

You Might Not Need an Effect

WTS, 2022/9/28

<https://beta.reactjs.org/learn/you-might-not-need-an-effect>

Table of Contents

- Effects overview
- When you can use an effect
- When you might not need an effect

Effects Overview

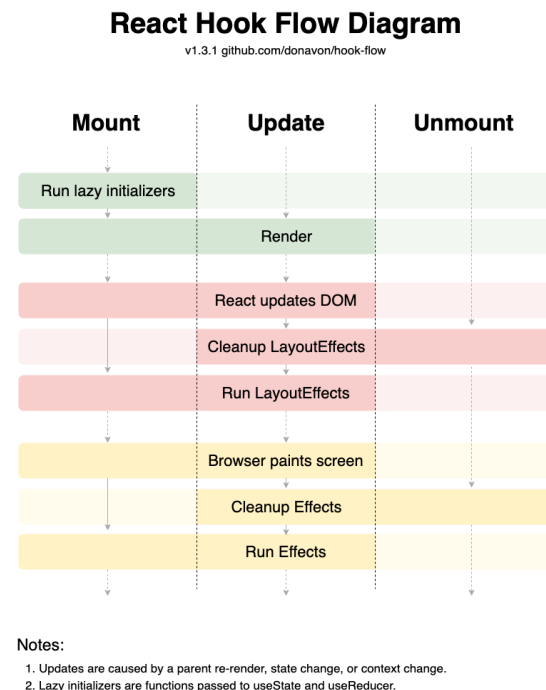
Types of logic inside React components:

- **Rendering code**
 - Where you take the props and state, transform them, and return the JSX you want to see on the screen
- **Event handlers**
 - Add interactivity
- **Effects**
 - Specify side effects that are caused by rendering itself

Effects Overview

Effects let you run some code **after rendering** so that you can synchronize your component with some system **outside of React**.

- **After rendering**
 - after the screen updates
- **Outside of React**
 - e.g. network, third-party libraries, browser APIs
- **Effects don't run on the server**
 - codes inside effects won't run in SSR/SSG



*Diagram from <https://github.com/donavon/hook-flow>

Effects Overview

Your components should be **resilient to being remounted**.

- When Strict Mode is on, React mounts components **twice** (in development only) to stress-test your Effects
- If your Effect breaks because of remounting, you need to implement a **cleanup** function

When You Can Use an Effect

Run side effects after render:

- **Directly interact with the DOM after render** (like play a video)
- **Connect/Disconnect to an external server after render** (like a chat room)

When You Can Use an Effect

Directly interact with the DOM after render: play a video

```
1  function VideoPlayer({ src, isPlaying }) {  
2    const ref = useRef(null);  
3  
4    useEffect(() => {  
5      if (isPlaying) {  
6        ref.current.play();  
7      } else {  
8        ref.current.pause();  
9      }  
10   }, [isPlaying]);  
11  
12   return <video ref={ref} src={src} loop playsInline />;  
13 }
```

When You Can Use an Effect

Connect/Disconnect to an external server after render: connect to a chat room server

```
1  function ChatRoom() {  
2    useEffect(() => {  
3      const connection = createConnection();  
4  
5      connection.connect();  
6  
7      return () => {  
8        connection.disconnect();  
9      };  
10   }, []);  
11  
12   return <h1>Welcome to the chat!</h1>;  
13 }
```


When You Might Not Need an Effect

- **Data fetching**
 - Use framework's built-in data fetching mechanism or data fetching libraries
- **Calculate something during render**
 - Write in the component function body
 - For expensive calculations, Use `useMemo``
- **Reset all state when a prop changes**
 - Pass a different `key``
- **Adjust some state when a prop changes**
 - Set state while rendering
 - Or modify the logic
- **Notify parent components about state changes**
 - Write in the event handler
- **Send an event-specific POST request**
 - Write in the event handler
- **Subscribe to an external store**
 - Prefer using `useSyncExternalStore``
- **Initialize the application**
 - Write outside the component

When You Might Not Need an Effect

Data fetching

● Avoid: Fetch data via effect without a cleanup function

```
1  function SearchResults({ query }) {  
2    const [results, setResults] = useState([]);  
3  
4    // ● Race conditions: bug when the newer request finishes first  
5    useEffect(() => {  
6      fetchResults(query).then((json) => {  
7        setResults(json);  
8      });  
9    }, [query]);  
10  
11    // ...  
12  }
```

When You Might Not Need an Effect

Data fetching

✓ Better: Add cleanup function to avoid race conditions

```
1  function SearchResults({ query }) {
2    const [results, setResults] = useState([]);
3
4    useEffect(() => {
5      let ignore = false;
6      fetchResults(query).then((json) => {
7        // Skip UI update if there is a newer request
8        if (!ignore) {
9          setResults(json);
10        }
11      });
12      // If a new request starts, ignore the current one
13      return () => {
14        ignore = true;
15      };
16    }, [query]);
17
18    // ...
19  }
```

