How to fetch data with React Hooks?

MARCH 07, 2019 BY ROBIN WIERUCH - EDIT THIS POST

In this tutorial, I want to show you **how to fetch data in React with Hooks** by using the state and effect hooks. We will use the widely known Hacker News API to fetch popular articles from the tech world. You will also implement your custom hook for the data fetching that can be reused anywhere in your application or published on npm as standalone node package.

If you don't know anything about this new React feature, checkout this introduction to React Hooks. If you want to checkout the finished project for the showcased examples that show how to fetch data in React with Hooks, checkout this GitHub repository.

If you just want to have a ready to go React Hook for data fetching: `npm install use-data-api` and follow the documentation. Don't forget to star it if you use it :-)

great way to learn more about state and effect hooks in React.

DATA FETCHING WITH REACT HOOKS

If you are not familiar with data fetching in React, checkout my extensive data fetching in React article. It walks you through data fetching with React class components, how it can be made reusable with Render Prop Components and Higher-Order Components, and how it deals with error handling and loading spinners. In this article, I want to show you all of it with React Hooks in function components.

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

function App() {
  const [data, setData] = useState({ hits: [] });

  return (
    <ul>
      {data.hits.map(item => (
        <li key={item.objectID}>
          <a href={item.url}>{item.title}</a>
        </li>
      ))}
    </ul>
  );
}

export default App;
```

The App component shows a list of items (hits = Hacker News articles). The state and state update function come from the state hook called `useState` that is responsible to manage the local state for the data that we are going to fetch for the App component. The initial state is an empty list of hits in an object that represents the data. No one is setting any state for this data yet.

We are going to use axios to fetch data, but it is up to you to use another data fetching library or the native fetch API of the browser. If you haven't installed axios yet, you can do so by on the command line with `npm install axios`. Then implement your effect hook for the data fetching:

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

```
  const [data, setData] = useState({ hits: [] });

  useEffect(async () => {
    const result = await axios(
      'https://hn.algolia.com/api/v1/search?query=redux',
    );

    setData(result.data);
  });

  return (
    <ul>
      {data.hits.map(item => (
        <li key={item.objectID}>
          <a href={item.url}>{item.title}</a>
        </li>
      ))}
    </ul>
  );
}

export default App;
```

The effect hook called useEffect is used to fetch the data with axios from the API and to set the data in the local state of the component with the state hook's update function. The promise resolving happens with async/await.

However, when you run your application, you should stumble into a nasty loop. The effect hook runs when the component mounts but also when the component updates. Because we are setting the state after every data fetch, the component updates and the effect runs again. It fetches the data again and again. That's a bug and needs to be avoided. **We only want to fetch data when the component mounts.** That's why you can provide an empty array as second argument to the effect hook to avoid activating it on component updates but only for the mounting of the component.

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

function App() {
  const [data, setData] = useState({ hits: [] });

  useEffect(async () => {
    const result = await axios(
      'https://hn.algolia.com/api/v1/search?query=redux',
    );
```

```
  return (
    <ul>
      {data.hits.map(item => (
        <li key={item.objectID}>
          <a href={item.url}>{item.title}</a>
        </li>
      ))}
    </ul>
  );
}

export default App;
```

The second argument can be used to define all the variables (allocated in this array) on which the hook depends. If one of the variables changes, the hook runs again. If the array with the variables is empty, the hook doesn't run when updating the component at all, because it doesn't have to watch any variables.

There is one last catch. In the code, we are using async/await to fetch data from a third-party API. According to the documentation every function annotated with async returns an implicit promise: "*The async function declaration defines an asynchronous function, which returns an AsyncFunction object. An asynchronous function is a function which operates asynchronously via the event loop, using an implicit Promise to return its result.* ". However, an effect hook should return nothing or a clean up function. That's why you may see the following warning in your developer console log: **07:41:22.910 index.js:1452 Warning: useEffect function must return a cleanup function or nothing. Promises and useEffect(async () => ...) are not supported, but you can call an async function inside an effect.**. That's why using async directly in the `useEffect` function isn't allowed. Let's implement a workaround for it, by using the async function inside the effect.

