USE CASES OF REACT HOOKS

1. `useState`

Use for: Managing local component state.

```
const [count, setCount] = useState(0);
```

Gotcha: Updates are async. Avoid directly logging `count` right after `setCount`.

2. `useEffect`

Use for:

- Data fetching
- Event listeners
- Subscriptions
- Manual DOM manipulation
- Syncing props to state

```
useEffect(() => {
  fetchData();
}, []);
```

Gotchas:

- **Dependency array confusion** — always include everything you use inside the effect unless it's stable or intentionally omitted.
- **Re-renders** can cause effects to run multiple times if dependencies change.

3. `useRef`

Use for:

- Accessing DOM elements
- Keeping mutable values between renders (but not causing re-renders)

```
const inputRef = useRef<HTMLInputElement>(null);
```

Gotcha: `ref.current` doesn't trigger a render on change — don't treat it like state.

4. `useMemo`

Use for:

- Expensive computations you don't want to recalculate on every render
- Memoizing derived values that depend on props/state

```
const sortedItems = useMemo(() => sortItems(items),
[items]);
```

Gotcha: `useMemo` is a **performance optimization**, not a correctness tool — don't use it just to stop re-renders unless there's actual cost.

5. `useCallback`

Use for: Memoizing functions passed to child components or dependencies of `useEffect`.

```
const handleClick = useCallback(() => doSomething(id),
[id]);
```

Gotcha: Don't overuse — it's useful only when functions are re-created unnecessarily and causing re-renders or effect triggers.

6. `useContext`

Use for: Accessing global/shared state like themes, user data, etc.

Gotcha: Context changes trigger **all consumers** to re-render. Use selectors or split context if perf suffers.

7. `useReducer`

Use for: Complex state logic (e.g., multiple related values, undo stacks).

```
const [state, dispatch] = useReducer(reducer,
initialState);
```

Gotcha: Slightly harder to set up than `useState`, but more scalable for logic-heavy state.

CAVEATS / “GOTCHAS” IN HOOKS

1. **Renders first, effects second.** `useEffect` does not block paint, so anything async won’t stop flicker.
2. **Too many effects can lead to “effect soup.”** Combine where logical.
3. **Infinite loops happen when dependency arrays are wrong.**
4. **Avoid side effects inside render.** Never call effects or setters outside `useEffect`, `useCallback`, etc.
5. **Avoid `useMemo`/`useCallback` for premature optimization.** Only use if there's a measured benefit.

ALTERNATIVES TO `useEffect`

Sometimes you **don’t need `useEffect`** — here's when:

1. Derived values → use computed properties instead:

```
// WRONG Don't do this
useEffect(() => {
  setFullName(first + ' ' + last);
}, [first, last]);

// Do this instead
const fullName = `${first} ${last}`;
```

2. Initialize state from props → do this directly in `useState`

```
const [value, setValue] = useState(() =>
props.initialValue);
```

3. Reacting to controlled inputs → manage state via props/handlers instead of syncing with useEffect

Other (not obvious) Uses of `useMemo`

1. Memoizing a config object

```
const config = useMemo(() => ({ theme, size }), [theme,
size]);
// Avoids unnecessary re-renders in children receiving
`config`
```

2. Filtering or transforming data

```
const visibleTodos = useMemo(() => {
  return todos.filter(todo => !todo.completed);
}, [todos]);
```

3. Avoiding stale closures

```
const interval = useMemo(() => {
  return setInterval(() => {
    console.log('Running');
  }, 1000);
}, []);
```

4. Stable dependency for `useEffect`

```
const stableQuery = useMemo(() => createQuery(params),
[params]);

useEffect(() => {
  runQuery(stableQuery);
}, [stableQuery]);
```

When to use Which Hook

Hook	When to Use It	Example Use Case	Avoid If...
`useState`	Component needs to re-render based on internal	Tracking form fields, toggling a modal	You don’t need UI to update or you need refs
`useEffect`	Side effects (fetch, subscribe, sync external	Fetching data on mount, updating document title	You can derive the value directly from props/state
`useRef`	Persist values between renders, or reference a DOM node	Focus an input, track animation frame ID	You mistakenly expect changes to trigger re-renders

