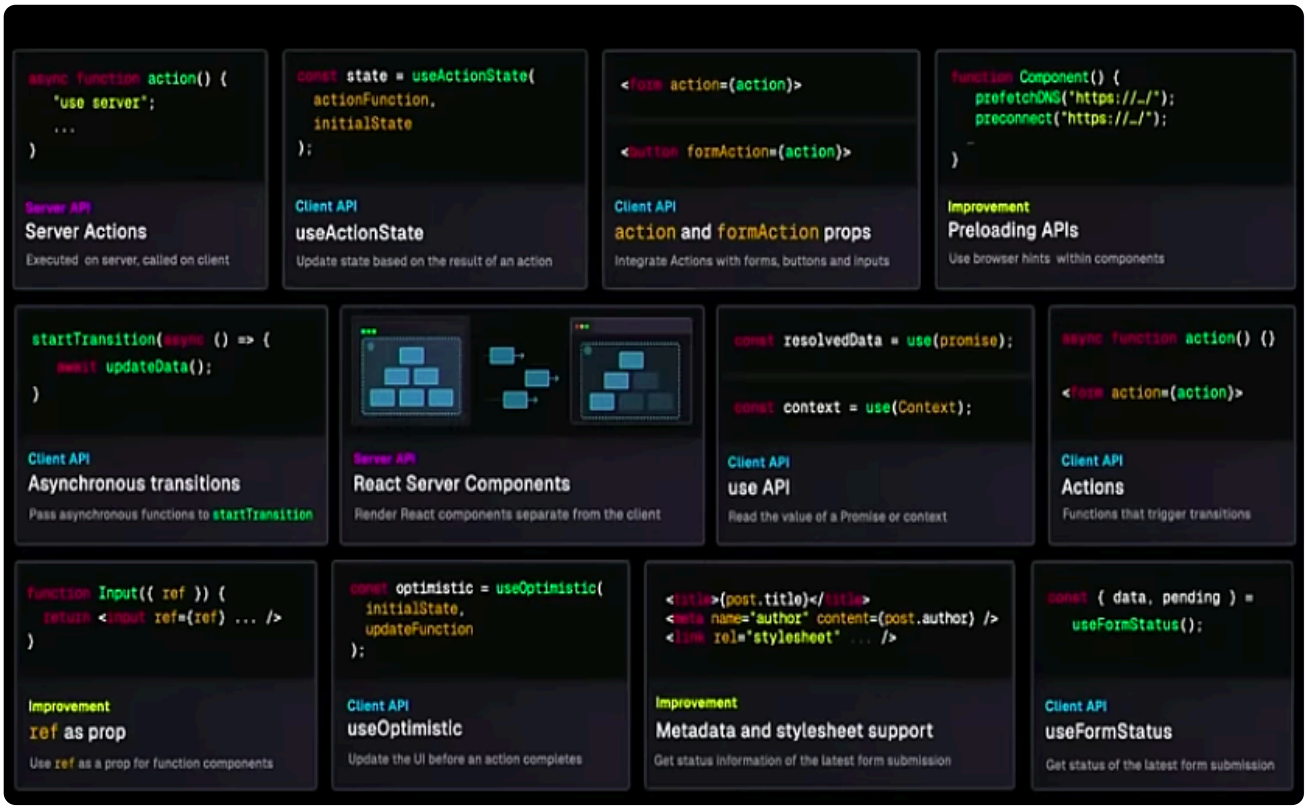
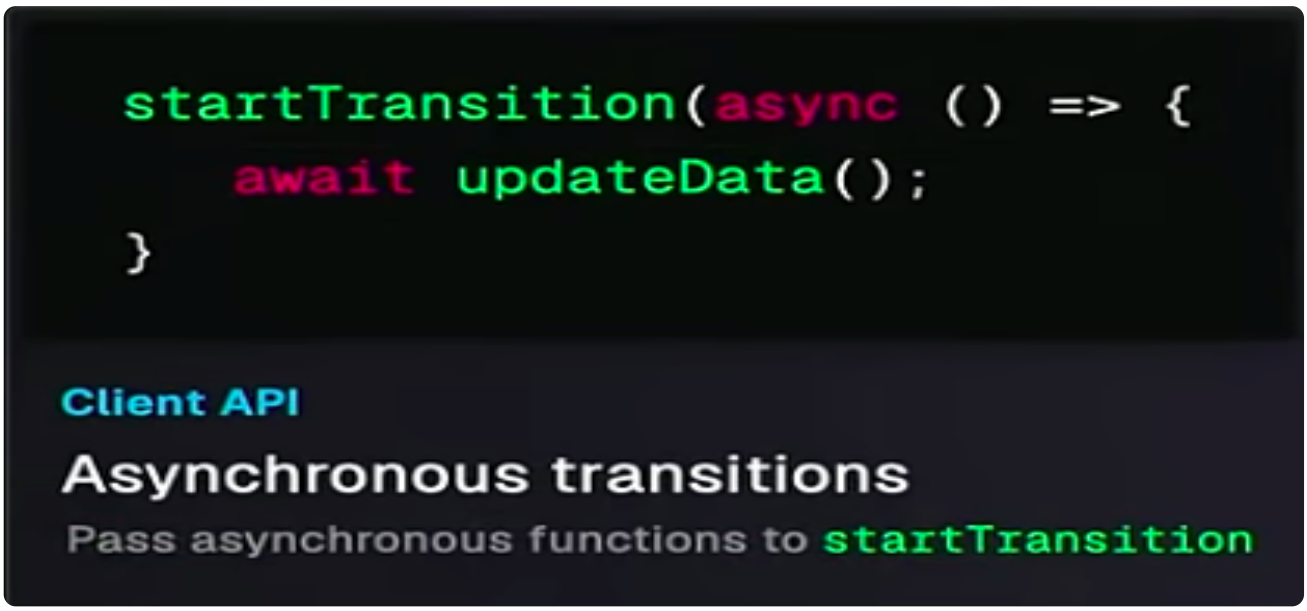
📑 What is the new in React 19



