventBind extends Component {
  constructor() {
   super()
   this.state = {
   message: 'Hello'
   }
   this.clickHandler =
this.clickHandler.bind(this)
  }
 clickHandler() {
   console.log(this)
this.setState({message: 'Goodbye'})
  }
  render() {
   return (
   <div>
        <div>{this.state.message}</div>
      <button
onClick={this.clickHandler}>Click</button>
   </div>
   )
  }
}
 export default EventBind
```

Constructor Binding

It ensures class methods maintain the correct **this** context by explicitly binding them. This prevents **this** from being undefined in event handlers. The following example demonstrates this behavior:

```
import React, { Component } from 'react'
class EventBind extends Component {
  constructor() {
   super()
   this.state = {
   message: 'Hello'
   }
   this.clickHandler = this.clickHandler.bind(this)
  }
 clickHandler() {
   console.log(this)
```

The code in the next slide demonstrates how constructor binding affects event handlers and state updates in React.

Constructor Binding

```
this.setState({message: 'Goodbye'})
  }
  render() {
   return (
   <div>
        <div>{this.state.message}</div>
      <button
onClick={this.clickHandler}>Click</button>
   </div>
   )
  }
}
 export default EventBind
```

Constructor binding improves performance and ensures the correct **this** context in React class components.

Conditional Rendering and Best Practices

Conditional Rendering in HTML and JavaScript

It dynamically determines which components to display based on conditions or values, enabling selective UI rendering. The primary approaches include:

Element variables

Ternary conditional operator

if/else statements

Short-circuit operator

if/else Statements

Conditional rendering evaluates specific conditions to determine which components should be rendered dynamically. The following program demonstrates conditional rendering using an If/else statement:

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

const UserGreeting = () => {
  const [isLoggedIn] = useState(false);

  if (isLoggedIn) {
    return <div>Welcome John</div>;
  } else {
    return <div>Welcome Guest</div>;
  }
};

export default UserGreeting;
```

if/else Statements

In the App.jsx file, components can be dynamically rendered based on conditional logic.

Below is the remaining part of the code:

```
import './App.css';
import UserGreeting from './components/UserGreeting';

function App() {
  return (
    <div className="App">
      <UserGreeting />
    </div>
  );
}

export default App;
```

Element Variables

It facilitates conditional rendering in JavaScript by storing JSX elements in a variable, allowing dynamic component updates based on conditions.

Example: In a functional component, use state with hooks:
If isLoggedIn is true, assign Welcome John to the message variable; otherwise, assign Welcome Guest

Element Variables

Conditional rendering with element variables enables developers to selectively display elements based on conditions. The following example demonstrates this approach:

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

function UserGreeting() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  let message;
  if (isLoggedIn) {
    message = <div>Welcome John</div>;
  } else {
    message = <div>Welcome Guest</div>;
  }

  return message;
}
export default UserGreeting;
```

Ternary Conditional Operator

It can be used within JSX by returning a conditional expression wrapped in parentheses to determine which element to display. The following code demonstrates this approach:

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

function UserGreeting() {
    const [isLoggedIn, setIsLoggedIn] = useState(false);

    return (
        <div>
            {isLoggedIn ? <div>Welcome John</div> :
<div>Welcome Guest</div>}
        </div>
    );
}

export default UserGreeting;
```

Ternary Conditional Operator

Once the code is executed, the text dynamically appears in the browser based on the evaluated condition, ensuring the appropriate message is displayed:

Short Circuit Operator

It is a concise way to perform conditional rendering, displaying an element only when a condition is true. If the condition is false, nothing is rendered. The following code demonstrates this approach:

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

function UserGreeting() {
  const [isLoggedIn, setIsLoggedIn] = useState(true);

  return isLoggedIn && <div>Welcome John</div>;
}

export default UserGreeting;
```

Short Circuit Operator

Once the code is executed, **Welcome John** dynamically appears in the browser because the condition evaluates to true. If the condition were false, no text would be displayed:

Assisted Practice

Creating a React Application Using Conditional Rendering **Duration: 15 Min.**

Problem statement:

You have been tasked with developing a React application that demonstrates conditional rendering. The goal is to create a component that dynamically updates the UI based on user interactions, utilizing the Vite framework for efficient performance.

Outcome:

By the end of this task, you will be able to set up a React project using Vite, implement conditional rendering in a component, and dynamically update the UI based on state changes.

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

Assisted Practice: Guidelines

Steps to be followed:

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

Developing a React Note-Taking App

Duration: 25 Mins.

Project agenda: The goal is to develop a simple and efficient note-taking application using React. The app will allow users to add and delete notes dynamically while demonstrating the use of props, event handlers, and conditional rendering.

Description: : You are tasked with building a React-based note-taking app to enhance usability and efficiency. This involves implementing state management, event handling for adding and deleting notes, and conditional rendering for UI feedback. The project ensures a seamless user experience with dynamic updates and an intuitive interface.

LESSON-END PROJECT

Developing a React Note-Taking App

Duration: 25 Min.

Perform the following:

1. Set up a new React project using Vite
2. Implement core components using Props
3. Modify the App.jsx file
4. Update the main.jsx file
5. Run and verify the application

Expected deliverables: A fully functional React-based note-taking application with core features, including dynamic note management using state and event handling. The application will support adding and deleting notes with UI feedback and conditional rendering. It will be structured with reusable components, efficient state updates using useState, and a clean, intuitive user interface.

LESSON-END PROJECT

Key Takeaways

- React simplifies frontend development with a component-based architecture for dynamic web apps.
- Vite boosts React development with fast setup, optimized performance, and HMR.
- JSX enables writing HTML-like syntax in JavaScript for easier component management.
- State manages dynamic data, while props enable reusable component communication.
- React uses synthetic events and bindings for efficient UI updates.

Thank You