Remember Last Searches

Task: Remember the last five search terms to hit the API, and provide a button to move quickly between searches. When the buttons are clicked, stories for the search term are fetched again.

Optional Hints:

- Don't use a new state for this feature. Instead, reuse the `url` state and `setUrl` state updater function to fetch stories from the API. Adapt them to multiple `urls` as state, and to set multiple `urls` with `setUrls`. The last URL from `urls` can be used to fetch the data, and the last five URLs from `urls` can be used to display the buttons.

First, we will refactor all `url` to `urls` state and all `setUrl` to `setUrls` state updater functions. Instead of initializing the state with a `url` as a string, make it an array with the initial `url` as its only entry:

```
const App = () => {
  ...
  const [urls, setUrls] = React.useState([
    `${API_ENDPOINT}${searchTerm}`,
  ]);

  ...
};
```

src/App.js

Second, instead of using the current `url` state for data fetching, use the last `url` entry from the `urls` array. If another `url` is added to the list of `urls`, it is used to fetch data instead:

```
const App = () => {

  ...

  const handleFetchStories = React.useCallback(async () => {
```

```
  const handleFetchStories = React.useCallback(async () => {
    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    try {
      const lastUrl = urls[urls.length - 1];
      const result = await axios.get(lastUrl);

      dispatchStories({
        type: 'STORIES_FETCH_SUCCESS',
        payload: result.data.hits,
      });
    } catch {
      dispatchStories({ type: 'STORIES_FETCH_FAILURE' });
    }

  }, [urls]);

  ...
};
```

src/App.js

And third, instead of storing `url` string as state with the state updater function, concat the new `url` with the previous `urls` in an array for the new state:

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

  const handleSearchSubmit = event => {
    const url = `${API_ENDPOINT}${searchTerm}`;
    setUrls(urls.concat(url));

    event.preventDefault();
  };

  ...
};
```

src/App.js

With each search, another URL is stored in our state of `urls`. Next, render a button for each of the last five URLs. We'll include a new universal handler for these buttons, and each passes a specific `url` with a more specific inline handler:

```
const getLastSearches = urls => urls.slice(-5);

...

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

```
  const handleLastSearch = url => {
    // do something
  };

  const lastSearches = getLastSearches(urls);

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <SearchForm ... />

      {lastSearches.map(url => (
        <button
          key={url}
          type="button"
          onClick={() => handleLastSearch(url)}
        >
          {url}
        </button>
      ))}

      ...
    </div>
  );
};
```

src/App.js

Next, instead of showing the whole URL of the last search in the button as button text, show only the search term by replacing the API's endpoint with an empty string:

```
const extractSearchTerm = url => url.replace(API_ENDPOINT, '');
const getLastSearches = urls =>
  urls.slice(-5).map(url => extractSearchTerm(url));

...

const App = () => {
  ...

  const lastSearches = getLastSearches(urls);

  return (
    <div>
      ...

      {lastSearches.map(searchTerm => (
        <button
          key={searchTerm}

          type="button"

          onClick={() => handleLastSearch(searchTerm)}

        >
          {searchTerm}
```

```
        </button>
      ))}


      ...
    </div>
  );
};
```

src/App.js

The `getLastSearches` function now returns search terms instead of URLs. The actual `searchTerm` is passed to the inline handler instead of the `url`. By mapping over the list of `urls` in `getLastSearches`, we can extract the search term for each `url` within the array's map method. Making it more concise, it can also look like this:

```
const getLastSearches = urls =>
  urls.slice(-5).map(extractSearchTerm);
```

src/App.js

Now we'll provide functionality for the new handler used by every button, since clicking one of these buttons should trigger another search. Since we use the `urls` state for fetching data, and since we know the last URL is always used for data fetching, concat a new `url` to the list of `urls` to trigger another search request:

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

  const handleLastSearch = searchTerm => {
    const url = `${API_ENDPOINT}${searchTerm}`;
    setUrls(urls.concat(url));
  };

  ...
};
```

src/App.js

If you compare this new handler's implementation logic to the `handleSearchSubmit`, you may see some common functionality. Extract this common functionality to a new handler and a new extracted utility function:

```
const getUrl = searchTerm => `${API_ENDPOINT}${searchTerm}`;
```

```
...

const App = () => {
  ...

  const handleSearchSubmit = event => {
    handleSearch(searchTerm);
    event.preventDefault();
  };

  const handleLastSearch = searchTerm => {

    handleSearch(searchTerm);

  };

  const handleSearch = searchTerm => {
    const url = getUrl(searchTerm);
    setUrls(urls.concat(url));
  };

  ...
};
```

src/App.js

The new utility function can be used somewhere else in the App component. If you extract functionality that can be used by two parties, always check to see if it can be used by a third party.

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

  // important: still wraps the returned value in []
  const [urls, setUrls] = React.useState([getUrl(searchTerm)]);

  ...
};
```

src/App.js

The functionality should work, but it complains or breaks if the same search term is used more than once, because `searchTerm` is used for each button element as `key` attribute. Make the key more specific by concatenating it with the `index` of the mapped array.