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

function App() {
  const [data, setData] = useState({ hits: [] });

  useEffect(() => {
    const fetchData = async () => {
      const result = await axios(
        'https://hn.algolia.com/api/v1/search?query=redux',
      );

      setData(result.data);
    };
```

```
  return (
    <ul>
      {data.hits.map(item => (
        <li key={item.objectID}>
          <a href={item.url}>{item.title}</a>
        </li>
      ))}
    </ul>
  );
}

export default App;
```

That's data fetching with React hooks in a nutshell. But continue reading if you are interested about error handling, loading indicators, how to trigger the data fetching from a form, and how to implement a reusable data fetching hook.

HOW TO TRIGGER A HOOK PROGRAMMATICALLY / MANUALLY?

Great, we are fetching data once the component mounts. But what about using an input field to tell the API in which topic we are interested in? "Redux" is taken as default query. But what about topics about "React"? Let's implement an input element to enable someone to fetch other stories than "Redux" stories. Therefore, introduce a new state for the input element.

```
import React, { Fragment, useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [data, setData] = useState({ hits: [] });
  const [query, setQuery] = useState('redux');

  useEffect(() => {
    const fetchData = async () => {
      const result = await axios(
        'https://hn.algolia.com/api/v1/search?query=redux',
      );

      setData(result.data);
    };

    fetchData();
```

```
      <Fragment>
        <input
          type="text"
          value={query}
          onChange={event => setQuery(event.target.value)}
        />
        <ul>
          {data.hits.map(item => (
            <li key={item.objectID}>
              <a href={item.url}>{item.title}</a>
            </li>
          ))}
        </ul>
      </Fragment>
    );
}

export default App;
```

At the moment, both states are independent from each other, but now you want to couple them to only fetch articles that are specified by the query in the input field. With the following change, the component should fetch all articles by query term once it mounted.

```
...

function App() {
  const [data, setData] = useState({ hits: [] });
  const [query, setQuery] = useState('redux');

  useEffect(() => {
    const fetchData = async () => {
      const result = await axios(
        `http://hn.algolia.com/api/v1/search?query=${query}`,
      );

      setData(result.data);
    };

    fetchData();
  }, []);

  return (
    ...
  );
}

export default App;
```

the empty array as second argument to the effect. The effect depends on no variables, so it is only triggered when the component mounts. However, now the effect should depend on the query. Once the query changes, the data request should fire again.

```
...

function App() {
  const [data, setData] = useState({ hits: [] });
  const [query, setQuery] = useState('redux');

  useEffect(() => {
    const fetchData = async () => {
      const result = await axios(
        `http://hn.algolia.com/api/v1/search?query=${query}`,
      );

      setData(result.data);
    };

    fetchData();
  }, [query]);

  return (
    ...
  );
}

export default App;
```

The refetching of the data should work once you change the value in the input field. But that opens up another problem: On every character you type into the input field, the effect is triggered and executes another data fetching request. How about providing a button that triggers the request and therefore the hook manually?

```
function App() {
  const [data, setData] = useState({ hits: [] });
  const [query, setQuery] = useState('redux');
  const [search, setSearch] = useState('');

  useEffect(() => {
    const fetchData = async () => {
      const result = await axios(
        `http://hn.algolia.com/api/v1/search?query=${query}`,
      );

      setData(result.data);
```

```
  }, [query]);

  return (
    <Fragment>
      <input
        type="text"
        value={query}
        onChange={event => setQuery(event.target.value)}
      />
      <button type="button" onClick={() => setSearch(query)}>
        Search
      </button>

      <ul>
        {data.hits.map(item => (
          <li key={item.objectID}>
            <a href={item.url}>{item.title}</a>
          </li>
        ))}
      </ul>
    </Fragment>
  );
}
```

Now, make the effect dependant on the search state rather than the fluctuant query state that changes with every key stroke in the input field. Once the user clicks the button, the new search state is set and should trigger the effect hook kinda manually.

