TABLE OF CONTENTS

1. INTRODUCTION TO REACT

1.1 What is React?

React is a JavaScript library for building user interfaces. It was developed by Facebook in 2013 and is now maintained by Meta and an active open-source community.

Unlike traditional JavaScript applications, where developers directly manipulate the DOM (Document Object Model), React introduces a more efficient approach. With React, developers build components that describe what the UI should look like, and React updates the DOM automatically when data changes.

React is widely used by companies like Facebook, Instagram, Netflix, Airbnb, Uber, and many startups around the world.

1.2 Why React?

Before React, web applications often became hard to maintain as they grew larger. React solved this problem by introducing a component-based architecture.

Benefits of React:

1. Component-Based – Breaks UIs into small, reusable parts.

2. Virtual DOM – Faster rendering compared to manipulating the real DOM directly.

3. Declarative – You describe what you want, and React figures out how to do it.

4. Reusable Code – Use the same components across multiple projects.

5. Large Ecosystem – Plenty of libraries, tools, and community support.

1.3 The Virtual DOM

One of React's biggest innovations is the Virtual DOM.

1. The Virtual DOM is a lightweight copy of the real DOM.
2. When something changes, React updates the Virtual DOM first.
3. It then compares the Virtual DOM with the real DOM using a process called reconciliation.
4. Only the differences (called a diff) are applied to the real DOM.

This makes React fast and efficient.

1.4 React vs Other Frameworks

React vs Angular: React is a library (flexible, lightweight), while Angular is a full framework (batteries included but heavier).

React vs Vue: Vue is simpler for small projects but React has a larger ecosystem and more job demand.

1.5 Real-World Examples of React

React powers many popular apps:

1. Facebook – Built and maintained by Meta.

2. Instagram – UI components and stories are powered by React.
3. Netflix – Optimized streaming UI.
4. Airbnb – Reusable components for booking system.
5. WhatsApp Web – Chat interface built with React.

---

2. SETTING UP THE DEVELOPMENT ENVIRONMENT

2.1 Introduction

Before building React applications, you need to prepare your development environment. React requires Node.js and a package manager (npm or yarn) to manage dependencies and run development servers. You will also need a good code editor and some browser tools.

2.2 Steps to Create a new React Environment

1. Open a terminal.

2. Run the following command:

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

3. Move into the folder:

cd my-app

4. Install dependencies:

npm install

5. Start the development server:

npm run dev

6. Open the URL (**http://localhost:5173**) in your browser to see the app.

2.3 Editing and Running the App

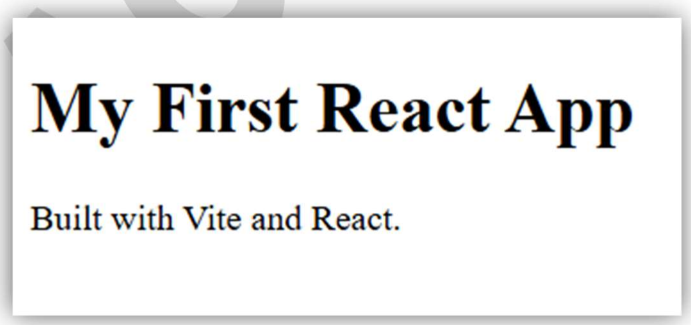
1. Open **App.jsx**.
2. Replace the default content with your own.

   Example:

```
function App() {
  return (
    <div>
      <h1>My First React App</h1>
      <p>Built with Vite and React.</p>
    </div>
  )};
export default App;
```

3. Save the file.
4. The browser will automatically reload and display the new content.

Output:



My First React App

Built with Vite and React.

3. JSX AND RENDERING

3.1 Introduction

JSX (JavaScript XML) is a syntax extension that allows you to write HTML-like code inside JavaScript. It makes React code easier to read and write. JSX is not HTML – it gets compiled into JavaScript under the hood. In this chapter, you will learn how JSX works, how rendering happens in React, and the rules you must follow.

3.2 What is JSX?

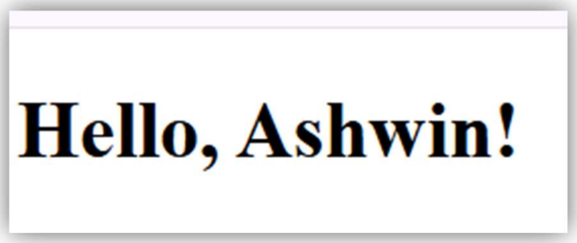
JSX lets you combine HTML-like syntax with JavaScript. This means you can embed variables, functions, and expressions directly inside your UI code.

Example:

App.jsx:

```
const name = "Ashwin";
const element = <h1>Hello, {name}!</h1>;
function App() {
  return (
    <div>{element}</div>
  )};
export default App;
```

Output:


Hello, Ashwin!

3.3 Rendering Elements

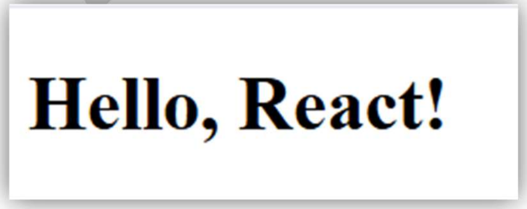
React elements are the smallest building blocks of React apps. They are plain JavaScript objects that describe what should appear on the screen.

Example:

App.jsx:

```
const element = <h1>Hello, React!</h1>;

function App() {

        return (

          <div>{element}</div>)};

export default App;
```

Output:


Hello, React!

3.4 Rules of JSX

When writing JSX, keep these rules in mind:

1. JSX must return a single root element.
   Example (incorrect):

   ```
   return <h1>Hello</h1><p>World</p>;
   ```

   Example (correct):

   ```
   return (
     <div>
       <h1>Hello</h1>
       <p>World</p>
     </div>
   );
   ```

2. Use className instead of class.

   ```
   <p className="intro">Welcome</p>
   ```

3. All tags must be closed.

   ```
   <img src="logo.png" />
   ```

4. Use {} to insert JavaScript expressions.

   ```
   <h2>{2 + 2}</h2>
   ```

3.5 Fragments

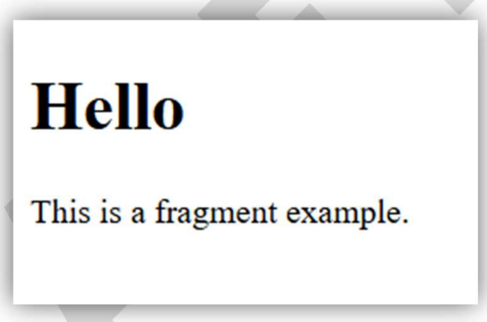
Sometimes you don't want to wrap elements in an extra <div>. In that case, use **fragments**.

Example:

App.jsx

```
function App() {

return (<>

   <h1>Hello</h1>

   <p>This is a fragment example. </p>

  </>)}

export default App;
```

Output:



```
Hello

This is a fragment example.
```

3.6 Dynamic Rendering with Expressions

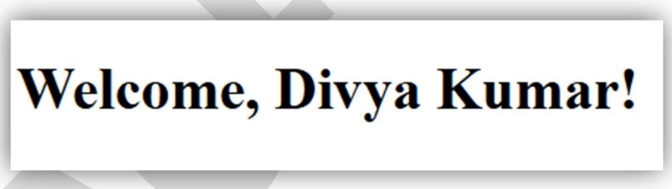
JSX allows embedding functions and variables inside curly braces.

Example:

App.jsx:

```
const user = { firstName: "Divya", lastName: "Kumar" };

function getFullName(user) {

  return user.firstName + " " + user.lastName;

}

function App() {

        return (

            <div><h1>Welcome, {getFullName(user)}!</h1></div>

)};

export default App;
```

Output:


Welcome, Divya Kumar!

4. COMPONENTS & PROPS

4.1 Introduction

In React, everything is built with **components**. A component is a small, reusable piece of UI that can be combined with other components to build complete applications. Think of components as Lego blocks – you can use them separately or combine them to create something bigger.

4.2 Types of Components

React supports two main types of components:

1. **Functional Components** – These are plain JavaScript functions that return JSX. They are the most common and modern way to write components.
   Example:
   ```
   function Greeting() {
     return <h1>Hello, React!</h1>;
   }
   ```
2. **Class Components** – These were used in older versions of React. They are less common today but still supported.

   Example:
   ```
   class Greeting extends React.Component {
     render() {
       return <h1>Hello, React!</h1>;  }}
   ```

For new projects, you should always use **functional components**.

