Check out this video for a walkthrough of the assignment.
Note: The topic shown in the video might be different.

Requirement:

1. Analyze the reference learning experience design table here thoroughly to understand clearly what kind of activities and use cases are we looking for
2. Learn about useCallback Hook in React.
    a. You'll need to do this only for the necessary topics needed to fill in the learning experience design table given below
    b. You don't have to master useCallback hook but the necessary topics you must know thoroughly
3. Complete the learning experience design table below by filling up (B) and (C) columns for each of the expected learning outcome provided in (A) column

(A) Expected Learning Outcome	**(B) Activity or Practical usecase**	**(C) Demo idea or Code snippet (with new learnings mentioned)**

User understands that in scenarios when child component has functions passed as prop, child components gets re-rendered unnecessarily when parent re-renders, due to the function reference changing

Activity #1:How would you implement a counter system with two components, Parent and Child, where the counter function is passed as a prop from the parent to the child component?

```
//File : Parent.jsx

import React, {useState } from 'react'
import Child from './Child'
const Parent = () => {
    const[counter,setCounter] = useState(0);
    const[childCounter,setChildCounter] = useState(0);

    //when we update usestate variable it will re-render this
component along with its child component

    const handleOnClickCount =()=>{
        setCounter(counter + 1);
    }

   // function that will be called from the child component
    const handleOnClickChild =()=>{
console.log("Hello this is function reference");
     setChildCounter(childCounter + 1);
    }

    return (
        <div>
            <Child childCounter={childCounter}
handleOnClickChild={handleOnClickChild} />
            <h1>This is Parent Component: {counter} </h1>
            <button onClick={handleOnClickCount}>Increment
Parent</button>
        </div>
    )
}
export default Parent

//When the increment button is clicked in the parent, it will
re-render both the parent and the child. Additionally, it recreates a
function every time, causing the child component to re-render
because it receives a new function in its props. If you run the
code, open the console to better understand.

//File:Child.jsx
import React from 'react'
const Child = ({childCounter,handleOnClickChild}) => {
    console.log("This is child component")
  return (
    <div>
        <h1>This is Child Component: {childCounter}</h1>
        <button onClick={handleOnClickChild}>Increment
child</button>
    </div>
  )
```

		}
		export default Child;

User should be able to use useCallback() to avoid function reference change when a component re-renders	Activity #2:How can you utilize the useCallback Hook to maintain a consistent function reference and avoid unnecessary changes during the rendering of a parent component?	//File : Parent.jsx
		import React, { **useCallback**, useState } from 'react'
		import Child from './Child'
		const Parent = () => {
		const[counter,setCounter] = useState(0);
		const[childCounter,setChildCounter] = useState(0);
		//when we update usestate variable it will re-render this component along with its child component
		const handleOnClickCount =()=>{
		setCounter(counter + 1);
		}
		
		//Use the useCallback() Hook to prevent the function reference from changing when a component re-renders. This function should only be triggered when the 'childCounter' state changes.;
		
		const handleOnClickChild = useCallback(()=>{
		setChildCounter(childCounter + 1);
		},[childCounter]); // Dependency Array
		
		return (
		<div>
		<Child childCounter={childCounter} handleOnClickChild={handleOnClickChild} />
		<h1>This is Parent Component: {counter} </h1>
		<button onClick={handleOnClickCount}>Increment Parent</button>
		</div>
		)
		}
		export default Parent
		
		//File:Child.jsx
		Same as Above code.
		
		The useCallback() Hook prevents the recreation of the callback function within the parent component during a re-render. However, it does not address the unnecessary re-rendering of the child component. To resolve this issue, please refer to Activity #3.

