## 3.REACTJS-HOL

## Introduction

This document serves as a comprehensive guide to understanding fundamental React concepts and applying them in a practical, hands-on lab. We will begin by exploring the theoretical foundations of React, such as components, their types, and their core functions. Following the conceptual overview, we will proceed with a detailed, step-by-step walkthrough to build a "Student Score Calculator" application using React. This guide is designed to be thorough, providing in-depth explanations at every stage to ensure a complete understanding of the principles and practices involved.

## Part 1: Core React Concepts

Before diving into the practical lab, it's essential to grasp the foundational concepts that make React a powerful library for building user interfaces. The following sections address the key learning objectives outlined in your assignment document.

### 1. What are React Components?

At the very core of every React application are **components**. Think of components as independent, reusable building blocks that make up a user interface (UI). A complete UI, like a webpage, is typically formed by combining multiple components. For example, a social media page might have a NavigationBar component, a ProfileInfo component, a Feed component, and a ChatWindow component. Each of these is self-contained, managing its own structure, style, and logic.

The primary advantages of using a component-based architecture are:

* **Reusability:** A component can be used multiple times throughout an application, and even across different projects. For instance, a custom Button component can be used everywhere you need a button, ensuring consistency and saving development time.
* **Separation of Concerns:** Components allow you to break down a complex UI into smaller, manageable pieces. Each piece is responsible for only one thing, making the code easier to develop, debug, and maintain.
* **Composition:** React's design philosophy is built on composition. Simple components can be combined to create more complex components. This hierarchical structure makes it easy to reason about the application's architecture.

### 2. Differences Between Components and JavaScript Functions

While React components are often written as JavaScript functions, there are key distinctions that set them apart from regular JavaScript functions:

| **Feature** | **Regular JavaScript Function** | **React Component** |
| --- | --- | --- |
| **Purpose** | To perform a specific task, calculate a value, or manipulate data. It's a general-purpose block of code. | Its primary purpose is to describe a part of the user interface. It defines what should be rendered on the screen. |
| **Return Value** | Can return any type of data (string, number, object, array, boolean, etc.) or nothing at all (undefined). | Must return a description of the UI. This is typically done using **JSX** (JavaScript XML), which looks like HTML, or null if it should render nothing. |
| **Input** | Accepts arbitrary arguments of any type. | Accepts a single argument: an object called **props** (short for properties). Props are read-only and are used to pass data from a parent component to a child component. |
| **State** | A regular function is typically stateless. It produces the same output for the same input and has no memory of past executions (unless it uses closures or external variables). | Components can have an internal "memory" or **state**. State allows a component to keep track of changing data and re-render itself when that data changes, making the UI interactive. |
| **Lifecycle** | A function is simply called and executed. It has no concept of a "lifecycle." | Components have a lifecycle, which includes phases like mounting (being added to the DOM), updating (re-rendering due to state or prop changes), and unmounting (being removed from the DOM). |

In essence, a React component is a specialized JavaScript function designed specifically for building UIs, with added features like props, state, and lifecycle methods to manage the appearance and behavior of UI elements over time.

### 3. Types of Components

In React, there are two primary ways to create components:

1. **Class Components**
2. **Function Components**

Historically, class components were the only way to create stateful and complex components. However, with the introduction of **Hooks** in React 16.8, function components can now do everything class components can do, and they are the modern, recommended approach for building React applications.

### 4. Explaining Class Components

A class component is an ES6 class that extends React.Component. It must include a render() method that returns the JSX to be displayed.

**Key Features:**

* **State Management:** They manage their internal state using this.state, which is initialized in the constructor. State is updated using the this.setState() method. Calling this.setState() triggers a re-render of the component.
* **Lifecycle Methods:** They use lifecycle methods like componentDidMount(), componentDidUpdate(), and componentWillUnmount() to run code at specific points in the component's life.
* **Props:** They access props passed from a parent component via this.props.

**Example of a Class Component:**

import React from 'react';  
  
class WelcomeMessage extends React.Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 message: 'Welcome!'  
 };  
 }  
  
 componentDidMount() {  
 // This runs after the component is added to the DOM  
 console.log('Component has mounted.');  
 }  
  
 render() {  
 return <h1>{this.state.message} My name is {this.props.name}.</h1>;  
 }  
}  
  
export default WelcomeMessage;

### 5. Explaining Function Components

A function component is a plain JavaScript function that accepts a props object as its argument and returns a React element (JSX).

**Key Features:**

* **Simplicity:** They have a much simpler and more concise syntax compared to class components, making them easier to read and write.
* **Hooks:** They use **Hooks** to manage state and other React features.
  + **useState:** This Hook allows function components to have local state.
  + **useEffect:** This Hook lets you perform side effects, such as fetching data or subscribing to events, which serves the same purpose as lifecycle methods in class components.
* **Props:** They receive props directly as the first argument of the function.

