Paginated Fetch

Searching for popular stories via Hacker News API is only one step towards a fully-functional search engine, and there are many ways to fine-tune the search. Take a closer look at the data structure and observe how the Hacker News API returns more than a list of `hits`.

Specifically, it returns a paginated list. The page property, which is `0` in the first response, can be used to fetch more paginated lists as results. You only need to pass the next page with the same search term to the API.

The following shows how to implement a paginated fetch with the Hacker News data structure. If you are used to **pagination** from other applications, you may have a row of buttons from 1-10 in your mind – where the currently selected page is highlighted 1-[3]-10 and where clicking one of the buttons leads to fetching and displaying this subset of data.

In contrast, we will implement the feature as **infinite pagination**. Instead of rendering a single paginated list on a button click, we will render *all paginated lists as one list* with *one* button to fetch the next page. Every additional *paginated list* is concatenated at the end of the *one list*.

Task: Rather than fetching only the first page of a list, extend the functionality for fetching succeeding pages. Implement this as an infinite pagination on button click.

Optional Hints:

- Extend the `API_ENDPOINT` with the parameters needed for the paginated fetch.
- Store the `page` from the `result` as state after fetching the data.
- Fetch the first page (`0`) of data with every search.
- Fetch the succeeding page (`page + 1`) for every additional request

triggered with a new HTML button.

First, extend the API constant so it can deal with paginated data later. We will turn this one constant:

```
const API_ENDPOINT = 'https://hn.algolia.com/api/v1/search?query=';

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

Into a composable API constant with its parameters:

```
const API_BASE = 'https://hn.algolia.com/api/v1';
const API_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
// careful: notice the ? in between

const getUrl = searchTerm =>
  `${API_BASE}${API_SEARCH}?${PARAM_SEARCH}${searchTerm}`;
```

src/App.js

Fortunately, we don't need to adjust the API endpoint, because we extracted a common `getUrl` function for it. However, there is one spot where we must address this logic for the future:

```
const extractSearchTerm = url => url.replace(API_ENDPOINT, '');
```

src/App.js

In the next steps, it won't be sufficient to replace the base of our API endpoint, which is no longer in our code. With more parameters for the API endpoint, the URL becomes more complex. It will change from X to Y:

```
// X
https://hn.algolia.com/api/v1/search?query=react

// Y
https://hn.algolia.com/api/v1/search?query=react&page=0
```

It's better to extract the search term by extracting everything between `?` and `&`. Also consider that the `query` parameter is directly after the `?` and all other parameters like `page` follow it

```
const extractSearchTerm = url =>
  url
    .substring(url.lastIndexOf('?') + 1, url.lastIndexOf('&'));
```

The key (`query=`) also needs to be replaced, leaving only the value (`searchTerm`):

```
const extractSearchTerm = url =>
  url
    .substring(url.lastIndexOf('?') + 1, url.lastIndexOf('&'));
    .replace(PARAM_SEARCH, '');
```

src/App.js

Essentially, we'll trim the string until we leave only the search term:

```
// url
https://hn.algolia.com/api/v1/search?query=react&page=0

// url after  substring
query=react

// url after replace
react
```

The returned result from the Hacker News API delivers us the `page` data:

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

  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: {
          list: result.data.hits,
          page: result.data.page,
        },

      });
    } catch {
      dispatchStories({ type: 'STORIES_FETCH_FAILURE' });
```

```
    }
  }, [urls]);

  ...
};
```

src/App.js

We need to store this data to make paginated fetches later:

```
const storiesReducer = (state, action) => {
  switch (action.type) {
    case 'STORIES_FETCH_INIT':
      ...
    case 'STORIES_FETCH_SUCCESS':
      return {
        ...state,
        isLoading: false,
        isError: false,

        data: action.payload.list,
        page: action.payload.page,

      };
    case 'STORIES_FETCH_FAILURE':
      ...
    case 'REMOVE_STORY':
      ...
    default:
      throw new Error();
  }
};

const App = () => {
  ...

  const [stories, dispatchStories] = React.useReducer(
    storiesReducer,

    { data: [], page: 0, isLoading: false, isError: false }
  );

  ...
};
```

src/App.js

Extend the API endpoint with the new `page` parameter. This change was covered by our premature optimizations earlier, when we extracted the search term from the URL.

```
const API_BASE = 'https://hn.algolia.com/api/v1';
const API_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
```

```
const PARAM_PAGE = 'page=';

// careful: notice the ? and & in between

const getUrl = (searchTerm, page) =>
  `${API_BASE}${API_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${page}`;
```

src/App.js

Next, we must adjust all `getUrl` invocations by passing the `page` argument. Since the initial search and last search always fetch the first page (`0`), we pass this page as an argument to the function for retrieving the appropriate URL:

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

  const [urls, setUrls] = React.useState([getUrl(searchTerm, 0)]);

  ...

  const handleSearchSubmit = event => {
    handleSearch(searchTerm, 0);
    event.preventDefault();
  };

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

    handleSearch(searchTerm, 0);
  };

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

  ...
};
```

src/App.js

To fetch the next page when a button is clicked, we'll need to increment the `page` argument in this new handler:

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

  const handleMore = () => {
    const lastUrl = urls[urls.length - 1];
    const searchTerm = extractSearchTerm(lastUrl);
    handleSearch(searchTerm, stories.page + 1);
  };

  ...
```

```
  return (
    <div>

      ...

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List list={stories.data} onRemoveItem={handleRemoveStory} />
      )}

      <button type="button" onClick={handleMore}>
        More
      </button>

    </div>
  );
};
```

src/App.js

We've implemented data fetching with the dynamic `page` argument. The initial and last searches always use the first page, and every fetch with the new "More" button uses an incremented page. There is one crucial bug when trying the feature, though: the new fetches don't extend the previous list, but completely replace it.

