How To Create A Custom React Hook To Fetch And Cache Data

Quick summary: There is a high possibility that a lot of components in your React application will have to make calls to an API to retrieve data that will be displayed to your users. It's already possible to do that using the `componentDidMount()` lifecycle method, but with the introduction of Hooks, you can use `useEffect` or build **a custom hook** that will fetch and cache the data for you. That's what this session will cover.

Fetching Data In A React Component

Before React hooks, it was conventional to fetch initial data in the `componentDidMount()` lifecycle method, and data based on prop or state changes in `componentDidUpdate()` lifecycle method.

Here's how it works:

```
componentDidMount() {
  const fetchData = async () => {
    const response = await fetch(
      `https://hn.algolia.com/api/v1/search?query=JavaScript`
    );
    const data = await response.json();
    this.setState({ data });
  };

  fetchData();
}


componentDidUpdate(previousProps, previousState) {
    if (previousState.query !== this.state.query) {
      const fetchData = async () => {
        const response = await fetch(
          `https://hn.algolia.com/api/v1/search?query=${this.state.query}`
        );
        const data = await response.json();
        this.setState({ data });
      };

      fetchData();
    }
  }
```

The `componentDidMount` lifecycle method gets invoked as soon as the component gets mounted, and when that is done, what we did was to make a request to search for "JavaScript" via the API and update the state based on the response.

The `componentDidUpdate` lifecycle method, on the other hand, gets invoked when there's a change in the component. We compared the previous query in the state with the current query to prevent the method from getting invoked every time we set `data` in state. One thing we get from using hooks is to combine both lifecycle methods in a cleaner way — meaning that we won't need to have two lifecycle methods for when the component mounts and when it updates.

Fetching Data With useEffect Hook

The `useEffect` hook gets invoked as soon as the component is mounted. If we need the hook to rerun based on some prop or state changes, we'll need to pass them to the dependency array (which is the second argument of the `useEffect` hook).

Let's explore how to fetch data with hooks:

```
import { useState, useEffect } from 'react';

const [status, setStatus] = useState('idle');
const [query, setQuery] = useState('');
const [data, setData] = useState([]);

useEffect(() => {
    if (!query) return;

    const fetchData = async () => {
        setStatus('fetching');
        const response = await fetch(
            `https://hn.algolia.com/api/v1/search?query=${query}`
        );
        const data = await response.json();
        setData(data.hits);
        setStatus('fetched');
    };

    fetchData();
}, [query]);
```

In the example above, we passed query as a dependency to our `useEffect` hook. By doing that, we're telling `useEffect` to track query changes. If the previous query value isn't the same as the current value, the `useEffect` get invoked again.

With that said, we're also setting several status on the component as needed, as this will better convey some message to the screen based on some finite states status. In the idle state, we could let users know that they could make use of the search box to get started. In the fetching state, we could show a spinner. And, in the fetched state, we'll render the data.

It's important to set the data before you attempt to set status to fetched so that you can prevent a flicker which occurs as a result of the data being empty while you're setting the fetched status.

Creating A Custom Hook (useFetch)

"A custom hook is a JavaScript function whose name starts with 'use' and that may call other Hooks." That's really what it is, and along with a JavaScript function, it allows you to reuse some piece of code in several parts of your app.

The definition from the React Docs has given it away but let's see how it works in practice with a custom hook:

```
const useFetch = (query) => {
    const [status, setStatus] = useState('idle');
    const [data, setData] = useState([]);

    useEffect(() => {
        if (!query) return;

        const fetchData = async () => {
            setStatus('fetching');
            const response = await fetch(
                `https://hn.algolia.com/api/v1/search?query=${query}`
            );
            const data = await response.json();
            setData(data.hits);
            setStatus('fetched');
        };

        fetchData();
    }, [query]);

    return { status, data };
};
```

It's pretty much the same thing we did above with the exception of it being a function that takes in query and returns status and data. And, that's a `useFetch` hook that we could use in several components in our React application.

This works, but the problem with this implementation now is, it's specific to an API so we might to update a bit. What we intend to do is, to create a useFetch hook that can be used to call any URL. Let's revamp it to take in a URL instead!

```
const useFetch = (url) => {
    const [status, setStatus] = useState('idle');
    const [data, setData] = useState([]);

    useEffect(() => {
        if (!url) return;
        const fetchData = async () => {
            setStatus('fetching');
            const response = await fetch(url);
            const data = await response.json();
            setData(data);
            setStatus('fetched');
        };

        fetchData();
    }, [url]);

    return { status, data };
};
```

Now, our useFetch hook is generic and we can use it as we want in our various components.

Here's one way of consuming it:

```
const [query, setQuery] = useState('');

const url = query && `https://hn.algolia.com/api/v1/search?query=${query}`;
const { status, data } = useFetch(url);
```

In this case, if the value of query is truthy, we go ahead to set the URL and if it's not, we're fine with passing undefined as it'd get handled in our hook. The effect will attempt to run once, regardless.

Memoizing Fetched Data

Memoization is a technique we would use to make sure that we don't hit the endpoint if we have made some kind of request to fetch it at some initial phase. Storing the result of expensive fetch calls will save the users some load time, therefore, increasing overall performance.

Let's explore how we could do that!

```
const cache = {};

const useFetch = (url) => {
    const [status, setStatus] = useState('idle');
    const [data, setData] = useState([]);

    useEffect(() => {
        if (!url) return;

        const fetchData = async () => {
            setStatus('fetching');
            if (cache[url]) {
                const data = cache[url];
                setData(data);
                setStatus('fetched');
            } else {
                const response = await fetch(url);
                const data = await response.json();
                cache[url] = data; // set response in cache;
                setData(data);
                setStatus('fetched');
            }
        };

        fetchData();
    }, [url]);

    return { status, data };
};
```

Here, we're mapping URLs to their data. So, if we make a request to fetch some existing data, we set the data from our local cache, else, we go ahead to make the request and set the result in the cache. This ensures we do not make an API call when we have the data available to us locally. We'll also notice that we're killing off the effect if the URL is falsy, so it makes sure we don't proceed to fetch data that doesn't exist. We can't do it before the useEffect hook as that will go against one of the rules of hooks, which is to always call hooks at the top level.

Declaring cache in a different scope works but it makes our hook go against the principle of a pure function. Besides, we also want to make sure that React helps in cleaning up our mess when we no longer want to make use of the component. The tip that we can use `useRef` to help us in achieving that.

Conclusion

We've explored several hooks concepts to help fetch and cache data in our components.