Effects in React

What in an effect in react?

Effects let you specific side effects (changes) that are caused by rendering / or re-rendering of a component.

Example of an effect can be:

- Downloading data
- Reading data from local storage

When your component is being rendered, we want to download some data.

To implement effects in react, we have a hook called as useEffect.

According to the official docs:

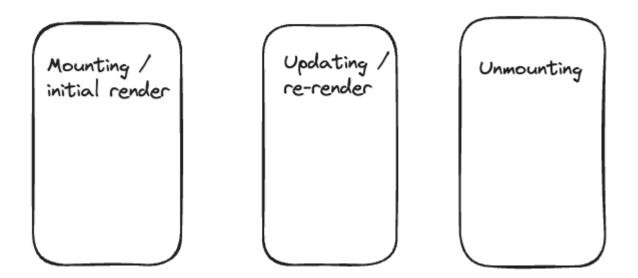
Effects can help you to synchronize your frontend with external systems.

Lifecycle events in a component

Whenever a component is brought into the picture, there are multiple lifecycle events that it goes through

- 1. Mounting / Initial render This is the phase of the first time loading of the component, i.e. when the component is added to the dom for the first time.
- 2. Re-render / Updates When due to a state update or parent re-render, a component re-renders then this is the phase of updating / re-rendering a component
- 3. Unmounting This is the phase of removing component from dom.

Lifecycle of a component



If we want to control, what should happen or what logic needs to be executed while mounting a component, unmounting or re-rendering a component, we can control this by useEffect

State vs Effects

States can cause a component to re-render, but effects can helps us to control when a rerender happens, what to do.

In a nutshell, states are one of the causes of re-render and effects are consequences of re-render.

XMLHttpRequest

For a very long time to make a network request from the browser, we used to use an object of XMLHttpRequest. This object is capable of making a network call and downloading content.

```
const xhr = new XMLHttpRequest(); // we need to create an object of
XMLHttpRequest

// This open function initiates a connection request when we call send
function

// This open function takes the method type and url for the network call
xhr.open("GET", "https://jsonplaceholder.typicode.com/todos/2");

// Once we have the response from server, then the call back of onload is
```

```
executed
xhr.onload = function () {
    if(xhr.status >= 200 && xhr.status < 300) {
        console.log("Response", xhr.responseText);
    } else {
        console.log("seomthing went wrong");
    }
}
xhr.send(); // This triggers the final call.</pre>
```

Fetch - Alternative to XMLHttpRequest

So, XMLHttpRequest is an old way to make network calls. It is more callback based and syntax is also not so simple. That's why modern browsers support fetch.

Using fetch we can write promise based syntax to make network calls.

```
async function download() {
    const response = await
fetch("https://jsonplaceholder.typicode.com/todos/2");
    console.log(response);
    const result = await response.json();
    console.log(result);
}
downlod();
```

Here, fetch returns a response, which is a promise, once that promise is resolve from te resolved we need to call <code>.json()</code> which is again a promise based call which gives us the final resultant json.

Note:

Fetch doesn't internally use HTMLHttpRequest instead it is just an alternative.