NAME: JANHVI SINGH

REGISTRATION NUMBER: 22BCE8544

Assignment 15

NameForm.jsx

// src/NameForm.jsx

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

const NameForm = () => {

  const inputRef = useRef(null); // useRef to reference the DOM element

  const initialNameRef = useRef('Guest'); // useRef to store a persistent initial value

  const [submittedName, setSubmittedName] = useState('');

  const handleSubmit = (e) => {

    e.preventDefault();

    if (inputRef.current) {

      setSubmittedName(inputRef.current.value);

    }

  };

  return (

    <div className="component-container">

      <h2>1. Using `useRef` to Store Persistent Values</h2>

      <p>Initial name (persisted with useRef): <strong>{initialNameRef.current}</strong></p>

      <form onSubmit={handleSubmit}>

        <label htmlFor="nameInput">Enter your name:</label>

        <input

          id="nameInput"

          type="text"

          ref={inputRef}

          defaultValue={initialNameRef.current} // Set initial value from useRef

          placeholder="e.g., John Doe"

        />

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

      </form>

      {submittedName && <p>Submitted Name: <strong>{submittedName}</strong></p>}

      <p>

        \*`useRef` is used to store `initialNameRef.current` which persists across renders.

        It also references the input DOM element to get its value on submission.\*

      </p>

    </div>

  );

};

export default NameForm;

WatchEffect.jsx

// src/WatchEffect.jsx

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

const WatchEffect = () => {

  const [inputValue, setInputValue] = useState('');

  // Custom "watch" effect using useEffect

  useEffect(() => {

    console.log('WatchEffect: Input value changed to:', inputValue);

    // You could also display it on the screen:

    // document.getElementById('watched-value-display').textContent = inputValue;

  }, [inputValue]); // Dependency array: runs when inputValue changes

  const handleChange = (e) => {

    setInputValue(e.target.value);

  };

  return (

    <div className="component-container">

      <h2>2. Implement a Custom "Watch" Effect with `useEffect`</h2>

      <label htmlFor="watchInput">Type something:</label>

      <input

        id="watchInput"

        type="text"

        value={inputValue}

        onChange={handleChange}

        placeholder="Type here..."

      />

      <p>Current Input Value: <strong>{inputValue}</strong></p>

      <p>

        \*Check your browser's console to see the `useEffect` logging changes to the input value.\*

      </p>

      <p id="watched-value-display"></p> {/\* Optional: for displaying on screen \*/}

    </div>

  );

};

export default WatchEffect;

useWatch:

// src/useWatch.jsx

import { useState, useEffect } from 'react';

/\*\*

 \* A custom hook to observe and return the latest value of a specific state.

 \* Similar in concept to a "watch" effect in other frameworks,

 \* but specifically for tracking a single value.

 \*

 \* @param {any} value The value to watch for changes.

 \* @returns {any} The latest watched value.

 \*/

const useWatch = (value) => {

  const [watchedValue, setWatchedValue] = useState(value);

  useEffect(() => {

    // Only update watchedValue if the provided value has actually changed

    // This is similar to how a framework's watch might behave,

    // ensuring we only capture specific changes.

    if (watchedValue !== value) {

      setWatchedValue(value);

      console.log('useWatch Hook: Value has changed to:', value);

    }

  }, [value, watchedValue]); // Depend on both value and watchedValue to ensure correct update logic

  return watchedValue;

};

export default useWatch;

WatchInput:

// src/WatchInput.jsx

import React, { useState } from 'react';

import useWatch from './usewatch';

const WatchInput = () => {

  const [inputValue, setInputValue] = useState('');

  const watchedInput = useWatch(inputValue); // Use our custom useWatch hook

  const handleChange = (e) => {

    setInputValue(e.target.value);

  };

  return (

    <div className="component-container">

      <h2>3. Implement a Custom `useWatch` Hook</h2>

      <label htmlFor="trackInput">Track this input:</label>

      <input

        id="trackInput"

        type="text"

        value={inputValue}

        onChange={handleChange}

        placeholder="Watch me change..."

      />

      <p>Current Input Value (from state): <strong>{inputValue}</strong></p>

      <p>Watched Input Value (from `useWatch` hook): <strong>{watchedInput}</strong></p>

      <p>

        \*The `useWatch` hook specifically tracks and updates `watchedInput` only when `inputValue` changes.