**Example of a Function Component:**

import React, { useState, useEffect } from 'react';  
  
function WelcomeMessage(props) {  
 const [message, setMessage] = useState('Welcome!');  
  
 useEffect(() => {  
 // This runs after the component renders  
 console.log('Component has rendered or updated.');  
 }, []); // The empty array means it only runs once, like componentDidMount  
  
 return <h1>{message} My name is {props.name}.</h1>;  
}  
  
export default WelcomeMessage;

This is the modern and preferred way to write React components.

### 6. Defining the Component constructor()

The constructor() method is specific to **class components**. It is a special method that is called automatically when a component instance is created, before it is mounted to the DOM.

Its primary purposes are:

1. **Initializing State:** It is the only place where you should assign this.state directly. In all other methods, you must use this.setState().
2. **Binding Event Handlers:** To ensure that this works correctly inside an event handler method, you need to bind the method to the component instance in the constructor.

**Important Rules for constructor():**

* You must always call super(props) as the very first line. This calls the constructor of the parent class (React.Component) and ensures that this.props is correctly initialized.
* If you don't need to initialize state or bind methods, you don't need to implement a constructor for your React component.

### 7. Defining the render() Function

The render() function is the most important method in a **class component**. It is the only required method.

**Functionality:**

* The render() method is responsible for describing the UI. It returns the elements that should be displayed on the screen.
* It is called whenever there is a change in the component's state or props, which triggers a re-render.
* The return value is typically JSX, but it can also be other types like arrays, fragments, strings, numbers, booleans, or null.
* The render() function should be **pure**, meaning that for the same inputs (props and state), it should always return the same output. It should not modify the component's state or interact with the browser directly (e.g., no API calls). Such side effects should be placed in lifecycle methods like componentDidMount().

## Part 2: Hands-On Lab: Building the Score Calculator App

Now, we will apply the concepts above to build the scorecalculatorapp as described in your assignment.

### Prerequisites

Ensure you have the following software installed on your computer:

* **Node.js and npm:** You can download them from [nodejs.org](https://nodejs.org/). npm (Node Package Manager) is included with Node.js.
* **Visual Studio Code:** A popular code editor. You can download it from [code.visualstudio.com](https://code.visualstudio.com/).

### Step 1: Create the React Project

First, we need to create a new React application. Open your terminal (or the integrated terminal in Visual Studio Code) and navigate to the directory where you want to create your project. Then, run the following command:

npx create-react-app scorecalculatorapp

This command uses create-react-app, a tool maintained by Facebook, to set up a complete, ready-to-use React project with all the necessary configurations. It will create a new folder named scorecalculatorapp.

After the process is complete, navigate into your new project directory:

cd scorecalculatorapp

### Step 2: Create the Component File

For better organization, it's a good practice to keep your components in a dedicated folder.

1. Inside the src folder of your project, create a new folder named Components.
2. Inside this new Components folder, create a new file named CalculateScore.js.

Your project structure should look something like this:

scorecalculatorapp/  
├── node\_modules/  
├── public/  
├── src/  
│ ├── Components/  
│ │ └── CalculateScore.js <-- Your new component file  
│ ├── App.css  
│ ├── App.js  
│ ├── App.test.js  
│ ├── index.css  
│ ├── index.js  
│ └── ...  
└── package.json

### Step 3: Write the CalculateScore Function Component

Open the CalculateScore.js file and type the following code. Below the code block, you'll find a detailed explanation of what each part does.

**File: src/Components/CalculateScore.js**

import React from 'react';  
import '../Stylesheets/mystyle.css'; // Importing the CSS file for styling  
  
// This is a function component named CalculateScore.  
// It receives 'props' as an argument, which is an object containing all the data passed to it.  
function CalculateScore(props) {  
 // We are using object destructuring to extract the properties from the props object.  
 // This makes the code cleaner than writing props.name, props.school, etc.  
 const { name, school, total, goal } = props;  
  
 // This function calculates the percentage score.  
 // It takes the 'total' marks obtained and the 'goal' (which is the maximum possible marks).  
 // It returns the calculated percentage.  
 const calcScore = () => {  
 return (total / goal) \* 100;  
 };  
  
 // The component returns JSX, which is a syntax extension for JavaScript.  
 // It looks like HTML and describes what the UI should look like.  
 return (  
 <div className="student-card">  
 <div className="card-header">  
 <h1>Student Details</h1>  
 </div>  
 <div className="card-body">  
 <p><strong>Name:</strong> {name}</p>  
 <p><strong>School:</strong> {school}</p>  
 <p><strong>Total Marks:</strong> {total}</p>  
 <p><strong>Maximum Marks:</strong> {goal}</p>  
 <p><strong>Percentage:</strong> <span className="score">{calcScore().toFixed(2)}%</span></p>  
 </div>  
 </div>  
 );  
}  
  
// We export the component to make it available for use in other files, like App.js.  
export default CalculateScore;

### Step 4: Add Styles to the Component

To make our application look good, we will add some CSS styles.