When You Might Not Need an Effect

Data fetching

Downsides of data fetching in effects:

- Need to care about race conditions
- Effects don't run on the server
- Fetching directly in Effects makes it easy to create “network waterfalls”
- Fetching directly in Effects usually means you don't preload or cache data

Better options:

- If you use a framework like Next.js, use its built-in data fetching mechanism
 - e.g. `getServerSideProps`` & `getStaticProps`` in Next.js
- Otherwise, consider using or building a client-side cache
 - e.g. useSWR, React Query, React Router 6.4+

When You Might Not Need an Effect

Calculate something during render

● Avoid: Filter TODO using effects

```
1  function getFilteredTodos(todos, filter) {  
2    // ...  
3  }  
4  
5  function TodoList({ todos, filter }) {  
6    // ● Redundant state and unnecessary Effect  
7    const [visibleTodos, setVisibleTodos] = useState([]);  
8    useEffect(() => {  
9      setVisibleTodos(getFilteredTodos(todos, filter));  
10   }, [todos, filter]);  
11  
12   // ...  
13 }
```

When You Might Not Need an Effect

Calculate something during render

✓ Do: Calculate in the function body

```
1  function getFilteredTodos(todos, filter) {  
2    // ...  
3  }  
4  
5  function TodoList({ todos, filter }) {  
6    const visibleTodos = getFilteredTodos(todos, fi  
7  
8    // ...  
9  }
```

✓ Do: Cache using `useMemo``

```
1  function getFilteredTodos(todos, filter) {  
2    // ...  
3  }  
4  
5  function TodoList({ todos, filter }) {  
6    const visibleTodos = useMemo(  
7      () => getFilteredTodos(todos, filter),  
8      [todos, filter]  
9    );  
10  
11    // ...  
12  }
```

When You Might Not Need an Effect

Reset all state when a prop changes

● Avoid: Reset state via effect

```
1  export default function ProfilePage({ userId }) {  
2    const [comment, setComment] = useState('');  
3  
4    // ● Comment is old during the first render  
5    useEffect(() => {  
6      setComment('');  
7    }, [userId]);  
8  
9    // ...  
10 }
```

When You Might Not Need an Effect

Reset all state when a prop changes

✓ Do: Reset state via ``key``

```
1  export default function ProfilePage({ userId }) {
2    return (
3      <Profile
4        userId={userId}
5        key={userId} // userId change will recreate the <Profile>
6      />
7    );
8  }
9
10 function Profile({ userId }) {
11   const [comment, setComment] = useState('');
12
13   // ...
14 }
```


When You Might Not Need an Effect

Adjust some state when a prop changes

● Avoid: Adjust state via effect

```
1  function List({ items }) {  
2    const [isReverse, setIsReverse] = useState(false);  
3    const [selection, setSelection] = useState(null);  
4  
5    // ● When items prop changes, selection state is stale at first  
6    useEffect(() => {  
7      setSelection(null);  
8    }, [items]);  
9  
10   // ...  
11 }
```

When You Might Not Need an Effect

Adjust some state when a prop changes

🟡 Better: Adjust the state while rendering

```
1  function List({ items }) {
2    const [isReverse, setIsReverse] = useState(false);
3    const [selection, setSelection] = useState(null);
4
5    // When items prop changes, List will immediately re-render
6    //
7    const [prevItems, setPrevItems] = useState(items);
8    if (items !== prevItems) {
9      setPrevItems(items);
10     setSelection(null);
11   }
12
13   // ...
14 }
```

When You Might Not Need an Effect

Adjust some state when a prop changes

✅ Best: Modify your logic so that you can do one of the following:

- Calculate everything during rendering
- Reset all state with a `key`

```
1  function List({ items }) {  
2    const [isReverse, setIsReverse] = useState(false);  
3    const [selectedId, setSelectedId] = useState(null);  
4    // Calculate everything during rendering  
5    const selection = items.find((item) => item.id === selectedId) ?? null;  
6  
7    // ...  
8  }
```

When You Might Not Need an Effect

Notify parent components about state changes

● Avoid: Notify parent via effect

```
1  function Toggle({ onChange }) {  
2    const [isOn, setIsOn] = useState(false);  
3  
4    // ● The onChange handler runs too late  
5    useEffect(() => {  
6      onChange(isOn);  
7    }, [isOn, onChange]);  
8  
9    function handleClick() {  
10     setIsOn(!isOn);  
11   }  
12  
13   // ...  
14 }
```

