



What is the new in React 19

<pre>async function action() { "use server"; ... }</pre> <p>Server API Server Actions Executed on server, called on client</p>	<pre>const state = useActionState(actionFunction, initialState);</pre> <p>Client API useActionState Update state based on the result of an action</p>	<pre><form action={action}> <button formAction={action}></pre> <p>Client API action and formAction props Integrate Actions with forms, buttons and inputs</p>	<pre>function Component() { prefetchDNS('https://.../'); preconnect("https://.../"); }</pre> <p>Improvement Preloading APIs Use browser hints within components</p>
<pre>startTransition(async () => { await updateData(); })</pre> <p>Client API Asynchronous transitions Pass asynchronous functions to <code>startTransition</code></p>	 <p>Server API React Server Components Render React components separate from the client</p>	<pre>const resolvedData = use(promise); const context = use(Context);</pre> <p>Client API use API Read the value of a Promise or context</p>	<pre>async function action() {} <form action={action}></pre> <p>Client API Actions Functions that trigger transitions</p>
<pre>function Input({ ref }) { return <input ref={ref} ... /> }</pre> <p>Improvement ref as prop Use <code>ref</code> as a prop for function components</p>	<pre>const optimistic = useOptimistic(initialState, updateFunction);</pre> <p>Client API useOptimistic Update the UI before an action completes</p>	<pre><title>(post.title)</title> <meta name="author" content={post.author} /> <link rel="stylesheet" ... /></pre> <p>Improvement Metadata and stylesheet support Get status information of the latest form submission</p>	<pre>const { data, pending } = useFormStatus();</pre> <p>Client API useFormStatus Get status of the latest form submission</p>

✓ Asynchronous transitions

```
startTransition(async () => {
  await updateData();
})
```

