BLOG >

React 19 Beta

React 19 Beta is now available on npm! In this post, we'll give an overview of the new features in React 19, and how you can adopt them.

April 25, 2024 by The React Team

Note

This beta release is for libraries to prepare for React 19. App developers should upgrade to 18.3.0 and wait for React 19 stable as we work with libraries and make changes based on feedback.

React 19 Beta is now available on npm!

In our React 19 Beta Upgrade Guide, we shared step-by-step instructions for upgrading your app to React 19 Beta. In this post, we'll give an overview of the new features in React 19, and how you can adopt them.

- What's new in React 19
- Improvements in React 19
- How to upgrade

For a list of breaking changes, see the Upgrade Guide.

What's new in React 19

Actions

A common use case in React apps is to perform a data mutation and then update state in response. For example, when a user submits a form to change their name, you will make an API request, and then handle the response. In the past, you would need to handle pending states, errors, optimistic updates, and sequential requests manually.

For example, you could handle the pending and error state in `useState`:

```
// Before Actions
function UpdateName({}) {
  const [name, setName] = useState("");
  const [error, setError] = useState(null);
  const [isPending, setIsPending] = useState(false);

  const handleSubmit = async () => {
    setIsPending(true);
    const error = await updateName(name);
    setIsPending(false);
    if (error) {
      setError(error);
      return;
    }
    redirect("/path");
  };

  return (
    <div>
      <input value={name} onChange={(event) => setName(event.target.value)} /
      <button onClick={handleSubmit} disabled={isPending}>
        Update
      </button>
      {error && <p>{error}</p>}
```

```
    </div>
  );
}
```

In React 19, we're adding support for using async functions in transitions to handle pending states, errors, forms, and optimistic updates automatically.

For example, you can use `useTransition` to handle the pending state for you:

```
// Using pending state from Actions
function UpdateName({}) {
  const [name, setName] = useState("");
  const [error, setError] = useState(null);
  const [isPending, startTransition] = useTransition();

  const handleSubmit = async () => {
    startTransition(async () => {
      const error = await updateName(name);
      if (error) {
        setError(error);
        return;
      }
      redirect("/path");
    })
  };

  return (
    <div>
      <input value={name} onChange={(event) => setName(event.target.value)} /
      <button onClick={handleSubmit} disabled={isPending}>
        Update
      </button>
      {error && <p>{error}</p>}
    </div>
  );
}
```

The async transition will immediately set the `isPending` state to true, make the async request(s), and switch `isPending` to false after any transitions. This allows you to keep the current UI responsive and interactive while the data is changing.

Note

By convention, functions that use async transitions are called "Actions".

Actions automatically manage submitting data for you:

- **Pending state**: Actions provide a pending state that starts at the beginning of a request and automatically resets when the final state update is committed.
- **Optimistic updates**: Actions support the new `useOptimistic` hook so you can show users instant feedback while the requests are submitting.
- **Error handling**: Actions provide error handling so you can display Error Boundaries when a request fails, and revert optimistic updates to their original value automatically.
- **Forms**: `<form>` elements now support passing functions to the `action` and `formAction` props. Passing functions to the `action` props use Actions by default and reset the form automatically after submission.

Building on top of Actions, React 19 introduces `useOptimistic` to manage optimistic updates, and a new hook `React.useActionState` to handle common cases for Actions. In `react-dom` we're adding `<form>` Actions to manage forms automatically and `useFormStatus` to support the common cases for Actions in forms.

In React 19, the above example can be simplified to:

```
// Using <form> Actions and useActionState
function ChangeName({ name, setName }) {
  const [error, submitAction, isPending] = useActionState(
    async (previousState, formData) => {
      const error = await updateName(formData.get("name"));
      if (error) {
        return error;
      }
      redirect("/path");
    }
  );

  return (
    <form action={submitAction}>
      <input type="text" name="name" />
      <button type="submit" disabled={isPending}>Update</button>
      {error && <p>{error}</p>}
    </form>
  );
}
```

In the next section, we'll break down each of the new Action features in React 19.