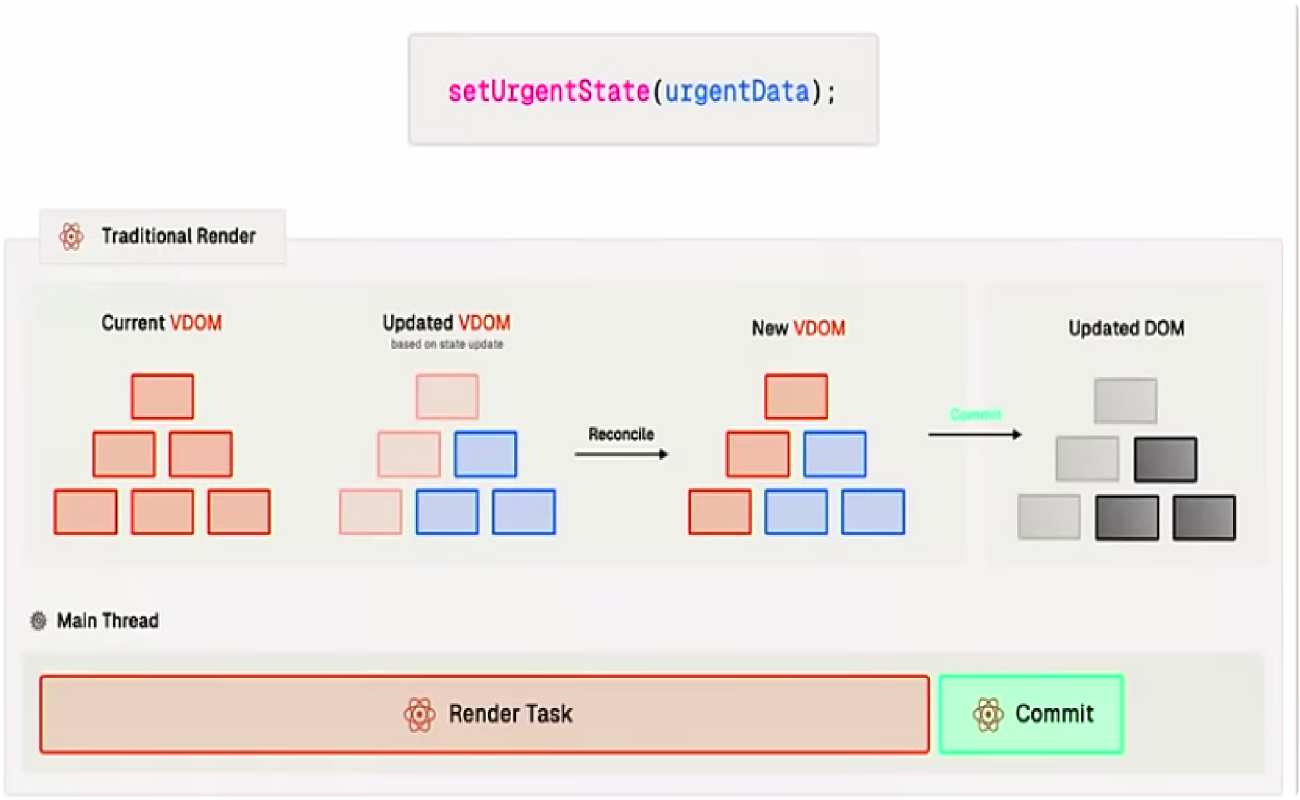
✅ Asynchronous transitions



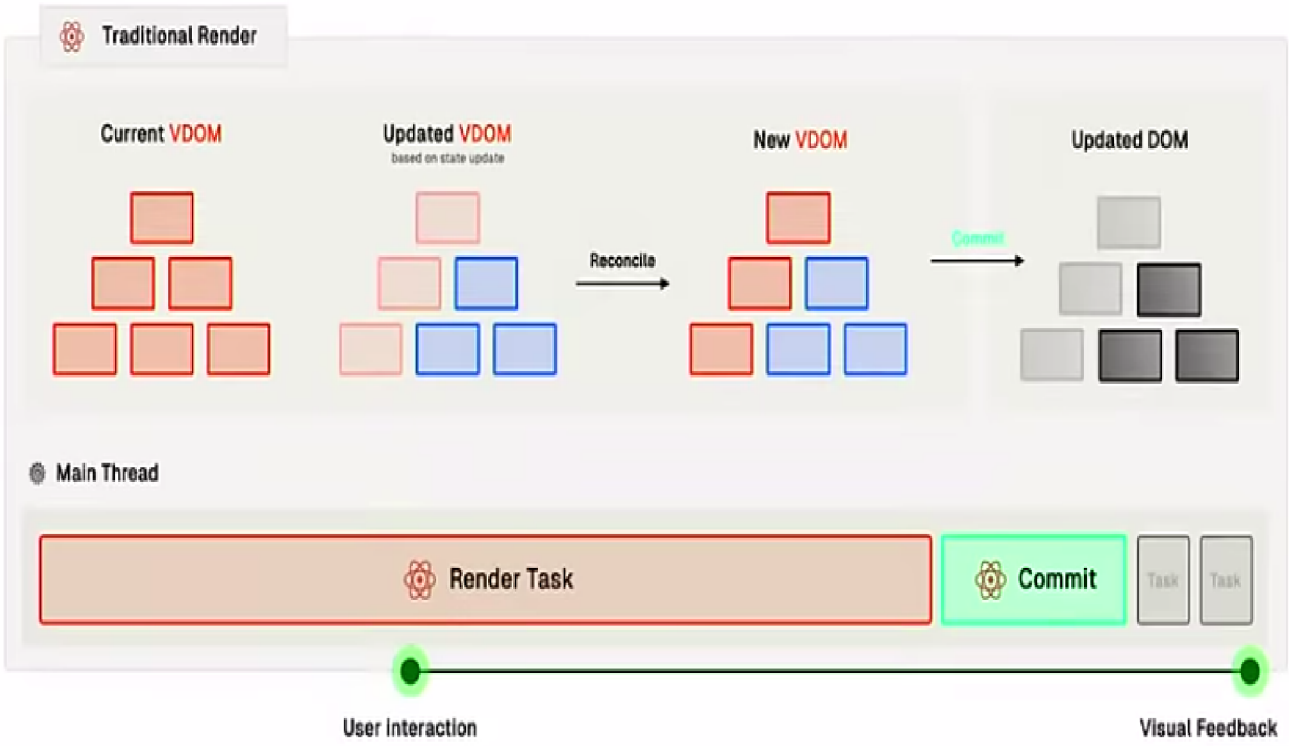
Transitions were introduced in React 18 as a way to mark given state updates as a non-urgent.

```
const [isPending, startTransition] = useTransition();

function handleData() {
  setUrgentState(urgentData);

  startTransition(() => {
    setNonUrgentState(nonUrgentData);
  });
}
```