4.3 Creating and Using Components

Function Component Examples:

Example 1:

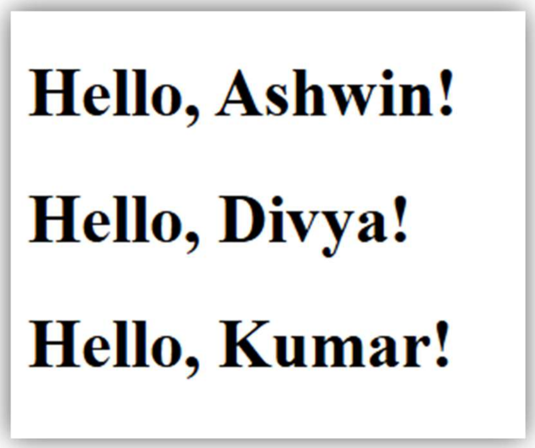
Hello.jsx:

```
function Hello(props) {
  return <h1>Hello, {props.name}!</h1>;
}
export default Hello;
```

App.jsx:

```
import Hello from "./Hello";
function App() {
  return (
    <div>
      <Hello name="Ashwin" />
      <Hello name="Divya" />
      <Hello name="Kumar" />
    </div>  )}
export default App;
```

Output:



```
Hello, Ashwin!
Hello, Divya!
Hello, Kumar!
```

Example 2:

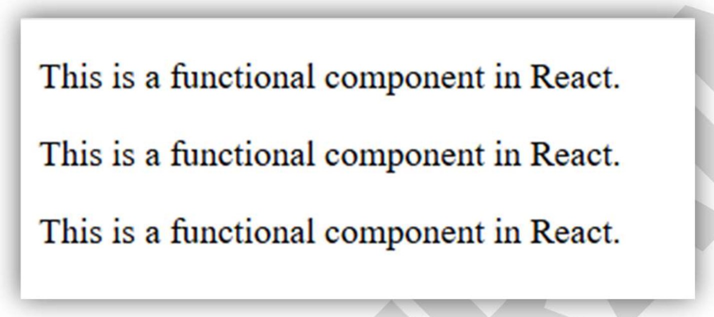
Info.jsx:

```
function Info() {
  return <p>This is a functional component in React.</p>;
}
export default Info;
```

App.jsx:

```
import Info from "./Info";
function App() {
  return (
    <div>
      <Info />
    <Info />
    <Info />
   </div>
 )}
export default App;
```

Output:



```
This is a functional component in React.
This is a functional component in React.
This is a functional component in React.
```

Class Component Examples:

Example 1:

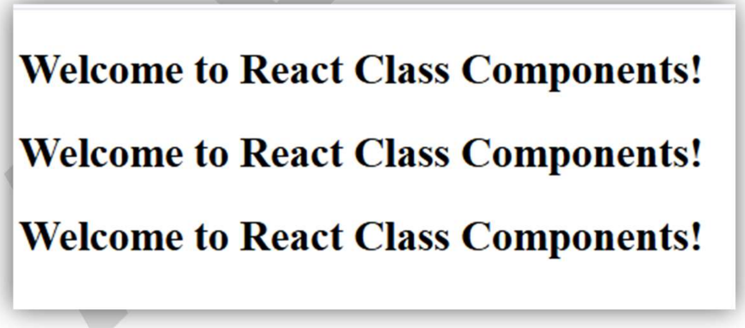
WelcomeClass.jsx:

```
import React, { Component } from "react";
class WelcomeClass extends Component {
 render() {
   return <h2>Welcome to React Class Components!</h2>;}}
export default WelcomeClass;
```

App.jsx:

```
import WelcomeClass from "./WelcomeClass";

function App() {

  return (

    <div>

      <WelcomeClass />

      <WelcomeClass />

      <WelcomeClass />

    </div>

  )}

export default App;
```

Output:



```
Welcome to React Class Components!

Welcome to React Class Components!

Welcome to React Class Components!
```

Example 2:

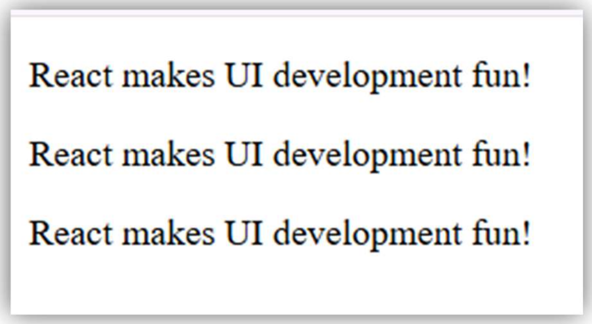
MessageClass.jsx:

```
import React, { Component } from "react";
class MessageClass extends Component {
  render() {
    return <p>React makes UI development fun!</p>;  }
}
export default MessageClass;
```

App.jsx:

```
import MessageClass from "./MessageClass";
function App() {
  return (
    <div>
      <MessageClass />
      <MessageClass />
      <MessageClass />
    </div>
  );}
export default App;
```

Output:



```
React makes UI development fun!

React makes UI development fun!

React makes UI development fun!
```

4.4 Props – Passing Data to Components

Props (short for properties) allow you to pass data from a parent component to a child component.

Example 1:

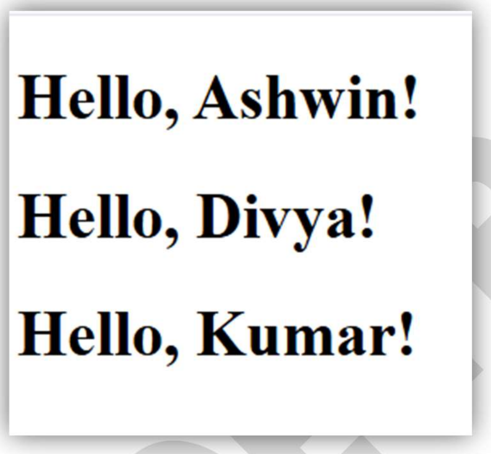
Greeting.jsx:

```
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}
export default Greeting;
```

App.jsx:

```
import Greeting from "./Greeting";
function App() {
  return (
  <div>
    <Greeting name="Ashwin" />
    <Greeting name="Divya" />
    <Greeting name="Kumar" />
  </div>)}
export default App;
```

Output:



```
Hello, Ashwin!
Hello, Divya!
Hello, Kumar!
```

Example 2:

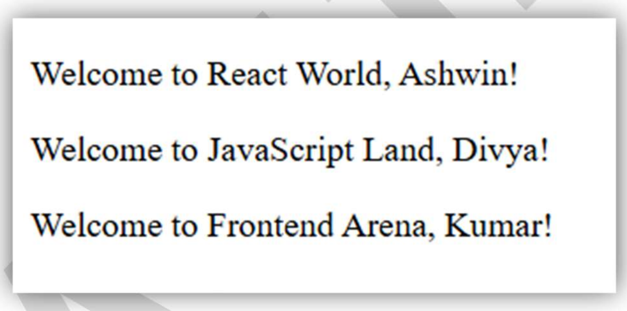
Welcome.jsx:

```
function Welcome(props) {
  return <p>Welcome to {props.place}, {props.user}!</p>;
}
export default Welcome;
```

App.jsx:

```
import Welcome from "./Welcome";

function App() {

 return (

   <div>

     <Welcome user="Ashwin" place="React World" />

     <Welcome user="Divya" place="JavaScript Land" />

     <Welcome user="Kumar" place="Frontend Arena" />

   </div>)}

export default App;
```

Output:



```
Welcome to React World, Ashwin!

Welcome to JavaScript Land, Divya!

Welcome to Frontend Arena, Kumar!
```

5. STATE AND LIFECYCLE

5.1 Introduction

Components often need to manage data that changes over time. In React, this is handled using **state**. State allows a component to "remember" information between renders and update the UI when the data changes. React also provides lifecycle behavior, which lets you run code when a component mounts, updates, or unmounts.

5.2 What is State?

State is a built-in object in React components that is used to store data. Unlike props, which are passed from parent to child, state is managed inside the component itself.

Example:

Counter.jsx:

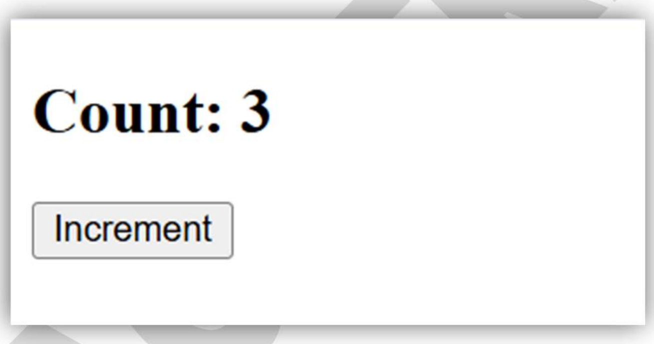
```
import { useState } from "react";

function Counter() {

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

  return (

    <div>

      <h2>Count: {count}</h2>

      <button onClick={() => setCount(count + 1)}>Increment</button>
```

```
   </div>
  )}
export default Counter;
```

App.jsx:

```
import Counter from "./Counter"
function App() {
  return (<Counter />)}
export default App;
```

Output:



```
Count: 3
```

5.3 Updating State

The setState function (or setCount in the above example) is used to update state.

Important: Never modify state directly. Always use the state updater function.

Incorrect:

count = count + 1; // ❌ This won’t update the UI

Correct:

setCount(count + 1); // ✅ Correct way

5.4 Multiple State Variables

You can use multiple useState hooks to manage different pieces of state.

