# React Hooks useEffect Basics

## *Defining useEffect*

You've likely performed data fetching, subscriptions, or manually changing the DOM from React components before. We call these operations **"side effects" (or "effects" for short) because they can affect other components and can't be done during rendering**. Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects. Whether or not you're used to calling these operations "side effects" (or just "effects"), you've likely performed them in your components before.

## *Hello World React Hooks useEffect app*

### STEP 1: create the react-redux directory w/ the *npx*

In the outer directory type: *npx create-react-app use_effect_initial_demo*

### STEP 2: view the skeletal files:

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

App.js

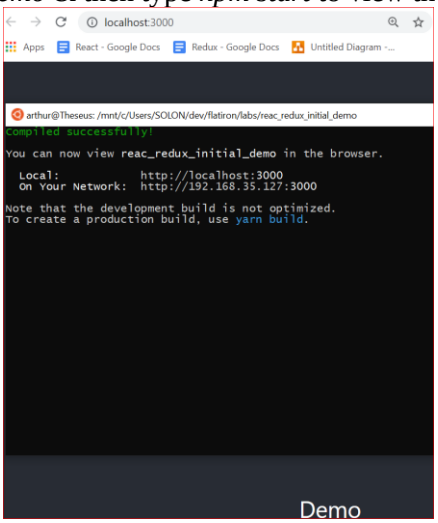
```
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        Demo
      </header>
    </div>
  );
}

export default App;
```

## STEP 3: view the running app

cd into the directory *react_redux_initial_demo* & then type *npm start* to view the demo page




## STEP 4: useEffect API

From the API [1]

```
useEffect(didUpdate);
```

Accepts a function that contains imperative, possibly effectful code.

Mutations, subscriptions, timers, logging, and other side effects are not allowed inside the main body of a function component (referred to as React's *render phase*). Doing so will lead to confusing bugs and inconsistencies in the UI.

Instead, use useEffect. The function passed to useEffect will run after the render is committed to the screen. Think of effects as an escape hatch from React's purely functional world into the imperative world.

By default, effects run after every completed render, but you can choose to fire them only when certain values have changed.

**Instead of thinking in terms of "mounting" and "updating", you might find it easier to think that effects happen "after render". React guarantees the DOM has been updated by the time it runs the effects.**

### *Cleaning Up After an Effect*

Often, effects create resources that need to be cleaned up before the component leaves the screen, such as a login / subscription. To do this, the function passed to *useEffect* may return a clean-up function. For example, to check a friend's online status, this functional component logs in and then after checking the status of our friend logs out:

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

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);
  let cleanup = () => {API.logout();};

  useEffect( () => {
                      let friendIsloggedIn = API.login(props.friend.id);
                      if (friendIsloggedIn) setIsOnline(true);
                      return cleanup;
                    }
           );

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

The clean-up function runs before the component is removed from the UI to prevent memory leaks. Additionally, if a component renders multiple times (as they typically do), the **previous effect is cleaned up before executing the next effect**. In our example, this means a new login is attempted on every update. To avoid firing an effect on every update, refer to the next section.

### *Conditionally firing an effect*

The default behavior for effects is to fire the effect after every completed render. That way an effect is always recreated if one of its dependencies changes.

However, this may be overkill in some cases, like the example from the previous section where you login to check the status of your friend. We don't need to create a new login on every update, only if the *id* of the *friend* we are checking the status on has changed.

To implement this, pass a second argument to *useEffect* that is the array of values that the effect depends on. Our updated example now looks like this:

```
  useEffect( () => {
                     let friendIsloggedIn = API.login(props.friend.id);
                     if (friendIsloggedIn) setIsOnline(true);
                     return cleanup;
                    },
                    [props.friend.id]
           );
```

So the updated useEffect API looks so:

```
useEffect( () => { . . . return cleanup; },[var_n_whose_change_triggers_useEffect . . .] );
```

If you want to run an effect and clean it up only once (on mount and unmount), you can pass an empty array ([]) as a second argument. This tells React that your effect doesn't depend on *any* values from props or state, so it never needs to re-run. This isn't handled as a special case — it follows directly from how the dependencies array always works.

## *The Source*

## *References:*


https://reactjs.org/docs/hooks-reference.html#useeffect

https://reactjs.org/docs/hooks-effect.html

https://reactjs.org/docs/hooks-overview.html