Before moving to Machine coding problems we will see one hook i.e the useRef hook and then we will solve these problems

StopWatch
Automatic Image Carousel
Creating your Own Hook (Custom Hook)

title: useRef Hook

The useRef hook in React is a powerful tool for directly accessing and interacting with DOM elements and for persisting values across renders without causing re-renders.

Unlike state, updating a useRef value doesn't trigger a component re-render.

how useRef works

1. Creating a Reference: You can create a reference using useRef(initialValue). This returns a mutable object with a current property, which is initialized to the passed initialValue.
2. Persisting Values: Unlike state, changes to the current property of the ref do not trigger a re-render. This makes refs perfect for

storing values that should persist across renders without affecting the UI.

3. Accessing DOM Elements: You can assign a ref to a DOM element using the ref attribute in JSX. This allows you to directly interact with the DOM element, similar to document.querySelector in vanilla JavaScript.

Example: Using useRef to Access a DOM Element

```
import React, { useEffect, useRef } from 'react';

function FocusInput() {
 const inputRef = useRef(null);

 const focusInput = () => {
   inputRef.current.focus();
 };
// can add useEffect for auto focus
// useEffect(() => {
//     inputRef.current.focus();
// },[]);

 return (
   <div>
     <input ref={inputRef} type="text" />
     <button onClick={focusInput}>Focus Input</button>
   </div>
 );
}

export default FocusInput;
```

In this example:

1. useRef(null) creates a ref object with a current property initialized to null.
2. The ref attribute is used to attach this ref to the <input> element.
3. When the button is clicked, inputRef.current.focus() is called, which focuses the input field.

Example: Persisting Values Across Renders

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

function Timer() {
 const [seconds, setSeconds] = useState(0);
 const intervalRef = useRef(null);

 useEffect(() => {
   intervalRef.current = setInterval(() => {
     setSeconds((prevSeconds) => prevSeconds + 1);
   }, 1000);

   return () => {
     clearInterval(intervalRef.current);
   };
 }, []);

 return (
   <div>
     <p>Seconds: {seconds}</p>
     <button onClick={() => clearInterval(intervalRef.current)}>Stop Timer</button>
   </div>
 );
}

export default Timer;
```

In this example:

1. useRef(null) creates a ref to store the interval ID.
2. intervalRef.current is set to the interval ID returned by setInterval.
3. This ref persists across renders without causing re-renders, allowing the timer to run in the background.

Use Cases for useRef

1. Accessing DOM Elements: When you need to interact directly with a DOM element, such as focusing an input, measuring dimensions, or manipulating the element in other ways.
2. Storing Mutable Values: When you need to store a mutable value that persists across renders but doesn't need to cause a re-render when updated. Examples include timers, intervals, previous state values, or any other non-UI related data.
3. Avoiding Re-Initialization: When you want to initialize a value only once and keep it around across renders, like initializing a library or maintaining a stable reference to a callback function.

Conclusion

The useRef hook is a versatile tool in React for dealing with direct DOM manipulations and persisting values without triggering re-renders. It is particularly useful in scenarios where you need a persistent, mutable reference or direct access to DOM elements.

title: StopWatch

Create a stopwatch application using React. The stopwatch should have the following features:

Start the timer.
Stop the timer.
Reset the timer.
Display the elapsed time in a format of hours:minutes:seconds.

Steps to Create the Stopwatch

1. Set Up the React Project
    a. First, ensure you have Node.js and npm installed. Then, create a new React project using Create React App:
    b. Npm create vite@latest
    c. cd stopwatch
    d. npm run dev

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

const Stopwatch = () => {
 const [time, setTime] = useState(0);
 const [isRunning, setIsRunning] = useState(false);
 const timerRef = useRef(null);

 const startTimer = () => {
   if (!isRunning) {
     setIsRunning(true);
     timerRef.current = setInterval(() => {
       setTime((prevTime) => prevTime + 1);
     }, 1000);
   }
```

```
  };

  const stopTimer = () => {
    if (isRunning) {
      clearInterval(timerRef.current);
      setIsRunning(false);
    }
  };

  const resetTimer = () => {
    clearInterval(timerRef.current);
    setIsRunning(false);
    setTime(0);
  };

  const formatTime = (time) => {
    const getSeconds = `0${time % 60}`.slice(-2);
    const minutes = Math.floor(time / 60);
    const getMinutes = `0${minutes % 60}`.slice(-2);
    const getHours = `0${Math.floor(time / 3600)}`.slice(-2);
    return `${getHours}:${getMinutes}:${getSeconds}`;
  };

  return (
    <div>
      <h1>{formatTime(time)}</h1>
      <button onClick={startTimer}>Start</button>
      <button onClick={stopTimer}>Stop</button>
      <button onClick={resetTimer}>Reset</button>
    </div>
  );
};

export default Stopwatch;
```

Explanation

1. useState: This hook is used to manage the state of the time and the running status of the stopwatch.