Example:

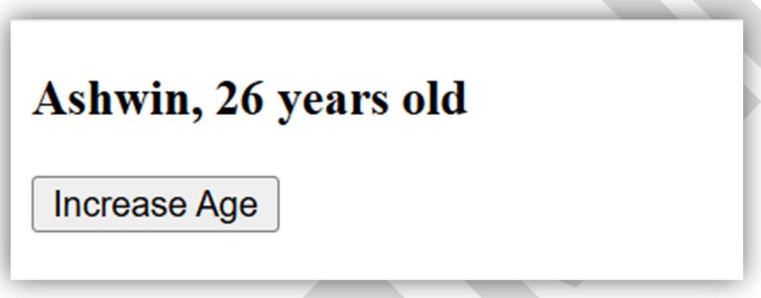
Profile.jsx:

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

function Profile() {

  const [name, setName] = useState("Ashwin");

  const [age, setAge] = useState(22);

  return (

    <div>

      <h3>{name}, {age} years old</h3>

      <button onClick={() => setAge(age + 1)}>Increase Age</button>

    </div>

  );

}

export default Profile;
```

App.jsx:

```
import Profile from "./Profile"

function App() {

  return (<Profile />)

}

export default App;
```

Output:


Ashwin, 26 years old

5.5 Lifecycle with useEffect

React's useEffect hook lets you perform side effects (such as fetching data, updating the DOM, or setting timers).

Runs after the component renders.

Can be configured to run only once or when specific values change.

Example:

Timer.jsx:

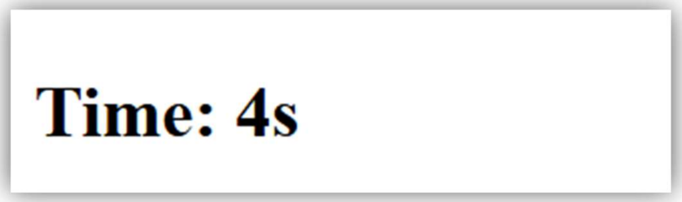
```
import { useState, useEffect } from "react";
```

```
function Timer() {
  const [seconds, setSeconds] = useState(0);
  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds(prev => prev + 1);
    }, 1000);

    return () => clearInterval(interval);
  }, []);
  return <h2>Time: {seconds}s</h2>;
}
export default Timer;
```

App.jsx:

```
import Timer from "./Timer"
function App() {
  return (<Timer />);
}
export default App;
```

Output:


```
Time: 4s
```

5.6 Lifecycle Phases

React components go through three lifecycle phases:

Mounting – When the component is added to the DOM.

Example: fetching initial data.

Updating – When state or props change.

Example: re-rendering when a counter changes.

Unmounting – When the component is removed.

Example: clearing timers or unsubscribing.

---

6. HANDLING EVENTS & FORMS

6.1 Introduction

React allows you to handle user interactions such as clicks, typing, and form submissions using **event handlers**. Forms are an essential part of web applications, and React provides a way to manage input fields through controlled and uncontrolled components.

6.2 Handling Events

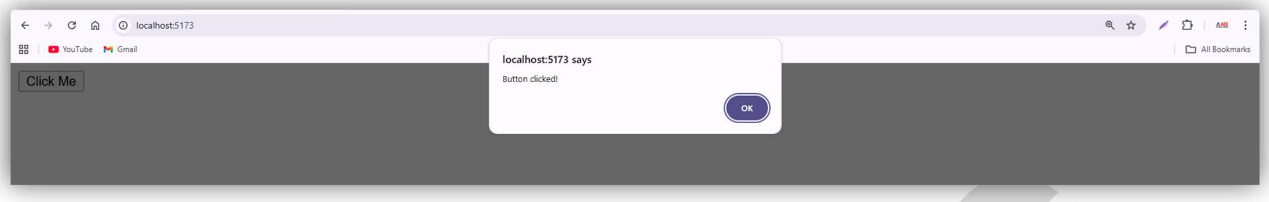
In React, events are written in **camelCase** and passed as functions.

Example:

App.jsx:

```
function App() {
 function handleClick() {
  alert("Button clicked!");
 }
 return (
  <div><button onClick={handleClick}>Click Me</button></div>
 );
}
export default App;
```

Output:



localhost:5173 says

Button clicked!

6.3 Passing Arguments to Event Handlers

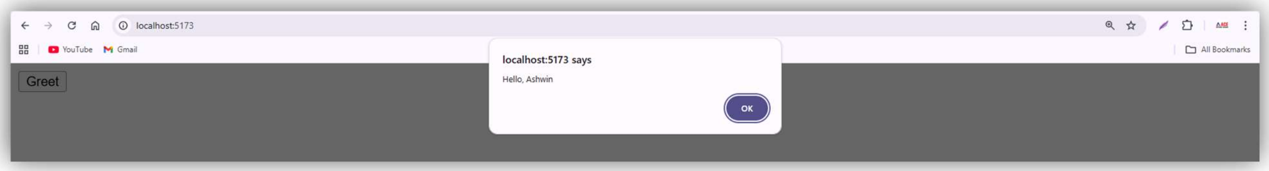
You can pass arguments by wrapping the handler in an arrow function.

Example:

App.jsx:

```
function App() {
 function greetUser(name) {
  alert("Hello, " + name);
 }
 return (
  <div>
   <button onClick={() => greetUser("Ashwin")}>Greet</button>
  </div>
 );
}
export default App;
```

Output:



localhost:5173 says

Hello, Ashwin

6.4 Handling Forms

Forms are used to collect user input. React manages form inputs in two ways:

Controlled Components – The input value is managed by React state.

Uncontrolled Components – The input value is managed by the DOM directly using ref.

6.5 Controlled Components

In controlled components, the input value is linked to state.

Example:

FormExample.jsx:

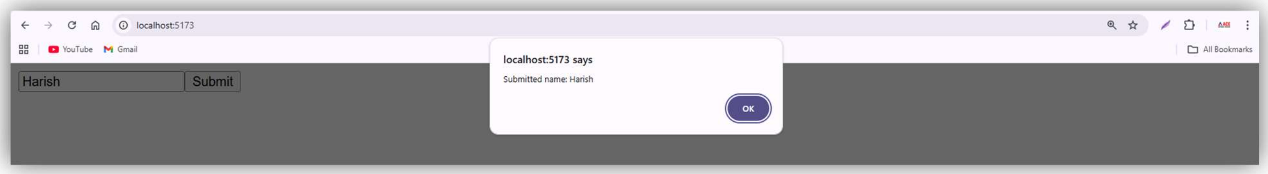
```
import { useState } from "react";

function FormExample() {

 const [name, setName] = useState("");

 function handleSubmit(event) {

   event.preventDefault();

   alert("Submitted name: " + name)

}
```

```
 return (
  <form onSubmit={handleSubmit}>
   <input
    type="text"
    value={name}
    onChange={(e) => setName(e.target.value)}
    placeholder="Enter your name" />
   <button type="submit">Submit</button>
  </form>
 )}
export default FormExample;
```

App.jsx:

```
import FormExample from "./FormExample"
function App() {
 return (<FormExample />);
}
export default App;
```

Output:



localhost:5173 says
Submitted name: Harish

6.6 Uncontrolled Components

Uncontrolled components use refs to access form values directly.

Example:

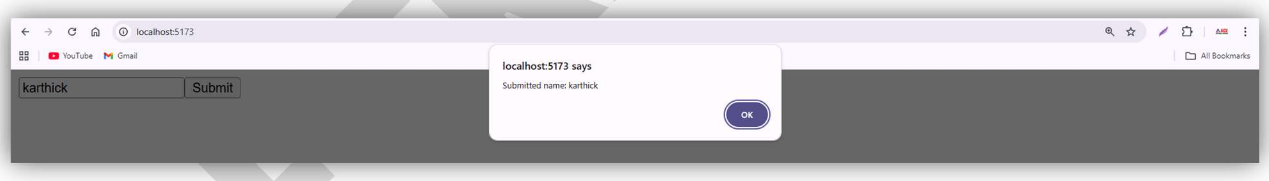
UncontrolledForm.jsx:

```
import { useRef } from "react";

function UncontrolledForm() {

  const inputRef = useRef();

  function handleSubmit(event) {

    event.preventDefault();

    alert("Submitted name: " + inputRef.current.value);

  }

  return (

    <form onSubmit={handleSubmit}>

      <input type="text" ref={inputRef} placeholder="Enter your name" />

      <button type="submit">Submit</button>

    </form>
  )}
export default UncontrolledForm;
```

App.jsx:

```
import UncontrolledForm from "./UncontrolledForm"
function App() {
  return (<UncontrolledForm />);
}
export default App;
```

Output:



```
localhost:5173 says
Submitted name: karthick
```

6.7 Multiple Inputs in Forms

You can handle multiple inputs using one state object.

Example:

MultiForm.jsx:

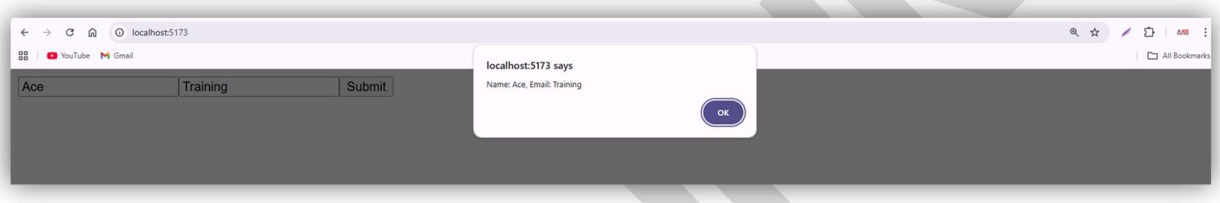
```
import { useState } from "react";
function MultiForm() {
 const [formData, setFormData] = useState({ name: "", email: "" });

 function handleChange(e) {

  const { name, value } = e.target;

  setFormData({ ...formData, [name]: value });

 }

 function handleSubmit(e) {

  e.preventDefault();

  alert(`Name: ${formData.name}, Email: ${formData.email}`);