We solve this in the reducer by avoiding the replacement of current `data` with new `data`, concatenating the paginated lists:

```
const storiesReducer = (state, action) => {
  switch (action.type) {
    case 'STORIES_FETCH_INIT':
      ...
    case 'STORIES_FETCH_SUCCESS':
      return {
        ...state,
        isLoading: false,
        isError: false,

        data:
          action.payload.page === 0
            ? action.payload.list
            : state.data.concat(action.payload.list),

        page: action.payload.page,
      };
    case 'STORIES_FETCH_FAILURE':
      ...
    case 'REMOVE_STORY':
      ...
    default:
      throw new Error();
  }
};
```

src/App.js

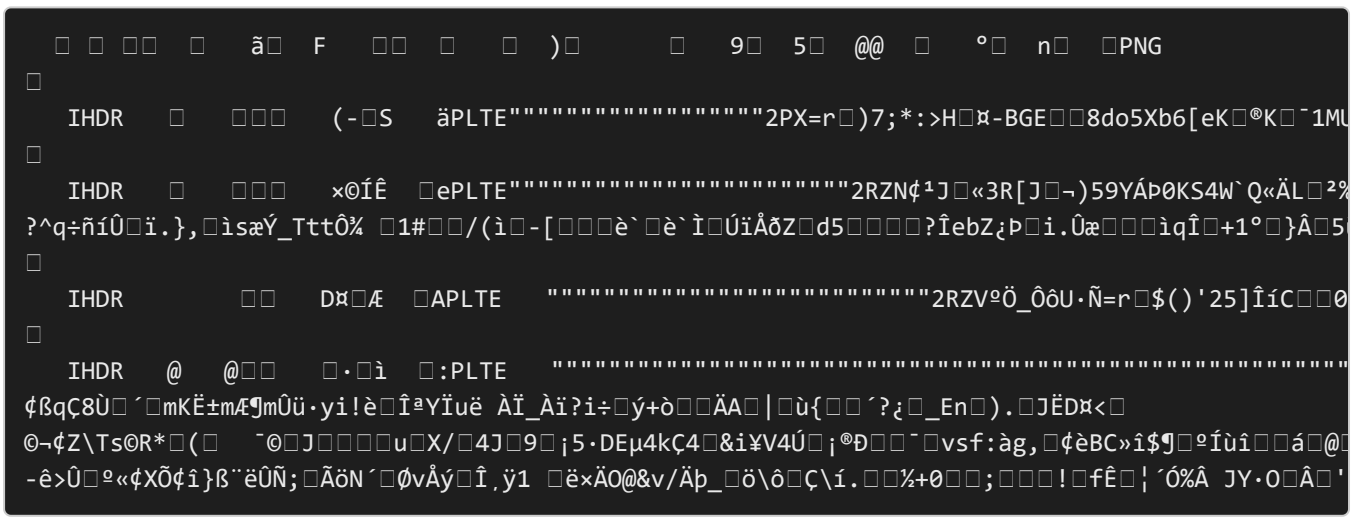
The displayed list grows after fetching more data with the new button. However, there is still a flicker straining the UX. When fetching paginated data, the list disappears for a moment because the loading indicator appears and reappears after the request resolves.

The desired behavior is to render the list–which is an empty list in the beginning–and replace the “More” button with the loading indicator only for pending requests. This is a common UI refactoring for conditional rendering when the task evolves from a single list to paginated lists.

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

  return (
    <div>
      ...
      <List list={stories.data} onRemoveItem={handleRemoveStory} />

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

        <button type="button" onClick={handleMore}>
          More
        </button>

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

src/App.js

The complete demonstration of the above concepts:



It's possible to fetch ongoing data for popular stories now. When working with third-party APIs, it's always a good idea to explore its boundaries. Every remote API returns different data structures, so its features may vary, and can be used in applications that consume the API.

Exercises:

- Confirm the changes from the last section.
- Revisit the Hacker News API documentation: Is there a way to fetch more items in a list for a page by just adding further parameters to the API endpoint?
- Revisit the beginning of this section which speaks about pagination and infinite pagination. How would you implement a normal pagination component with buttons from 1-[3]-10, where each button fetches and displays only one page of the list.
- Instead of having one "More" button, how would you implement an infinite pagination with an infinite scroll technique? Rather than clicking a button for fetching the next page explicitly, the infinite scroll could fetch the next page once the viewport of the browser hits the bottom of the displayed list.