New hook: `useActionState`

To make the common cases easier for Actions, we've added a new hook called `useActionState`:

```
const [error, submitAction, isPending] = useActionState(async (previousState,
  const error = await updateName(newName);
  if (error) {
    // You can return any result of the action.
```

```
    // Here, we return only the error.
    return error;
  }

  // handle success
});
```

`useActionState` accepts a function (the "Action"), and returns a wrapped Action to call. This works because Actions compose. When the wrapped Action is called, `useActionState` will return the last result of the Action as `data`, and the pending state of the Action as `pending`.

Note

`React.useActionState` was previously called `ReactDOM.useFormState` in the Canary releases, but we've renamed it and deprecated `useFormState`.

See #28491 for more info.

For more information, see the docs for `useActionState`.

React DOM: `<form>` Actions

Actions are also integrated with React 19's new `<form>` features for `react-dom`. We've added support for passing functions as the `action` and `formAction` props of `<form>`, `<input>`, and `<button>` elements to automatically submit forms with Actions:

```
<form action={actionFunction}>
```

When a `<form>` Action succeeds, React will automatically reset the form for uncontrolled components. If you need to reset the `<form>` manually, you can call the new `requestFormReset` React DOM API.

For more information, see the `react-dom` docs for `<form>`, `<input>`, and `<button>`.

React DOM: New hook: `useFormStatus`

In design systems, it's common to write design components that need access to information about the `<form>` they're in, without drilling props down to the component. This can be done via Context, but to make the common case easier, we've added a new hook `useFormStatus`:

```
import {useFormStatus} from 'react-dom';

function DesignButton() {
  const {pending} = useFormStatus();
  return <button type="submit" disabled={pending} />
}
```

`useFormStatus` reads the status of the parent `<form>` as if the form was a Context provider.

For more information, see the `react-dom` docs for `useFormStatus`.

New hook: `useOptimistic`

Another common UI pattern when performing a data mutation is to show the final state optimistically while the async request is underway. In React 19, we're adding a new hook called `useOptimistic` to make this easier:

```
function ChangeName({currentName, onUpdateName}) {
```

```
  const [optimisticName, setOptimisticName] = useOptimistic(currentName);

  const submitAction = async formData => {
    const newName = formData.get("name");
    setOptimisticName(newName);
    const updatedName = await updateName(newName);
    onUpdateName(updatedName);
  };

  return (
    <form action={submitAction}>
      <p>Your name is: {optimisticName}</p>
      <p>
        <label>Change Name:</label>
        <input
          type="text"
          name="name"
          disabled={currentName !== optimisticName}
        />
      </p>
    </form>
  );
}
```

The `useOptimistic` hook will immediately render the `optimisticName` while the `updateName` request is in progress. When the update finishes or errors, React will automatically switch back to the `currentName` value.

For more information, see the docs for `useOptimistic`.

New API: `use`

In React 19 we're introducing a new API to read resources in render: `use`.

For example, you can read a promise with `use`, and React will Suspend until the promise resolves:

```jsx
import {use} from 'react';

function Comments({commentsPromise}) {
  // `use` will suspend until the promise resolves.
  const comments = use(commentsPromise);
  return comments.map(comment => <p key={comment.id}>{comment}</p>);
}

function Page({commentsPromise}) {
  // When `use` suspends in Comments,
  // this Suspense boundary will be shown.
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <Comments commentsPromise={commentsPromise} />
    </Suspense>
  )
}
```

Note

`use` does not support promises created in render.

If you try to pass a promise created in render to `use`, React will warn:

Console

```
A component was suspended by an uncached promise. Creating promises inside a Client Component or hook is not yet supported, except via a Suspense-compatible library or framework.
```

To fix, you need to pass a promise from a suspense powered library or framework that supports caching for promises. In the future we plan to ship features to make it easier to cache promises in render.