User should be able to use React.Memo() along with useCallback() to resolve the unnecessary re-render issue	Activity #3: Develop a code snippet to optimize the performance of a child component by preventing unnecessary re-renders, utilizing React.memo() and useCallback()	**//Using {memo} that is React.memo() will cause React to skip rendering a component if its props have not changed.** **//File Parent.jsx - //it will be same as in Activity no #2 with the use of useCallback Hook**

		//File: Child.jsx
		import React,{**memo**} from 'react'
		const Child = ({childCounter,handleOnClickChild}) => {
		    console.log("This is child component")
		  return (
		    <div>
		        <h1>This is Child Component: {childCounter}</h1>
		        <button onClick={handleOnClickChild}>Increment child</button>
		    </div>
		  )
		}
		//To prevent re-rendering, we use React.memo(). Now, the component will only re-render if there are changes in its props.
		
		export default **memo(Child); // Bind the child component in memo**
		
		//memo helps bind the entire component and skips re-rendering unless there are changes in the props.

User understands the need of dependency array in useCallback Hook	Activity #4: Demonstrate the difference between an empty dependency array and a non-empty dependency array in the useCallback hook.	//File:Parent.jsx:
		import React, { useCallback, useState } from 'react'
		import Child from './Child'
		const Parent = () => {
		    const[counter,setCounter] = useState(0);
		    const[childCounter,setChildCounter] = useState(0);
		
		    const handleOnClickCount =()=>{
		        setCounter(counter+1);
		    }
		
		    const handleOnClickChild = useCallback(()=>{
		        setChildCounter(childCounter + 1);
		    },**[ ]**); ***//Empty Dependency Array***
		
		    return (
		        <div>
		            <Child childCounter={childCounter} handleOnClickChild={handleOnClickChild} />
		            <h1>This is Parent Component: {counter} </h1>
		            <button onClick={handleOnClickCount}>Increment Parent</button>
		        </div>
		    )
		}
		export default Parent

		//When we click the button in the child to increment the counter and call this function, it will only be created once and will not be recreated. Therefore, this demonstrates that with an empty dependency array, the function is created only during the initial rendering.
		
		//For the seamless use of the counter in the child component, we have to pass the dependency..
		
		//So pass the 'childCounter' in dependency array
		
		const handleOnClickChild = useCallback(()=>{
		setChildCounter(childCounter + 1);
		},**[childCounter]**); **// *Dependency Array***
		
		// Now, the function will be recreated when the childCounter variable changes, working as intended.

User should be able to use dependency array with useCallback	Use Case #1: In a Todo List Application, to prevent unnecessary recreation of the function.	//File:Todo.jsx
		
		import React, { useState, useCallback } from 'react';
		const Todo = () => {
		const [todos, setTodos] = useState([]);
		const [newTodo, setNewTodo] = useState('');
		
		const addTodo = useCallback(() => {
		if (newTodo.trim() !== '') {
		setTodos((prevTodos) => [...prevTodos, newTodo]);
		setNewTodo('');
		console.log("In AddTodo");
		}
		}, [newTodo]); // *Dependency Array*
		
		const handleInputChange = (e) => {
		setNewTodo(e.target.value);
		};
		return (
		<div>
		<h1>Todo List</h1>
		<ul>
		{todos.map((todo, index) => (
		<li key={index}>{todo}</li>
		))}
		</ul>
		<div>
		<input type="text" value={newTodo}
		onChange={handleInputChange} />
		<button onClick={addTodo}>Add Todo</button>
		</div>

		</div>
		);
		};
		export default Todo;
		
		//**Include the newTodo variable in the dependency array, as the addTodo function should only be called when there is an addition to the newTodo.**

Link to your final solution code:**https://github.com/Prince-kumar-pk/Crio.Do-Assignment**