 }

 return (

  <form onSubmit={handleSubmit}>

   <input name="name" onChange={handleChange}
value={formData.name} />

   <input name="email" onChange={handleChange}
value={formData.email} />

   <button type="submit">Submit</button>

  </form>

 )}

export default MultiForm;
```

App.jsx:

```
import MultiForm from "./MultiForm"
```

```
function App() {
  return (<MultiForm />)
}
export default App;
```

Output:



localhost:5173 says

Name: Ace, Email: Training

7. LISTS, KEYS, AND CONDITIONAL RENDERING

7.1 Introduction

In most applications, you will need to display lists of data (such as products, users, or messages) and control what gets shown based on certain conditions. React provides tools to handle these tasks efficiently using **lists, keys, and conditional rendering**.

7.2 Rendering Lists

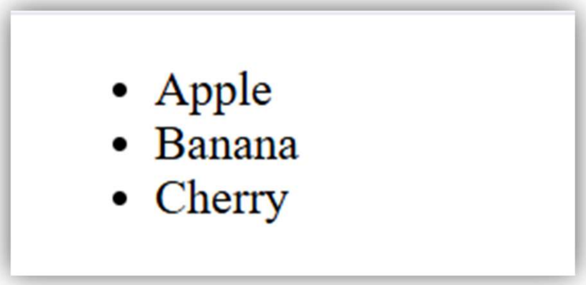
You can render a list of items in React using JavaScript's `map()` function.

Example:

App.jsx :

```
function App() {
 const fruits = ["Apple", "Banana", "Cherry"];
 return (
  <ul>
   {fruits.map((fruit, index) => (
    <li key={index}>{fruit}</li>
   ))}
  </ul>  )}
export default App;
```

Output:



```
• Apple
• Banana
• Cherry
```

7.3 The Role of Keys

Keys help React identify which items have changed, been added, or removed.

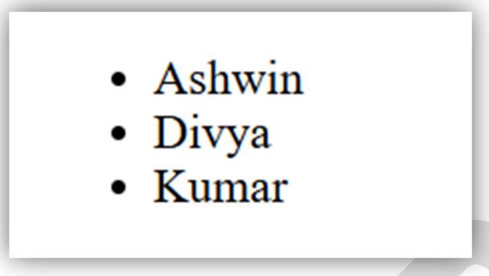
- Keys must be **unique** among siblings.
- Do not use indexes as keys if the list may change.

Example:

App.jsx:

```
const users = [
  { id: 1, name: "Ashwin" },
  { id: 2, name: "Divya" },
  { id: 3, name: "Kumar" }
];
function App() {
  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  )}
export default App;
```

Output:



```
• Ashwin
• Divya
• Kumar
```

7.4 Conditional Rendering

Conditional rendering means showing UI elements only if a certain condition is true.

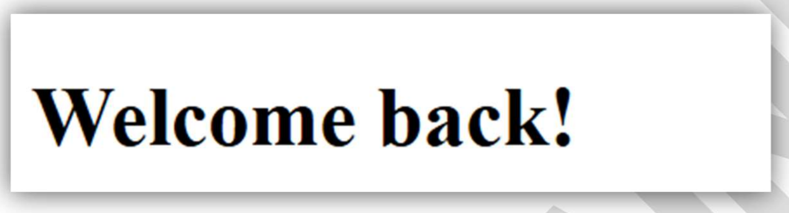
1. **Using if statements**

App.jsx:

```
function App() {
  const isLoggedIn = true;
  if (isLoggedIn) {
    return <h1>Welcome back!</h1>;
```

```
 } else {
   return <h1>Please log in.</h1>;
 }}
export default App;
```

Output:

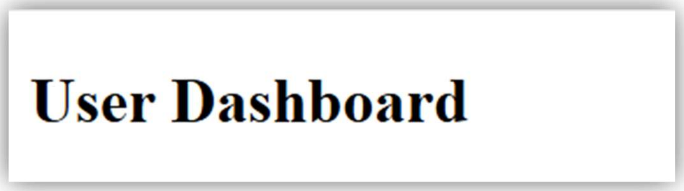

Welcome back!

2. Using ternary operator

App.jsx:

```
function App() {
 const isAdmin = false;
 return (
  <div>
   {isAdmin ? <h2>Admin Dashboard</h2> : <h2>User Dashboard</h2>}
  </div>
 )}
export default App;
```

Output:


User Dashboard

7.5 Rendering Lists with Conditions

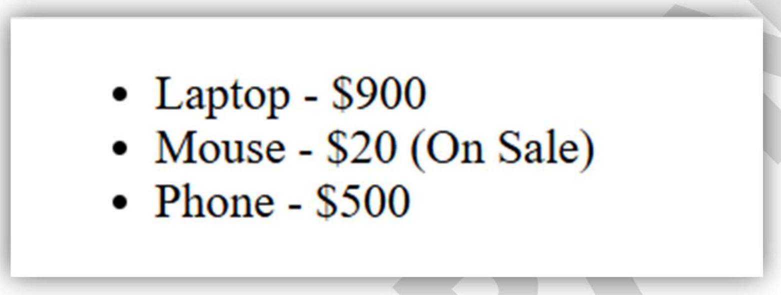
You can combine lists with conditions for more complex rendering.

Example:

App.jsx:

```
const products = [
  { id: 1, name: "Laptop", price: 900 },
  { id: 2, name: "Mouse", price: 20 },
  { id: 3, name: "Phone", price: 500 }
];
function App() {
  return (
    <ul>
      {products.map(product => (
        <li key={product.id}>
          {product.name} - ${product.price}
          {product.price < 100 && <span> (On Sale)</span>}
      </li>
    ))}
  </ul>
 )}
export default App;
```

Output:



```
• Laptop - $900
• Mouse - $20 (On Sale)
• Phone - $500
```

8. HOOKS

8.1 Introduction

Hooks are special functions in React that let you "hook into" React features such as state and lifecycle methods without using class components. They were introduced in React 16.8 and are now the standard way of building components.

The most used hooks are:

1. useState
2. useEffect
3. useRef
4. useMemo
5. useCallback
6. useContext
7. useReducer

8.2 useState – Managing State

The `useState` hook allows you to add state to functional components.

Example 1:

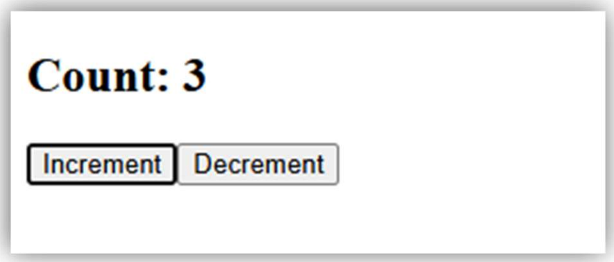
Counter.jsx:

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

function Counter() {

  const [count, setCount] = useState(0);
  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
    </div>
  );
}
export default Counter;
```

App.jsx:

```
import Counter from "./Counter";
function App() {
  return (
    <div>
      < Counter />
    </div>
  )}
export default App;
```

Output:


Count: 3

Example 2:

FormInput.jsx:

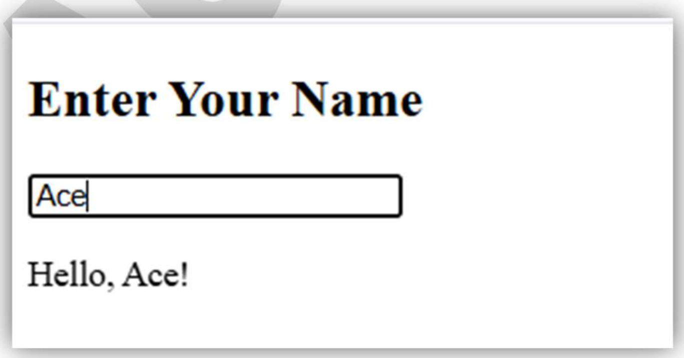
```
import { useState } from "react";

function FormInput() {

 const [name, setName] = useState("");

 return (

  <div>

   <h2>Enter Your Name</h2>

   <input

    type="text"

    value={name}

    onChange={(e) => setName(e.target.value)}

    placeholder="Type your name"

   />

   <p>Hello, {name || "Guest"}!</p>
```

```
    </div>
  )}
export default FormInput;
```

App.jsx:

```
import FormInput from "./FormInput";
function App() {
  return (
    <div>
      <FormInput />
    </div>
  )}
export default App;
```

Output:



```
Enter Your Name

Ace

Hello, Ace!
```

8.3 useEffect – Side Effects and Lifecycle

The useEffect hook lets you perform side effects such as fetching data, setting timers, or updating the DOM.