        Check console for `useWatch` logs.\*

      </p>

    </div>

  );

};

export default WatchInput;

FocusInput:

// src/FocusInput.jsx

import React, { useRef } from 'react';

const FocusInput = () => {

  const inputToFocusRef = useRef(null);

  const handleFocusClick = () => {

    if (inputToFocusRef.current) {

      inputToFocusRef.current.focus(); // Directly manipulate DOM to focus the input

      inputToFocusRef.current.value = 'Focused!'; // Optionally set a value

    }

  };

  return (

    <div className="component-container">

      <h2>4. Focus Input Field Using `useRef` (DOM Manipulation)</h2>

      <label htmlFor="domInput">Interact with me:</label>

      <input

        id="domInput"

        type="text"

        ref={inputToFocusRef} // Attach useRef to the input element

        placeholder="I can be focused..."

      />

      <button onClick={handleFocusClick}>Focus the Input</button>

      <p>

        \*Clicking the button uses `useRef` to directly call the `focus()` method on the input field's DOM element.\*

      </p>

    </div>

  );

};

export default FocusInput;

App.jsx

// src/App.jsx

import React from 'react';

import NameForm from './NameForm';

import WatchEffect from './WatchEffect';

import WatchInput from './WatchInput';

import FocusInput from './FocusInput';

import './App.css'; // Import the CSS file

function App() {

  return (

    <div className="App">

      <h1>React Hooks Demo: `useRef`, `useEffect`, `useWatch`</h1>

      <div className="components-grid">

        <NameForm />

        <WatchEffect />

        <WatchInput />

        <FocusInput />

      </div>

    </div>

  );

}

export default App;

App.css

/\* src/App.css \*/

body {

  font-family: Arial, sans-serif;

  line-height: 1.6;

  margin: 0;

  padding: 0;

  background-color: #f4f4f4;

  color: #333;

}

.App {

  max-width: 1200px;

  margin: 20px auto;

  padding: 20px;

  background-color: #fff;

  border-radius: 8px;

  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);

}

h1 {

  text-align: center;

  color: #2c3e50;

  margin-bottom: 40px;

}

.components-grid {

  display: grid;

  grid-template-columns: repeat(auto-fit, minmax(300px, 1fr));

  gap: 30px;

}

.component-container {

  padding: 25px;

  border: 1px solid #e0e0e0;

  border-radius: 8px;

  background-color: #fcfcfc;

  box-shadow: 0 1px 3px rgba(0, 0, 0, 0.05);

}

.component-container h2 {

  color: #34495e;

  margin-top: 0;

  margin-bottom: 20px;

  font-size: 1.5em;

}

.component-container label {

  display: block;

  margin-bottom: 8px;

  font-weight: bold;

}

.component-container input[type="text"] {

  width: calc(100% - 20px);

  padding: 10px;

  margin-bottom: 15px;

  border: 1px solid #ccc;

  border-radius: 4px;

  box-sizing: border-box;

}

.component-container button {

  padding: 10px 15px;

  background-color: #007bff;

  color: white;

  border: none;

  border-radius: 4px;

  cursor: pointer;

  font-size: 1em;

  transition: background-color 0.2s ease;

}

.component-container button:hover {

  background-color: #0056b3;

}

.component-container p {

  margin-top: 15px;

  font-size: 0.9em;

  color: #666;

}

.component-container p strong {

  color: #333;

}

README

**# React Watch Effect Demo**

This project demonstrates the usage of `useRef`, `useEffect`, and a custom `useWatch` hook in React to manage form inputs, track state changes, and perform direct DOM manipulation.

**## How to Run**

1.  Clone the repository:

    `git clone <repository-url>`

2.  Navigate to the project directory:

    `cd react-watch-effect-demo`

3.  Install dependencies:

    `npm install`

4.  Start the development server:

    `npm run dev`

The application will be accessible at `http://localhost:5173/` (or another port as indicated by Vite).

**## Assignment Tasks Implementation**

**### 1. Using `useRef` to Store Persistent Values (`src/NameForm.jsx`)**

\* **\*\*Purpose\*\***: `useRef` is used to store values that persist across component renders without causing re-renders when the ref's `.current` value changes. It's also used to directly reference DOM elements.