`useMemo`	Cache expensive computation or stable	Filtered list, options array, config object passed to a	Computation is cheap or there's no measurable
`useCallback`	Memoize event handlers or functions used in deps	Button handler passed to child component	The function is local and not causing re-renders
`useReducer`	Complex state logic, especially related values	Shopping cart, undo history, toggles with	Simple state like a counter
`useContext`	Share state across tree without prop drilling	Theme, user session, app-wide settings	Too many re-renders: consider memoized
`useLayoutEffect`	Same as useEffect but fires *before* paint	Measuring DOM size before display (e.g. tooltip placement)	You don't need layout-dependent operations

Alternatives to `useEffect`

INSTEAD OF:

```
useEffect(() => {
  setFiltered(data.filter(d => d.isActive));
}, [data]);
```

USE:

```
const filtered = useMemo(() => data.filter(d =>
d.isActive), [data]);
```

Use `useMemo` to

Prevent unnecessary recalculations:

```
const sortedList = useMemo(() => sortHeavy(list), [list]);
```

You are passing derived props to children:

```
const chartConfig = useMemo(() => ({ theme, data }),
[theme, data]);
<Chart config={chartConfig} />
```

You're using the same object as a dependency in `useEffect`:

```
const filters = useMemo(() => buildFilters(query),
[query]);

useEffect(() => {
  fetchResults(filters);
}, [filters]);
```

Bad `useMemo` usage

```
//Unnecessary memoization
const doubled = useMemo(() => num * 2, [num]);
//Instead, write: const doubled = num * 2;
```

Real-life Scenarios

Scenario	Recommended Hook/Pattern
Debounced search bar	`useEffect` + debounce with `setTimeout`
Form validation	`useState` or `useReducer`
Focus on mount	`useRef` + `useEffect`
Animated progress bar	`useEffect` + requestAnimationFrame
Load-once config fetch	`useEffect(() => { fetch() }, [])`
Avoid re-filtering/sorting large arrays	`useMemo`
Sharing user theme state across app	`useContext`

1. Debounced Search Input (via `useEffect` + `setTimeout`)

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

function SearchBox() {
  const [query, setQuery] = useState('');
  const [debouncedQuery, setDebouncedQuery] =
useState(query);

  useEffect(() => {
    const handler = setTimeout(() => {
      setDebouncedQuery(query);
    }, 500); // debounce delay

    return () => clearTimeout(handler); // cleanup if query
changes before 500ms
  }, [query]);

  useEffect(() => {
    if (debouncedQuery) {
      fetch(`/api/search?q=${debouncedQuery}`);
    }
  }, [debouncedQuery]);

  return <input value={query} onChange={(e) =>
setQuery(e.target.value)} />;
}
```

First effect delays syncing `query` into `debouncedQuery`. The second effect triggers actual fetch.

2. Filtered List with `useMemo`

```
const filteredUsers = useMemo(() => {
  return users.filter(user =>
user.name.toLowerCase().includes(searchTerm.toLowerCase()))
;
}, [users, searchTerm]);
```

Why useMemo: Prevents re-filtering on every render unless `users` or `searchTerm` changes.

3. Derived Value Without `useEffect`

```
//Don't use
useEffect(() => {
  setIsValid(email.includes('@'));
}, [email]);
```

```
// Use
const isValid = email.includes('@');
```

4. Memoized Event Handler with `useCallback`

```
const handleAddToCart = useCallback(() => {
  dispatch({ type: 'add', item });
}, [item, dispatch]);

<Button onClick={handleAddToCart}>Add</Button>
```

Why useCallback: Keeps the same function reference across renders — especially important if `<Button>` is memoized.

5. Avoiding Layout Shift with `useLayoutEffect`

```
const boxRef = useRef<HTMLDivElement>(null);

useLayoutEffect(() => {
  const box = boxRef.current;
  if (box) {
    const width = box.getBoundingClientRect().width;
    console.log('Box width before paint:', width);
  }
}, []);
```

Layout effect: Runs synchronously before paint — avoids flicker if measuring DOM dimensions.

6. Preserve Value Between Renders with `useRef`

```
const renderCount = useRef(0);
renderCount.current++;

console.log('Component rendered', renderCount.current,
'times');
```

Why useRef: Doesn't trigger re-render. Great for tracking things like render count, timers, or external libs.

7. Imperative DOM Focus with `useRef` + `useEffect`