Example 1:

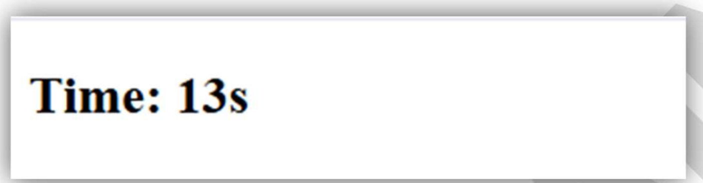
Timer.jsx:

```
import { useState, useEffect } from "react";
function Timer() {
  const [seconds, setSeconds] = useState(0);
  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds(prev => prev + 1);
    }, 1000);
    return () => clearInterval(interval); // cleanup
  }, []);
  return <h2>Time: {seconds}s</h2>;
}
export default Timer;
```

App.jsx:

```
import Timer from "./Timer";
function App() {
  return (
    <div>
      <Timer />
    </div>
  )}
export default App;
```

Output:



```
Time: 13s
```

Example 2:

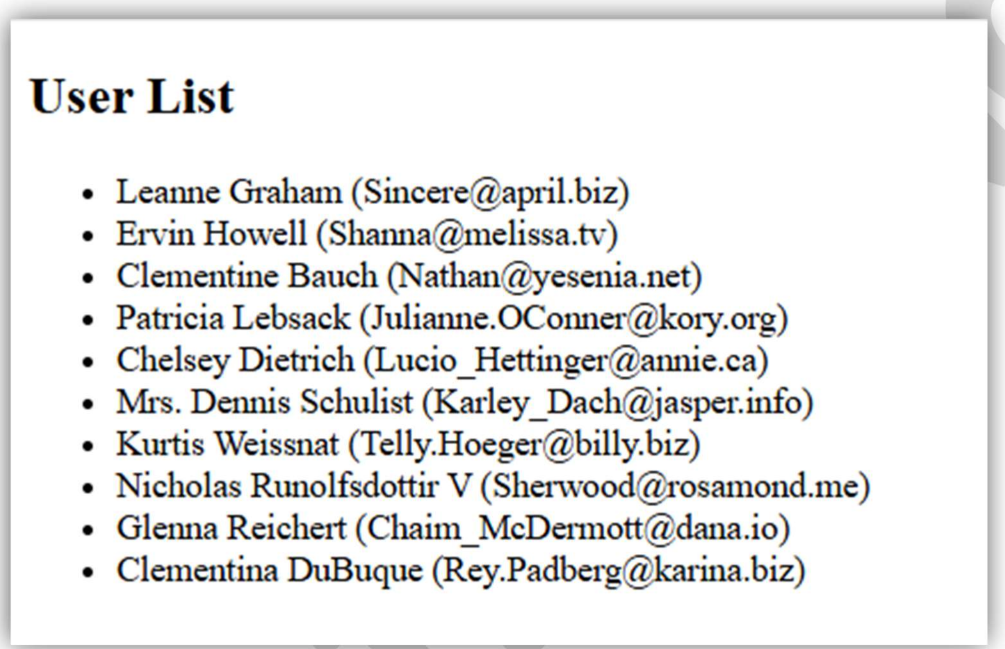
UserList.jsx:

```
import { useState, useEffect } from "react";
function UserList() {
  const [users, setUsers] = useState([]);
  useEffect(() => {
    // Fetching fake users from JSONPlaceholder API
    fetch("https://jsonplaceholder.typicode.com/users")
      .then((res) => res.json())
      .then((data) => setUsers(data))
    .catch((err) => console.error("Error fetching users:", err));
  }, []);
  return (
    <div>
      <h2>User List</h2>
      <ul>
        {users.map((user) => (
          <li key={user.id}>{user.name} ({user.email})</li>
        ))}
      </ul>
    </div>
  )}
export default UserList;
```

App.jsx:

```
import UserList from "./UserList";
function App() {
  return (
    <div>
      <UserList />
    </div>
  )}
export default App;
```

Output:



```
User List

  • Leanne Graham (Sincere@april.biz)
  • Ervin Howell (Shanna@melissa.tv)
  • Clementine Bauch (Nathan@yesenia.net)
  • Patricia Lebsack (Julianne.OConner@kory.org)
  • Chelsey Dietrich (Lucio_Hettinger@annie.ca)
  • Mrs. Dennis Schulist (Karley_Dach@jasper.info)
  • Kurtis Weissnat (Telly.Hoeger@billy.biz)
  • Nicholas Runolfsdottir V (Sherwood@rosamond.me)
  • Glenna Reichert (Chaim_McDermott@dana.io)
  • Clementina DuBuque (Rey.Padberg@karina.biz)
```

8.4 useRef – Referencing DOM Elements and Values

useRef provides a way to reference DOM elements or store mutable values without causing re-renders.

Example 1:

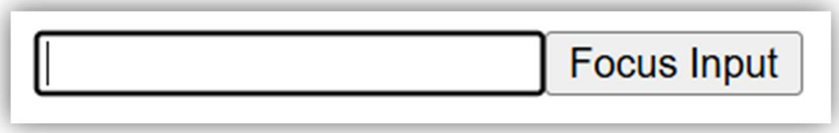
InputFocus.jsx:

```
import { useRef } from "react";
function InputFocus() {
 const inputRef = useRef();
 function handleClick() {
  inputRef.current.focus();
 }
 return (
  <div>
   <input type="text" ref={inputRef} />
   <button onClick={handleClick}>Focus Input</button>
  </div>
 );
}
export default InputFocus;
```

App.jsx:

```
import InputFocus from "./InputFocus";
function App() {
 return (
  <div>
   <InputFocus />
  </div>  )}
export default App;
```

Ouput:



Example 2:

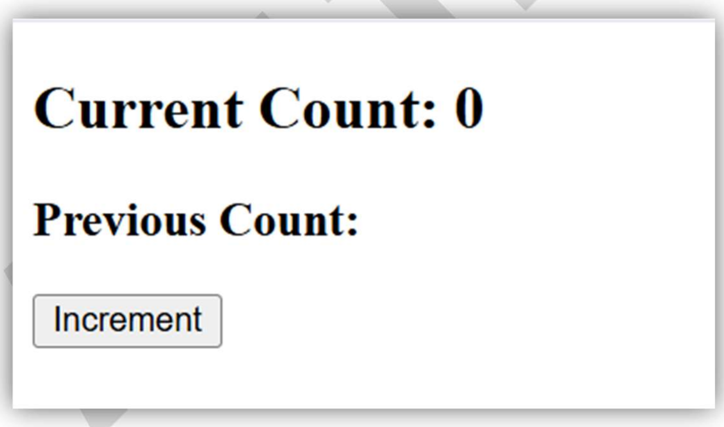
PreviousValue.jsx:

```
import { useState, useRef, useEffect } from "react";
function PreviousValue() {
  const [count, setCount] = useState(0);
  const prevCountRef = useRef();
  useEffect(() => {
    prevCountRef.current = count; // update the ref with current count
  }, [count]);
  return (
    <div>
      <h2>Current Count: {count}</h2>
      <h3>Previous Count: {prevCountRef.current}</h3>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>)}
export default PreviousValue;
```

App.jsx:

```
import PreviousValue from "./PreviousValue";

function App() {

  return (

    <div>

      <PreviousValue />

    </div>

  );

}

export default App;
```

Output:



```
Current Count: 0

Previous Count:
```

8.5 useMemo – Optimizing Expensive Calculations

useMemo memoizes the result of a function so it only recalculates when its dependencies change.

Example 1:

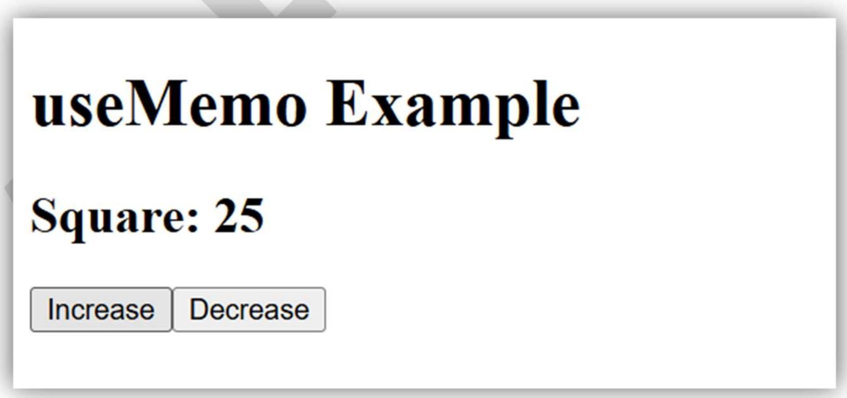
ExpensiveCalculation.jsx:

```
import { useState, useMemo } from "react";

function ExpensiveCalculation() {

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

 const squared = useMemo(() => {

  console.log("Calculating...");

  return count * count;

 }, [count]);

 return (

  <div>

   <h2>Square: {squared}</h2>

   <button onClick={() => setCount(count + 1)}>Increase</button>

   <button onClick={() => setCount(count - 1)}>Decrease</button>

  </div>

 )}

export default ExpensiveCalculation;
```

App.jsx:

```
import ExpensiveCalculation from "./ExpensiveCalculation";

function App() {

  return (

    <div>

      <h1>useMemo Example</h1>

      <ExpensiveCalculation />

    </div>

  );

}

export default App;
```

Output:



```
useMemo Example

Square: 25
```

Example 2:

DoubleValue.jsx:

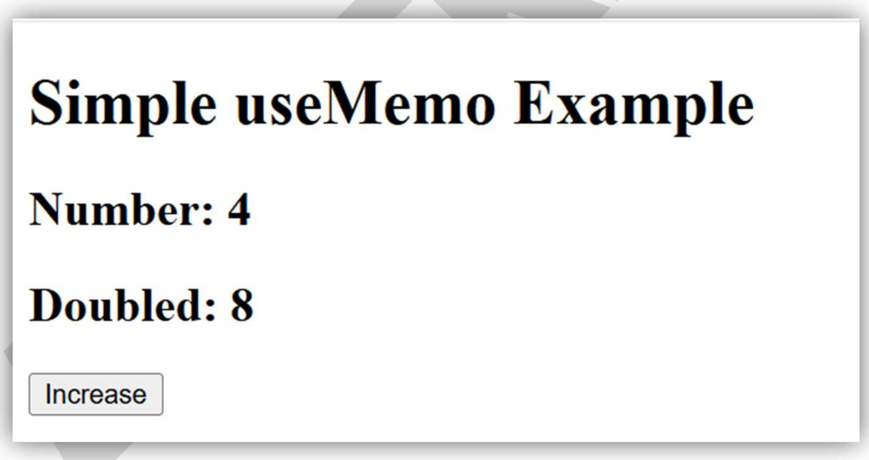
```
import { useState, useMemo } from "react";

function DoubleValue() {

  const [number, setNumber] = useState(0);

  const doubled = useMemo(() => {

    console.log("Calculating...");

    return number * 2;

  }, [number]);

  return (

    <div>

      <h2>Number: {number}</h2>

      <h2>Doubled: {doubled}</h2>

      <button onClick={() => setNumber(number + 1)}>Increase</button>

    </div>

  )

}

export default DoubleValue;
```

App.jsx:

```
import DoubleValue from "./DoubleValue";
```

```
function App() {
  return (
    <div>
      <h1>Simple useMemo Example</h1>
      <DoubleValue />
    </div>
  )
}
export default App;
```

Output:



```
Simple useMemo Example
Number: 4
Doubled: 8
```

8.6 useCallback – Memoizing Functions

useCallback returns a memoized version of a function so it doesn't get recreated on every render.

Example 1:

CallbackExample.jsx:

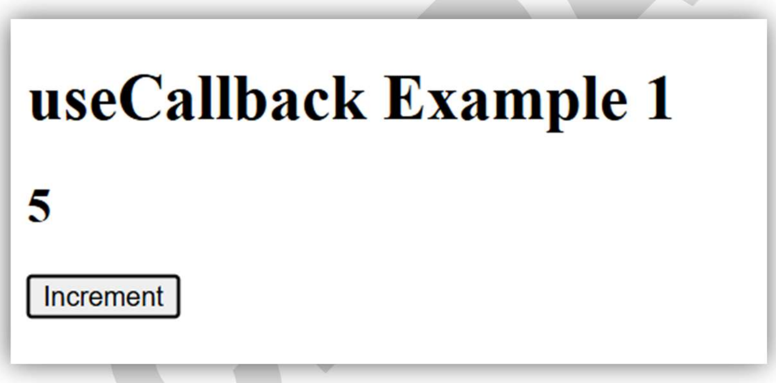
```
import { useState, useCallback } from "react";
function CallbackExample() {
  const [count, setCount] = useState(0);
  const increment = useCallback(() => {
    setCount(c => c + 1);
  }, []);
  return (
    <div>
      <h2>{count}</h2>
      <button onClick={increment}>Increment</button>
    </div>
  )}
export default CallbackExample;
```

App.jsx:

```
import CallbackExample from "./CallbackExample";
```

```
function App() {
  return (
    <div>
      <h1>useCallback Example 1</h1>
      <CallbackExample />
    </div>
  )}
export default App;
```

Output:



useCallback Example 1

5

Example 2:

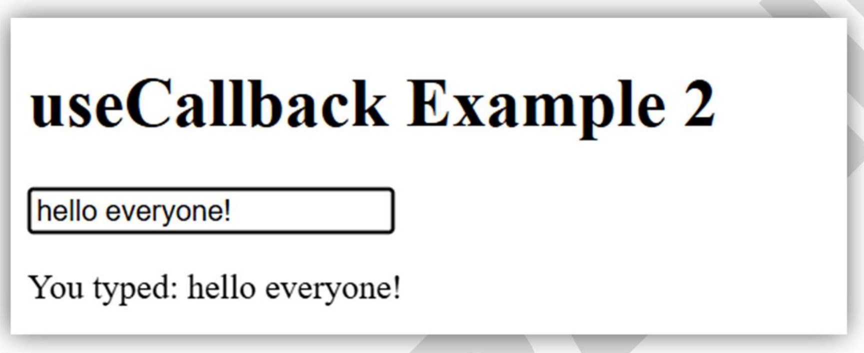
CallbackButton.jsx:

```
import { useState, useCallback } from "react";
function CallbackButton() {
  const [text, setText] = useState("");

  const handleChange = useCallback((e) => {

    setText(e.target.value);

  }, []);

  return (

    <div>

      <input type="text" value={text} onChange={handleChange}
placeholder="Type something" />

      <p>You typed: {text}</p>

    </div>

  );

}

export default CallbackButton;
```

App.jsx:

```
import CallbackButton from "./CallbackButton";

function App() {

  return (

    <div>

      <h1>useCallback Example 2</h1>

      <CallbackButton />
    </div>
  );
}
export default App;
```

Output:



useCallback Example 2

hello everyone!

You typed: hello everyone!

8.7 useContext – Accessing Context Values

The `useContext` hook allows you to consume data from a Context without prop drilling.

Example 1:

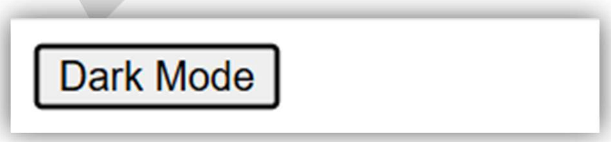
ThemeExample.jsx:

```
import { createContext, useContext } from "react";
export const ThemeContext = createContext("light");
function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button>{theme === "light" ? "Light Mode" : "Dark Mode"}</button>;
}
export default ThemedButton;
```

App.jsx:

```
import { ThemeContext } from "./ThemeExample";
import ThemedButton from "./ThemeExample";
function App() {
  return (
    <ThemeContext.Provider value="dark">
      <ThemedButton />
    </ThemeContext.Provider>
  );
}
export default App;
```

Output:



Example 2:

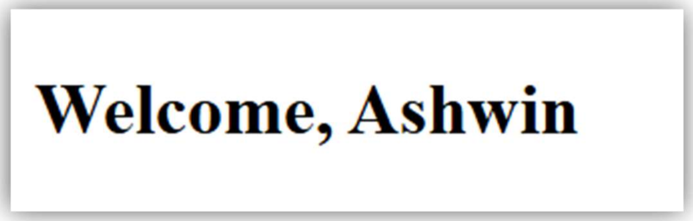
UserExample.jsx:

```
import { createContext, useContext } from "react";

export const UserContext = createContext({name:"Guest"});

function UserProfile() {

  const user = useContext(UserContext);

  return <h2>Welcome, {user.name}</h2>;

}

export default UserProfile;
```

App.jsx:

```
import { UserContext } from "./UserExample";

import UserProfile from "./UserExample";

function App() {

  return (

    <UserContext.Provider value={{name:"Ashwin"}}>

      <UserProfile />

    </UserContext.Provider>

  );

}

export default App;
```

Output:


Welcome, Ashwin

8.8 useReducer – Complex State Management

The useReducer hook is useful for managing more complex state logic, especially when multiple state variables are related.

Example:

Counter.jsx:

```
import { useReducer } from "react";

function reducer(state, action) {

 switch (action.type) {

  case "increment":

   return { count: state.count + 1 };

  case "decrement":

   return { count: state.count - 1 };

  default:

   return state;

 }}
```

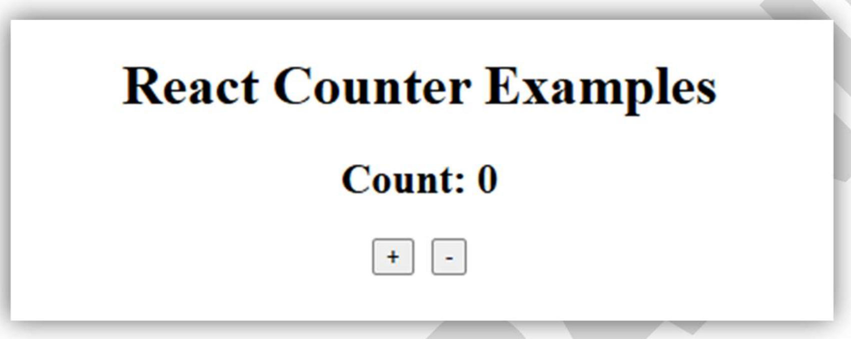
```
function Counter() {
 const [state, dispatch] = useReducer(reducer, { count: 0 });
 return (
  <div style={{ textAlign: "center", margin: "20px 0" }}>
   <h2>Count: {state.count}</h2>
   <button
    onClick={() => dispatch({ type: "increment" })}
    style={{ marginRight: "10px" }}
   >+</button>
   <button onClick={() => dispatch({ type: "decrement" })}>-</button>
  </div>
 )}
export default Counter;
```

App.jsx:

```
import Counter from "./Counter"
function App() {
 return (
  <div>
   <h1 style={{ textAlign: "center" }}>React Counter Examples</h1>
   <Counter />
```

```
    </div>
  );
}
export default App;
```

Output:



React Counter Examples
Count: 0

9. ROUTING WITH REACT ROUTER

9.1 Introduction

By default, React applications are single-page applications (SPA). This means everything is rendered on one HTML page, and navigation is handled inside React without reloading the browser.

To add navigation between pages (like Home, About, Contact), we use **React Router**, the most popular routing library for React.

9.2 Installing React Router

To use React Router, install it in your project:

npm install react-router-dom

9.3 Basic Setup

Example: Creating Home and About pages with navigation.

