ReactJS Evolution

React 15 (2016)

Refinement of the DOM rendering model

- Improved DOM nesting validation: More accurate enforcement of HTML rules.
- Better SVG support
- Bug fixes in controlled vs uncontrolled components
- **React.createClass deprecated in 15.5 (moved to create-react-class)**
- Preparation for future improvements in layout/DOM handling

React 16 (2017) – "React Fiber"

Massive internal rewrite (Fiber architecture)

- **New Reconciliation Engine ("Fiber")** – allows interruption, prioritization, and async rendering.
- **Error Boundaries** – use `componentDidCatch()` to handle errors in the component tree.
- **Fragments** – `<>...</>` syntax to return multiple elements without a wrapping div.
- **Render Return Types Expanded** – can now return arrays, strings, numbers, etc.
- **Portals** – render children into a DOM node outside the parent hierarchy.
- **Complete rewrite of server-side rendering (SSR)** – smaller and faster.

React 17 (2020) – "No New Features"

Foundation for gradual upgrades. No new developer-facing features, but:

- New event system (delegated to root rather than document)
- Support for gradual upgrades – multiple React versions can coexist
- Better tooling integration and backward compatibility
- Focused on making it easier for libraries and apps to upgrade over time.

React 18 (2022) – "Concurrency Begins"

Lays the foundation for Concurrent React

- Automatic Batching – React batches state updates even in async events.
- Concurrent Rendering enabled via `<StrictMode>` and `createRoot()`
- `startTransition()` API – mark updates as non-urgent (transitions).
- `useDeferredValue()` – defer part of the UI for smoother rendering.
- `useId()` – generate unique, deterministic IDs for SSR hydration.
- Streaming Server-Side Rendering (SSR) with support for Suspense
- Suspense for Data Fetching (experimental) – helps prepare for React Server Components.

React 19 (2024) - "Full-Stack React"

Origin

- XHP (2008 - 2010)
- PHP/Hack + XHP (2014) ➔ ReactJS + JSX (2013)
- BigPipe (2007 - 2010) ➔ RSC + Suspense (2020 - 2023)

Features

Latest: View Transitions and Activity

- **View Transitions** API – Enables smooth animations between UI states, making transitions more fluid.
- **Activity** Component – A new way to manage state and interactions, simplifying complex UI behaviors. When an Activity is hidden it is unmounted, but will save its state and continue to render at a lower priority than anything visible on screen.

Two different concepts for startTransition()

Originally startTransition() was introduced to control state update priority, allowing developers to mark state updates as low priority, meaning they won’t block urgent interactions. Previously, all state updates were treated as urgent, but now developers can defer UI changes to enhance performance.

View Transitions API (CSS-based): This is a browser-level feature that enables smooth animations when transitioning between different views. The team decide to reuse the same function name i.e. startTransition()

#	Topic	Role in learning
1	React Server Components	Foundation: server-first rendering
2	Server Functions	How server logic works
3	Asynchronous Transitions	Handling async UI patterns early
4	Actions	Core of mutations
5	action and formAction Props	Hooking up forms
6	use API	Handling async results when needed
7	useActionState()	Managing Action state
8	useFormStatus()	Feedback for forms
9	useOptimistic()	Make UIs feel faster
10	`ref` as a Prop	Advanced interop
	React 19 introduces the ability to pass `ref` as a prop to function components, simplifying component composition and ref forwarding.	
11	Stylesheet Support	DX improvements
	React 19 supports defining document metadata tags, such as `<title>` and `<meta>`, directly within components. React will automatically hoist these tags into the `<head>` section of the document.	
12	Document Metadata	SEO/accessibility
	React 19 enhances support for rendering stylesheets directly within components, ensuring styles are applied correctly and improving the developer experience.	

1. React Server Components

Understanding the Rendering Strategies

- Static website rendering
- Multi-Page Applications (MPA)
- Single-Page Applications (SPA)
- Server-Side Rendering (SSR)
- Static Site Generation (SSG)
- React Server Components (RSC)

The problem with SSR

When users see the website as fully interactive, but it actually is not, the time they spend waiting for the app to become fully interactive could be even longer than in the SPA version of the app.

