Memoized Handler in React (Advanced)

The previous sections have taught you about handlers, callback handlers, and inline handlers. Now we'll introduce a **memoized handler**, which can be applied on top of handlers and callback handlers. For the sake of learning, we will move all the data fetching logic into a standalone function outside the side-effect (A); wrap it into a `useCallback` hook (B); and then invoke it in the `useEffect` hook (C):

```
const App = () => {
  ...

  // A
  const handleFetchStories = React.useCallback(() => {
    if (!searchTerm) return;

    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    fetch(`${API_ENDPOINT}${searchTerm}`)
      .then(response => response.json())
      .then(result => {
        dispatchStories({
          type: 'STORIES_FETCH_SUCCESS',
          payload: result.hits,
        });
      })
      .catch(() =>
        dispatchStories({ type: 'STORIES_FETCH_FAILURE' })
      );

  }, [searchTerm]); // E


  React.useEffect(() => {

    handleFetchStories(); // C
  }, [handleFetchStories]); // D


  ...
};
```

src/App.js

The application behaves the same; only the implementation logic has been

The application behaves the same; only the implementation logic has been refactored. Instead of using the data fetching logic anonymously in a side-effect, we made it available as a function for the application.

Let's explore why React's `useCallback` Hook is needed here. This hook creates a memoized function every time its dependency array (E) changes. As a result, the `useEffect` hook runs again (C) because it depends on the new function (D):

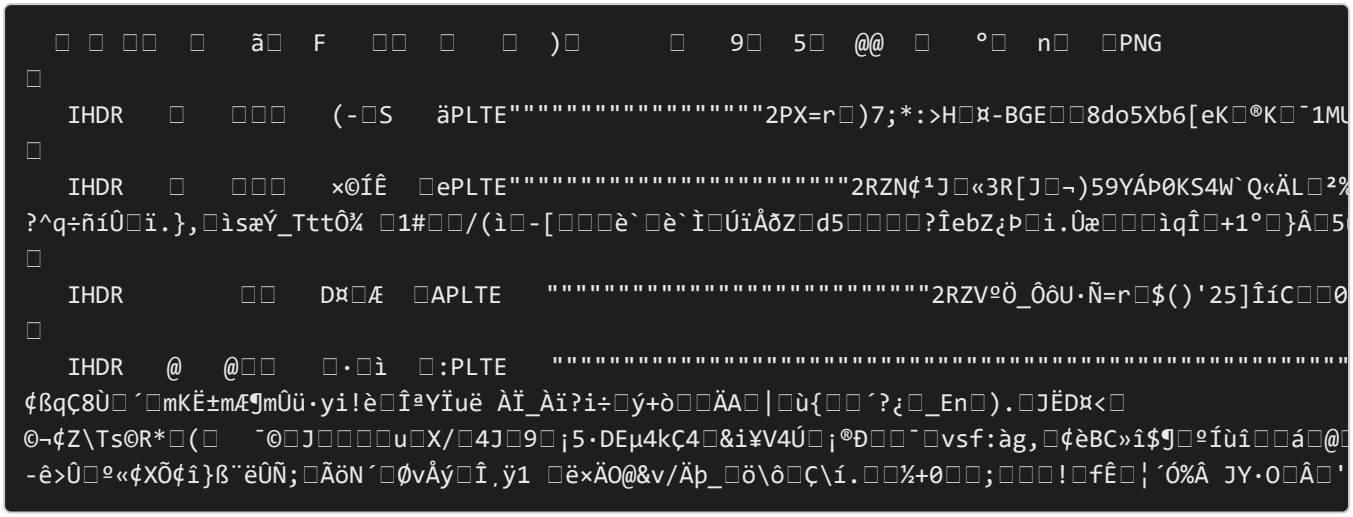
```
1. change: searchTerm
2. implicit change: handleFetchStories
3. run: side-effect
```

If we didn't create a memoized function with React's `useCallback` Hook, a new `handleFetchStories` function would be created with each App component is rendered. The `handleFetchStories` function would be created each time, and would be executed in the `useEffect` hook to fetch data. The fetched data is then stored as state in the component. Because the state of the component changed, the component re-renders and creates a new `handleFetchStories` function. The side-effect would be triggered to fetch data, and we'd be stuck in an endless loop:

```
1. define: handleFetchStories
2. run: side-effect
3. update: state
4. re-render: component
5. re-define: handleFetchStories
6. run: side-effect
...
```

The `useCallback` hook changes the function only when the search term changes. That's when we want to trigger a re-fetch of the data, because the input field has new input and we want to see the new data displayed in our list.

By moving the data fetching function outside the `useEffect` hook, it becomes reusable for other parts of the application. We won't use it just yet, but it is a way to understand the `useCallback` hook. Now the `useEffect` hook runs implicitly when the `searchTerm` changes, because the `handleFetchStories` is re-defined each time the `searchTerm` changes. Since the `useEffect` hook depends on the `handleFetchStories`, the side-effect for data fetching runs again.



Exercises:

- Confirm the changes from the last section.
- Read more about React's useCallback Hook.