2. time stores the elapsed time in seconds.
3. isRunning keeps track of whether the stopwatch is currently running.
4. useRef: This hook is used to store a reference to the timer interval. It allows us to keep track of the timer between renders without causing re-renders.
5. startTimer Function: This function starts the stopwatch.
    a. It checks if the stopwatch is not already running.
    b. If not running, it sets isRunning to true and starts an interval that increments the time state every second (1000 milliseconds).
6. stopTimer Function: This function stops the stopwatch.
    a. It checks if the stopwatch is running.
    b. If running, it clears the interval and sets isRunning to false.
7. resetTimer Function: This function resets the stopwatch.
    a. It clears the interval, sets isRunning to false, and resets the time state to 0.
8. formatTime Function: This function formats the elapsed time into a human-readable format (hh:mm:ss).
    a. time % 60 computes the remaining seconds when time is divided by 60.
    b. Adding 0 in front (0${time % 60}) ensures that any single-digit second (e.g., 5) becomes a two-digit string (e.g., 05).

c. .slice(-2) extracts the last two characters of the string, ensuring that the result is always two characters long. This handles both single-digit and double-digit seconds correctly.

Optimizations

1. Every time the component renders, the functions are getting created

```
const startTimer = useCallback(() => {
  if (!isRunning) {
    setIsRunning(true);
    timerRef.current = setInterval(() => {
      setTime((prevTime) => prevTime + 1);
    }, 1000);
  }
}, [isRunning]);

const stopTimer = useCallback(() => {
  if (isRunning) {
    clearInterval(timerRef.current);
    setIsRunning(false);
  }
}, [isRunning]);
```

title: Automatic Image Carousel

Create a simple image carousel component in React that automatically cycles through a list of images, displaying one image at a time. The carousel should also allow users to manually navigate to the next or previous image using buttons.

Features:

Automatically cycles through images every 2 seconds.
Manual navigation to the next or previous image using buttons.
Displays image, title, and description for each item.

Step 1: Setting Up the Component

First, you need to import the necessary hooks and define the list of items (images, titles, and descriptions) that will be displayed in the carousel.

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

const items = [
 {
   id: 1,
   imageUrl:
'https://images.pexels.com/photos/14286166/pexels-photo-14286166.jpeg?auto=compress&cs
=tinysrgb&w=1260&h=750&dpr=1',
   title: 'Item 1',
   description: 'Description of item 1',
 },
 {
   id: 2,
   imageUrl:
'https://images.pexels.com/photos/13455799/pexels-photo-13455799.jpeg?auto=compress&cs
=tinysrgb&w=1260&h=750&dpr=1',
   title: 'Item 2',
   description: 'Description of item 2',
 },
 {
   id: 3,
   imageUrl:
'https://images.pexels.com/photos/15582923/pexels-photo-15582923.jpeg?auto=compress&cs
=tinysrgb&w=1260&h=750&dpr=1',
   title: 'Item 3',
   description: 'Description of item 3',
```

```
 },
];
```

Step 2: Initializing State

```
const Carousel = () => {
   const [currentItem, setCurrentItem] = useState(0);
}
export default Carousel;
```

Step 3: Navigation Functions

Define functions to navigate to the next and previous items in the carousel.

```
const Carousel = () => {
   const [currentItem, setCurrentItem] = useState(0);
   function nextItem() {
       if (currentItem === items.length - 1) {
         setCurrentItem(0);
       } else {
         setCurrentItem((curr) => curr + 1);
       }
     }

     function prevItem() {
       if (currentItem === 0) {
         setCurrentItem(items.length - 1);
       } else {
         setCurrentItem((curr) => curr - 1);
       }
     }
}
export default Carousel;
```

```
```

Step 4: Rendering the Carousel

```
return (
  <div className="carousel">
    <button onClick={prevItem}>Prev</button>
    <div className="carousel-item">
      <img
        height="300"
        width="400"
        src={items[currentItem].imageUrl}
        alt={items[currentItem].title}
      />
      <h2>{items[currentItem].title}</h2>
      <p>{items[currentItem].description}</p>
    </div>
    <button onClick={nextItem}>Next</button>
  </div>
);
```

Step 5: Automatic Cycling with useEffect

Use the useEffect hook to automatically cycle through the images every 2 seconds. Clear the interval when the component unmounts to avoid memory leaks.

