Data Re-Fetching in React

Learn to re-fetch data using a dynamic query for the API request.

We'll cover the following

- Exercises:

So far, the App component fetches a list of stories once with a predefined query ('react'). After that, users can search for stories on the client-side. Now we'll move this feature from client-side to server-side searching, using the actual `searchTerm` as a dynamic query for the API request.

First, remove `searchedStories`, because we will receive the stories searched from the API. Pass only the regular stories to the List component:

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

  return (
    <div>
      ...

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (

        <List list={stories.data} onRemoveItem={handleRemoveStory} />
      )}
    </div>
  );
};
```

src/App.js

And second, instead of using a hardcoded search term like before, use the actual `searchTerm` from the component's state. If `searchTerm` is an empty string, do nothing:

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

  React.useEffect(() => {
```

```
    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' })
      );
  }, []);

  ...
};
```

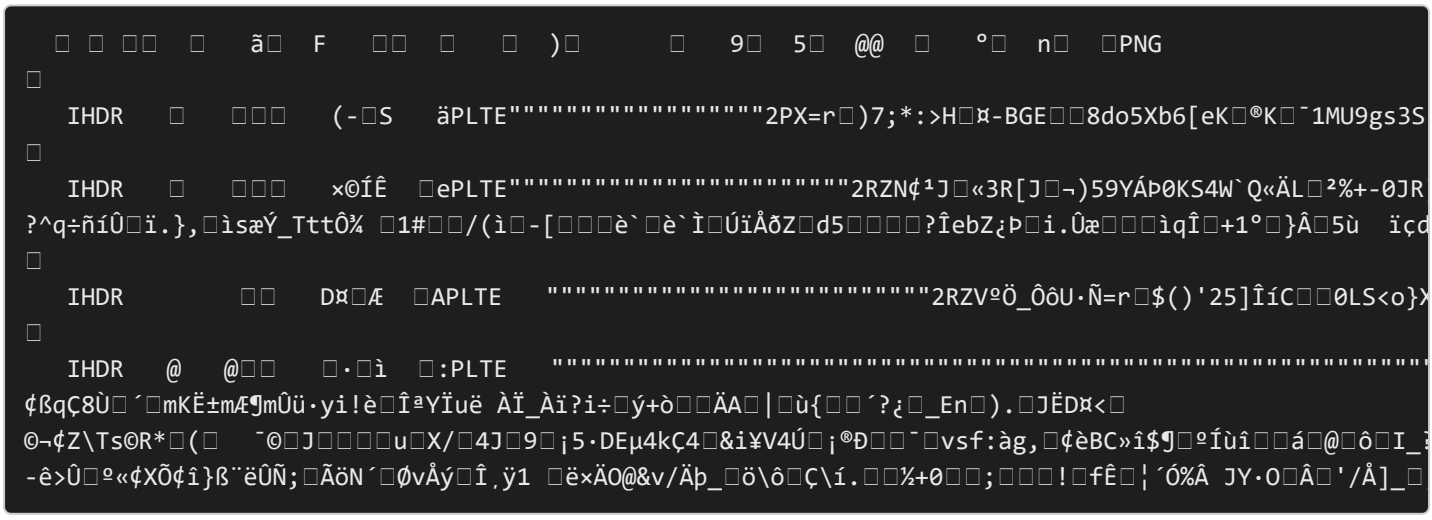
src/App.js

The initial search respects the search term now, so we'll implement data refetching. If the `searchTerm` changes, run the side-effect for the data fetching again. If `searchTerm` is not present (e.g. null, empty string, undefined), do nothing (as a more generalized condition):

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

  React.useEffect(() => {

    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]);


  ...
};
```

src/App.js

We changed the feature from a client-side to server-side search. Instead of filtering a predefined list of stories on the client, the `searchTerm` is used to fetch a server-side filtered list. The server-side search happens not only for the initial data fetching but also for data fetching if the `searchTerm` changes. The feature is fully server-side now.



Re-fetching data each time someone types into the input field isn't optimal, so we'll correct that soon. Because this implementation stresses the API, you might experience errors if you use requests too often.

Exercises:

- Confirm the changes from the last section.