SSR Workflow

- User requests a site
- Server creates ready HTML files
- Browser renders HTML but it's not interactive
- Browser downloads JavaScript
- Browser executes JavaScript
- Website is fully interactive

Strategies to Improve

- **Partial Hydration:** Only hydrate some components; others stay static
- **Progressive Hydration:** Hydrate components in stages (e.g., top-down, or as-needed)
- **Selective Hydration:** Prioritize hydration based on user interaction (React-specific optimization introduced in React 18 with Concurrent Mode)

Streaming

- Available from React 18 and big part of RSC performance
- Split a request into chunks
- Send each chunk as a stream to the client
- Render each chunk as it is streamed in

Out-of-Order Streaming

- **Streaming** refers to the ability of the server to send the RSC payload to the client incrementally as it's generated, instead of waiting for the entire page or component tree to be ready.
- **Out-of-order streaming** is a **specific capability within streaming** where different parts of the UI (wrapped in Suspense boundaries) are **rendered and sent to the client as they become ready, rather than in a fixed sequential order**.

The idea behind RSC

React for Two Computers by Dan Abramov

An app in which the data is initially hardcoded on the client.

Then a defined server endpoint to which the client will make a request.

In a lot of cases, this is fine, but we want to have control over the UI, where we want to be able to say: this particular interaction has to be synchronous, like we want the data to be already there with that screen, and we don't want to wait for it to come later.

So the question is: How do we achieve this?

So there is not much that we could do from the client-side perspective because this is the workflow, i.e., wait for the data to be fetched and visualize it after that. This is why we go to the server.

Previously, we have looked at the server code and client code as two separate programs that talk to each other. But now, the different perspective is: a single program that spans two devices, i.e., computers—server and client machines. And it is evaluated in steps.

This approach provides optimization opportunities, i.e., to simplify it. Similar to "lifting the state up" in React, we could do "lifting the data up", i.e., move the fetching of the data to the server, where we would just get it directly from, let's say, a database and not have to call another API endpoint to do it.

And now we don't have an additional network request from the client because the data is already there on the server, and it includes it in the HTML that will return to the client.

And for the server to have data ahead of time, it means that the server could process it and return only the relevant data, i.e., a small amount if needed, to the client, i.e., from the whole array of cat names that was sent before, it could send only one name.

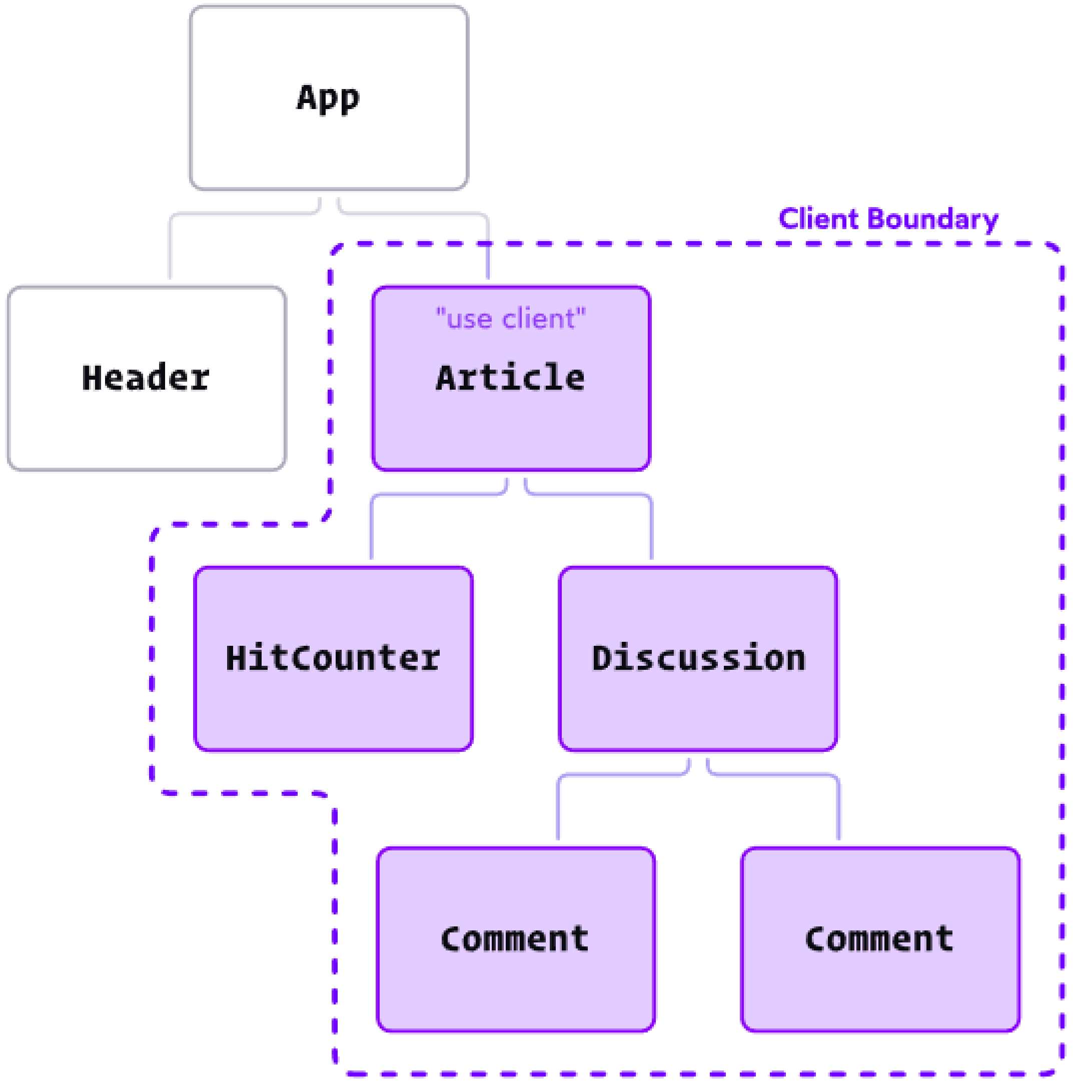
So we have code that executes first on the server, and it emits another code that will be sent to the client and executed later, and we could pass some data together with this client code.

