

Error Handling

Errors can be divided into two categories: **expected errors** and **uncaught exceptions**:

- **Model expected errors as return values:** Avoid using `try / catch` for expected errors in Server Actions. Use `useActionState` [↗] to manage these errors and return them to the client.
- **Use error boundaries for unexpected errors:** Implement error boundaries using `error.tsx` and `global-error.tsx` files to handle unexpected errors and provide a fallback UI.

Good to know: These examples use React's `useActionState` hook, which is available in React 19 RC. If you are using an earlier version of React, use `useFormState` instead. See the [React docs](#) [↗] for more information.

Handling Expected Errors

Expected errors are those that can occur during the normal operation of the application, such as those from [server-side form validation](#) or failed requests. These errors should be handled explicitly and returned to the client.

Handling Expected Errors from Server Actions

Use the `useActionState` [↗] hook to manage the state of Server Actions, including handling errors. This approach avoids `try / catch` blocks for expected errors, which should be modeled as return values rather than thrown exceptions.

TS app/actions.ts

TypeScript ▾



```
1  'use server'
2
3  import { redirect } from 'next/navigation'
4
5  export async function createUser(prevState: any, formData: FormData) {
6    const res = await fetch('https://...')
7    const json = await res.json()
8
9    if (!res.ok) {
10     return { message: 'Please enter a valid email' }
11   }
12
13   redirect('/dashboard')
14 }
```

Then, you can pass your action to the `useActionState` hook and use the returned `state` to display an error message.

TS app/ui/signup.tsx

TypeScript ▾



```
1  'use client'
2
3  import { useActionState } from 'react'
4  import { createUser } from '@app/actions'
5
6  const initialState = {
7    message: '',
8  }
9
10 export function Signup() {
11   const [state, formAction] = useActionState(createUser, initialState)
12
13   return (
14     <form action={formAction}>
15       <label htmlFor="email">Email</label>
16       <input type="text" id="email" name="email" required />
17       { /* ... */ }
18       <p aria-live="polite">{state?.message}</p>
19       <button>Sign up</button>
20     </form>
21   )
22 }
```

You could also use the returned state to display a toast message from the client component.

Handling Expected Errors from Server Components

When fetching data inside of a Server Component, you can use the response to conditionally render an error message or redirect.

TS app/page.tsx

TypeScript ▾



```
1 export default async function Page() {
2   const res = await fetch(`https://...`)
3   const data = await res.json()
4
5   if (!res.ok) {
6     return 'There was an error.'
7   }
8
9   return '...'
10 }
```

Uncaught Exceptions

Uncaught exceptions are unexpected errors that indicate bugs or issues that should not occur during the normal flow of your application. These should be handled by throwing errors, which will then be caught by error boundaries.

- **Common:** Handle uncaught errors below the root layout with `error.js`.
- **Optional:** Handle granular uncaught errors with nested `error.js` files (e.g. `app/dashboard/error.js`)
- **Uncommon:** Handle uncaught errors in the root layout with `global-error.js`.

Using Error Boundaries

Next.js uses error boundaries to handle uncaught exceptions. Error boundaries catch errors in their child components and display a fallback UI instead of the component tree that crashed.

Create an error boundary by adding an `error.tsx` file inside a route segment and exporting a React component:

TS app/dashboard/error.tsx

TypeScript ▾

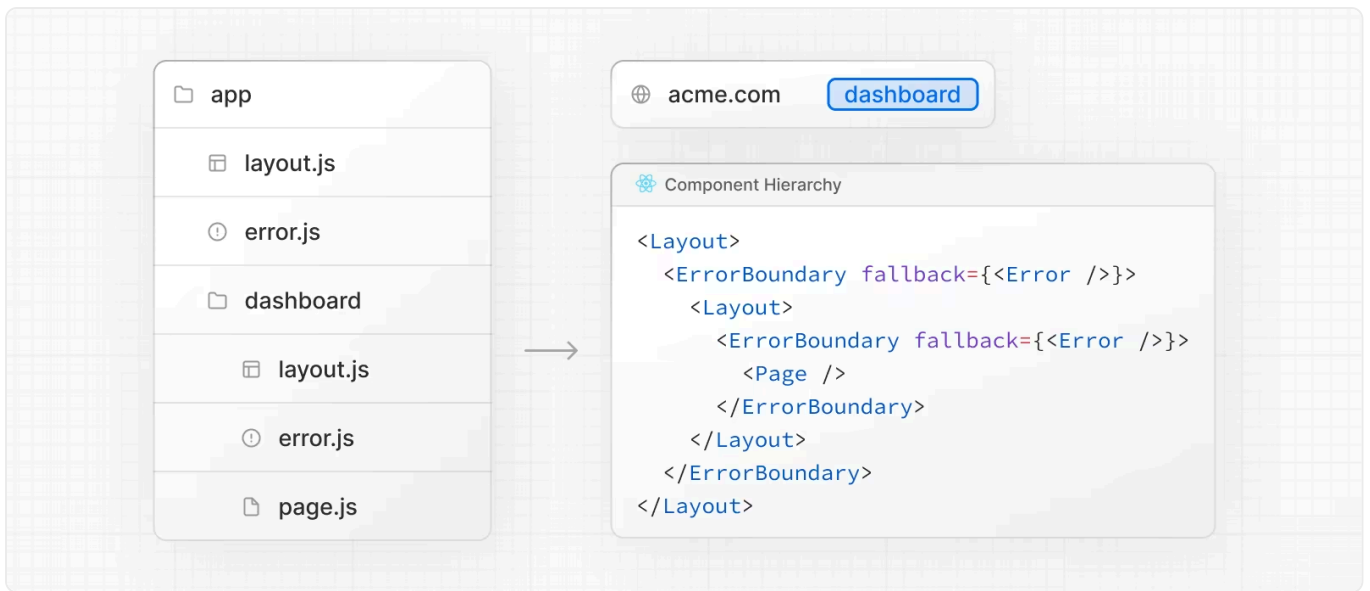


```
1  'use client' // Error boundaries must be Client Components
2
3  import { useEffect } from 'react'
4
5  export default function Error({
6    error,
7    reset,
8  }: {
9    error: Error & { digest?: string }
10    reset: () => void
11  }) {
12    useEffect(() => {
13      // Log the error to an error reporting service
14      console.error(error)
15    }, [error])
16
17    return (
18      <div>
19        <h2>Something went wrong!</h2>
20        <button
21          onClick={
22            // Attempt to recover by trying to re-render the segment
23            () => reset()
24          }
25        >
26          Try again
27        </button>
28      </div>
29    )
30  }
```

If you want errors to bubble up to the parent error boundary, you can `throw` when rendering the `error` component.

Handling Errors in Nested Routes

Errors will bubble up to the nearest parent error boundary. This allows for granular error handling by placing `error.tsx` files at different levels in the [route hierarchy](#).



Handling Global Errors

While less common, you can handle errors in the root layout using `app/global-error.js`, located in the root app directory, even when leveraging [internationalization](#). Global error UI must define its own `<html>` and `<body>` tags, since it is replacing the root layout or template when active.

TS app/global-error.tsx

TypeScript ▾



```
1  'use client' // Error boundaries must be Client Components
2
3  export default function GlobalError({
4    error,
5    reset,
6  }: {
7    error: Error & { digest?: string }
8    reset: () => void
9  }) {
10   return (
11     // global-error must include html and body tags
12     <html>
13       <body>
14         <h2>Something went wrong!</h2>
15         <button onClick={() => reset()}>Try again</button>
16       </body>
17     </html>
18   )
19 }
```

Next Steps

App Router > ... > File Conventions

[error.js](#)





API reference for the error.js special file.


Previous

< Linking and Navigating

Next

Loading UI and Streaming >

Was this helpful?    

	Resources	More	About Vercel	Legal
	Docs	Next.js Commerce	Next.js + Vercel	Privacy Policy
	Learn	Contact Sales	Open Source Software	
	Showcase	GitHub	GitHub	
	Blog	Releases	X	
	Analytics	Telemetry		
	Next.js Conf	Governance		
	Previews			

Subscribe to our newsletter

Stay updated on new releases and features, guides, and case studies.

you@domain.com

Subscribe