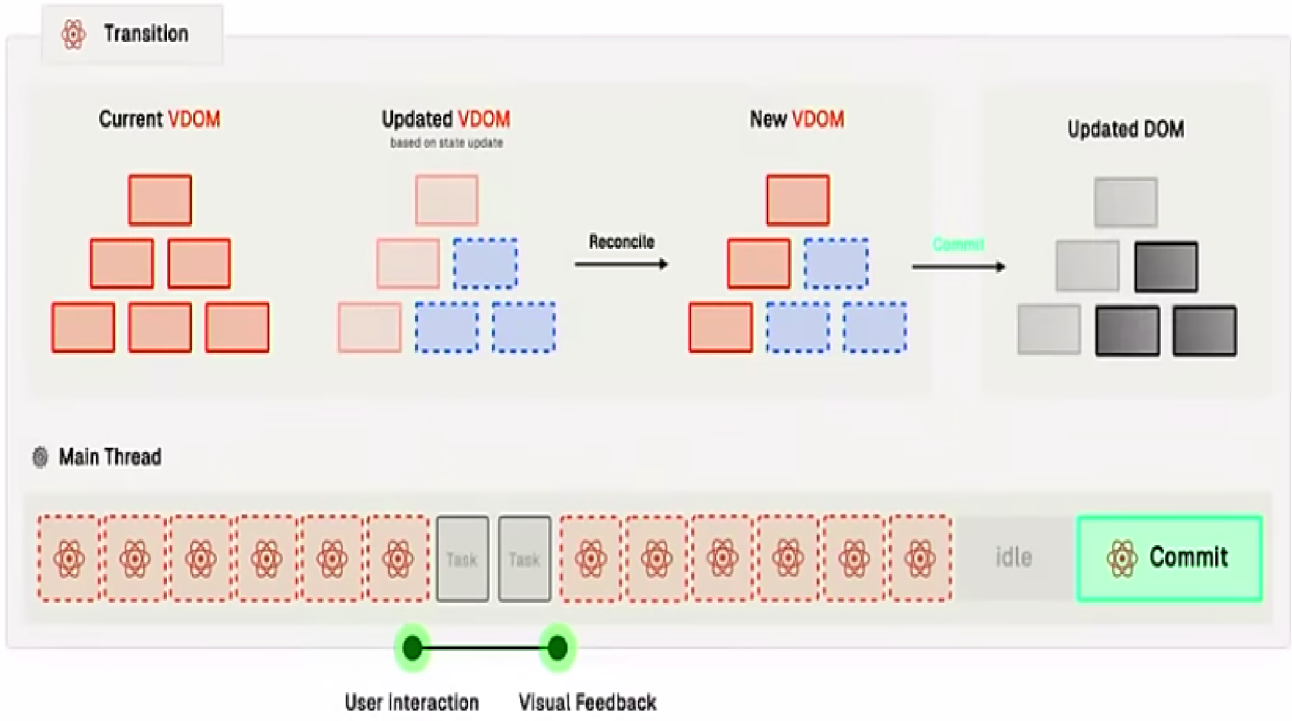
By default React renders all the necessary components in a single uniterruptable task.



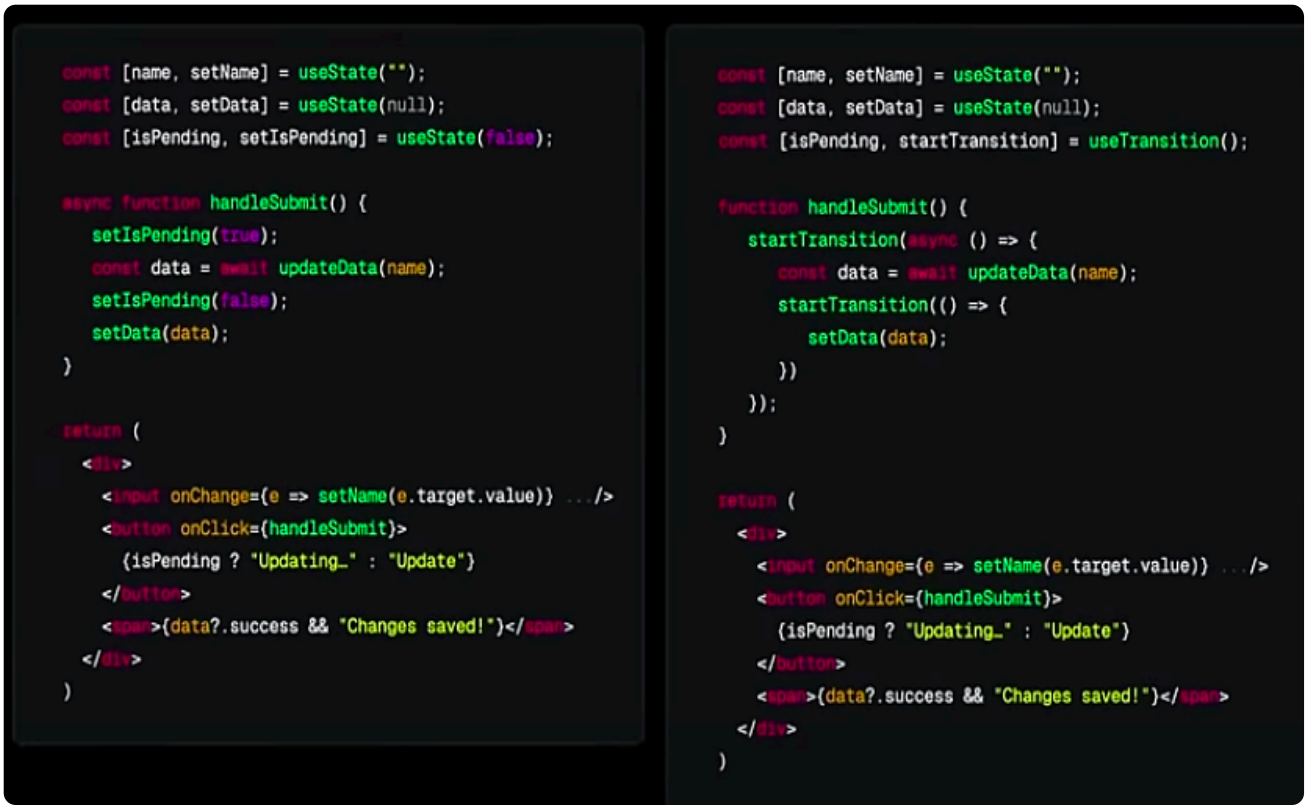
Depending on the complexity of the update, it could take quite a while before React actually renders these updates and commits them to the DOM. Since this is all happening on the main thread, our application is unable to handle any other tasks during this time, which leaves the UI unresponsive. If the user interacts with the UI while React is rendering an update, they will experience a significant visual feedback delay.