React Server Components Behaviour

They run exclusively on the server and then streamed to the client. They never re-render just run once on the server to generate the UI. Their JavaScript is never send to the client. The server is part of the full-stack framework i.e. NextJS. The components can run during the build process of the app as an alternative to the server. The server components eliminate the need to write API.

When we want that we use client components and they are marked with **"use client"** directive.

In the context of RSC, the term **"boundary"** refers to the distinction between client components and server components—essentially defining where the client-server responsibilities are divided in a React app.


App
Header
Client Boundary
"use client"
Article
HitCounter
Discussion
Comment
Comment

Stronger distinction between server components and client components

Server Components: These still handle data-fetching but must return serializable data (not JSX that contains client-side interactivity).

Client Components: If a component needs interactivity (event handlers, hooks like useState, etc.), it must include "use client" and be imported into the server component.

This shift reinforces React’s architectural intent—keeping the server responsible for rendering static content while ensuring dynamic UI logic stays in client components.

React Server Components can still return JSX, but only serializable JSX—meaning it must be pure markup and cannot include client-side interactivity (like event handlers or hooks).

RSC Payload

React renders RSC into a special format called RSC Payload. It is streamable format that represents the DOM, like serialized version of the React tree. The RSC Payload contains the rendered server components with "holes" of references to client components and the props they passed to them. It's used both to generate the pre-rendered HTML on the server and to update the DOM on the client without a need for JavaScript.

When RSC needs to be re-rendered due to a data change it refreshes on the server and seamlessly merges into the existing DOM without a hard refresh, updating only the parts that have changed. As a result the client state is preserved even as parts of the view are updated from the server. This is possible because the RSC Payload only contains references to the client components and can leave them untouched.

Traditional CSR Workflow

The browser loads `index.html` from a static server.

React downloads JavaScript and initializes the app in the client.

The client makes API calls to fetch data.

React renders everything on the client, updating state dynamically.

RSC Workflow

Client Requests a Page

Instead of fetching a static HTML file, the client sends a request to a React-aware server (e.g., using Next.js, running on Node.js or an edge runtime).

This server understands how to process Server Components and deliver an RSC payload.

RSC Workflow

Server Processes Server Components

The server evaluates React Server Components:

- Executes data fetching, logic, and component rendering entirely on the server
- No Server Component logic or data fetching is sent to the client

The result is an RSC payload—a structured stream of data representing the server-rendered React tree.

