Client Cache

This lesson will teach you how to optimize your search functionality by using the client-side cache.

Each search submit makes a request to the Hacker News API. You might search for “redux”, followed by “react” and eventually “redux” again. In total it makes 3 requests. But you searched for “redux” twice and both times it took a whole asynchronous roundtrip to fetch the data. In a client-side cache, you would store each result. When a request to the API is made, it checks if a result is already there and uses the cache if it is. Otherwise, an API request is made to fetch the data.

To have a client cache for each result, you have to store multiple `results` rather than one `result` in your local component state. The results object will be a map with the search term as key and the result as value. Each result from the API will be saved by the search term (key).

At the moment, your result in the local state looks like this:

```
result: {
  hits: [ ... ],
  page: 2,
}
```

Imagine you have made two API requests. One for the search term “redux” and another one for “react”. The results object should look like the following:

```
results: {
  redux: {
    hits: [ ... ],
    page: 2,
  },
  react: {
    hits: [ ... ],
    page: 1,
  },
  ...
}
```

Let's implement a client-side cache with React `setState()`. First, rename the `result` object to `results` in the initial component state. Second, define a temporary `searchKey` to store each `result`.

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
    };

    ...

  }

  ...

}
```

The `searchKey` has to be set before each request is made. It reflects the `searchTerm`. Understanding why we don't use `searchTerm` here is a crucial part to understand before continuing with the implementation. The `searchTerm` is a fluctuant variable, because it gets changed every time you type into the search input field. We want a non fluctuant variable that determines the recent submitted search term to the API and can be used to retrieve the correct result from the map of results. It is a pointer to your current result in the cache, which can be used to display the current result in the `render()` method.

```
componentDidMount() {
  const { searchTerm } = this.state;
  this.setState({ searchKey: searchTerm });
  this.fetchSearchTopStories(searchTerm);
}

onSearchSubmit(event) {
  const { searchTerm } = this.state;
  this.setState({ searchKey: searchTerm });
  this.fetchSearchTopStories(searchTerm);
  event.preventDefault();
}
```

Now we will change where the result is stored to the local component state. It should store each result by `searchKey`.

```
class App extends Component {

  ...

  setSearchTopStories(result) {
    const { hits, page } = result;
    const { searchKey, results } = this.state;

    const oldHits = results && results[searchKey]
      ? results[searchKey].hits
      : [];

    const updatedHits = [
      ...oldHits,
      ...hits
    ];

    this.setState({
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      }
    });
  }

  ...

}
```

The `searchKey` will be used as the key to save the updated hits and page in a `results` map.

First, you have to retrieve the `searchKey` from the component state. Remember that the `searchKey` gets set on `componentDidMount()` and `onSearchSubmit()`.

Second, the old hits have to get merged with the new hits as before. But this time the old hits get retrieved from the `results` map with the `searchKey` as key.

Third, a new result can be set in the `results` map in the state. Let's examine the `results` object in `setState()`.

```
results: {
  ...results,
  [searchKey]: { hits: updatedHits, page }
}
```

The bottom part makes sure to store the updated result by `searchKey` in the results map. The value is an object with a hits and page property. The `searchKey` is the search term. You already learned the `[searchKey]: ...` syntax. It is an ES6 computed property name. It helps you allocate values dynamically in an object.

The upper part needs to spread all other results by `searchKey` in the state using the object spread operator. Otherwise, you would lose all the results that you have stored before.

Now you store all results by search term. That's the first step to enable your cache. In the next step, you can retrieve the result depending on the non fluctuant `searchKey` from your map of results. That's why you had to introduce the `searchKey` in the first place as non fluctuant variable. Otherwise, the retrieval would be broken when you would use the fluctuant `searchTerm` to retrieve the current result because this value might change when we use the Search component.

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey
    } = this.state;

    const page = (
      results &&
      results[searchKey] &&
      results[searchKey].page
    ) || 0;

    const list = (
      results &&
      results[searchKey] &&
      results[searchKey].hits
    ) || [];

    return (
      <div className="page">
        <div className="interactions">
          ...
        </div>
        <Table
          list={list}
          onDismiss={this.onDismiss}
        />
        <div className="interactions">
```

```
          <Button onClick={() => this.fetchSearchTopStories(searchKey, page + 1)}>
            More
          </Button>

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

Since you default to an empty list when there is no result by `searchKey`, you can spare the conditional rendering for the Table component. Additionally, we need to pass the `searchKey` rather than the `searchTerm` to the "More" button. Otherwise, your paginated fetch depends on the `searchTerm` value which is fluctuant. Moreover, make sure to keep the fluctuant `searchTerm` property for the input field in the Search component.

The search functionality should store all results from the Hacker News API now.

Now, we can see the `onDismiss()` method needs improvement, as it still deals with the `result` object. We want it deal with multiple `results`:

```
  onDismiss(id) {
    const { searchKey, results } = this.state;
    const { hits, page } = results[searchKey];

    const isNotId = item => item.objectID !== id;
    const updatedHits = hits.filter(isNotId);

    this.setState({
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      }
    });
  }
```

Check to make sure the "Dismiss" button is working again.

Note that nothing will stop the application from sending an API request on each search submit. Though there might be already a result, there is no check that prevents the request, so the cache functionality is not complete yet. It caches the results, but it doesn't make use of them. The last step is to prevent the API request when a result is available in the cache:

```
class App extends Component {
```

```
  constructor(props) {

    ...

    this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);
    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  needsToSearchTopStories(searchTerm) {
    return !this.state.results[searchTerm];
  }

  ...

  onSearchSubmit(event) {
    const { searchTerm } = this.state;
    this.setState({ searchKey: searchTerm });

    if (this.needsToSearchTopStories(searchTerm)) {
      this.fetchSearchTopStories(searchTerm);
    }

    event.preventDefault();
  }

  ...

}
```

Now your client only makes a request to the API once, though you searched for a term twice. Even paginated data with several pages gets cached that way, because you always save the last page for each result in the `results` map. This is a powerful approach to introduce caching into an application. The Hacker News API provides you with everything you need to cache paginated data effectively.