You can also read context with `use`, allowing you to read Context conditionally such as after early returns:

```
import {use} from 'react';
import ThemeContext from './ThemeContext'

function Heading({children}) {
  if (children == null) {
    return null;
  }

  // This would not work with useContext
  // because of the early return.
  const theme = use(ThemeContext);
  return (
    <h1 style={{color: theme.color}}>
      {children}
    </h1>
  );
}
```

The `use` API can only be called in render, similar to hooks. Unlike hooks, `use` can be called conditionally. In the future we plan to support more ways to consume resources in render with `use`.

For more information, see the docs for `use`.

React Server Components

Server Components

Server Components are a new option that allows rendering components ahead of time, before bundling, in an environment separate from your client application or SSR server. This separate environment is the “server” in React

Server Components. Server Components can run once at build time on your CI server, or they can be run for each request using a web server.

React 19 includes all of the React Server Components features included from the Canary channel. This means libraries that ship with Server Components can now target React 19 as a peer dependency with a `react-server` export condition for use in frameworks that support the Full-stack React Architecture.

> **Note**
>
> ### How do I build support for Server Components?
>
> While React Server Components in React 19 are stable and will not break between major versions, the underlying APIs used to implement a React Server Components bundler or framework do not follow semver and may break between minors in React 19.x.
>
> To support React Server Components as a bundler or framework, we recommend pinning to a specific React version, or using the Canary release. We will continue working with bundlers and frameworks to stabilize the APIs used to implement React Server Components in the future.

For more, see the docs for React Server Components.

Server Actions

Server Actions allow Client Components to call async functions executed on the server.

When a Server Action is defined with the `"use server"` directive, your framework will automatically create a reference to the server function, and pass that reference to the Client Component. When that function is called on the client, React will send a request to the server to execute the function, and return the result.

> **Note**
>
> **There is no directive for Server Components.**
>
> A common misunderstanding is that Server Components are denoted by `"use server"`, but there is no directive for Server Components. The `"use server"` directive is used for Server Actions.
>
> For more info, see the docs for Directives.

Server Actions can be created in Server Components and passed as props to Client Components, or they can be imported and used in Client Components.

For more, see the docs for React Server Actions.

Improvements in React 19

`ref` as a prop

Starting in React 19, you can now access `ref` as a prop for function components:

```
function MyInput({placeholder, ref}) {
  return <input placeholder={placeholder} ref={ref} />
}
```

```
//...
<MyInput ref={ref} />
```

New function components will no longer need `forwardRef`, and we will be publishing a codemod to automatically update your components to use the new `ref` prop. In future versions we will deprecate and remove `forwardRef`.

> **Note**
>
> `refs` passed to classes are not passed as props since they reference the component instance.

Diffs for hydration errors

We also improved error reporting for hydration errors in `react-dom`. For example, instead of logging multiple errors in DEV without any information about the mismatch:

Console

```
Warning: Text content did not match. Server: “Server” Client: “Client”
  at span
  at App

Warning: An error occurred during hydration. The server HTML was
replaced with client content in <div>.

Warning: Text content did not match. Server: “Server” Client: “Client”
  at span
  at App

Warning: An error occurred during hydration. The server HTML was
replaced with client content in <div>.

Uncaught Error: Text content does not match server-rendered HTML.
  at checkForUnmatchedText
  …
```

We now log a single message with a diff of the mismatch:

Console

```
Uncaught Error: Hydration failed because the server rendered HTML
didn't match the client. As a result this tree will be regenerated on
the client. This can happen if an SSR-ed Client Component used:

- A server/client branch  if (typeof window !== 'undefined') .
- Variable input such as  Date.now()  or  Math.random()  which changes each
time it's called.
- Date formatting in a user's locale which doesn't match the server.
- External changing data without sending a snapshot of it along with
the HTML.
- Invalid HTML tag nesting.

It can also happen if the client has a browser extension installed
which messes with the HTML before React loaded.

https://react.dev/link/hydration-mismatch

  <App>
    <span>
+    Client
-    Server

  at throwOnHydrationMismatch
  …
```