RSC Workflow

Server Streams the RSC Payload to the Client

The RSC payload is streamed to the client:

- It may include some HTML (e.g. for layout, static text)
- It contains serialized metadata and component instructions for React to reconstruct the UI
- Suspense boundaries allow parts of the UI to progressively load

This approach improves perceived performance by rendering and displaying parts of the UI as soon as they’re ready.

RSC Workflow

Client Hydrates and Enhances the UI

On the client:

- React reconstructs the UI from the streamed RSC payload
- Client Components (marked with "use client") are hydrated — making them interactive
- These handle state, user interactions, animations, etc.

Server Components remain static on the client — they are never hydrated or re-executed in the browser.

For each client request, does the server run the Server Components and generate a dynamic bundle?

Every time the client makes a request, the server processes the React Server Components dynamically, rendering the necessary UI before sending it back to the client. Instead of delivering a pre-built JavaScript bundle, the server generates UI content on demand, based on the request.

Benefits of using RSC

- **Faster data fetching and access to backend resources**, before they RSC is send to the client i.e. ahead of time.
- **Security:** all sensitive logic and API keys are kept on the server witout risk to expose them
- **Caching:** by rendering on the server the result can be cached and reused on subsequent request
- **Bundle size and performance:** large dependencies could be kept on the server and to be excluded from the client side bundle
- **Streaming and Suspence integration:** server components can be streamed meaning they can be sent in chunks and viewed as they become ready
- **Developer experience:** a lot less hassle on the client with effects and keeping data in the state

Drawbacks of RSC

- Require a framework i.e. NextJS
- Increased complexity with the new concepts
- Learning curve
- Early adoption

Since the client components exists on the client and server components on the server, props need to be serializable. That means that we can't pass functions through the server boundary. Instead we must create API endpoints or better - server actions to communicate between them when we can't use props.

If we use the same data in multiple components in a tree, we don't have to fetch it on the root and pass it down. We can fetch it in each component because it is automatically memoized.

Server Actions are the preferred way to mutate data

- Define a function/file with "use server"
- NextJS creates "hidden" API endpoint
- Use the function in any component
- Provides a type-safe RPC-experience

Summary: RSC in app router allow you to do whatever you need to do whether is MPA or SPA and include SSR or SEO all in the same stack optimized.

2. Server Functions

React Docs

Until September 2024, we referred to all Server Functions as "Server Actions". If a Server Function is passed to an action prop or called from inside an action then it is a Server Action, but not all Server Functions are Server Actions.

NextJS Docs

A Server Function is an asynchronous function that is executed on the server. Server Functions are inherently asynchronous because they are invoked by the client using a network request.

When invoked as part of an <u>action</u>, they are also called Server Actions.

By convention, an action is an asynchronous function passed to `startTransition()`. Server Functions are automatically wrapped with `startTransition()` when:

- Passed to a `<form>` using the `action` prop,
- Passed to a `<button>` using the `formAction` prop
- Passed to `useActionState()`

Creating Server Functions

A Server Function can be defined by using the "use server" directive. You can place the directive at the top of an asynchronous function to mark the function as a Server Function, or at the top of a separate file to mark all exports of that file.

There are two main ways you can invoke a Server Function:

- Forms in Server and Client Components
- Event Handlers in Client Components

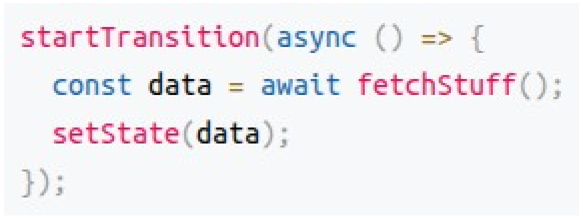
Good to know: When passed to the action prop, Server Functions are also known as Server Actions.

3. Asynchronous Transitions

React 19 support asynchronous transitions by allowing asynchronous functions to be passed to `startTransition()`.

This is the low-level mechanism.

UI waits for async logic to finish in transitions.



```
startTransition(async () => {
  const data = await fetchStuff();
  setState(data);
});
```

4. Actions

In React 19, when we talk about "actions", the term can refer to two related but distinct concepts:

- **"Client-side" Actions:** These are functions triggered by **user interactions** (e.g. button clicks, form submissions) that can start asynchronous UI transitions
- **Server Actions:** These are async functions you define in React Server Components, typically **used for data mutation**s on the server, like form submissions or database updates. They're often defined with the async function syntax and marked to run on the server.

