Sorting and Pagination

Prior Knowledge

- Know how to use the `useState` hook in React.
- Know how to use the `useEffect` hook in React.
- Know how to make API calls from a React component.
- Understand how a REST API accepts queries.

Learning Objectives

- Understand some common/ preferred approaches to sorting rendered data.
- Understand some common/ preferred approaches to rendering paginated data.

Sorting

Rendering a list of items that needs to be displayed in a particular order is a common task for most websites and apps.

Avoid Mutation of State!

Consider a React component that renders a basic list:

```
const List = () => {
  const [items, setItems] = useState(['banana', 'aardvark']);

  return (
    <div>
      <ul>
        {items.map((item) => (
          <li key={item}>{item}</li>
        ))}
      </ul>
      <button
        onClick={() => {
          setItems((items) => items.sort()); // This is bad
        }}
      >
        Sort alphabetically
      </button>
    </div>
  );
};
```

One easy mistake to make is to use the sort method to change the order of an array. This is problematic because:

The sort() method sorts the elements of an array in place

-- MDN

Therefore, state has been mutated! **"Do not mutate state"** is an important rule to follow when working with React.

To prevent mutation, it would be necessary to create a shallow copy of the array ***before*** sorting it. This can be done in a number of ways. One method being to use the spread operator to put all of the elements of the original array into a brand new array, which can then be mutated:

```
<button
  onClick={() => {
    setItems((items) => [...items].sort());
  }}
>
  Sort alphabetically
</button>
```

A Single Source of Truth

In the example above, some user action (clicking a button) directly caused a change in the order of the rendered list. However, it could be useful for the component to indicate to the user how the list is currently sorted.

Consider the following example:

```
const List = () => {
  const [items, setItems] = useState(['banana', 'aardvark']);
  const [sort, setSort] = useState('a -> z');

  return (
    <div>
      {/* render items */}
      <p>Sorted {sort}</p>
      <button
        onClick={() => {
          setSort('a -> z');
          setItems((items) => [...items].sort());
        }}
      >
        Sort a to z
      </button>
      <button
        onClick={() => {
          setSort('z -> a');
          setItems((items) => {
            return [...items].sort((a, b) => {
              return b.localeCompare(a);
            });
          });
        }}
      >
        Sort z to a
      </button>
    </div>
  );
};
```

Here, the list of items and the sort value are stored in state. When a user clicks a button both parts of state are updated independently. This is not ideal as there is a possibility for the `sort` and actual order of items to become "out of sync" as more features are added to the component. Using a **single source of truth** can help to avoid this situation.

In the example below, instead of setting the `sort` ***and*** updating the list in the click handlers, only the `sort` value is altered. Then, the useEffect hook is used to trigger an update to the items array whenever the `sort` value changes:

```
const List = () => {
  const [items, setItems] = useState(['banana', 'aardvark']);
  const [sort, setSort] = useState('a -> z');

  useEffect(() => {
    if (sort === 'a -> z') {
      setItems((items) => [...items].sort());
    } else {
      setItems((items) => {
        return [...items].sort((a, b) => {
          return b.localeCompare(a);
        });
      });
    }
  }, [sort]);

  return (
    <div>
      {/* render items */}
      <button onClick={() => setSort('a -> z')}>Sort a to z</button>
      <button onClick={() => setSort('z -> a')}>Sort z to a</button>
    </div>
  );
};
```

With API Calls

A dynamic list in a React application is usually generated from some call to an API. In this case it would not be appropriate to just re-order the array on the client-side because the API may not serve up all of the items in a single request. Instead, it would be better to make a new API call to request data sorted in the correct order:

```
useEffect(() => {
  setIsLoading(true);
  api
    .getItems({ sort })
    .then((items) => {
      setItems(items);
      setIsLoading(false);
    })
    .catch((err) => {
      // error handle
    });
}, [sort]);
```

Pagination

API Strategies

Often, APIs do not serve up all of the data that they have for a particular resource type in a single request. Imagine how big the response would be if you could make a request to get all books from the Google Books API! There are many different ways that RESTful APIs can do this. A few of the common options are:

- Page numbers
  The client requests a particular "page" of results:
  - `/items?p=1` responds with the first 10 items
  - `/items?p=2` responds with items 11 -> 20
- Limit / Offset
  The client provides how many items they want to receive and where the server should start
  - `/items?limit=10` responds with the first 10 items
  - `/items?limit=10&start=11` responds with items 11 -> 20
- Cursors
  By providing an order, and an id for the last item that was received, the API figures out the next set of items the client should receive
  - `/items?last_viewed_id=123`

UI Strategies

Whichever way the back-end deals with pagination, it shouldn't have any impact on design of the user interface. There are endless possibilities for displaying paginated data but there are a few common patterns to choose from:

- A "load more" button
- "Next" & "back" buttons
- Page number buttons
- Infinite scroll (new "pages" of data are loaded when the user scrolls close to the bottom of the list).

Next & Back Buttons

Just like when dealing with sorting, it is usually a good idea to keep track of the current page in state. There should be some way for the user to update the current page. Ensure users can't end up requesting pages that don't exist by conditionally disabling the appropriate buttons:

```
<button
  onClick={() => {
    setPage((currentPage) => currentPage - 1);
  }}
  disabled={page === 0}
>
  Previous Page
</button>
<button
  onClick={() => {
    setPage((currentPage) => currentPage + 1);
  }}
  disabled={PAGE_LENGTH * page >= totalCount}
>
  Next Page
</button>
```

When the current page is `0`, the "back" button should be disabled.

The total number of pages can be calculated using the total count of items and the number of items per page. Because the total count could change, it would be good to keep this information in state and update it each time that more data is requested.

Whenever the current page is updated in state, it can trigger a new request for a specific page of data:

```
useEffect(() => {
  api.getItems({ page }).then(({ items, total_count }) => {
    setItems(items);
    setTotalCount(total_count);
  });
}, [page]);
```

Some Considerations for Infinite Scroll

Infinite scroll is a design pattern in which new "pages" of data are loaded in the background as a user approaches the bottom of a page. This can give the impression that there is a single page with infinite items already loaded on it. Many social media sites use infinite scrolling.

Clean up event listeners

There are many ways to implement an infinite scroll. Sometimes necessary to set up a scroll event listener on the window, or some other scrollable element. If this is the case, it will be important to ensure the event listener is removed from the page when the component is unmounted. To do so, ensure a "cleanup" function is returned from the `useEffect` that removes the event listener:

```
useEffect(() => {
  function handleScroll() {
    // check whether user is close to bottom of page
    // if so, fetch new "page" of data and add to state
  }

  window.addEventListener('scroll', handleScroll);

  return () => {
    window.removeEventListener('scroll', handleScroll);
  };
}, []);
```

Check out the React hooks documentation for more detail about using effects with cleanups.

Prevent too many API requests

A scroll event handler is triggered many times as a user navigates a page. So, if a scroll event handler is being used to trigger an API request it is important to prevent multiple requests being sent at once. To stop this from happening, use some conditional logic based on the `isLoading` boolean or debounce the function to prevent it from being called too frequently.

copy original markdown to clipboard

v 0.1