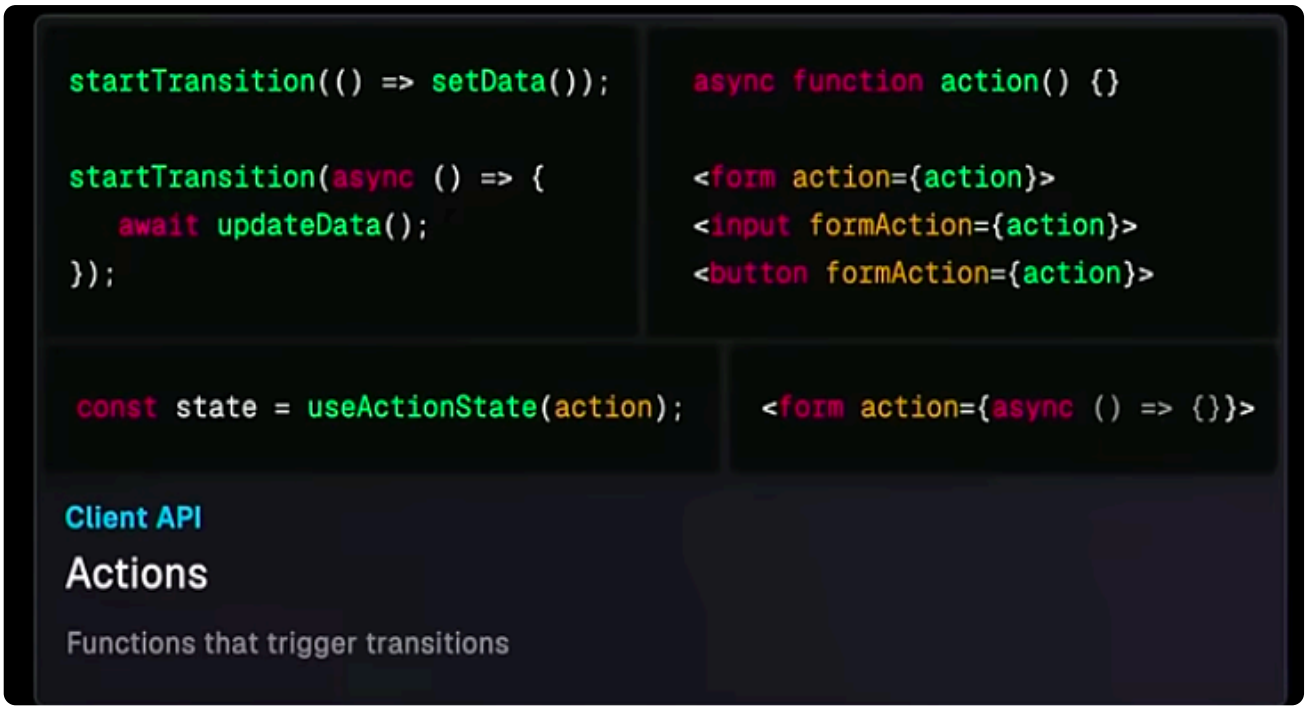
The transitions provides a solution to this. Instead of rendering the update as a single, uninterruptible task, React will yield back to the main thread every 5ms to see if there are other tasks waiting to be handled instead.



We could make the following improvement



In React 19 functions that trigger transitions are called actions



```
const [state, action, isPending] = useActionState(
  actionFunction,
  initialState
);
```

Client API

useActionState

Update state based on the result of an action

Using startTransition

```
const [name, setName] = useState("");
const [data, setData] = useState(null);
const [isPending, startTransition] = useTransition();

function handleSubmit() {
  startTransition(async () => {
    const data = await updateData(name);
    startTransition(() => {
      setData(data);
    })
  });
}

return (
  <div>
    <input onChange={e => setName(e.target.value)} ... />
    <button onClick={handleSubmit}>
      {isPending ? "Updating..." : "Update"}
    </button>
    <span>{data?.success && "Changes saved!"}</span>
  </div>
)
```

Using useActionState

```
const [data, action, isPending] = useActionState(
  async (currentState, formData) => {
    const name = await formData.get("name");
    const data = await updateData(name);
    return data;
  },
  null
);

return (
  <form action={action}>
    <input name="name" .../>
    <button type="submit">
      {isPending ? "Updating..." : "Update"}
    </button>
    <span>{data?.success && "Changes saved!"}</span>
  </form>
)
```