\* **\*\*Implementation\*\***:

    \* `initialNameRef = useRef('Guest')`: Stores the initial name "Guest" persistently. Its value (`initialNameRef.current`) remains available even after re-renders, and changing it doesn't trigger a re-render.

    \* `inputRef = useRef(null)`: This ref is attached to the input element (`<input ref={inputRef} />`). When the form is submitted, `inputRef.current.value` is accessed to get the current value of the input field, demonstrating direct DOM access for reading its value.

**### 2. Implement a Custom "Watch" Effect with `useEffect` (`src/WatchEffect.jsx`)**

\* **\*\*Purpose\*\***: `useEffect` is used to create side effects in functional components. By providing a dependency array, it can "watch" specific state variables or props and run the effect callback whenever they change.

\* **\*\*Implementation\*\***:

    \* `const [inputValue, setInputValue] = useState('')`: A state variable to hold the current input value.

    \* `useEffect(() => { console.log('WatchEffect: Input value changed to:', inputValue); }, [inputValue]);`: This `useEffect` hook acts as a "watch" effect. The callback function inside `useEffect` will execute every time `inputValue` (specified in the dependency array) changes. This logs the new value to the console, demonstrating a simple way to react to state changes.

**### 3. Implement a `useWatch` Hook (`src/useWatch.jsx` and `src/WatchInput.jsx`)**

\* **\*\*Purpose\*\***: The custom `useWatch` hook is designed to specifically observe and return the latest value of a particular state, providing a dedicated mechanism for tracking changes.

\* **\*\*Implementation\*\***:

    \* **\*\*`src/useWatch.jsx` (Custom Hook Definition)\*\***:

        \* It takes `value` as an argument, which is the state variable we want to watch.

        \* It maintains an internal `watchedValue` state.

        \* The `useEffect` inside `useWatch` compares the incoming `value` with its internal `watchedValue`. If they differ, it updates `watchedValue` and logs the change. This ensures that the `useWatch` hook only signals a change when the actual `value` passed to it has changed.

    \* **\*\*`src/WatchInput.jsx` (Usage)\*\***:

        \* `const watchedInput = useWatch(inputValue);`: The `useWatch` hook is called with `inputValue` as the value to track.

        \* `watchedInput` will then hold the latest value of `inputValue` as reported by the `useWatch` hook, and it's displayed on the page. This clearly differentiates the current state from the value explicitly "watched" by the custom hook.

**### 4. Focus Input Field Using `useRef` (DOM Manipulation) (`src/FocusInput.jsx`)**

\* **\*\*Purpose\*\***: `useRef` is not only for persisting values but also for getting a direct reference to a DOM element. This allows for imperative DOM manipulations that are sometimes necessary (e.g., focusing an input, playing media, measuring elements).

\* **\*\*Implementation\*\***:

    \* `const inputToFocusRef = useRef(null)`: A ref is created and attached to the input element (`<input ref={inputToFocusRef} />`).

    \* `handleFocusClick` function: When the "Focus the Input" button is clicked, `inputToFocusRef.current.focus()` is called. This directly calls the `focus()` method available on the native HTML input element, causing the input field to gain focus. This demonstrates direct, imperative DOM manipulation via `useRef`.

**## Learning Outcomes**

By completing this assignment, I have:

\* **\*\*`useRef` Mastery\*\***: Gained a clear understanding of how `useRef` can be used to store and persist values across component renders without triggering re-renders, and crucially, how to use it to obtain and interact with direct references to DOM elements for imperative manipulations.

\* **\*\*Custom Watch Effect with `useEffect`\*\***: Solidified knowledge of `useEffect`'s role in creating side effects and how its dependency array can be leveraged to "watch" specific state or prop changes, enabling reactive behavior in components.

\* **\*\*`useWatch` Hook Implementation\*\***: Experienced implementing a custom `useWatch` hook, understanding the logic behind observing specific value changes and encapsulating this behavior into a reusable hook, similar to patterns seen in state management libraries.

\* **\*\*Efficient DOM Manipulation\*\***: Developed practical skills in performing necessary direct DOM manipulations efficiently and correctly in React using `useRef`, understanding when and why such direct access might be required while adhering to React's declarative nature.

Screenshot