"Client-side" Actions

Originally, `startTransition()` was introduced in React to manage state update priority—it allowed developers to mark certain state updates as low priority, preventing UI lag during expensive renders.

Later, with React 19, the concept of "Actions" was added, extending `startTransition()` to work seamlessly with async functions. This introduced automatic pending states, error handling, and optimistic updates, making it much easier to handle async operations without manual state management.

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

  const handleSubmit = () => {
    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>
  );
}
```

Server Actions

A Server Function is an asynchronous function that is executed on the server. Server Functions are inherently asynchronous because they are invoked by the client using a network request.

When invoked as part of an action, they are also called Server Actions.

By convention, an action is an asynchronous function passed to startTransition. Server Functions are automatically wrapped with startTransition when:

- Passed to a `<form>` using the `action` prop,
- Passed to a `<button>` using the `formAction` prop
- Passed to `useActionState()`

Creating Server Functions

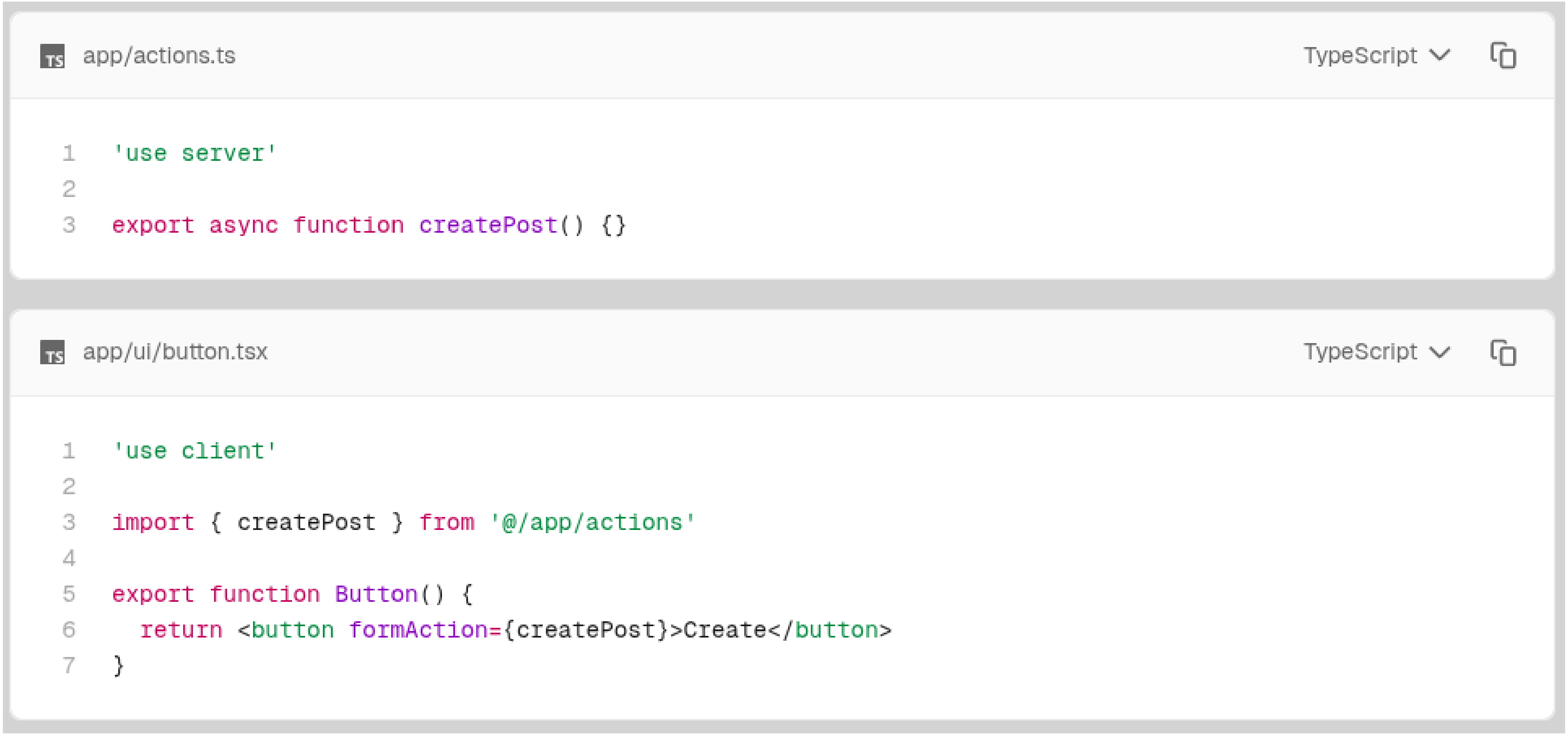
A Server Function can be defined by using the use server directive. You can place the directive at the top of an asynchronous function to mark the function as a Server Function, or at the top of a separate file to mark all exports of that file.

Invoking Server Functions

There are two main ways you can invoke a Server Function:

- Forms in Server and Client Components
- Event Handlers in Client Components

When passed to the action prop, Server Functions are also known as Server Actions.