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

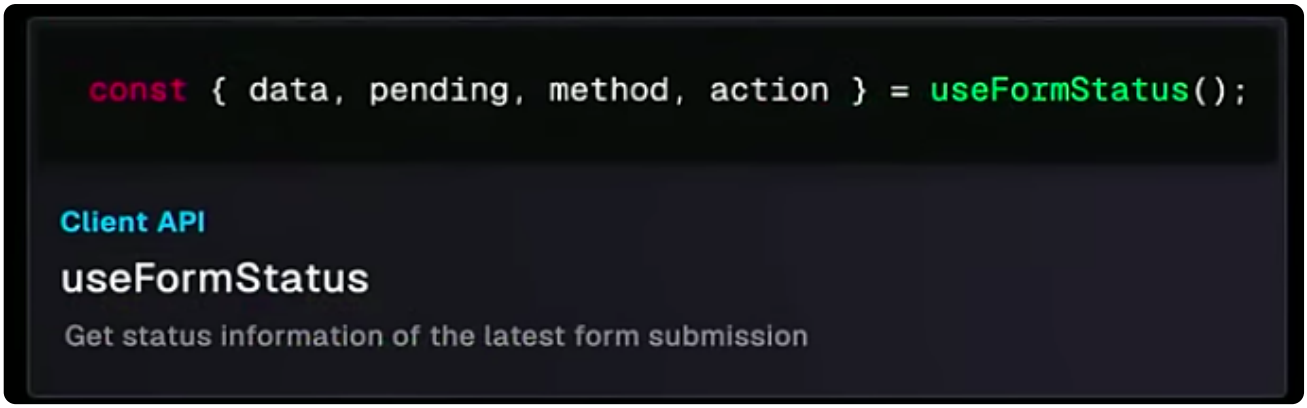
```
<button formAction={action}>
```

Client API

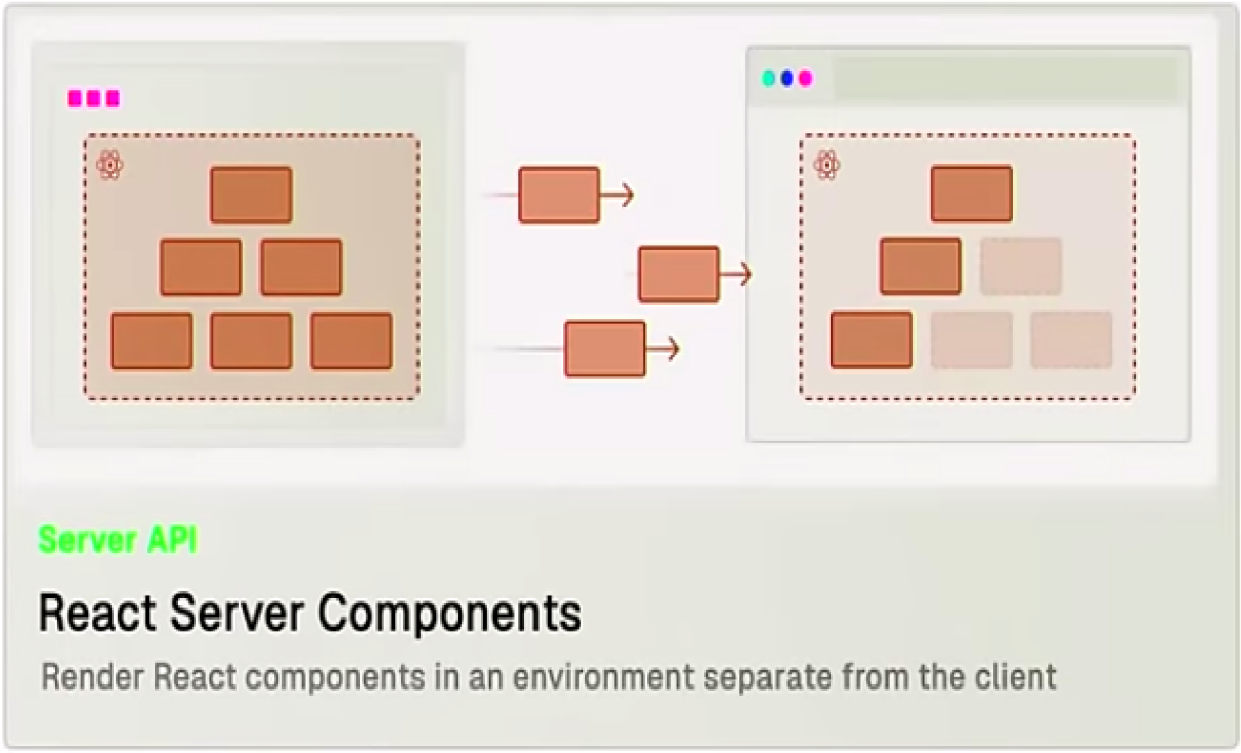
action and formAction props

Integrate Actions with forms, buttons and inputs

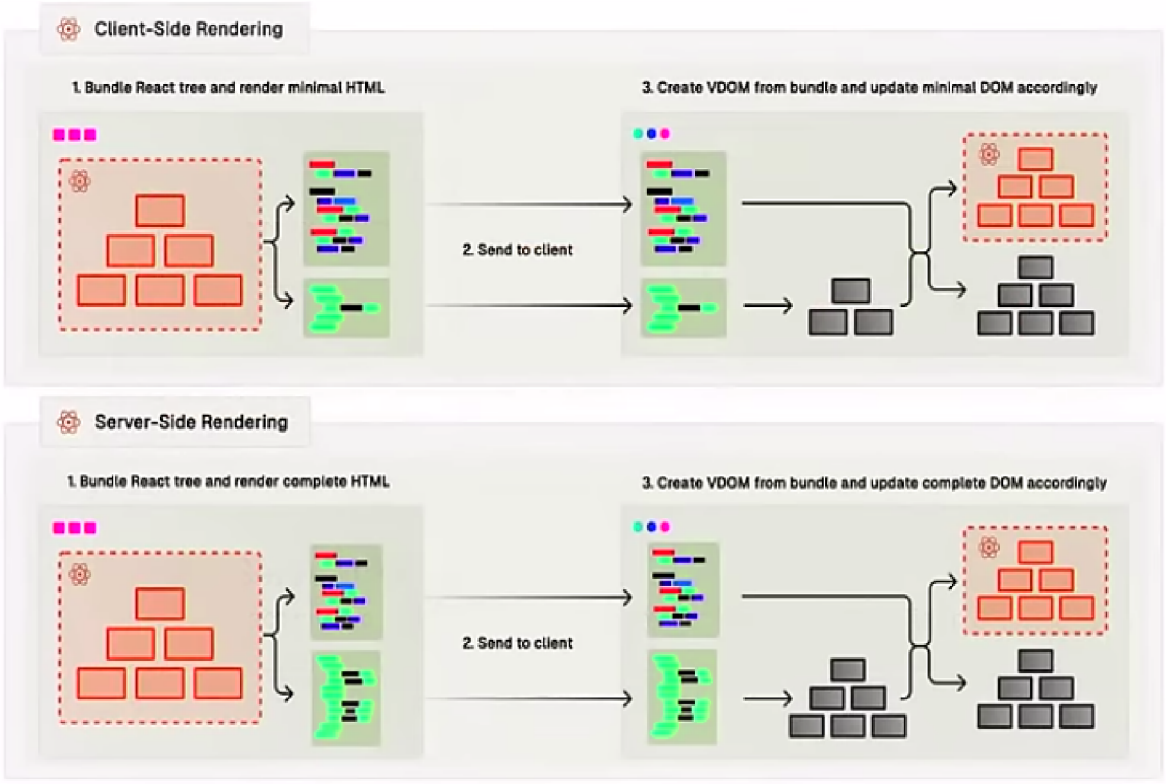

async function action(formData) {
const email = formData.get("email");
await updateEmail(email);
}
<form action={action}>
const [state, action] = useActionState(
(currentData, formData) => {
const email = formData.get("email");
await updateEmail(email);
}
);
<form action={action}>
<form action={async formData => {
const email = formData.get("email");
await updateEmail(email);
}}>
<form action={formData => {
const email = formData.get("email");
}}>


const { data, pending, method, action } = useFormStatus();
Client API
useFormStatus
Get status information of the latest form submission

