useLocalStorageWithExpiry:

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

const useLocalStorageWithExpiry = (key, defaultValue, expiryInMinutes) => {
  const [value, setValue] = useState(() => {
    const item = window.localStorage.getItem(key);
    if (item) {
      const parsedItem = JSON.parse(item);
      if (parsedItem.expiry && Date.now() > parsedItem.expiry) {
        window.localStorage.removeItem(key);
        return defaultValue;
      }
      return parsedItem.value;
    }
    return defaultValue;
  });

  useEffect(() => {
    const expiryTime = Date.now() + expiryInMinutes * 60 * 1000;
    const item = { value, expiry: expiryTime };
    window.localStorage.setItem(key, JSON.stringify(item));
  }, [key, value, expiryInMinutes]);

  return [value, setValue];
};

export default useLocalStorageWithExpiry;
```

Explanation:

The `useLocalStorageWithExpiry` hook allows you to store data in the browser's `localStorage` with an expiry time. It takes three parameters: `key` (the key for storing the data), `defaultValue` (the initial value if no value is found in `localStorage`), and `expiryInMinutes` (the duration in minutes after which the stored value should expire). It returns a stateful value and a function to update that value. The hook retrieves the stored value from `localStorage` and checks if it has expired. If it has expired, the value is removed from `localStorage` and the default value is returned. Otherwise, the stored value is returned. Whenever the value changes, it is stored in `localStorage` with the expiry time.

Use Case:

This hook can be useful when you want to store temporary data in the browser that automatically expires after a certain period. For example, you can use it to store a user's session token that needs to be valid only for a specific duration.

usePrevious:

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

const usePrevious = (value) => {
  const ref = useRef();

  useEffect(() => {
    ref.current = value;
  }, [value]);

  return ref.current;
};

export default usePrevious;
```

Explanation:

The `usePrevious` hook allows you to get the previous value of a state or prop. It uses the `useRef` hook to create a mutable ref object. The effect updates the ref object whenever the value changes. The previous value is stored in the ref object, which is then returned.

Use Case:

This hook can be useful when you need to compare the current and previous values of a state or prop. For example, you can use it to track changes in a specific prop and perform some actions based on the comparison.

useOnlineStatus:

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

const useOnlineStatus = () => {
  const [isOnline, setIsOnline] = useState(navigator.onLine);

  useEffect(() => {
    const handleOnline = () => setIsOnline(true);
    const handleOffline = () => setIsOnline(false);
    window.addEventListener('online', handleOnline);
    window.addEventListener('offline', handleOffline);

    return () => {
      window.removeEventListener('online', handleOnline);
      window.removeEventListener('offline', handleOffline);
    };
  }, []);

  return isOnline;
};

export default useOnlineStatus;
```

Explanation:

The `useOnlineStatus` hook allows you to detect the online/offline status of the user's browser. It initializes the `isOnline` state with the current online status using `navigator.onLine

`. It sets up event listeners for the `online` and `offline` events and updates the `isOnline` state accordingly. The effect runs only once (on mount) and removes the event listeners when the component unmounts.

Use Case:

This hook can be useful when you want to provide real-time feedback to the user based on their online/offline status. For example, you can show a notification when the user loses internet connectivity.

useIdle:

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

const useIdle = (timeoutInMs) => {
  const [isIdle, setIsIdle] = useState(false);

  useEffect(() => {
    let timeoutId;

    const handleIdle = () => setIsIdle(true);
    const handleActive = () => {
      setIsIdle(false);
      clearTimeout(timeoutId);
      startTimeout();
    };

    const startTimeout = () => {
      timeoutId = setTimeout(handleIdle, timeoutInMs);
    };

    startTimeout();

    window.addEventListener('mousemove', handleActive);
    window.addEventListener('keydown', handleActive);

    return () => {
      clearTimeout(timeoutId);
      window.removeEventListener('mousemove', handleActive);
      window.removeEventListener('keydown', handleActive);
    };
  }, [timeoutInMs]);

  return isIdle;
};

export default useIdle;
```

Explanation:

The `useIdle` hook allows you to detect when the user becomes idle, i.e., there is no mouse movement or keyboard activity for a specified duration. It takes a `timeoutInMs` parameter representing the duration of inactivity after which the user is considered idle. It initialises the `isIdle` state as `false` and sets up event listeners for `mousemove` and `keydown` events. When there is activity, the `isIdle` state is set to `false`, the previous timeout is cleared, and a new timeout is started. If there is no activity within the specified duration, the `isIdle` state is set to `true`.