```
...

function App() {
  const [data, setData] = useState({ hits: [] });
  const [query, setQuery] = useState('redux');
  const [search, setSearch] = useState('redux');

  useEffect(() => {
    const fetchData = async () => {
      const result = await axios(
        `http://hn.algolia.com/api/v1/search?query=${search}`,
      );

      setData(result.data);
    };

    fetchData();
  }, [search]);

  return (
```

```
export default App;
```

Also the initial state of the search state is set to the same state as the query state, because the component fetches data also on mount and therefore the result should mirror the value in the input field. However, having a similar query and search state is kinda confusing. Why not set the actual URL as state instead of the search state?

```
function App() {
  const [data, setData] = useState({ hits: [] });
  const [query, setQuery] = useState('redux');
  const [url, setUrl] = useState(
    'https://hn.algolia.com/api/v1/search?query=redux',
  );

  useEffect(() => {
    const fetchData = async () => {
      const result = await axios(url);

      setData(result.data);
    };

    fetchData();
  }, [url]);

  return (
    <Fragment>
      <input
        type="text"
        value={query}
        onChange={event => setQuery(event.target.value)}
      />
      <button
        type="button"
        onClick={() =>
          setUrl(`http://hn.algolia.com/api/v1/search?query=${query}`)
        }
      >
        Search
      </button>

      <ul>
        {data.hits.map(item => (
          <li key={item.objectID}>
            <a href={item.url}>{item.title}</a>
          </li>
        ))}
      </ul>
```

That's if for the implicit programmatic data fetching with the effect hook. You can decide on which state the effect depends. Once you set this state on a click or in another side-effect, this effect will run again. In this case, if the URL state changes, the effect runs again to fetch stories from the API.

LOADING INDICATOR WITH REACT HOOKS

Let's introduce a loading indicator to the data fetching. It's just another state that is manage by a state hook. The loading flag is used to render a loading indicator in the App component.

```
import React, { Fragment, useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [data, setData] = useState({ hits: [] });
  const [query, setQuery] = useState('redux');
  const [url, setUrl] = useState(
    'https://hn.algolia.com/api/v1/search?query=redux',
  );
  const [isLoading, setIsLoading] = useState(false);

  useEffect(() => {
    const fetchData = async () => {
      setIsLoading(true);

      const result = await axios(url);

      setData(result.data);
      setIsLoading(false);
    };

    fetchData();
  }, [url]);

  return (
    <Fragment>
      <input
        type="text"
        value={query}
        onChange={event => setQuery(event.target.value)}
      />
      <button
        type="button"
```

```
        >
          Search
        </button>

        {isLoading ? (
          <div>Loading ...</div>
        ) : (
          <ul>
            {data.hits.map(item => (
              <li key={item.objectID}>
                <a href={item.url}>{item.title}</a>
              </li>
            ))}
          </ul>
        )}
      </Fragment>
    );
}

export default App;
```

Once the effect is called for data fetching, which happens when the component mounts or the URL state changes, the loading state is set to true. Once the request resolves, the loading state is set to false again.

ERROR HANDLING WITH REACT HOOKS

What about error handling for data fetching with a React hook? The error is just another state initialized with a state hook. Once there is an error state, the App component can render feedback for the user. When using async/await, it is common to use try/catch blocks for error handling. You can do it within the effect:

```
import React, { Fragment, useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [data, setData] = useState({ hits: [] });
  const [query, setQuery] = useState('redux');
  const [url, setUrl] = useState(
    'https://hn.algolia.com/api/v1/search?query=redux',
  );
  const [isLoading, setIsLoading] = useState(false);
  const [isError, setIsError] = useState(false);
```