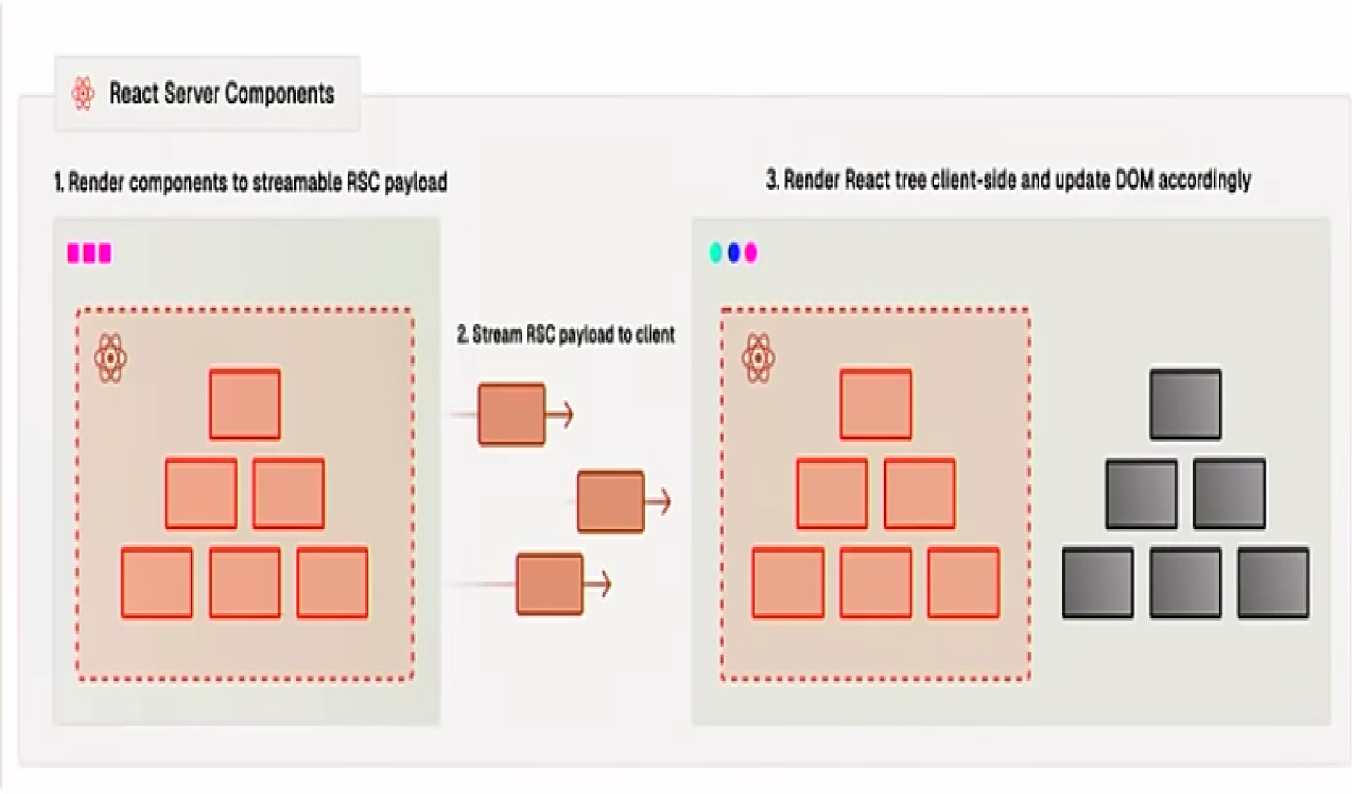

function Form() {
const [data, action] = useActionState(...);
return (
<form action={action}>
<input name="name" .../>
<StyledButton />
{data.success && "Changes saved!"}
</form>
)
}
function StyledButton() {
const { pending } = useFormStatus();
return (
<button style={…}>
{pending ? "Updating…" : "Update"}
</button>
)
}



Traditionally we had two ways to render applications: Client-Side Rendering and Server-Side Rendering



Both approaches rely on the fact that the React renderer needs to rebuild the React tree client-side, even though it already has it on the server. The introduction of the concurrent renderer in React 18 allowed for a different approach - React Server Components.



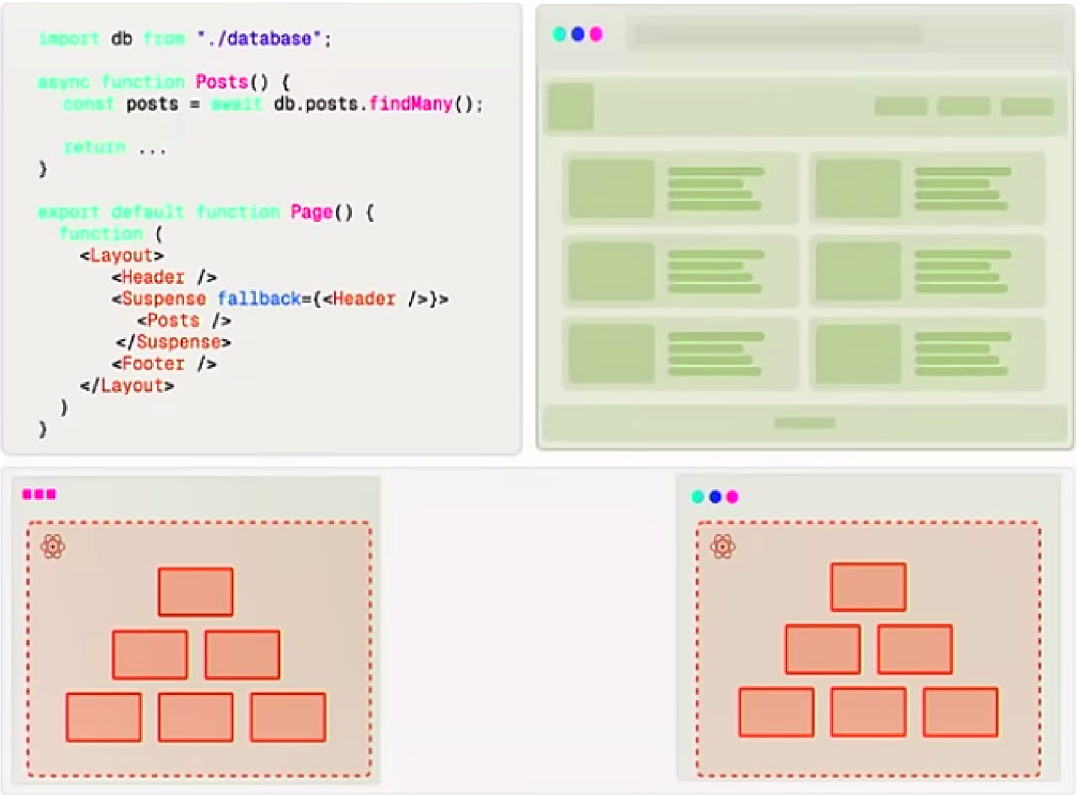
RSC allows React to send the actual serialized component tree to the client using a special JSON-like format. The client-side renderer

understands this format and can rebuild the tree on the client-side without the need for HTML or JavaScript.

