react-use-async: A React Hook for Async Operations That Doesn't Overthink It

Kabui, Charles

2026-02-17

Table of Contents

A lightweight React hook for managing async operations with loading states, error handling, and data merging — in under 2KB.

The Pattern

Every React app has this code somewhere:

```
const [data, setData] = useState(null);
const [loading, setLoading] = useState(true);
const [error, setError] = useState(null);

useEffect(() => {
  setLoading(true);
  fetchData()
    .then(setData)
    .catch(setError)
    .finally(() => setLoading(false));
}, []);
```

It's boilerplate. You write it once, then copy-paste it everywhere with slight variations. Sometimes you need pagination (merge new data with old). Sometimes you need manual re-execution. Sometimes you want to wait before executing. Every time, you rewrite the same loading/error/data dance.

The Hook

`useAsync` wraps this pattern into a single hook:

```
import { useAsync } from '@mckabue/react-use-async';

const UserProfile = ({ userId }) => {
  const { data, isLoading, error } = useAsync(
    () => fetch(`/api/users/${userId}`).then(r => r.json()),
    [userId],
  );

  if (isLoading) return <Spinner />;
  if (error) return <Error message={error.message} />;
  return <Profile user={data} />;
};
```

That's it. Executes on mount, re-executes when `userId` changes, tracks loading and error state.

Two Variants

`useAsync` — Execute on Mount

Runs the async function immediately when the component mounts (and on dependency changes):

```
const { data, isLoading, error } = useAsync(
  () => api.getWorkflows(),
  [],
);
```

`useDelayedAsync` — Execute on Demand

Same API, but doesn't run until you call `execute`:

```
const { data, isLoading, execute } = useDelayedAsync(
  (searchTerm: string) => api.search(searchTerm),
  [],
);

// Later, in response to user action:
<button onClick={() => execute('photos')}>Search</button>
```

Arguments you pass to `execute()` are forwarded to your async function.

Pagination with `continueWith`

The hook's `continueWith` method appends new data to existing data using a merge function:

```
const { data, isContinuing, continueWith } = useAsync(
  () => api.listItems({ page: 1 }),
  [],
);

const loadMore = continueWith(
  () => api.listItems({ page: nextPage }),
  (existing, newData) => ({
    ...newData,
    items: [...(existing?.items ?? []), ...newData.items],
```

```
  }),
);

return (
  <>
    {data?.items.map(item => <Item key={item.id} {...item} />)}
    <button onClick={loadMore} disabled={isContinuing}>
      {isContinuing ? 'Loading...' : 'Load More'}
    </button>
  </>
);
```

The `isContinuing` state is separate from `isExecuting`, so your initial content stays visible while more loads.

Full Return Type

```
{
  data: R | null;           // Latest resolved value
  isExecuting: boolean;     // true during execute() calls
  isContinuing: boolean;    // true during continueWith() calls
  isLoading: boolean;       // isExecuting || isContinuing
  error: Error | null;      // Latest error (cleared on next call)
  args: A;                  // Last arguments passed to execute()
  execute: (...args) => Promise<void>;
  continueWith: (callback, merger) => () => Promise<void>;
}
```

Why Not Use X?

TanStack Query / SWR: If you need caching, deduplication, background refetching, optimistic updates — use those. They're excellent.

This hook: If you need loading/error/data for a single async call without a query client wrapping your app. No providers. No cache keys. No configuration. Just a hook.

Installation

```
npm install @mckabue/react-use-async
```

Zero dependencies beyond React as a peer dependency. Under 2KB minified.

GitHub: https://github.com/mckabue/react-use-async **npm:** https://www.npmjs.com/package/@mckabue/react-use-async

---