```
      setIsError(false);
      setIsLoading(true);

      try {
        const result = await axios(url);

        setData(result.data);
      } catch (error) {
        setIsError(true);
      }

      setIsLoading(false);
    };

    fetchData();
  }, [url]);

  return (
    <Fragment>
      <input
        type="text"
        value={query}
        onChange={event => setQuery(event.target.value)}
      />
      <button
        type="button"
        onClick={() =>
          setUrl(`http://hn.algolia.com/api/v1/search?query=${query}`)
        }
      >
        Search
      </button>

      {isError && <div>Something went wrong ...</div>}

      {isLoading ? (
        <div>Loading ...</div>
      ) : (
        <ul>
          {data.hits.map(item => (
            <li key={item.objectID}>
              <a href={item.url}>{item.title}</a>
            </li>
          ))}
        </ul>
      )}
    </Fragment>
  );
}

export default App;
```

error yourself, you can alter the URL into something invalid. Then check whether the error message shows up.

FETCHING DATA WITH FORMS AND REACT

What about a proper form to fetch data? So far, we have only a combination of input field and button. Once you introduce more input elements, you may want to wrap them with a form element. In addition, a form makes it possible to trigger the button with "Enter" on the keyboard too.

```
function App() {
  ...

  return (
    <Fragment>
      <form
        onSubmit={() =>
          setUrl(`http://hn.algolia.com/api/v1/search?query=${query}`)
        }
      >
        <input
          type="text"
          value={query}
          onChange={event => setQuery(event.target.value)}
        />
        <button type="submit">Search</button>
      </form>

      {isError && <div>Something went wrong ...</div>}

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

But now the browser reloads when clicking the submit button, because that's the native behavior of the browser when submitting a form. In order to prevent the default behavior, we can invoke a function on the React event. That's how you do it in React class components too.

```
function App() {
  ...
```

```
        <form onSubmit={event => {
          setUrl(`http://hn.algolia.com/api/v1/search?query=${query}`);

          event.preventDefault();
        }}>
          <input
            type="text"
            value={query}
            onChange={event => setQuery(event.target.value)}
          />
          <button type="submit">Search</button>
        </form>

        {isError && <div>Something went wrong ...</div>}

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

Now the browser shouldn't reload anymore when you click the submit button. It works as before, but this time with a form instead of the naive input field and button combination. You can press the "Enter" key on your keyboard too.

CUSTOM DATA FETCHING HOOK

In order to extract a custom hook for data fetching, move everything that belongs to the data fetching, except for the query state that belongs to the input field, but including the loading indicator and error handling, to its own function. Also make sure you return all the necessary variables from the function that are used in the App component.

```
  const useHackerNewsApi = () => {
    const [data, setData] = useState({ hits: [] });
    const [url, setUrl] = useState(
      'https://hn.algolia.com/api/v1/search?query=redux',
    );
    const [isLoading, setIsLoading] = useState(false);
    const [isError, setIsError] = useState(false);

    useEffect(() => {
      const fetchData = async () => {
        setIsError(false);
        setIsLoading(true);
```

```
        setData(result.data);
      } catch (error) {
        setIsError(true);
      }

      setIsLoading(false);
    };

    fetchData();
  }, [url]);

  return [{ data, isLoading, isError }, setUrl];
}
```

Now, your new hook can be used in the App component again:

```
function App() {
  const [query, setQuery] = useState('redux');
  const [{ data, isLoading, isError }, doFetch] = useHackerNewsApi();

  return (
    <Fragment>
      <form onSubmit={event => {
        doFetch(`http://hn.algolia.com/api/v1/search?query=${query}`);

        event.preventDefault();
      }}>
        <input
          type="text"
          value={query}
          onChange={event => setQuery(event.target.value)}
        />
        <button type="submit">Search</button>
      </form>