```ts
'use server'

export async function createPost() {}
```

```tsx
'use client'

import { createPost } from '@/app/actions'

export function Button() {
  return <button formAction={createPost}>Create</button>
}
```

5. **action** and **formAction** props

React 19 introduces **action** and **formAction** props for `<form>`, `<input>`, and `<button>` elements. These props allow actions to be directly associated with form elements, streamlining form handling.

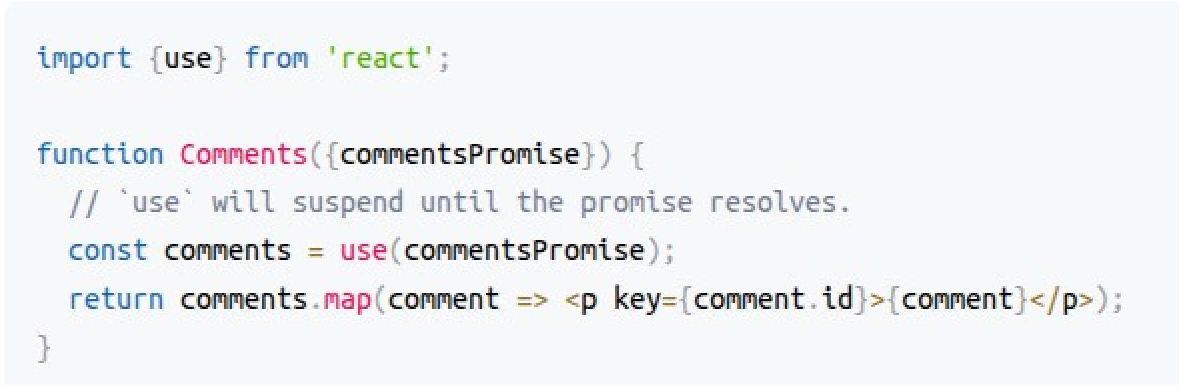
The **formAction** attribute on an `<input>` (or `<button>`) is used to override the parent `<form>`'s action when that specific button is used to submit the form.

Previously, `formAction` (or just action on the `<form>`) had to be a URL string pointing to a route that would handle the submission. With React 19, you can pass a server-defined function directly to the action prop, and React will serialize the form submission and invoke the server function — no custom API route needed.

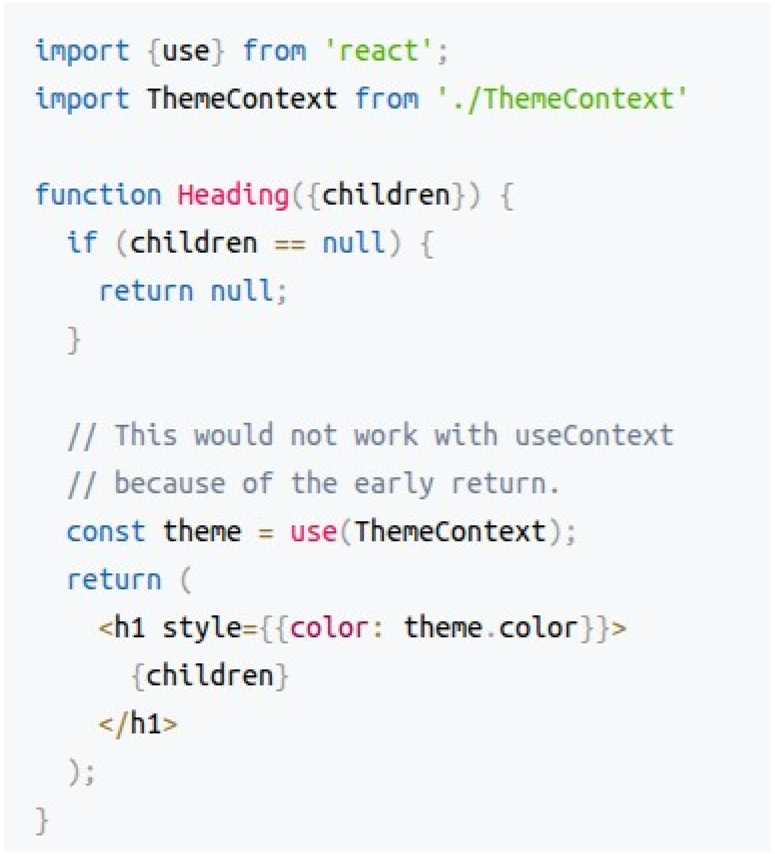
```
function saveDraft(data) {
  console.log("Saving draft:", Object.fromEntries(data));
}