`<Context>` as a provider

In React 19, you can render `<Context>` as a provider instead of `<Context.Provider>`:

```
const ThemeContext = createContext('');

function App({children}) {
  return (
    <ThemeContext value="dark">
      {children}
    </ThemeContext>
  );
}
```

New Context providers can use `<Context>` and we will be publishing a codemod to convert existing providers. In future versions we will deprecate `<Context.Provider>`.

Cleanup functions for refs

We now support returning a cleanup function from `ref` callbacks:

```
<input
  ref={(ref) => {
    // ref created

    // NEW: return a cleanup function to reset
    // the ref when element is removed from DOM.
    return () => {
      // ref cleanup
    };
  }}
/>
```

When the component unmounts, React will call the cleanup function returned from the `ref` callback. This works for DOM refs, refs to class components, and `useImperativeHandle`.

Note

Previously, React would call `ref` functions with `null` when unmounting the component. If your `ref` returns a cleanup function, React will now skip this step.

In future versions, we will deprecate calling refs with `null` when unmounting components.

Due to the introduction of ref cleanup functions, returning anything else from a `ref` callback will now be rejected by TypeScript. The fix is usually to stop using implicit returns, for example:

```
- <div ref={current => (instance = current)} />
+ <div ref={current => {instance = current}} />
```

The original code returned the instance of the `HTMLDivElement` and TypeScript wouldn't know if this was *supposed* to be a cleanup function or if you didn't want to return a cleanup function.

You can codemod this pattern with `no-implicit-ref-callback-return` .

`useDeferredValue` initial value

We've added an `initialValue` option to `useDeferredValue` :

```
function Search({deferredValue}) {
  // On initial render the value is ''.
  // Then a re-render is scheduled with the deferredValue.
  const value = useDeferredValue(deferredValue, '');

  return (
    <Results query={value} />
  );
}
```

When initialValue is provided, useDeferredValue will return it as value for the initial render of the component, and schedules a re-render in the background with the deferredValue returned.

For more, see useDeferredValue.

Support for Document Metadata

In HTML, document metadata tags like <title>, <link>, and <meta> are reserved for placement in the <head> section of the document. In React, the component that decides what metadata is appropriate for the app may be very far from the place where you render the <head> or React does not render the <head> at all. In the past, these elements would need to be inserted manually in an effect, or by libraries like react-helmet, and required careful handling when server rendering a React application.

In React 19, we're adding support for rendering document metadata tags in components natively:

```
function BlogPost({post}) {
  return (
    <article>
      <h1>{post.title}</h1>
      <title>{post.title}</title>
      <meta name="author" content="Josh" />
      <link rel="author" href="https://twitter.com/joshcstory/" />
      <meta name="keywords" content={post.keywords} />
      <p>
        Eee equals em-see-squared...
      </p>
    </article>
  );
}
```

When React renders this component, it will see the `<title>` `<link>` and `<meta>` tags, and automatically hoist them to the `<head>` section of document. By supporting these metadata tags natively, we're able to ensure they work with client-only apps, streaming SSR, and Server Components.

Note

You may still want a Metadata library

For simple use cases, rendering Document Metadata as tags may be suitable, but libraries can offer more powerful features like overriding generic metadata with specific metadata based on the current route. These features make it easier for frameworks and libraries like `react-helmet` to support metadata tags, rather than replace them.

For more info, see the docs for `<title>`, `<link>`, and `<meta>`.

Support for stylesheets

Stylesheets, both externally linked (`<link rel="stylesheet" href="...">`) and inline (`<style>...</style>`), require careful positioning in the DOM due to style precedence rules. Building a stylesheet capability that allows for composability within components is hard, so users often end up either loading all of their styles far from the components that may depend on them, or they use a style library which encapsulates this complexity.