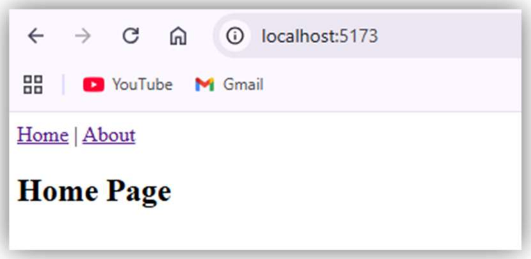
Example 1:

App.jsx:

```
import { BrowserRouter as Router, Routes, Route, Link } from "react-router-dom";

function Home() {

  return <h2>Home Page</h2>;

}
```

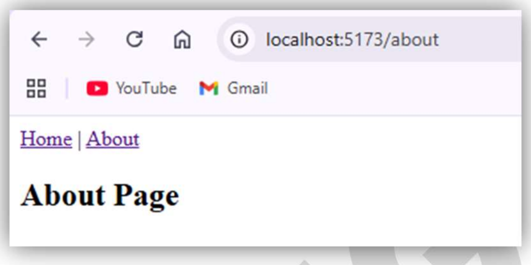
```
function About() {
 return <h2>About Page</h2>;
}
function App() {
 return (
  <Router>
   <nav>
    <Link to="/">Home</Link> | <Link to="/about">About</Link>
   </nav>
   <Routes>
    <Route path="/" element={<Home />} />
    <Route path="/about" element={<About />} />
   </Routes>
  </Router>
 )
}
export default App;
```

Output:


localhost:5173

Home | About

Home Page


localhost:5173/about

Home | About

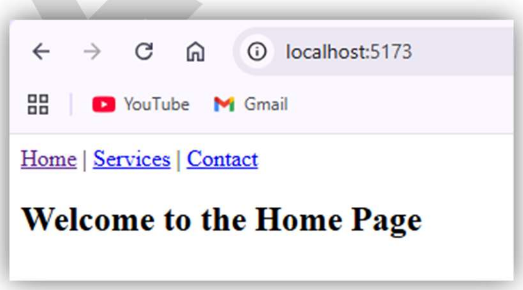
About Page

Example 2:

App.jsx:

```
import { BrowserRouter as Router, Routes, Route, Link } from "react-router-dom";

function Home() {
  return <h2>Welcome to the Home Page</h2>;
}

function Services() {
  return <h2>Our Services</h2>;
}

function Contact() {
  return <h2>Contact Us</h2>;
}

function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link> |{" "}
        <Link to="/services">Services</Link> |{" "}
        <Link to="/contact">Contact</Link>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/services" element={<Services />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
    </Router>
  )}
export default App;
```

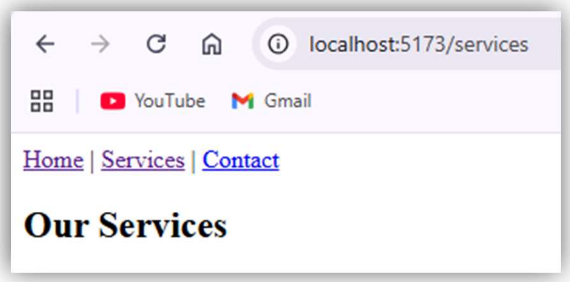
Output:


localhost:5173

YouTube Gmail

Home | Services | Contact

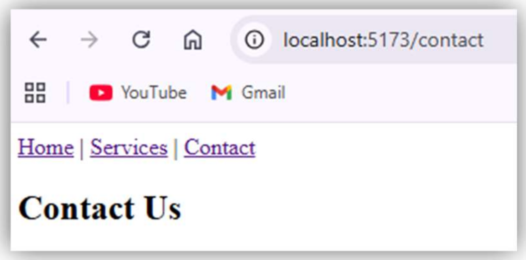
Welcome to the Home Page


localhost:5173/services

YouTube Gmail

Home | Services | Contact

Our Services



Home | Services | Contact

Contact Us

10. STYLING IN REACT

10.1 Introduction

Styling is an important part of building React applications. React does not enforce a specific way of styling components — you can use plain CSS, CSS Modules, CSS-in-JS libraries, or utility-first frameworks like Tailwind CSS.

10.2 Inline Styling

React allows you to apply styles directly using the `style` attribute. Styles are written as objects in camelCase.

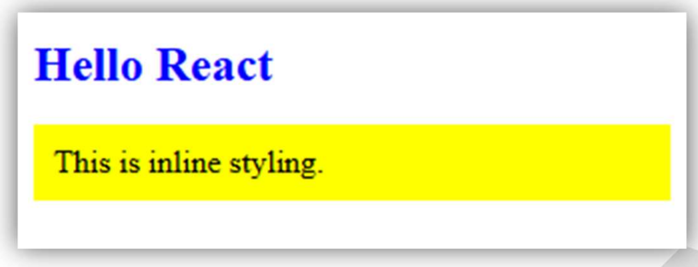
Example 1:

App.jsx:

```
function App() {
  const headingStyle = { color: "blue", fontSize: "24px" };
  return (
    <div>
      <h1 style={headingStyle}>Hello React</h1>
      <p style={{ backgroundColor: "yellow", padding: "10px" }}>
        This is inline styling.
      </p>
    </div>
```

```
 )}
export default App;
```

Output:



```
Hello React
This is inline styling.
```

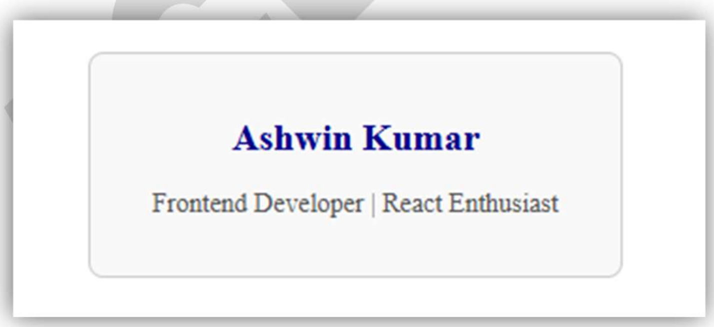
Example 2:

App.jsx:

```
function App() {
  const cardStyle = {
    border: "1px solid #ccc",
    borderRadius: "8px",
    padding: "16px",
    width: "250px",
    margin: "20px auto",
    textAlign: "center",
    backgroundColor: "#f9f9f9",
  };
```

```
 const nameStyle = { color: "darkblue",  fontSize: "20px",   fontWeight: "bold" };
 const descStyle = {
  color: "#555",
  fontSize: "14px",
 };
 return (
  <div style={cardStyle}>
   <h2 style={nameStyle}>Ashwin Kumar</h2>
   <p style={descStyle}>Frontend Developer | React Enthusiast</p>
  </div>
 )}
export default App;
```

Output:



Ashwin Kumar

Frontend Developer | React Enthusiast

10.3 External CSS Files

You can create a CSS file and import it into your component.

Example 1:

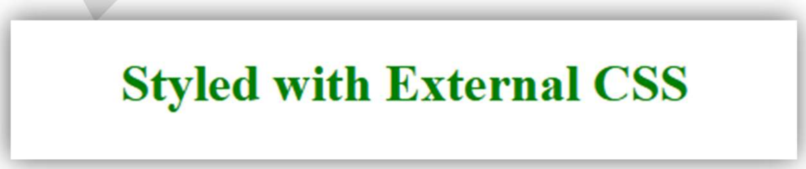
App.css:

```
.title {
  color: green;
  text-align: center;
}
```

App.jsx:

```
import "./App.css";
function App() {
  return <h1 className="title">Styled with External CSS</h1>;
}
export default App;
```

Output:



```
Styled with External CSS
```

Example 2:

Card.css:

```
.card {
  border: 1px solid #ccc;
  border-radius: 8px;
  padding: 16px;
  width: 250px;
  margin: 20px auto;
  text-align: center;
  box-shadow: 2px 2px 8px rgba(0, 0, 0, 0.1);
}
.card h2 {
  color: #2c3e50;
}
.card p {
  color: #555;
  font-size: 14px;
}
```

Card.jsx:

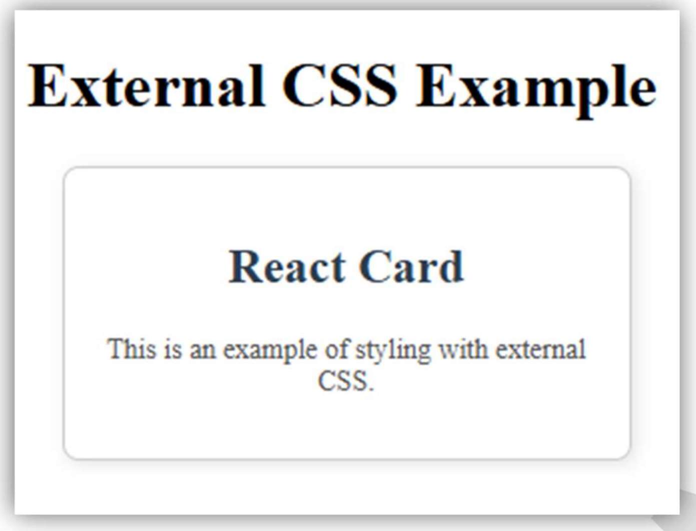
```
import "./Card.css";
```

```
function Card() {
  return (
    <div className="card">
      <h2>React Card</h2>
      <p>This is an example of styling with external CSS.</p>
    </div>
  );
}
export default Card;
```

App.jsx:

```
import Card from "./Card";
function App() {
  return (
    <div>
      <h1>External CSS Example</h1>
      <Card />
    </div>
  );
}
export default App;
```

Output:



10.4 CSS Modules

CSS Modules scope styles locally to a component, avoiding class name conflicts.

