# *REACT ASSIGNMENT ON PROPS, STATE AND USEEFFECT*

## OBJECTIVE:

Develop a React application showcasing individual button interactions and a global reset functionality. The application should consist of a collection of buttons, each with its unique identifier number as label. The objective is to track and display user interactions (clicks) on each button, as well as provide a mechanism to reset these interactions to their initial state.

**Detailed Requirements:**

1. **Child Component (ButtonCounter):**

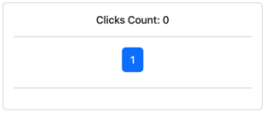
**Functionality:**

- Each instance of this component represents a unique button with its identifier.
- It should manage and display a count of how many times it has been clicked.
- Upon each click, it should increment its count.
- When the reset signal is received (via props), the count should return to zero.

**Display:**

- Use a Bootstrap card to contain each button.


- The card should display:
    - The click count at the top, separated by an **<hr/>** element.




- The button itself in the center.
- Utilize Bootstrap utilities to ensure the content of the card is centered both vertically and horizontally.

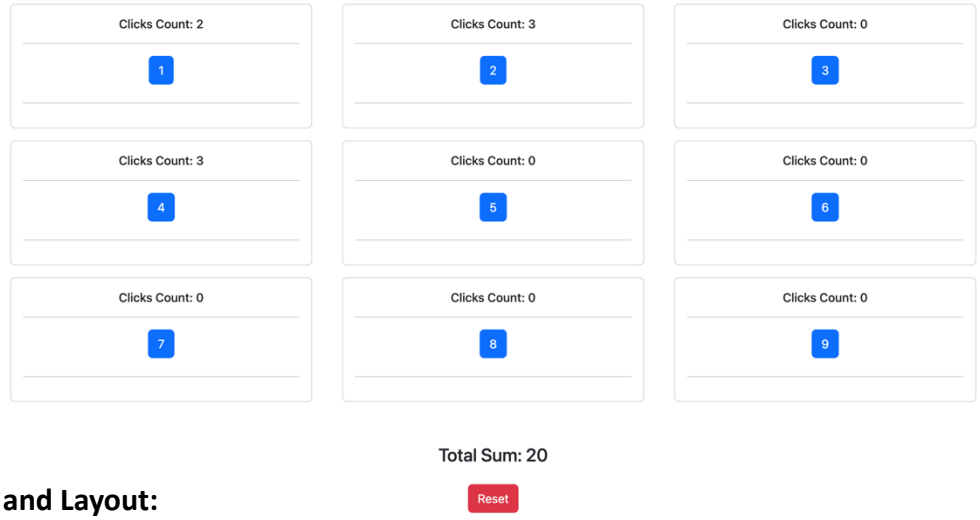
2. **Parent Component (App):**

**Functionality:**

- It should render a collection of **ButtonCounter** components in a grid layout.
- Provide a mechanism (state) to trigger the reset functionality across all child components.
- Have a reset button. When this button is clicked:
- All individual **ButtonCounter** component counts should be reset to their initial state.
- Any other global state, if present, should also be reset.

**Display:**

- Use a Bootstrap layout to arrange the **ButtonCounter** components in a grid (e.g., 3x3).
- Below the grid, centrally align the reset button.




3. **Styling and Layout:**
    - The application should be responsive, ensuring a good user experience on both mobile and desktop views.
    - Ensure a consistent color scheme and adequate spacing between elements using Bootstrap classes.
4. **Testing (Bonus Challenge):**
    - Write unit tests for both the **ButtonCounter** and **App** components.
    - The tests for **ButtonCounter** should ensure that:
    - It displays the correct count.
    - It correctly increments the count on being clicked.
    - It resets the count when the reset signal is received.
    - The tests for the **App** component should verify:
    - The reset functionality works across all child components.
    - Individual button counts and any global state are correctly reset.

**Key Concepts and Learning Outcomes:**

- **State Management:** Understand how to effectively manage and manipulate component state in React.
- **Props and Communication:** Learn how to pass data and signals between parent and child components.
- **Component Lifecycle and Effects:** Use the **useEffect** hook to listen to changes in props and adjust component state accordingly.
- **Testing in React:** Gain exposure to testing techniques and practices in a React environment.

**Tips:**

- Always consider the separation of concerns principle. Each component should have a clear and distinct responsibility.
- Think about how to efficiently propagate the reset signal from the parent to all child components without unnecessary re-renders.
- Ensure your application is modular, making it easy to expand or modify in the future.

**Detailed Instructions:**

1. **Setup:**

- Create a new React project using the Create React App command-line tool:

  ***npx create-react-app button-counter-app***

- Install Bootstrap:

  ***npm install bootstrap***

2. **ButtonCounter Component:**

- Create **ButtonCounter.js** inside the **src** directory.
- Design the component to receive two props:
  1. **number** - The label for the button.
  2. **onButtonClick** - A callback function to notify the parent component when a button is clicked.
- Use the **useState** hook to maintain a local state (**hitCount**) representing the number of times the button has been clicked.
- Render a button that, when clicked, increases the **hitCount** and calls the **onButtonClick** callback with the button's number.

```
// Sudo code for ButtonCounter.js
import necessary libraries and hooks
function ButtonCounter(props) {
    initialize hitCount with useState

    function handleClick() {
        increase hitCount
        call onButtonClick with the button's number
    }

    return button with number and hitCount
}
export ButtonCounter
```