In React 19, we're addressing this complexity and providing even deeper integration into Concurrent Rendering on the Client and Streaming Rendering on the Server with built in support for stylesheets. If you tell React the `precedence` of your stylesheet it will manage the insertion order of the

stylesheet in the DOM and ensure that the stylesheet (if external) is loaded before revealing content that depends on those style rules.

```
function ComponentOne() {
  return (
    <Suspense fallback="loading...">
      <link rel="stylesheet" href="foo" precedence="default" />
      <link rel="stylesheet" href="bar" precedence="high" />
      <article class="foo-class bar-class">
        {...}
      </article>
    </Suspense>
  )
}

function ComponentTwo() {
  return (
    <div>
      <p>{...}</p>
      <link rel="stylesheet" href="baz" precedence="default" />  <-- will be
    </div>
  )
}
```

During Server Side Rendering React will include the stylesheet in the `<head>`, which ensures that the browser will not paint until it has loaded. If the stylesheet is discovered late after we've already started streaming, React will ensure that the stylesheet is inserted into the `<head>` on the client before revealing the content of a Suspense boundary that depends on that stylesheet.

During Client Side Rendering React will wait for newly rendered stylesheets to load before committing the render. If you render this component from multiple places within your application React will only include the stylesheet once in the document:

```
function App() {
  return <>
    <ComponentOne />
    ...
    <ComponentOne /> // won't lead to a duplicate stylesheet link in the DOM
  </>
}
```

For users accustomed to loading stylesheets manually this is an opportunity to locate those stylesheets alongside the components that depend on them allowing for better local reasoning and an easier time ensuring you only load the stylesheets that you actually depend on.

Style libraries and style integrations with bundlers can also adopt this new capability so even if you don't directly render your own stylesheets, you can still benefit as your tools are upgraded to use this feature.

For more details, read the docs for `<link>` and `<style>`.

Support for async scripts

In HTML normal scripts (`<script src="...">`) and deferred scripts (`<script defer="" src="...">`) load in document order which makes rendering these kinds of scripts deep within your component tree challenging. Async scripts (`<script async="" src="...">`) however will load in arbitrary order.

In React 19 we've included better support for async scripts by allowing you to render them anywhere in your component tree, inside the components that actually depend on the script, without having to manage relocating and deduplicating script instances.

```
function MyComponent() {
  return (
    <div>
```

```
      <script async={true} src="..." />
      Hello World
    </div>
  )
}

function App() {
  <html>
    <body>
      <MyComponent>
      ...
      <MyComponent> // won't lead to duplicate script in the DOM
    </body>
  </html>
}
```

In all rendering environments, async scripts will be deduplicated so that React will only load and execute the script once even if it is rendered by multiple different components.

In Server Side Rendering, async scripts will be included in the `<head>` and prioritized behind more critical resources that block paint such as stylesheets, fonts, and image preloads.

For more details, read the docs for `<script>`.

Support for preloading resources

During initial document load and on client side updates, telling the Browser about resources that it will likely need to load as early as possible can have a dramatic effect on page performance.

React 19 includes a number of new APIs for loading and preloading Browser resources to make it as easy as possible to build great experiences that aren't held back by inefficient resource loading.

```
import { prefetchDNS, preconnect, preload, preinit } from 'react-dom'
function MyComponent() {
  preinit('https://.../path/to/some/script.js', {as: 'script' }) // loads and
  preload('https://.../path/to/font.woff', { as: 'font' }) // preloads this f
  preload('https://.../path/to/stylesheet.css', { as: 'style' }) // preloads
  prefetchDNS('https://...') // when you may not actually request anything fr
  preconnect('https://...') // when you will request something but aren't sur
}
```

```
<!-- the above would result in the following DOM/HTML -->
<html>
  <head>
    <!-- links/scripts are prioritized by their utility to early loading, not
    <link rel="prefetch-dns" href="https://...">
    <link rel="preconnect" href="https://...">
    <link rel="preload" as="font" href="https://.../path/to/font.woff">
    <link rel="preload" as="style" href="https://.../path/to/stylesheet.css">
    <script async="" src="https://.../path/to/some/script.js"></script>
  </head>
  <body>
    ...
  </body>
</html>
```