1. Inside the src folder, create a new folder named Stylesheets.
2. Inside the Stylesheets folder, create a new file named mystyle.css.
3. Add the following CSS code to mystyle.css.

**File: src/Stylesheets/mystyle.css**

/\* General body styling for a clean background \*/  
body {  
 font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, 'Helvetica Neue', Arial, sans-serif;  
 background-color: #f0f2f5;  
 display: flex;  
 justify-content: center;  
 align-items: center;  
 height: 100vh;  
 margin: 0;  
}  
  
/\* The main container for the student's information card \*/  
.student-card {  
 background-color: #ffffff;  
 border-radius: 12px;  
 box-shadow: 0 4px 12px rgba(0, 0, 0, 0.1);  
 width: 100%;  
 max-width: 400px;  
 overflow: hidden; /\* Ensures the header's background doesn't spill out \*/  
 transition: transform 0.3s ease, box-shadow 0.3s ease;  
}  
  
.student-card:hover {  
 transform: translateY(-5px);  
 box-shadow: 0 8px 20px rgba(0, 0, 0, 0.15);  
}  
  
/\* Styling for the header section of the card \*/  
.card-header {  
 background: linear-gradient(135deg, #4a90e2, #50e3c2);  
 color: white;  
 padding: 20px;  
 text-align: center;  
}  
  
.card-header h1 {  
 margin: 0;  
 font-size: 1.8em;  
 font-weight: 600;  
}  
  
/\* Styling for the body/content section of the card \*/  
.card-body {  
 padding: 25px;  
 font-size: 1.1em;  
 color: #333;  
}  
  
.card-body p {  
 margin: 15px 0;  
 line-height: 1.6;  
}  
  
.card-body strong {  
 color: #4a90e2;  
 margin-right: 8px;  
}  
  
/\* Special styling for the calculated score to make it stand out \*/  
.score {  
 font-weight: bold;  
 font-size: 1.2em;  
 color: #ffffff;  
 background-color: #50e3c2;  
 padding: 4px 10px;  
 border-radius: 8px;  
}

*Note: We have already imported this CSS file in CalculateScore.js with the line import '../Stylesheets/mystyle.css';.*

### Step 5: Render the Component in App.js

Now we need to use our CalculateScore component. We will do this inside the main App component. Open the src/App.js file and replace its content with the following code:

**File: src/App.js**

import React from 'react';  
import CalculateScore from './Components/CalculateScore'; // Import our new component  
import './App.css'; // You can keep or remove this if not used  
  
// The main App component which acts as a container for other components.  
function App() {  
 // The App component returns the CalculateScore component.  
 // We pass data to CalculateScore using 'props'.  
 // Here, 'name', 'school', 'total', and 'goal' are the props.  
 return (  
 <div className="App">  
 <CalculateScore  
 name="John Doe"  
 school="Springfield High School"  
 total={255}  
 goal={300}  
 />  
 </div>  
 );  
}  
  
export default App;

**Explanation:**

1. **import CalculateScore from './Components/CalculateScore';**: This line imports the component we created, making it available to use inside App.js.
2. **<CalculateScore ... />**: This is how we use or "render" the component. It looks like an HTML tag.
3. **name="John Doe"**, **school="..."**, etc.: These are the **props** we are passing down to our CalculateScore component. The CalculateScore component will receive these in its props object. You can change these values to see the output update.

### Step 6: Run the Application

You are now ready to see your application in action. In your terminal, make sure you are still in the scorecalculatorapp directory, and run the following command:

npm start

This command will start the React development server and automatically open a new tab in your default web browser pointing to http://localhost:3000.

### Step 7: View the Output

If you've followed all the steps correctly, your browser should display the styled Student Score Card, which will look like this:

The application will show the details for "John Doe," including the calculated percentage of **85.00%**.

## Conclusion

In this hands-on lab, we have successfully built a simple but complete React application. We started by understanding the theoretical concepts of React components, including the differences between class and function components, and the roles of the constructor() and render() methods. We then applied this knowledge practically by creating a CalculateScore function component, styling it with CSS, and rendering it within our main App component by passing data through props. This exercise demonstrates the power of component-based architecture and provides a solid foundation for building more complex applications with React.