Client API

Asynchronous transitions

Pass asynchronous functions to `startTransition`

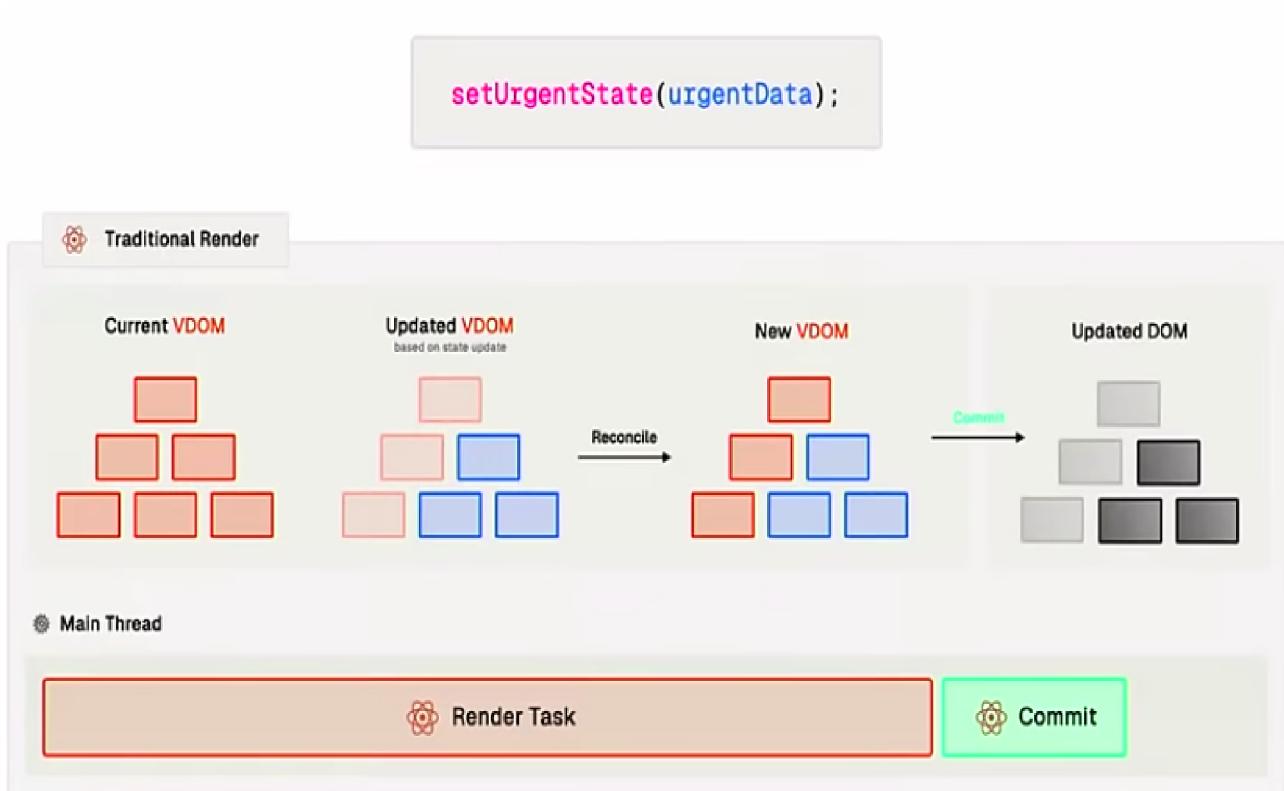
Transitions were introduced in React 18 as a way to mark given state updates as a non-urgent.

```

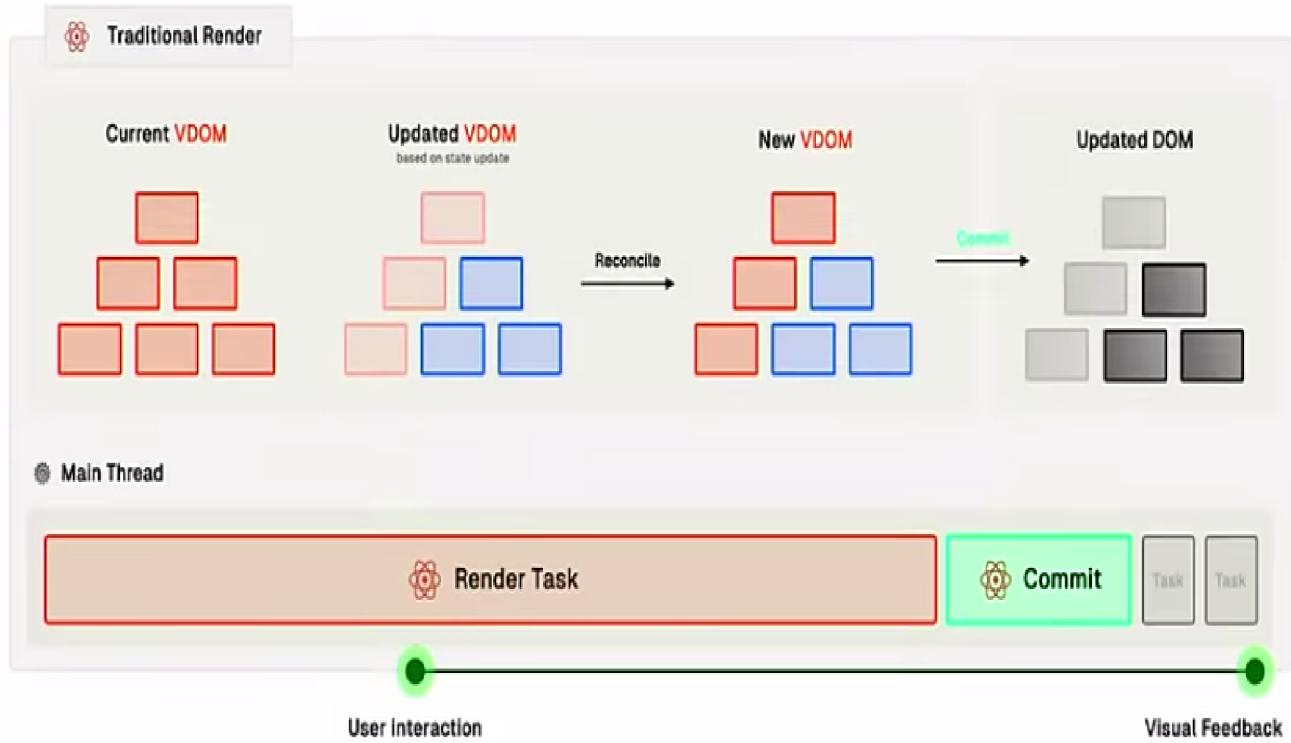
1 const [isPending, startTransition] = useTransition();
2
3 function handleData() {
4   setUrgentState(urgentData);
5
6   startTransition(() => {
7     setNonUrgentState(nonUrgentData);
8   });
9 }