These APIs can be used to optimize initial page loads by moving discovery of additional resources like fonts out of stylesheet loading. They can also make client updates faster by prefetching a list of resources used by an anticipated navigation and then eagerly preloading those resources on click or even on hover.

For more details see Resource Preloading APIs.

Compatibility with third-party scripts and extensions

We've improved hydration to account for third-party scripts and browser extensions.

When hydrating, if an element that renders on the client doesn't match the element found in the HTML from the server, React will force a client re-render to fix up the content. Previously, if an element was inserted by third-party scripts or browser extensions, it would trigger a mismatch error and client render.

In React 19, unexpected tags in the `<head>` and `<body>` will be skipped over, avoiding the mismatch errors. If React needs to re-render the entire document due to an unrelated hydration mismatch, it will leave in place stylesheets inserted by third-party scripts and browser extensions.

Better error reporting

We improved error handling in React 19 to remove duplication and provide options for handling caught and uncaught errors. For example, when there's an error in render caught by an Error Boundary, previously React would throw the error twice (once for the original error, then again after failing to automatically recover), and then call `console.error` with info about where the error occurred.

This resulted in three errors for every caught error:

```
Console

Uncaught Error: hit
  at Throws
  at renderWithHooks
  …

Uncaught Error: hit     <-- Duplicate
  at Throws
  at renderWithHooks
  …

The above error occurred in the Throws component:
  at Throws
  at ErrorBoundary
  at App
```

```
  React will try to recreate this component tree from scratch using the
  error boundary you provided, ErrorBoundary.
```

In React 19, we log a single error with all the error information included:

Console

```
Error: hit
  at Throws
  at renderWithHooks
  …

The above error occurred in the Throws component:
  at Throws
  at ErrorBoundary
  at App

React will try to recreate this component tree from scratch using the
error boundary you provided, ErrorBoundary.
  at ErrorBoundary
  at App
```

Additionally, we've added two new root options to complement `onRecoverableError`:

- `onCaughtError`: called when React catches an error in an Error Boundary.
- `onUncaughtError`: called when an error is thrown and not caught by an Error Boundary.
- `onRecoverableError`: called when an error is thrown and automatically recovered.

For more info and examples, see the docs for `createRoot` and `hydrateRoot`.

Support for Custom Elements

React 19 adds full support for custom elements and passes all tests on Custom Elements Everywhere.

In past versions, using Custom Elements in React has been difficult because React treated unrecognized props as attributes rather than properties. In React 19, we've added support for properties that works on the client and during SSR with the following strategy:

- **Server Side Rendering**: props passed to a custom element will render as attributes if their type is a primitive value like `string`, `number`, or the value is `true`. Props with non-primitive types like `object`, `symbol`, `function`, or value `false` will be omitted.
- **Client Side Rendering**: props that match a property on the Custom Element instance will be assigned as properties, otherwise they will be assigned as attributes.

Thanks to Joey Arhar for driving the design and implementation of Custom Element support in React.

How to upgrade

See the React 19 Upgrade Guide for step-by-step instructions and a full list of breaking and notable changes.

PREVIOUS
Blog

NEXT
React 19 Beta Upgrade Guide

Learn React

Quick Start

Installation

Describing the UI

Adding Interactivity

Managing State

Escape Hatches

API Reference

React APIs

React DOM APIs

Community

Code of Conduct

Meet the Team

Docs Contributors

Acknowledgements

More

Blog

React Native

Privacy

Terms