```
useEffect(() => {
  const timer = setInterval(() => {
    nextItem();
  }, 2000);
  return () => clearInterval(timer);
}, [currentItem]);
```

title: Custom Hook

What are custom hooks

sometimes, you have logic that needs to be reused across multiple components. Without custom hooks, you might end up copying and pasting the same logic, which is inefficient and hard to maintain.

Solution: Custom hooks allow you to encapsulate and reuse logic, making your code more modular and maintainable.

Key Characteristics of Custom Hooks

1.Function Definition: Custom hooks are JavaScript functions, often prefixed with the word use, following the naming convention of React hooks (e.g., useCustomHook).
2.Internal Use of React Hooks: Inside a custom hook, you can call other React hooks such as useState, useEffect, useContext, etc., to manage state, perform side effects, or use context.
3.Reusable Logic: Custom hooks encapsulate reusable logic. Instead of duplicating code across multiple components, you can create a custom hook and use it wherever needed.
4.Maintain State and Side Effects: Custom hooks can maintain their own state and side effects, just like components, making it easier to manage complex interactions and stateful logic.

Problem Statement

You need to create a custom hook in React that manages the visibility of an element, such as a modal or dropdown. This custom hook, useVisibility, should provide a simple interface to show and hide the element and should be reusable across different components.

Features Required

Initial Visibility State: The custom hook should allow setting an initial visibility state.
Toggle Visibility: The hook should provide a method to toggle the visibility state.
Show and Hide Methods: The hook should provide methods to explicitly show or hide the element.
Visibility State: The hook should return the current visibility state.

Let's create our custom hook, useVisibility.

```
// src/useVisibility.js

import { useState, useCallback } from 'react';

function useVisibility(initialVisibility = false) {
 const [isVisible, setIsVisible] = useState(initialVisibility);

 const show = useCallback(() => {
   setIsVisible(true);
 }, []);
```

```
  const hide = useCallback(() => {
    setIsVisible(false);
  }, []);

  const toggle = useCallback(() => {
    setIsVisible(prev => !prev);
  }, []);

  return {
    isVisible,
    show,
    hide,
    toggle,
  };
}

export default useVisibility;
```

Using the Custom Hook

1. Create a Modal Component

```
// src/Modal.js

import React from 'react';
import './Modal.css'; // Create a CSS file for styling

function Modal({ isVisible, hide }) {
  if (!isVisible) return null;

  return (
    <div className="modal-overlay">
      <div className="modal">
        <h2>Modal Title</h2>
        <p>This is a modal.</p>
        <button onClick={hide}>Close</button>
      </div>
    </div>
  );
}
```

```
export default Modal;
```

2. Create the CSS for Modal

```
/* src/Modal.css */

.modal-overlay {
   position: fixed;
   top: 0;
   left: 0;
   width: 100%;
   height: 100%;
   background: rgba(0, 0, 0, 0.5);
   display: flex;
   align-items: center;
   justify-content: center;
 }
  .modal {
   background: white;
   padding: 20px;
   border-radius: 5px;
   box-shadow: 0 2px 10px rgba(0, 0, 0, 0.1);
 }
```

3. Modify App.jsx to use Modal and the custom hook

```
function App() {
 const [count, setCount] = useState(0)
 const { isVisible, show, hide, toggle } = useVisibility(false);

 return (
   <>
     {/* <Stopwatch /> */}
     {/* <Carousel /> */}
   <div className="App">
     <h1>Custom Hook Example</h1>
      <button onClick={show}>Show Modal</button>
      <button onClick={toggle}>Toggle Modal</button>
      <Modal isVisible={isVisible} hide={hide} />
   </div>
```

```
    </>
  )
}
```

useVisibility: Initializes the visibility state of the modal.
show, hide, toggle: Methods to control the modal's visibility.
Modal Component: Receives isVisible and hide as props to conditionally render the modal and close it.

Finalizing
Your custom hook useVisibility is now complete and reusable. You can use it to manage the visibility state of any element in your React application.