When You Might Not Need an Effect

Notify parent components about state changes

✅ Do: Notify parent via event

```
1  function Toggle({ onChange }) {  
2    const [isOn, setIsOn] = useState(false);  
3  
4    function updateToggle(nextIsOn) {  
5      // Perform all updates during  
6      // the event that caused them  
7      setIsOn(nextIsOn);  
8      onChange(nextIsOn);  
9    }  
10  
11    function handleClick() {  
12      updateToggle(!isOn);  
13    }  
14  
15    // ...  
16  }
```

✅ Do: Let parent controls the state

```
1  function Toggle({ isOn, onChange }) {  
2    function handleClick() {  
3      onChange(!isOn);  
4    }  
5  
6    // ...  
7  }
```

When You Might Not Need an Effect

Send an event-specific POST request

● Avoid: Event-specific logic inside an Effect

```
1  function Form() {
2    const [firstName, setFirstName] = useState('');
3    const [lastName, setLastName] = useState('');
4    const [jsonToSubmit, setJsonToSubmit] = useState(null);
5
6    function handleSubmit(e) {
7      e.preventDefault();
8      setJsonToSubmit({ firstName, lastName });
9    }
10
11    // ● POST request is not caused by the form being displayed
12    useEffect(() => {
13      if (jsonToSubmit !== null) {
14        post('/api/register', jsonToSubmit);
15      }
16    }, [jsonToSubmit]);
17
18    // ...
19  }
```

When You Might Not Need an Effect

Send an event-specific POST request

✅ Do: Event-specific logic in the event handler

```
1  function Form() {  
2    const [firstName, setFirstName] = useState('');  
3    const [lastName, setLastName] = useState('');  
4  
5    function handleSubmit(e) {  
6      e.preventDefault();  
7      post('/api/register', { firstName, lastName });  
8    }  
9  
10   // ...  
11 }
```

When You Might Not Need an Effect

Subscribe to an external store

🟡 Not ideal: Subscribe to a browser event in an effect

```
1  function ChatIndicator() {
2    const [isOnline, setIsOnline] = useState(true);
3    useEffect(() => {
4      function updateState() {
5        setIsOnline(navigator.onLine);
6      }
7
8      updateState();
9
10     window.addEventListener('online', updateState);
11     window.addEventListener('offline', updateState);
12     return () => {
13       window.removeEventListener('online', updateState);
14       window.removeEventListener('offline', updateState);
15     };
16   }, []);
17   // ...
18 }
```


When You Might Not Need an Effect

Subscribe to an external store

✅ Do: Subscribe to a browser event using `useSyncExternalStore`

```
1  function subscribe(callback) {
2    window.addEventListener('online', callback);
3    window.addEventListener('offline', callback);
4    return () => {
5      window.removeEventListener('online', callback);
6      window.removeEventListener('offline', callback);
7    };
8  }
9
10 function ChatIndicator() {
11   const isOnline = useSyncExternalStore(
12     subscribe, // React won't resubscribe for as long as you pass the same function
13     () => navigator.onLine, // How to get the value on the client
14     () => true // How to get the value on the server
15   );
16
17   // ...
18 }
```

When You Might Not Need an Effect

Initialize the application

● Avoid: Effects with logic that should only ever run once

```
1  function App() {  
2    // ● It might run twice in the dev environment  
3    useEffect(() => {  
4      loadDataFromLocalStorage();  
5      checkAuthToken();  
6    }, []);  
7  
8    // ...  
9  }
```

When You Might Not Need an Effect

Initialize the application

✅ Do: Add a variable to keep track if the code has run

```
1  let didInit = false;
2
3  function App() {
4    useEffect(() => {
5      if (!didInit) {
6        didInit = true;
7        // Only runs once per app load
8        loadDataFromLocalStorage();
9        checkAuthToken();
10     }
11   }, []);
12   // ...
13 }
```

✅ Do: Run during module initialization and before app render

```
1  // Only runs once per app load
2  checkAuthToken();
3  loadDataFromLocalStorage();
4
5  function App() {
6    // ...
7  }
```

Recap

What is an effect

- Runs after render
- Doesn't run on the server
- Mount effect might run twice on the dev environment

Recap

When you can use an effect

Effects are designed to run side effects that connect your component with **external systems after render**. E.g.

- Network
- Browser APIs
- Third party libraries

Recap

When you might not need an effect

Before you write an effect, pause and ask yourself **if there is a better solution**. E.g.

Situation	Better Option
Data fetching	Use a library like <code>useSWR</code>
Calculate something during render	Run in the component function body
Reset all state when a prop changes	Use the <code>key</code> prop
Adjust some state when a prop changes	Set state while rendering or modify the logic
Notify parent components about state changes / Send an event-specific POST request	Use event handlers
Subscribe to an external store	Use the built-in <code>useSyncExternalStore</code> hook
Initialize the application	Run outside the app component

References

- [You Might Not Need an effect](#)
- [Synchronizing with Effects](#)
- [hook-flow](#)