The useEffect Cheatsheet

When I started ReactJS, I really enjoyed my first steps, component, props and many fun things about react. One of that features was `useEffect`; it was fun but complicated for me while I struggled to learn it.

Now I want to share my mental model in this small cheat sheet to help you learn `useEffect` better or own a better mental model.

Philosophy

> `useEffect` is not a lifecycle hook. It's a mechanism for synchronizing side-effects (`fetch`, `setTimeout`, ...) with the state of your app. EpicReact.dev

The main goal is not using `useEffect` for the component lifecycle but using it to do stuff when state-changes (re-renders).

```
useEffect(() => {
    // A: run whenever the deps changes
    return {
      // B: Optional, runs before 1, we call this the clean-up function
    }
}, deps); // deps is Optional too
```

`useEffect`'s running steps:

- 1: Run A
- 2: Wait for new state changes (component re-renders)
- 3: If the `deps` changed
    - Run B to cleanup the previous render's side-effects
    - Go to 2

Dependencies

- **No dependency**: the side-effect function (A) will run on every state-change (re-render)

```
useEffect(() => {
    // I depend on everything, I'll run on every re-render
});
```

- **Empty Array**: There's nothing to listen to its changes, so it'll run the side-effect function just one time at the state initialization (first render)

```
useEffect(() => {
    // I depend on nothing, I'll run just one time
}, []);
```

- **Non-Empty Array**: The side-effect function runs on every dependency changes (at least one of the dependencies)

```
useEffect(() => {
    // I depend on state1, state2 and prop1
    // I'll run on every change of these dependencies
}, [state1, state2, prop1]);
```

Each Render Has Its Own Effects

I really love the "Each Render has its own Effects" title; I think almost all hooks rely on that title. We should note that every render has its own function body and its own values. The same goes for the side-effect function; check this.

```
useEffect(() => {
    console.log(count)
}, [count]);
```

let's do some fake state changes and see what happens to the side-effect function.

```
// in the first render, `count` is 0
// The side-effect function is going to be like this
() => {
    console.log(0)
}
// assume we change `count` to 1 (setCount(1)), next render is like that
() => {
    console.log(1)
}
// and so on...
```

That's how `useEffect` works around dependencies.