Estimated creation effort: ~ 5 hours

Target audience: Learners with

1. Knowledge of React and programming in Javascript.
2. Knowledge of React Lifecycle, rendering and memoization.

Guidelines for the assignment

- Ensure the statements you add are clear enough for another person checking the template. Do include any additional context if needed.
- You can reorder the entries of (A) column for a better sequence of learning, if needed
- (B) column can have
    - Activity i.e, Exercise for learners to do
        - (C) column to include additional code needed for the activity with any new syntax or learning clearly called out, in this case
    - Practical usecase: Practical need for a concept shown
        - (C) column to include details and any code snippets for how to show/demo this usecase to learners, in this case
        - Note that this has to be in a way which can be shown to learners and not just explained in words in a tutorialish way
        - Can bring in relatable analogies additionally here as needed
- The activities should form a logical sequence with the same sample application instead of being disconnected
    - Have the activities mimic real-life scenarios as much as possible so that learners can apply it easily in other scenarios as well

- (B) column for an expected outcome (A) can have multiple items (activities or practical usecases) if a single item might become difficult for learners to grasp
    - If any syntax is not straightforward or needs knowledge of another topic, you must include an additional activity or practical use case to driver understanding of this (either usage or need) separately

What you'll be evaluated on?

1. How well does the practical use case convey the need
2. Technical accuracy of provided code snippets
3. Attention to detail and ability to design learning content by being in learner's shoes
    - Selection of a good theme/use-case for the entire content using which good set of activities and practical use cases can be created (eg: A counter application, Todo application, Stock quotes portfolio)
    - Good sequence of activities within and across each expected outcome, fitting in well like different episodes of a web series
    - Right difficulty level of activities with not too much to be learned in any single activity
    - Identifying any additional topics that comes in as a dependency to satisfy a given learning outcome and including Activities/Practical use-cases for them

Submission

- Start by making a copy of this document and update access so that "Anyone with the link" can access the doc

Learning Experience Design Table - React Fundamentals

(A) Expected Outcome	**(B) Activity/Analogy/Real-life usecase**	**(C) Demo idea or Code snippet (with new learnings mentioned)**

User should be able to create a simple unstyled static page with React	Activity #1: Write React to show a simple element on the screen (eg: a heading) to get started	// File: src/index.js
		
		// Note:
		// React.createElement used to help better understand need for JSX in next section
		// Also, makes concept of a React "element" more explicit
		
		// 1 - React.createElement() to create elements

		const element = React.***createElement***("h1", null, "React element");
		
		// 2.1 - ReactDOM.render() to render page
		// 2.2 - Linking public/index.html
		ReactDOM.***render***(element, document.querySelector("#root"));

	Activity #2: Write React to show a static Counter page with a count display & 2 buttons (w/o styles)	*// File: src/index.js*
		
		const counterDisplayElement = React.createElement("h1", null, 0);
		const incrementButtonElement = React.createElement("button", null, "+");
		const decrementButtonElement = React.createElement("button", null, "-");
		
		// 1 - Nesting elements
		const containerDivElement = React.createElement("div", null, **[counterDisplayElement, incrementButtonElement, decrementButtonElement]**);
		
		console.log(containerDivElement);
		
		ReactDOM.render(containerDivElement, document.querySelector("#root"));

User should be able to add styles to a React application	Activity #3: Write React to add styles to the static Counter page from a given CSS file	*// File: src/index.js* *// 1. Import syntax* **import** "./styles.css" *// 2.1. Setting "props" parameter of the React.createElement method for adding classes* *// 2.2. HTML attributes vs DOM properties* const counterDisplayElement = React.createElement("h1", **{className: "counter-display"}**, 0);
User understands the advantage of using JSX over React.createElement to create React elements	**Activity** #4: Write React code to create a simple element (eg: heading) with JSX - show comparison of React.createElement & JSX versions (Note: This along with Activity #3 will drive this outcome)	*// File: src/index.js* const counterDisplayElement = React.createElement("h1", { "className": "counter-display" }, 0); *// 1. JSX way of creating individual elements* const counterDisplayElementJSX = **(<h1 className="counter-display">0</h1>)**
User should be able to use JSX	**Activity** #5: Write React code to show result of	*// File: src/index.js*

expressions	an expression (eg: product of two numbers) on page using JSX	 *// 1. {} to be used for evaluating JS expressions* const counterDisplayElementJSX = (<h1 className="counter-display">**{2 * 5}**</h1>)

