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

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

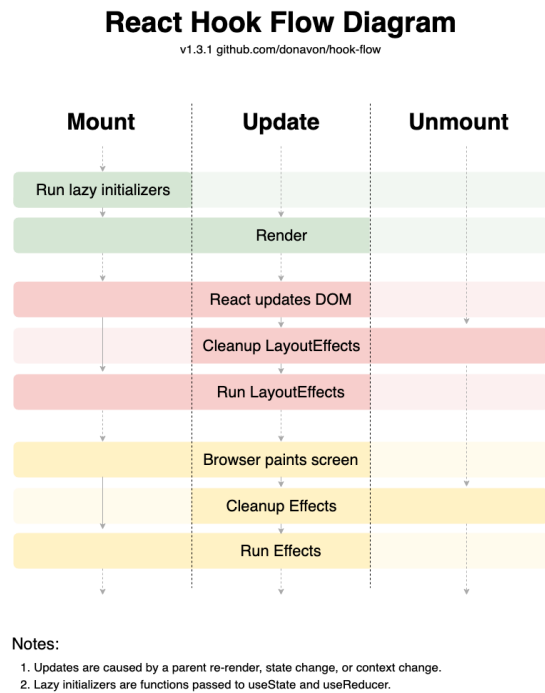

*Diagram from https://github.com/donavon/hook-flow

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

```
function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isPlaying) {
      ref.current.play();
    } else {
      ref.current.pause();
    }
  }, [isPlaying]);

  return <video ref={ref} src={src} loop playsInline />;
}
```

When You Can Use an Effect

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

```
function ChatRoom() {
  useEffect(() => {
    const connection = createConnection();

    connection.connect();

    return () => {
      connection.disconnect();
    };
  }, []);

  return <h1>Welcome to the chat!</h1>;
}
```

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

```
function SearchResults({ query }) {
  const [results, setResults] = useState([]);

  // 🔴 Race conditions: bug when the newer request finishes first
  useEffect(() => {
    fetchResults(query).then((json) => {
      setResults(json);
    });
  }, [query]);

  // ...
}
```

When You Might Not Need an Effect

Data fetching

✅ Better: Add cleanup function to avoid race conditions

```
function SearchResults({ query }) {
  const [results, setResults] = useState([]);

  useEffect(() => {
    let ignore = false;
    fetchResults(query).then((json) => {
      // Skip UI update if there is a newer request
      if (!ignore) {
        setResults(json);
      }
    });
    // If a new request starts, ignore the current one
    return () => {
      ignore = true;
    };
  }, [query]);

  // ...
}
```

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

```
function getFilteredTodos(todos, filter) {
  // ...
}

function TodoList({ todos, filter }) {
  // 🔴 Redundant state and unnecessary Effect
  const [visibleTodos, setVisibleTodos] = useState([]);
  useEffect(() => {
    setVisibleTodos(getFilteredTodos(todos, filter));
  }, [todos, filter]);

  // ...
}
```

When You Might Not Need an Effect

Calculate something during render

✅ Do: Calculate in the function body

```
function getFilteredTodos(todos, filter) {
  // ...
}

function TodoList({ todos, filter }) {
  const visibleTodos = getFilteredTodos(todos, fi
  // ...
}
```

✅ Do: Cache using `useMemo`

```
function getFilteredTodos(todos, filter) {
  // ...
}

function TodoList({ todos, filter }) {
  const visibleTodos = useMemo(
    () => getFilteredTodos(todos, filter),
    [todos, filter]
  );

  // ...
}
```

When You Might Not Need an Effect

Reset all state when a prop changes

🔴 Avoid: Reset state via effect

```
export default function ProfilePage({ userId }) {
  const [comment, setComment] = useState('');

  // 🔴 Comment is old during the first render
  useEffect(() => {
    setComment('');
  }, [userId]);

  // ...
}
```

When You Might Not Need an Effect

Reset all state when a prop changes

✅ Do: Reset state via `key`

```jsx
export default function ProfilePage({ userId }) {
  return (
    <Profile
      userId={userId}
      key={userId} // userId change will recreate the <Profile>
    />
  );
}

function Profile({ userId }) {
  const [comment, setComment] = useState('');

  // ...
}
```

When You Might Not Need an Effect

Adjust some state when a prop changes

🔴 Avoid: Adjust state via effect

```
function List({ items }) {
  const [selection, setSelection] = useState(null);

  // 🔴 When items prop changes, selection state is stale at first
  useEffect(() => {
    setSelection(null);
  }, [items]);

  // ...
}
```