function publishPost(data) {
  console.log("Publishing post:", Object.fromEntries(data));
}

function PostForm() {
  return (
    <form>
      <input name="title" placeholder="Title" />
      <input name="body" placeholder="Body" />

      <button type="submit" formAction={saveDraft}>
        Save Draft
      </button>

      <button type="submit" formAction={publishPost}>
        Publish
      </button>
    </form>
  );
}
```

6. use() API

- New API to read resources in render: `use()`
- You can read a promise with use, and React will Suspend until the promise resolves
- You can also read context with use, allowing you to read Context conditionally such as after early returns



```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>);
}
```

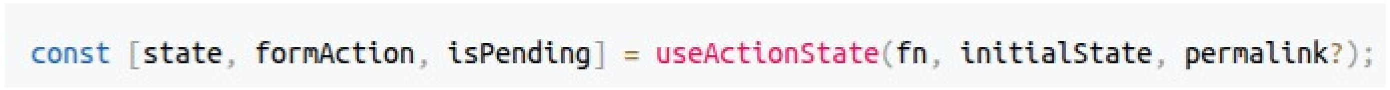


```jsx
import {use} from 'react';
import ThemeContext from './ThemeContext'

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>
  );
}
```

7. New hook: useActionState()

Building on top of Actions, React 19 introduces a new hook `useActionState()` to handle common cases for Actions and allows you to update state based on the result of a form action.

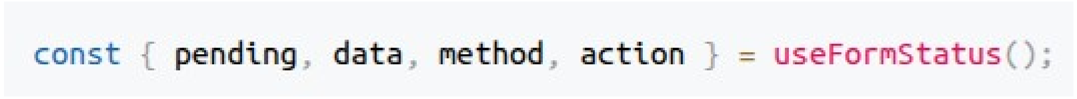
This hook provides a way to manage the state of actions, including tracking pending states and handling results. It simplifies the process of managing asynchronous operations within components.



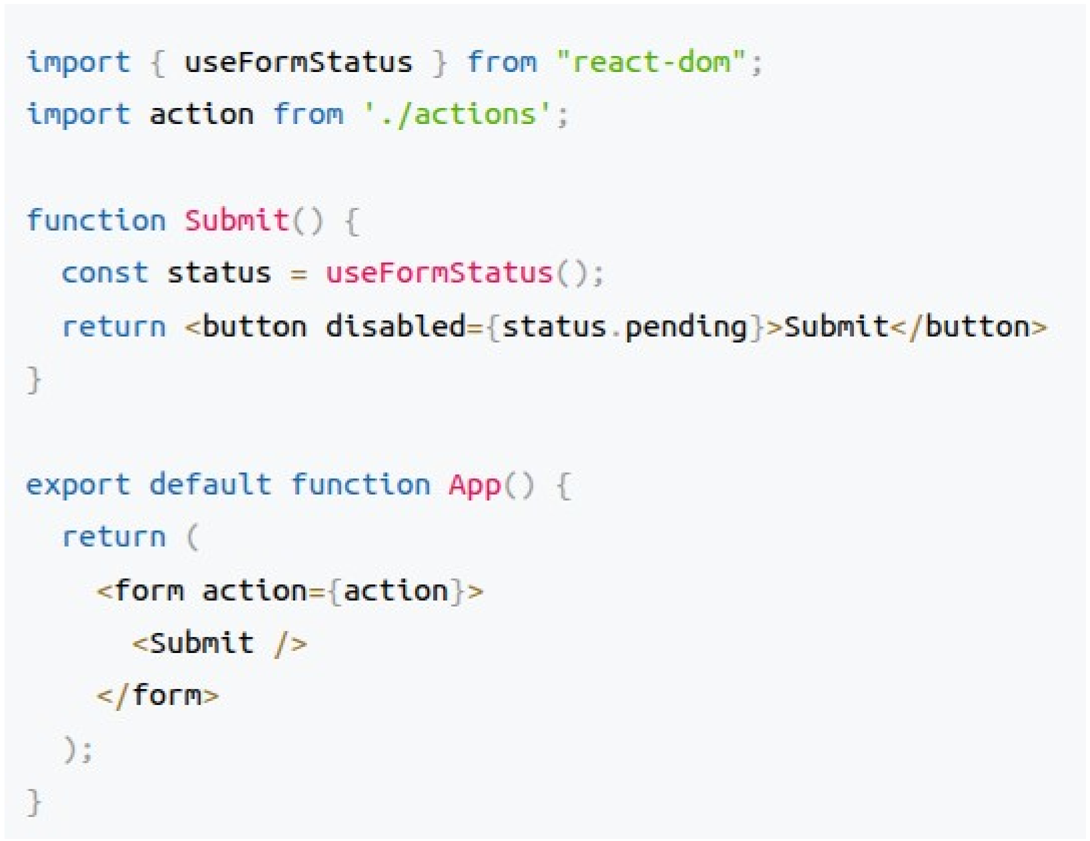
```
const [state, formAction, isPending] = useActionState(fn, initialState, permalink?);
```

8. New hook: useFormStatus()

useFormStatus() is a hook that gives you status information of the last form submission.



```
const { pending, data, method, action } = useFormStatus();
```



```jsx
import { useFormStatus } from "react-dom";
import action from './actions';

function Submit() {
  const status = useFormStatus();
  return <button disabled={status.pending}>Submit</button>
}

export default function App() {
  return (
    <form action={action}>
      <Submit />
    </form>