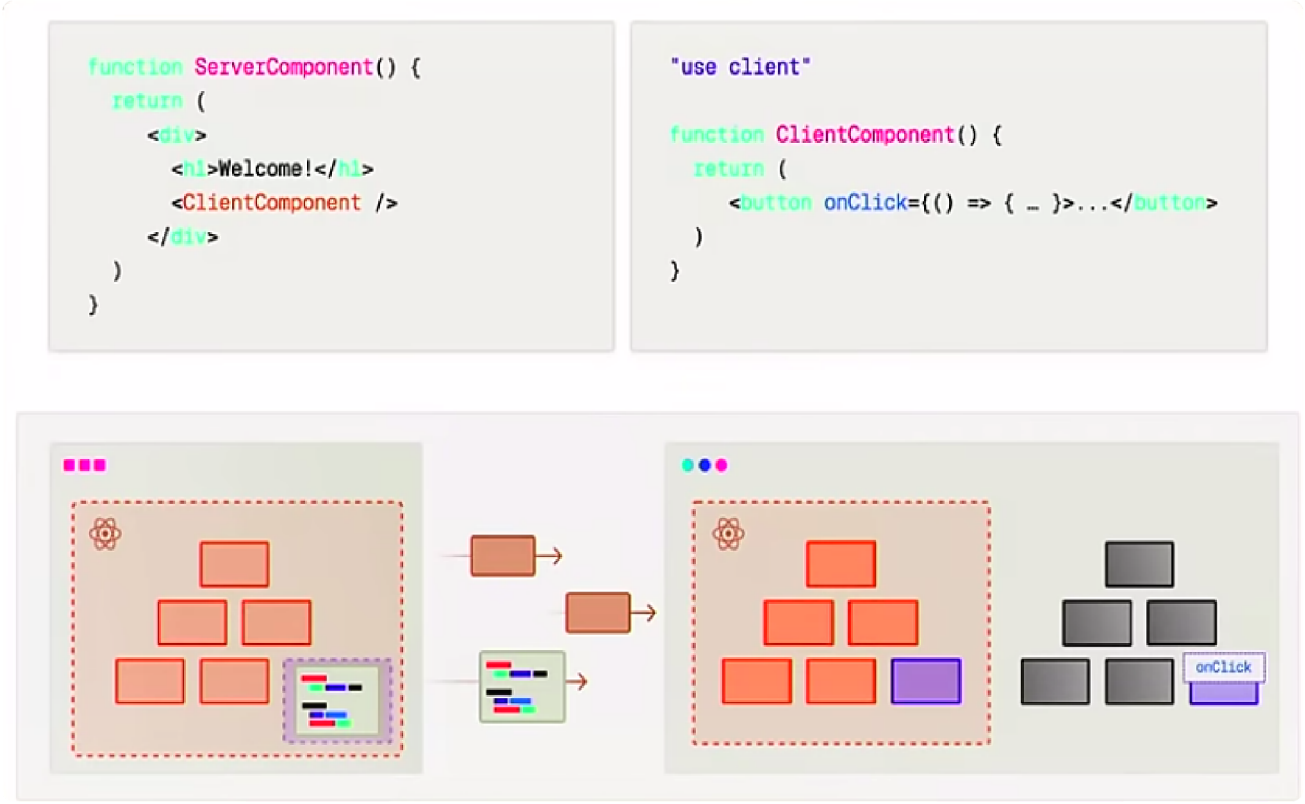
RSC is never sent to the client. It only returns the value in this JSON-like format. This means we can use server-side functionality directly within our component, such as accessing databases, file systems, etc.

This is especially easy because in React 19, we could use async/await to render server components.

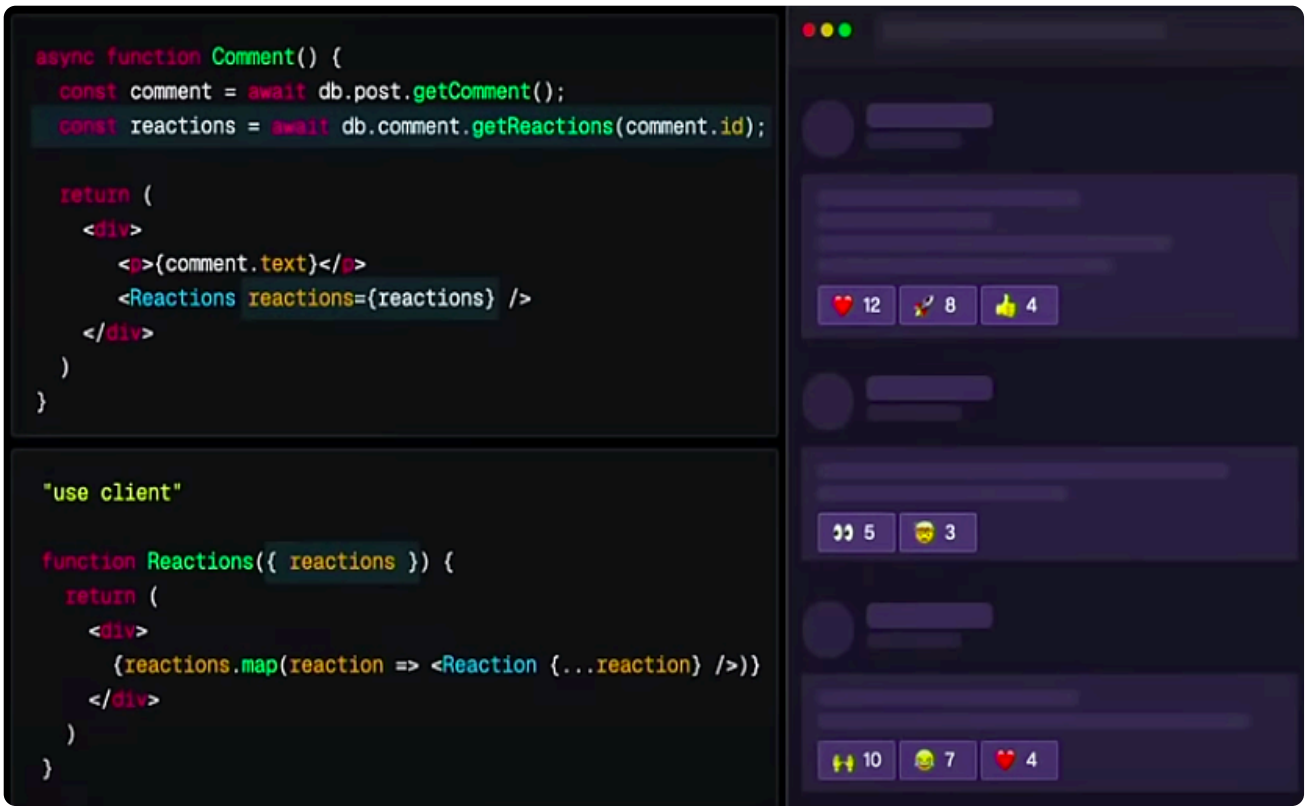
It can happen that this data lookup takes a while to complete. In that case, the entire React tree needs to wait before the data is completely fetched and sent to the client. To fix this, we can wrap the server component in a Suspense boundary. This allows React to stream the parts that were faster to the client and, in the meantime, render the fallback of Suspense. Only when the data has been fetched and the RSC payload has been generated, will this component be actually sent to the client.