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

  return (
    <div>
      ...
```

```
      {lastSearches.map((searchTerm, index) => (
        <button
          key={searchTerm + index}
          type="button"
          onClick={() => handleLastSearch(searchTerm)}
        >
          {searchTerm}
        </button>
      ))}

      ...
    </div>
  );
};
```

src/App.js

It's not the perfect solution, because the `index` isn't a stable key (especially when adding items to the list; however, it doesn't break in this scenario. The feature works now, but you can add further UX improvements by following the tasks below.

More Tasks:

- (1) Do not show the current search as a button, only the five preceding searches. Hint: Adapt the `getLastSearches` function.
- (2) Don't show duplicated searches. Searching twice for "React" shouldn't create two different buttons. Hint: Adapt the `getLastSearches` function.
- (3) Set the SearchForm component's input field value with the last search term if one of the buttons is clicked.

The source of the five rendered buttons is the `getLastSearches` function. There, we take the array of `urls` and return the last five entries from it. Now we'll change this utility function to return the last six entries instead of five, removing the last one. Afterward, only the five *previous* searches are displayed as buttons.

```
const getLastSearches = urls =>
  urls
    .slice(-6)
    .slice(0, -1)
    .map(extractSearchTerm);
```

src/App.js

If the same search is executed twice or more times in a row, duplicate buttons

appear, which is likely not your desired behavior. It would be acceptable to group identical searches into one button if they followed each other. We will solve this problem in the utility function as well. Before separating the array into the five previous searches, group the identical searches:

```
const getLastSearches = urls =>
  urls
    .reduce((result, url, index) => {
      const searchTerm = extractSearchTerm(url);

      if (index === 0) {
        return result.concat(searchTerm);
      }

      const previousSearchTerm = result[result.length - 1];

      if (searchTerm === previousSearchTerm) {
        return result;
      } else {
        return result.concat(searchTerm);
      }
    }, [])
    .slice(-6)
    .slice(0, -1);
```

src/App.js

The reduce function starts with an empty array as its `result`. The first iteration concats the `searchTerm` we extracted from the first `url` into the `result`. Every extracted `searchTerm` is compared to the one before it. If the previous search term is different from the current, concat the `searchTerm` to the result. If the search terms are identical, return the result without adding anything.

Lastly, the SearchForm's input field should be set with the new `searchTerm` if one of the last search buttons is clicked. We can solve this using the state updater function for the specific value used in the SearchForm component.

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

  const handleLastSearch = searchTerm => {
    setSearchTerm(searchTerm);

    handleSearch(searchTerm);
  };

  ...
};
```

src/App.js

Last, extract the feature's new rendered content from this section as a standalone component, to keep the App component lightweight:

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

  const lastSearches = getLastSearches(urls);

  return (
    <div>
      ...
      <LastSearches
        lastSearches={lastSearches}
        onLastSearch={handleLastSearch}
      />

      ...
    </div>
  );
};
const LastSearches = ({ lastSearches, onLastSearch }) => (
  <>
    {lastSearches.map((searchTerm, index) => (
      <button
        key={searchTerm + index}
        type="button"
        onClick={() => onLastSearch(searchTerm)}
      >
        {searchTerm}
      </button>
    ))}
  </>
);
```

src/App.js

The complete demonstration of the above concepts:

```
  □ □ □□  □   ã□  F   □□  □   □  )□     □   9□  5□  @@  □   °□  n□  □PNG
□
   IHDR   □   □□□   (-□S   äPLTE"""""""""""""""""""2PX=r□)7;*:>H□¤-BGE□□8do5Xb6[eK□®K□¯1MU
□
   IHDR   □   □□□   ×©ÍÊ  □ePLTE""""""""""""""""""""""""2RZN¢¹J□«3R[J□¬)59YÁÞ0KS4W`Q«ÄL□²%
?^q÷ñíÛ□ï.},□ìsæÝ_TttÔ¾ □1#□□/(ì□-[□□□è`□è`Ì□ÚïÅðZ□d5□□□□?ÎebZ¿Þ□i.Ûæ□□□ìqÎ□+1°□}Â□5
□
   IHDR        □□   D¤□Æ  □APLTE   """""""""""""""""""""""""""""2RZVºÖ_ÔôU·Ñ=r□$()'25]ÎíC□□0
□
   IHDR   @   @□□   □·□ì  □:PLTE   """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
¢ßqÇ8Ù□´□mKË±mÆ¶mÛü·yi!è□ÎªYÏuë ÀÏ_Àï?i÷□ý+ò□□ÄA□|□ù{□□´?¿□_En□).□JËD¤<□
©¬¢Z\Ts©R*□(□  ¯©□J□□□□u□X/□4J□9□¡5·DEµ4kÇ4□&i¥V4Ú□¡®Ð□□¯□vsf:àg,□¢èBC»î$¶□ºÍùî□□á□@□
-ê>Û□º«¢XÕ¢î}ß¨ëÛÑ;□ÃöN´□ØvÅý□Î¸ÿ1 □ë×ÄO@&v/Äþ_□ö\ô□Ç\í.□□½+0□□;□□□!□fÊ□¦´Ó%Â JY·O□Â□'
```

This feature wasn't an easy one. Lots of fundamental React but also JavaScript knowledge was needed to accomplish it. If you had no problems implementing it yourself or to follow the instructions, you are very well set. If you had one or the other issue, don't worry too much about it. Maybe you even figured out another way to solve this task and it may have turned out simpler than the one I showed here.

Exercises:

- Confirm the changes from the last section.