```
const inputRef = useRef<HTMLInputElement>(null);

useEffect(() => {
  inputRef.current?.focus();
}, []);

return <input ref={inputRef} />;
```

Why this works: You're directly accessing the DOM after it's mounted — classic `useEffect` case.

Mini Project: Product Search Dashboard

- **Debounced search input**
- **Filtered + sorted product list** using `useMemo`
- **Fetch API** with `useEffect`
- **Focus-first-input** with `useRef`
- **Track renders** with `useRef`
- **Memoized event handlers** with `useCallback`
- **Derived state** for input validation (no `useEffect`)
- **Progress bar animation** using `requestAnimationFrame` + `useEffect`

Components We Use

- `<App />` – Parent container

- `<SearchBar />` – Controlled input with debouncing
- `<ProductList />` – Filtered + sorted list of fetched products
- `<ProgressBar />` – Canvas animated loading bar
- `<ProductCard />` – Memoized card component

1. App.tsx

```
import React, { useState, useEffect } from 'react';
import SearchBar from './SearchBar';
import ProductList from './ProductList';
import ProgressBar from './ProgressBar';

const App = () => {
  const [products, setProducts] = useState([]);
  const [search, setSearch] = useState('');

  useEffect(() => {
    fetch('/products.json')
      .then(res => res.json())
      .then(setProducts);
  }, []);

  return (
    <div>
      <h1>Product Search</h1>
      <SearchBar value={search} onChange={setSearch} />
      <ProgressBar />
      <ProductList products={products} query={search} />
    </div>
  );
};

export default App;
```

2. SearchBar.tsx

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

```
const SearchBar = ({ value, onChange }) => {
  const [internal, setInternal] = useState(value);
  const inputRef = useRef(null);

  // Focus input on mount
  useEffect(() => {
    inputRef.current?.focus();
  }, []);

  useEffect(() => {
    const timeout = setTimeout(() => {
      onChange(internal);
    }, 500);
    return () => clearTimeout(timeout);
  }, [internal]);

  return (
    <input
      ref={inputRef}
      value={internal}
      onChange={e => setInternal(e.target.value)}
      placeholder="Search products..."
    />
  );
};

export default SearchBar;
```

3. ProductList.tsx

```
import React, { useMemo } from 'react';
import ProductCard from './ProductCard';

const ProductList = ({ products, query }) => {
  const filtered = useMemo(() => {
    const lower = query.toLowerCase();
    return products
      .filter(p => p.name.toLowerCase().includes(lower))
      .sort((a, b) => a.price - b.price);
  }, [products, query]);

  return (
    <div>
      {filtered.map(p => (
        <ProductCard key={p.id} product={p} />
      ))}
    </div>
  );
};

export default ProductList;
```

4. ProductCard.tsx

```
import React, { memo } from 'react';

const ProductCard = memo(({ product }) => {
  return (
    <div>
      <h2>{product.name}</h2>
      <p>${product.price.toFixed(2)}</p>
    </div>
  );
});

export default ProductCard;
```

5. ProgressBar.tsx

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

const ProgressBar = () => {
  const canvasRef = useRef(null);
  const progressRef = useRef(0);

  useEffect(() => {
    const canvas = canvasRef.current;
    const ctx = canvas.getContext('2d');
    const animate = () => {
      progressRef.current += 0.5;
      if (progressRef.current > 100) progressRef.current =
0;

      ctx.clearRect(0, 0, canvas.width, canvas.height);
      ctx.fillStyle = 'blue';
      ctx.fillRect(0, 0, (canvas.width *
progressRef.current) / 100, 20);

      requestAnimationFrame(animate);
    };

    animate();
  }, []);

  return <canvas ref={canvasRef} width="200" height="20" /
>;
};

export default ProgressBar;
```

6. products.json

Put this in `/public/products.json`:

```
[
  { "id": 1, "name": "MacBook Pro", "price": 1299 },
  { "id": 2, "name": "iPhone", "price": 899 },
  { "id": 3, "name": "Magic Mouse", "price": 79 },
  { "id": 4, "name": "iPad", "price": 499 }
]
```

Hooks We Used

Feature	Hook(s) Used
Debounced input	`useEffect`, `setTimeout`
API fetch	`useEffect`
DOM focus	`useRef`, `useEffect`

Memoized data + sorting	`useMemo`
Derived state (validation)	Plain expressions
Animation loop	`useRef`, `useEffect`, `requestAnimationFrame`
Memoized component	`React.memo`