3. **App Component:**

- Modify **App.js** to import **ButtonCounter** and Bootstrap's CSS.
- Initialize a state (**totalSum**) to keep track of the cumulative sum of the clicked button numbers.
- Create a function (**handleButtonClick**) that will update the **totalSum** state.
- Render 9 **ButtonCounter** components. Pass down the unique number and the **handleButtonClick** function as props to each.
- Display the total sum below the grid of buttons.

```
// Sudo code for App.js
import React, { useState } from 'react';
import ButtonCounter from './ButtonCounter'; // Assuming ButtonCounter is in the same directory
```

```
const App = () => {
  // State to manage the reset signal for all buttons
  const [resetSignal, setResetSignal] = useState(false);

  // Function to handle reset button click
  const handleReset = () => {
    // Logic to handle the reset functionality (will be fleshed out later)
  };

  return (
    <div className="container mt-5">
      <div className="row">
        {/* Dynamically render 9 ButtonCounter components */}
        {[...Array(9)].map((_, index) => (
          <div className="col-4 mb-3" key={index}>
            {/* Passing resetSignal prop to each child component */}
            <ButtonCounter buttonNumber={index + 1} resetSignal={resetSignal} />
          </div>
        ))}
      </div>

      {/* Reset Button */}
      <div className="text-center">
        <button className="btn btn-danger mt-3" onClick={handleReset}>Reset All</button>
      </div>
    </div>
  );
}
export default App;
```

The foundational code for the **ButtonCounter** child component, excluding the core business logic:

```
import React, { useState, useEffect } from 'react';
const ButtonCounter = (props) => {
  // State to manage the individual button click count
  const [clickCount, setClickCount] = useState(0);

  // useEffect to listen for the reset signal from the parent App component
  useEffect(() => {
    // Logic to reset the click count if the reset signal is received (will be fleshed out later)
  }, [props.resetSignal]);

  const handleClick = () => {
```

```
    // Logic to handle the button click (will be fleshed out later)
  };

  return (
    <div className="card">
      <div className="card-body text-center">
        {/* Display the click count */}
        <p>Clicks Count: {clickCount}</p>
        <hr />

        {/* Button with an onClick handler */}
        <button className="btn btn-primary" onClick={handleClick}>
          Button {props.buttonNumber}
        </button>
      </div>
    </div>
  );
}

export default ButtonCounter;
```

4. **Styling:**
    - Make use of Bootstrap classes to style your components. Each row should contain 3 buttons, so you'd utilize the Bootstrap grid system with **col-4** for each button.
    - You can enhance the styling by adding custom CSS or using more Bootstrap utilities.
5. **Testing:**
    - Start the React development server:

        ***npm start***

- Ensure that each button's click count updates correctly.
- Verify that the total sum updates appropriately when any button is clicked.

**Evaluation Criteria:**

- Correct implementation and functionality.
- Proper management of state (**props** and **state**).
- Effective integration of Bootstrap for responsive design.
- Code organization, readability, and consistency.

By the end of this assignment, you should have a deeper understanding of state management in React, component communication, and integration of external libraries like Bootstrap for styling. Happy coding!

## Need of child component? what kind of logic can move to child component and parent component from above assignment?

In the React framework, the concept of "components" allows for the breakdown of UI and associated logic into reusable and manageable pieces. This modular approach provides several advantages:

1. **Reusability:** Components can be reused across different parts of an application or even in other projects.
2. **Separation of Concerns:** By breaking down UI and logic into components, you can ensure that each component has a single responsibility, making the code more readable and maintainable.
3. **Performance Optimizations:** With the proper use of React's state and props, you can control when components re-render, optimizing performance.

In the context of the assignment:

**The Child Component (ButtonCounter):**

The **ButtonCounter** is designed to encapsulate the logic and UI related to individual button interactions. Here's what it's responsible for:

1. **Displaying the Button:** The component displays a button with its assigned number.
2. **Track Click Count:** The component manages a local state (**hitCount**) that keeps track of how many times that specific button has been clicked.
3. **UI Updates:** It updates the UI to reflect the number of times the button was clicked.
4. **Interaction:** It notifies the parent component (**App**) when the button is clicked to allow the parent to update the total sum.

**The Parent Component (App):**

The **App** component is designed to coordinate the overall layout and manage the aggregated data. Here's what it's responsible for:

1. **Rendering Child Components:** It renders multiple **ButtonCounter** components, each with a unique number.
2. **Track Overall Sum:** The component keeps track of the total sum of all the numbers of the clicked buttons.
3. **Reset Functionality:** It provides a mechanism to reset both the individual button counts and the overall sum.

**Logic Distribution:**

- **Child (ButtonCounter):** Local state and behaviors that pertain to individual button interactions should reside in the child. This includes:
    - Managing individual button click counts.
    - Rendering the button with its assigned number.
    - Notifying the parent when a button is clicked.
- **Parent (App):** Global or aggregated behaviors that need a holistic view of all child components should reside in the parent. This includes:
    - Managing the total sum of clicked button numbers.
    - Rendering all the **ButtonCounter** components.
    - Resetting the total sum and all individual button counts.

This separation of logic makes the components more understandable and reusable. For example, if you wanted to create another screen where buttons behaved similarly but had a different overall behavior, you could reuse the **ButtonCounter** without any changes.