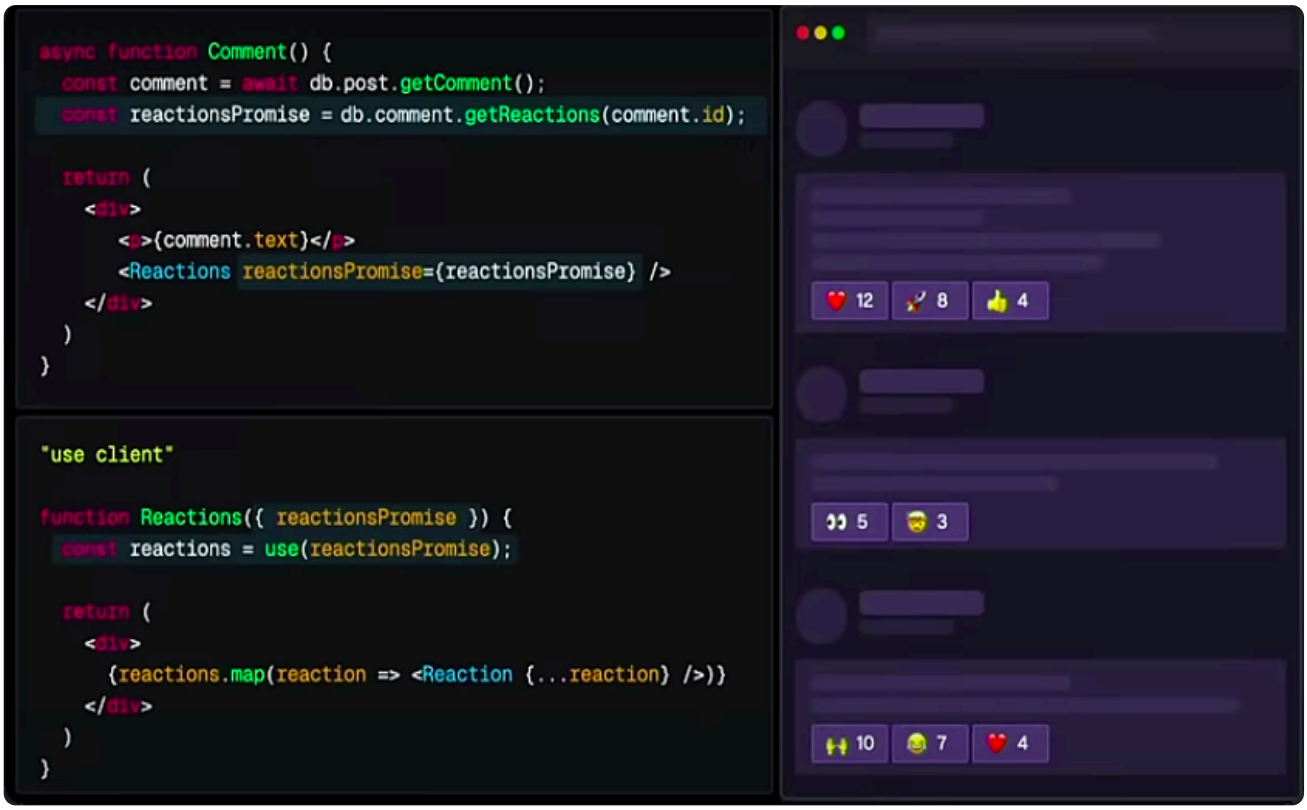
We can't use client-side functionality in server components i.e. hooks, event handlers. To use it on the client we have to specify "use client" directive and it will add the component to the javascript bundle.



Because we are actually shipping the component to the client and not just the return value (as server components do), we can't fetch data the same way that we could with a server component. First, we could fetch the data in the server component and then pass it as a prop to the client component.



Or we can use the "use" API within a client component and pass the promise as a prop instead of the actual data.



In server environment we can also create a server action. This is function that run server-side but can be called client-side.

Whenever server action is triggered, React will send the request to the server to execute this function. We can track the state of this request by getting more information about the action status.

```jsx
function ServerComponent() {
  async function serverAction(formData) {
    "use server"
    const email = formData.get("email");
    const data = await db.user.updateEmail(email);
    ...
  }

  return <ClientComponent serverAction={serverAction} />
}
```

```jsx
"use client"

function ClientComponent({ serverAction }) {
  const actionState = useActionState(serverAction);

  return <form action={serverAction}>...</form>
}
```

```jsx
"use client"

function ClientComponent({ serverAction }) {
  const [isPending, startTransition] = useTransition();

  function handleSubmission() {
    startTransition(async () => await serverAction());
  }

  return <form action={serverAction}>...</form>
}
```

```jsx
"use client"

function ClientComponent({ serverAction }) {
  return (
    <form action={serverAction}>
      <PendingText />
    </form>
  )
}

function PendingText() {
  const { pending } = useFormStatus();

  return isPending ? <span>Pending…</span> : null;
}
```

```jsx
function Input({ ref }) {
  return <input ref={ref} .../>
}
```

Improvements

ref as a prop

Access ref as a prop for function components

```jsx
function BlogPost({ post }) {
  return (
    ...
    <title>{post.title}</title>
    <meta name="author" content={post.author} />
    <meta property="og:image" content={post.image} />
    ...
  )
}
```

Improvements

Document metadata

Use document metadata tags within components

```jsx
function Component() {
  return (
    ...
    <link rel="stylesheet" href="/styles/styles.css" precedence="default" />
    <link rel="stylesheet" href="/styles/button.css" precedence="default" />
    <link rel="stylesheet" href="/styles/article.css" precedence="high" />
    ...
  )
}
```

Improvements

Stylesheet support

Render stylesheets directly within components

```jsx
function Component() {
  preinit("https://…/script.js", { as: "script" });
  preload("https://…/stylesheet.css", { as: "style" });
  prefetchDNS("https://…/");
  preconnect("https://…/");

  return ...
}
```

Client API

Preloading APIs

Load and preload browser resources within components