  );
}
```

9. New hook: useOptimistic()

Common UI pattern when performing a data mutation is to show the final state optimistically while the async request is underway.

`useOptimistic()` is a React Hook that lets you show a different state while an async action is underway. It accepts some state as an argument and returns a copy of that state that can be different during the duration of an async action such as a network request. You provide a function that takes the current state and the input to the action, and returns the optimistic state to be used while the action is pending.

This state is called the "optimistic" state because it is usually used to immediately present the user with the result of performing an action, even though the action actually takes time to complete.

1. Initial State
   const [messages, setMessages] = useState([]);
   const [optimisticMessages, addOptimisticMessage] = useOptimistic(messages, optimisticUpdateFn);

2. User Submits a Message
   → User types "Hello!" and submits a form.

3. Immediately Show Optimistic Message
   → addOptimisticMessage({ text: "Hello!", id: "temp-123" });

   UI shows:
   Hello! (pending...)

4. Send to Server via Action
   → form action={sendMessage}

5. Server Responds
   → React refreshes the UI with real server state
   → messages = [{ text: "Hello!", id: "real-456" }]

6. Optimistic State Reconciles
   → Optimistic message is replaced with confirmed one

UI shows:
Hello!

10. ref as a Prop

React 19 introduces the ability to pass `ref` as a prop to function components, simplifying component composition and ref forwarding.

11. Stylesheet Support

React 19 supports defining document metadata tags, such as `<title>` and `<meta>`, directly within components. React will automatically hoist these tags into the `<head>` section of the document.

12. Document Metadata

React 19 enhances support for rendering stylesheets directly within components, ensuring styles are applied correctly and improving the developer experience.