User should be able to use JSX to create a simple static page	**Activity** #6: Write React code with JSX for code in Activity #2 of Milestone #1- show comparison of React.createElement & JSX versions	*// File: src/index.js*
		
		// v1 - Create individual elements and nest under parent element
		// 1. Using “{“ to add array of child elements as an array
		const counterDisplayElementJSX = (<h1 className="counter-display">0</h1>)
		
		const incrementButtonElementJSX = (<button>+</button>);
		
		const decrementButtonElementJSX = (<button>-</button>);
		
		const containerDivElementJSX = (<div>**{[counterDisplayElementJSX, incrementButtonElementJSX, decrementButtonElementJSX]}**</div>);
		
		// v2 - Create single element by nesting within JSX
		// 2.1 JSX can have nested children just like nesting HTML elements
		// 2.2 Only 1 top-level parent valid for an element
		const containerDivElementJSX = (
		  <div>
		    <h1 className="counter-display">0</h1>
		    <button>+</button>
		    <button>-</button>
		  </div>
		)

User understands the need for components	**Usecase** #1: Needing to display multiple counters
	(Note: This along with Usecase #2 will drive this outcome)
		Show how we’ll need to duplicate JSX to have multiple counters otherwise with 2 versions compared
		
		// v1 - without components
		const containerDivElementJSX = (
		 <div>
		  <div>
		    <h1 className="counter-display">0</h1>
		    <button>+</button>
		    <button>-</button>
		  </div>
		  <div>
		    <h1 className="counter-display">0</h1>
		    <button>+</button>
		    <button>-</button>
		  </div>
		 </div>

		)
		
		ReactDOM.render(containerDivElementJSX,
		document.querySelector("#root"));
		
		// v2 - with components
		(Note: Learners need not understand the syntax at this point but just see the difference in lines of code)
		
		class Counter extends React.Component {
		  render() {
		     return (
		       <div>
		         <h1 className="counter-display">0</h1>
		         <button >+</button>
		         <button>-</button>
		       </div>
		    )
		     }
		}
		
		class Counter extends React.Component {
		  render() {
		     return (
		       <div>
		         <Counter />
		         <Counter />
		       </div>
		    )
		     }
		}
		
		ReactDOM.render(<Counter />,
		document.querySelector("#root"));

User should be able to create class component for a simple static page	**Activity** #7: Move the Counter static page JSX to a dedicated “Counter” component	*// File: src/index.js*
		
		// 1.1. Extending “React.Component”
		// 1.2. JSX to be returned from “render()”
		class Counter **extends React.Component** {
		  render() {
		     return (
		       <div>
		         <h1 className="counter-display">0</h1>
		         <button >+</button>
		         <button>-</button>
		       </div>
		    )
		     }
		}

		ReactDOM.render(<Counter />, document.querySelector("#root"));
User understands the need for state	**Usecase** #2: To store data within a component	Class variables get overwritten on re-render. Can utilise multiple ReactDOM.render() calls to show this

User should be able to create, read and update state for a class component	**Activity** #8: Initialise a state variable in "Counter" component for storing "count" to 0 and display its value as count on the page	*// File: src/index.js*
		
		// 1. State is initialised inside the constructor
		constructor(props) {
		  super(props);
		  this.state = {
		    count: 0
		  };
		}
		
		// 2. Reading state value
		<h1 className="counter-display">{this.state.count}</h1>

	Activity #9: Create event handlers for increasing & decreasing Counter count
	(Note: Can also have a transition here - set event handler to console.log; Next activity can be updating state)
		// File: src/index.js
		
		// 1.1. Class method syntax
		// 1.2. this.setState() syntax with callback
		increaseCount = () => {
		  this.setState((state, props) => {
		    return { count: state.count + 1 }
		  });
		};
		
		// 2. Syntax to pass event handler function reference
		<button onClick={**this.increaseCount**}>+</button>
		
		Note: Functional React Counter at this point