Use Case:

This hook can be useful when you want to trigger certain actions or UI changes when the user is idle. For example, you can display a screensaver or log out the user after a period of inactivity.

useDebounce:

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

const useDebounce = (value, delay) => {
  const [debouncedValue, setDebouncedValue] = useState(value);

  useEffect(() => {
    const handler = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    return () => {
      clearTimeout(handler);
    };
  }, [value, delay]);

  return debouncedValue;
};

export default useDebounce;
```

Explanation:

The `useDebounce` hook allows you to debounce a value, which means delaying the update of a value until a certain period of inactivity. It takes a `value` and `delay` as parameters. The hook initializes the `debouncedValue` state with the initial value. Whenever the `value` changes, a timeout is set with the specified `delay`. If the `value` changes again within the `delay`, the previous timeout is cleared. Once the `delay` elapses without any changes to the `value`, the `debouncedValue` is updated.

Use Case:

This hook can be useful when you want to delay the execution of a function or API call until the user has finished typing or interacting with an input field.

useClickOutside:

```
import { useEffect } from 'react';

const useClickOutside = (ref, callback) => {
  const handleClickOutside = (event) => {
    if (ref.current && !ref.current.contains

(event.target)) {
      callback();
    }
  };

  useEffect(() => {
    document.addEventListener('click', handleClickOutside);

    return () => {
      document.removeEventListener('click', handleClickOutside);
    };
  }, [ref, callback]);
};

export default useClickOutside;
```

Explanation:

The `useClickOutside` hook allows you to detect clicks that occur outside a specified ref element. It takes a `ref` and a `callback` function as parameters. The hook adds a click event listener to the document and checks if the target of the click is outside the `ref` element. If it is, the `callback` function is invoked.

Use Case:

This hook can be useful when you want to close a dropdown menu, modal, or any other component when the user clicks outside of it.

useGeolocation:

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

const useGeolocation = () => {
  const [position, setPosition] = useState({ latitude: null, longitude: null });

  const handleSuccess = (position) => {
    setPosition({
      latitude: position.coords.latitude,
      longitude: position.coords.longitude,
    });
  };

  const handleError = (error) => {
    console.error(error);
  };

  useEffect(() => {
    if (navigator.geolocation) {
      navigator.geolocation.getCurrentPosition(handleSuccess, handleError);
    } else {
      console.error('Geolocation is not supported by this browser.');
    }
  }, []);

  return position;
};

export default useGeolocation;
```

Explanation:

The `useGeolocation` hook allows you to retrieve the user's current geolocation coordinates. It initializes the `position` state with `null` latitude and longitude. If the browser supports geolocation, it calls the `getCurrentPosition` method to fetch the user's current position. The retrieved coordinates are then stored in the `position` state.

Use Case:

This hook can be useful when you need to provide location-based features or services in your application, such as displaying the user's current location on a map.

useKeyPress:

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

const useKeyPress = (targetKey) => {
  const [isKeyPressed, setIsKeyPressed] = useState(false);

  const handleKeyDown = ({ key }) => {
    if (key === targetKey) {
      setIsKeyPressed(true);
    }
  };

  const handleKeyUp = ({ key }) => {
    if (key === targetKey) {
      setIsKeyPressed(false);
    }
  };
  useEffect(() => {
    window.addEventListener('keydown', handleKeyDown);
    window.addEventListener('keyup', handleKeyUp);

    return () => {
      window.removeEventListener('keydown', handleKeyDown);
      window.removeEventListener('keyup', handleKeyUp);
    };
  }, [targetKey]);

  return isKeyPressed;
};

export default useKeyPress;
```

Explanation:

The `useKeyPress` hook allows you to detect when a specific key is pressed. It takes a `targetKey` parameter representing the key to detect. It initializes the `isKeyPressed` state as `false` and sets up event listeners for `keydown` and `keyup` events. When the target key is pressed, `isKeyPressed` is set to `true`, and when it is released, `isKeyPressed` is set to `false`.

Use Case:

This hook can be useful when you want to trigger actions based on specific key presses, such as keyboard shortcuts or game controls.

useHover:

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

const useHover = () => {
  const [isHovered, setIsHovered] = useState(false);
  const ref = useRef(null);

  const handleMouseEnter = () => {
    setIsHovered(true);
  };

  const handleMouseLeave = () => {
    setIsHovered(false);
  };

  useEffect(() => {
    const node = ref.current;
    if (node) {
      node.addEventListener('mouseenter', handleMouseEnter);
      node.addEventListener('mouseleave', handleMouseLeave);

      return () => {
        node.removeEventListener('mouseenter', handleMouseEnter);
        node.removeEventListener('mouseleave', handleMouseLeave);
      };
    }
  }, []);

  return [ref, isHovered];
};

export default useHover;
```

Explanation:

The `useHover` hook allows you to detect when an element is being hovered over. It initializes the `isHovered` state as `false` and creates a ref using `useRef` to reference the element. It sets up event listeners for `mouseenter` and `mouseleave` events on the element, updating the `isHovered` state accordingly.

Use Case:

This hook can be useful when you want to apply styles or trigger actions based on whether an element is being hovered over by the user.

Prepared By :
Divyansh Singh (LinkedIn : rgndunes)