Example 1:

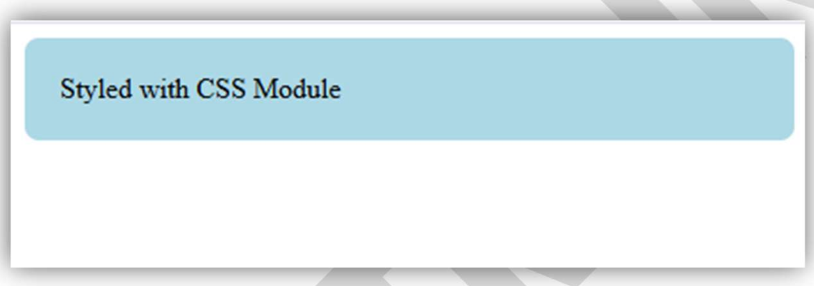
App.module.css:

```
.container {
  background: lightblue;
  padding: 20px;
  border-radius: 8px;
}
```

App.jsx:

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

function App() {

  return <div className={styles.container}>Styled with CSS Module</div>;

}

export default App;
```

Output:



```
Styled with CSS Module
```

Example 2:

Button.module.css:

```
.button {

  background-color: #4caf50;

  color: white;

  padding: 12px 24px;

  border: none;

  border-radius: 6px;
  cursor: pointer;
  font-size: 16px;
}
.button:hover {
  background-color: #45a049;
}
```

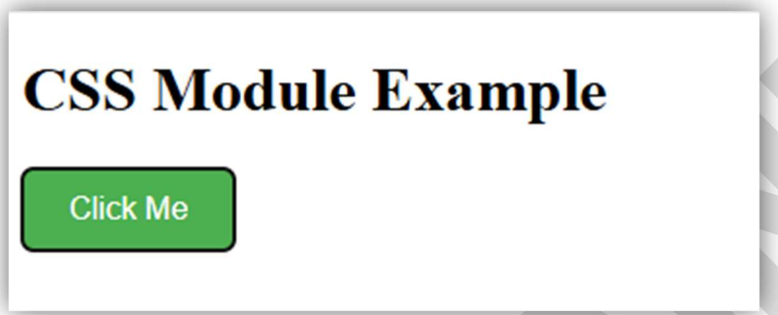
Button.jsx:

```
import styles from "./Button.module.css";
function Button() {
  return <button className={styles.button}>Click Me</button>;
}
export default Button;
```

App.jsx:

```
import Button from "./Button";
function App() {
  return (
    <div>
      <h1>CSS Module Example</h1>
      <Button />
  </div>
)}
export default App;
```

Output:



CSS Module Example

11. SAMPLE PROJECT

To-Do List with **CRUD** operations using React hooks and CSS Modules for styling.

App.jsx:

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

import styles from "./Todo.module.css";

function App() {

  const [task, setTask] = useState("");

  const [tasks, setTasks] = useState([]);

  const [editIndex, setEditIndex] = useState(null);


 // Load tasks from localStorage

  useEffect(() => {

    const saved = JSON.parse(localStorage.getItem("tasks"));

    if (saved) setTasks(saved);

  }, []);


// Save tasks to localStorage

  useEffect(() => {

    localStorage.setItem("tasks", JSON.stringify(tasks));

  }, [tasks]);
```

```
const handleAddOrEdit = () => {
  if (!task.trim()) return;
  if (editIndex !== null) {
    const updated = [...tasks];
    updated[editIndex].text = task;
    setTasks(updated);
    setEditIndex(null);
  } else {
    setTasks([...tasks, { text: task, completed: false }]);
  }
  setTask("");
};

const handleDelete = (index) => {
  setTasks(tasks.filter((_, i) => i !== index));
};

const handleEdit = (index) => {
  setTask(tasks[index].text);
  setEditIndex(index);
};
```

```
const toggleComplete = (index) => {
  const updated = [...tasks];
  updated[index].completed = !updated[index].completed;
  setTasks(updated);
};

return (
  <div className={styles.container}>
    <h2 className={styles.heading}>📝 To-Do List</h2>
    <div className={styles.inputBox}>
      <input
        type="text"
        value={task}
        onChange={(e) => setTask(e.target.value)}
        placeholder="Enter task..."
        className={styles.input}
      />
      <button onClick={handleAddOrEdit} className={styles.addBtn}>
        {editIndex !== null ? "Update" : "Add"}
      </button>
    </div>
```

```
<ul className={styles.list}>
 {tasks.map((t, i) => (
  <li
   key={i}
   className={`${styles.listItem} ${
    t.completed ? styles.completed : ""
   }`}>
   <span onClick={() => toggleComplete(i)}>{t.text}</span>
   <div>
    <button onClick={() => handleEdit(i)} className={styles.editBtn}>
     Edit
    </button>
    <button
     onClick={() => handleDelete(i)}
     className={styles.deleteBtn}>
     Delete
    </button>
   </div>
  </li>
 ))}
</ul>
```

```
    </div>
  )}
export default App;
```

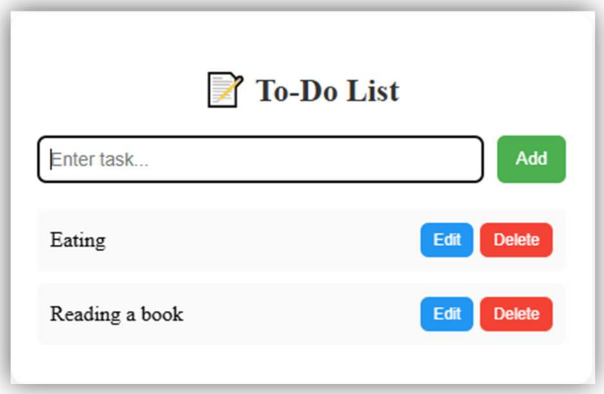
Todo.module.css:

```
.container {
  max-width: 400px;
  margin: 40px auto;
  padding: 20px;
  background: #fefefe;
  border-radius: 12px;
  box-shadow: 0 4px 10px rgba(0, 0, 0, 0.1);
}
.heading {
  text-align: center;
  color: #333;
  margin-bottom: 20px;
}
.inputBox {
  display: flex;
  gap: 10px;
  margin-bottom: 20px;
}
.input {
  flex: 1;
  padding: 8px;
  border: 2px solid #ccc;
  border-radius: 6px;
  font-size: 14px;
}
.addBtn {
  background: #4caf50;
  color: white;
  border: none;
  padding: 8px 14px;
  border-radius: 6px;
  cursor: pointer;
}
.addBtn:hover {
  background: #45a049;
}
.list {
  list-style: none;
  padding: 0;
  margin: 0;
}
.listItem {
  display: flex;
  justify-content: space-between;
  align-items: center;
  background: #fafafa;
  margin-bottom: 10px;
  padding: 10px;
  border-radius: 6px;
  transition: background 0.3s;
}
.listItem:hover {
  background: #f0f0f0;
}
.completed {
  text-decoration: line-through;
  color: gray;
}
```

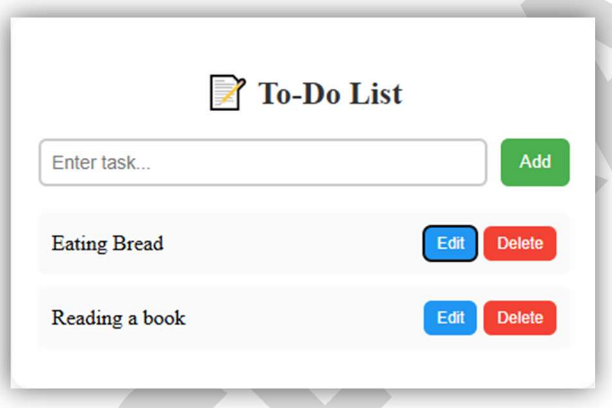
```
.editBtn, .deleteBtn {
  border: none;
  padding: 6px 10px;
  border-radius: 6px;
  margin-left: 5px;
  cursor: pointer;
  font-size: 12px;
}
.editBtn {
  background: #2196f3;
  color: white;
}
.editBtn:hover {
  background: #1976d2;
}
.deleteBtn {
  background: #f44336;
  color: white;
}
.deleteBtn:hover {
  background: #d32f2f;}
```

Output:

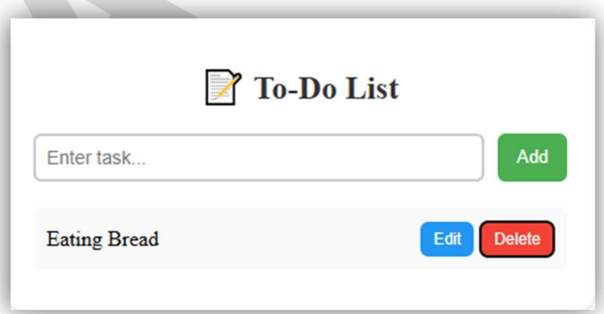
Add Todo:



Edit Todo:



Delete Todo:


To-Do List
Enter task...
Add
Eating Bread
Edit
Delete