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

The initial state can be made generic too. Pass it simply to the new custom hook:

```
import React, { Fragment, useState, useEffect } from 'react';
import axios from 'axios';

const useDataApi = (initialUrl, initialData) => {
```

```
  const [isError, setIsError] = useState(false);

  useEffect(() => {
    const fetchData = async () => {
      setIsError(false);
      setIsLoading(true);

      try {
        const result = await axios(url);

        setData(result.data);
      } catch (error) {
        setIsError(true);
      }

      setIsLoading(false);
    };

    fetchData();
  }, [url]);

  return [{ data, isLoading, isError }, setUrl];
};

function App() {
  const [query, setQuery] = useState('redux');
  const [{ data, isLoading, isError }, doFetch] = useDataApi(
    'https://hn.algolia.com/api/v1/search?query=redux',
    { hits: [] },
  );

  return (
    <Fragment>
      <form
        onSubmit={event => {
          doFetch(
            `http://hn.algolia.com/api/v1/search?query=${query}`,
          );

          event.preventDefault();
        }}
      >
        <input
          type="text"
          value={query}
          onChange={event => setQuery(event.target.value)}
        />
        <button type="submit">Search</button>
      </form>

      {isError && <div>Something went wrong ...</div>}
```

```
          <ul>
            {data.hits.map(item => (
              <li key={item.objectID}>
                <a href={item.url}>{item.title}</a>
              </li>
            ))}
          </ul>
        )}
      </Fragment>
    );
}

export default App;
```

That's it for the data fetching with a custom hook. The hook itself doesn't know anything about the API. It receives all parameters from the outside and only manages necessary states such as the data, loading and error state. It executes the request and returns the data to the component using it as custom data fetching hook.

REDUCER HOOK FOR DATA FETCHING

So far, we have used various state hooks to manage our data fetching state for the data, loading and error state. However, somehow all these states, managed with their own state hook, belong together because they care about the same cause. As you can see, they are all used within the data fetching function. A good indicator that they belong together is that they are used one after another (e.g. `setIsError`, `setIsLoading`). Let's combine all three of them with a Reducer Hook instead.

A Reducer Hook returns us a state object and a function to alter the state object. The function -- called dispatch function -- takes an action which has a type and an optional payload. All this information is used in the actual reducer function to distill a new state from the previous state, the action's optional payload and type. Let's see how this works in code:

```
import React, {
  Fragment,
  useState,
  useEffect,
  useReducer,
} from 'react';
import axios from 'axios';
```

```
const useDataApi = (initialUrl, initialData) => {
  const [url, setUrl] = useState(initialUrl);

  const [state, dispatch] = useReducer(dataFetchReducer, {
    isLoading: false,
    isError: false,
    data: initialData,
  });

  ...
};
```

The Reducer Hook takes the reducer function and an initial state object as parameters. In our case, the arguments of the initial states for the data, loading and error state didn't change, but they have been aggregated to one state object managed by one reducer hook instead of single state hooks.

```
const dataFetchReducer = (state, action) => {
  ...
};

const useDataApi = (initialUrl, initialData) => {
  const [url, setUrl] = useState(initialUrl);

  const [state, dispatch] = useReducer(dataFetchReducer, {
    isLoading: false,
    isError: false,
    data: initialData,
  });

  useEffect(() => {
    const fetchData = async () => {
      dispatch({ type: 'FETCH_INIT' });

      try {
        const result = await axios(url);

        dispatch({ type: 'FETCH_SUCCESS', payload: result.data });
      } catch (error) {
        dispatch({ type: 'FETCH_FAILURE' });
      }
    };

    fetchData();
  }, [url]);

  ...
```

Now, when fetching data, the dispatch function can be used to send information to the reducer function. The object being send with the dispatch function has a mandatory `type` property and an optional `payload` property. The type tells the reducer function which state transition needs to be applied and the payload can additionally be used by the reducer to distill the new state. After all, we only have three state transitions: initializing the fetching process, notifying about a successful data fetching result, and notifying about an errornous data fetching result.