When You Might Not Need an Effect

Adjust some state when a prop changes

🟡 Better: Adjust the state while rendering

```
function List({ items }) {
  const [selection, setSelection] = useState(null);

  // When items prop changes, List will immediately re-render
  //
  const [prevItems, setPrevItems] = useState(items);
  if (items !== prevItems) {
    setPrevItems(items);
    setSelection(null);
  }

  // ...
}
```

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`

```
function List({ items }) {
  const [selectedId, setSelectedId] = useState(null);
  // Calculate everything during rendering
  const selection = items.find((item) => item.id === selectedId) ?? null;

  // ...
}
```

When You Might Not Need an Effect

Notify parent components about state changes

🔴 Avoid: Notify parent via effect

```
function Toggle({ onChange }) {
  const [isOn, setIsOn] = useState(false);

  // 🔴 The onChange handler runs too late
  useEffect(() => {
    onChange(isOn);
  }, [isOn, onChange]);

  function handleClick() {
    setIsOn(!isOn);
  }

  // ...
}
```

When You Might Not Need an Effect

Notify parent components about state changes

✅ Do: Notify parent via event

```
function Toggle({ onChange }) {
  const [isOn, setIsOn] = useState(false);

  function updateToggle(nextIsOn) {
    // Perform all updates during
    // the event that caused them
    setIsOn(nextIsOn);
    onChange(nextIsOn);
  }

  function handleClick() {
    updateToggle(!isOn);
  }

  // ...
}
```

✅ Do: Let parent controls the state

```
function Toggle({ isOn, onChange }) {
  function handleClick() {
    onChange(!isOn);
  }

  // ...
}
```

When You Might Not Need an Effect

Send an event-specific POST request

🔴 Avoid: Event-specific logic inside an Effect

```
function Form() {
  const [firstName, setFirstName] = useState('');
  const [lastName, setLastName] = useState('');
  const [jsonToSubmit, setJsonToSubmit] = useState(null);

  function handleSubmit(e) {
    e.preventDefault();
    setJsonToSubmit({ firstName, lastName });
  }

  // 🔴 POST request is not caused by the form being displayed
  useEffect(() => {
    if (jsonToSubmit !== null) {
      post('/api/register', jsonToSubmit);
    }
  }, [jsonToSubmit]);

  // ...
}
```

When You Might Not Need an Effect

Send an event-specific POST request

✅ Do: Event-specific logic in the event handler

```
function Form() {
  const [firstName, setFirstName] = useState('');
  const [lastName, setLastName] = useState('');

  function handleSubmit(e) {
    e.preventDefault();
    post('/api/register', { firstName, lastName });
  }

  // ...
}
```

When You Might Not Need an Effect

Subscribe to an external store

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

```
function ChatIndicator() {
  const [isOnline, setIsOnline] = useState(true);
  useEffect(() => {
    function updateState() {
      setIsOnline(navigator.onLine);
    }

    updateState();

    window.addEventListener('online', updateState);
    window.addEventListener('offline', updateState);
    return () => {
      window.removeEventListener('online', updateState);
      window.removeEventListener('offline', updateState);
    };
  }, []);
  // ...
}
```

When You Might Not Need an Effect

Subscribe to an external store

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

```
function subscribe(callback) {
  window.addEventListener('online', callback);
  window.addEventListener('offline', callback);
  return () => {
    window.removeEventListener('online', callback);
    window.removeEventListener('offline', callback);
  };
}

function ChatIndicator() {
  const isOnline = useSyncExternalStore(
    subscribe, // React won't resubscribe for as long as you pass the same function
    () => navigator.onLine, // How to get the value on the client
    () => true // How to get the value on the server
  );

  // ...
}
```

When You Might Not Need an Effect

Initialize the application

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

```
function App() {
  // 🔴 It might run twice in the dev environment
  useEffect(() => {
    loadDataFromLocalStorage();
    checkAuthToken();
  }, []);

  // ...
}
```

When You Might Not Need an Effect

Initialize the application

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

```
let didInit = false;

function App() {
  useEffect(() => {
    if (!didInit) {
      didInit = true;
      // Only runs once per app load
      loadDataFromLocalStorage();
      checkAuthToken();
    }
  }, []);
  // ...
}
```

✅ Do: Run during module initialization and before app render

```
// Only runs once per app load
checkAuthToken();
loadDataFromLocalStorage();

function App() {
  // ...
}
```

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

Situation	**Better Option**
Data fetching	Use a library like `useSWR`
Calculate something during render	Run in the component function body
Reset all state when a prop changes	Use the `key` 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 `useSyncExternalStore` hook
Initialize the application	Run outside the app component

References

- You Might Not Need an effect
- Synchronizing with Effects
- hook-flow