In the end of the custom hook, the state is returned as before, but because we have a state object and not the standalone states anymore. This way, the one who calls the `useDataApi` custom hook still gets access to `data`, `isLoading` and `isError`:

```
const useDataApi = (initialUrl, initialData) => {
  const [url, setUrl] = useState(initialUrl);

  const [state, dispatch] = useReducer(dataFetchReducer, {
    isLoading: false,
    isError: false,
    data: initialData,
  });

  ...

  return [state, setUrl];
};
```

Last but not least, the implementation of the reducer function is missing. It needs to act on three different state transitions called `FETCH_INIT`, `FETCH_SUCCESS` and `FETCH_FAILURE`. Each state transition needs to return a new state object. Let's see how this can be implemented with a switch case statement:

```
const dataFetchReducer = (state, action) => {
  switch (action.type) {
    case 'FETCH_INIT':
      return { ...state };
    case 'FETCH_SUCCESS':
      return { ...state };
    case 'FETCH_FAILURE':
      return { ...state };
    default:
      throw new Error();
  }
};
```

So far, in out switch case statement each state transition only returns the previous state. A destructuring statement is used to keep the state object immutable -- meaning the state is never directly mutated -- to enforce best practices. Now let's override a few of the current's state returned properties to alter the state with each state transition:

```
const dataFetchReducer = (state, action) => {
  switch (action.type) {
    case 'FETCH_INIT':
      return {
        ...state,
        isLoading: true,
        isError: false
      };
    case 'FETCH_SUCCESS':
      return {
        ...state,
        isLoading: false,
        isError: false,
        data: action.payload,
      };
    case 'FETCH_FAILURE':
      return {
        ...state,
        isLoading: false,
        isError: true,
      };
    default:
      throw new Error();
  }
};
```

Now every state transition, decided by the action's type, returns a new state based on the previous state and the optional payload. For instance, in the case of a successful request, the payload is used to set the data of the new state object.

In conclusion, the Reducer Hook makes sure that this portion of the state management is encapsulated with its own logic. By providing action types and optional payloads, you will always end up with a predicatbale state change. In addition, you will never run into invalid states. For instance, previously it would have been possible to accidently set the `isLoading` and `isError` states to true. What should be displayed in the UI for this case? Now, each state transition defined by the reducer function leads to a valid state object.

It's a common problem in React that component state is set even though the component got already unmounted (e.g. due to navigating away with React Router). I have written about this issue previously over here which describes how to prevent setting state for unmounted components in various scenarios. Let's see how we can prevent to set state in our custom hook for the data fetching:

```
const useDataApi = (initialUrl, initialData) => {
  const [url, setUrl] = useState(initialUrl);

  const [state, dispatch] = useReducer(dataFetchReducer, {
    isLoading: false,
    isError: false,
    data: initialData,
  });

  useEffect(() => {
    let didCancel = false;

    const fetchData = async () => {
      dispatch({ type: 'FETCH_INIT' });

      try {
        const result = await axios(url);

        if (!didCancel) {
          dispatch({ type: 'FETCH_SUCCESS', payload: result.data });
        }
      } catch (error) {
        if (!didCancel) {
          dispatch({ type: 'FETCH_FAILURE' });
        }
      }
    };

    fetchData();

    return () => {
      didCancel = true;
    };
  }, [url]);

  return [state, setUrl];
};
```

Every Effect Hook comes with a clean up function which runs when a component unmounts. The clean up function is the one function returned from the hook. In our case, we use a

set to `true` which results in preventing to set the component state after the data fetching has been asynchronously resolved eventually.

Note: Actually not the data fetching is aborted -- which could be achieved with Axios Cancellation -- but the state transition is not performed anymore for the unmounted component. Since Axios Cancellation has not the best API in my eyes, this boolean flag to prevent setting state does the job as well.

. . .

You have learned how the React hooks for state and effects can be used in React for data fetching. If you are curious about data fetching in class components (and function components) with render props and higher-order components, checkout out my other article from the beginning. Otherwise, I hope this article was useful to you for learning about React Hooks and how to use them in a real world scenario.