```

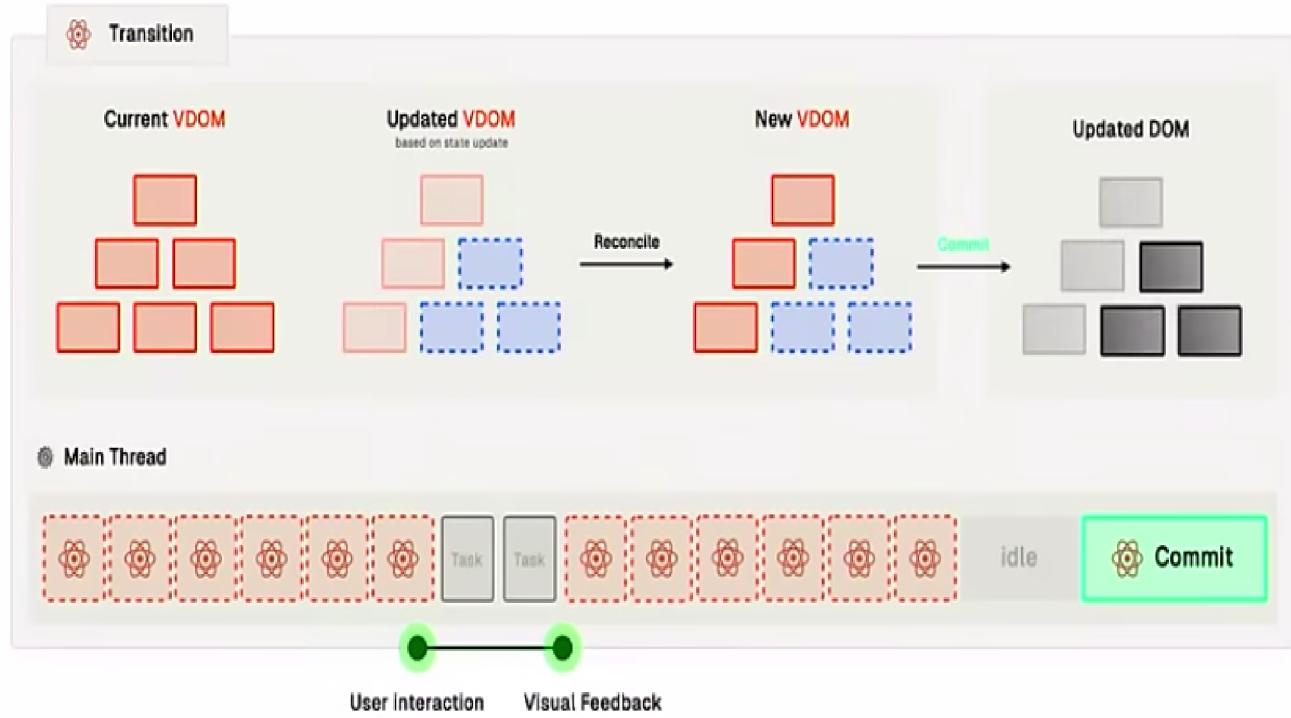
By default React renders all the necessary components in a single uninterrupted task.



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

```

const [name, setName] = useState("");
const [data, setData] = useState(null);
const [isPending, setIsPending] = useState(false);

async function handleSubmit() {
  setIsPending(true);
  const data = await updateData(name);
  setIsPending(false);
  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>
)

```

```

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

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>
)

```

In React 19 functions that trigger transitions are called actions

<code>startTransition(() => setData());</code>	<code>async function action() {}</code>
<code>startTransition(async () => { await updateData(); });</code>	<code><form action={action}> <input formAction={action}> <button formAction={action}></code>
<code>const state = useActionState(action);</code>	<code><form action={async () => {}}></code>

Client API

Actions

Functions that trigger transitions

```
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 name = 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}>

<form action={async formData => {
  const email = formData.get("email");
  await updateEmail(email);
}}>
```

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

<form action={action}>

<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 />
      <span>{data.success && "Changes saved!"}</span>
    </form>
  )
}

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

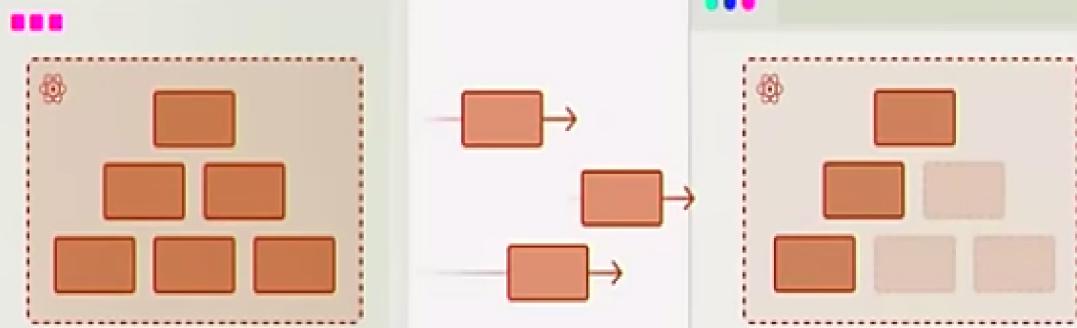
  return (
    <button style={...}>
      {pending ? "Updating..." : "Update"}
    </button>
  )
}
```

```
const [allValues, addOptimisticValue] = useOptimistic(  
  state,  
  updateFunction  
)
```

Client API

useOptimistic

Optimistically update the UI before an action completes

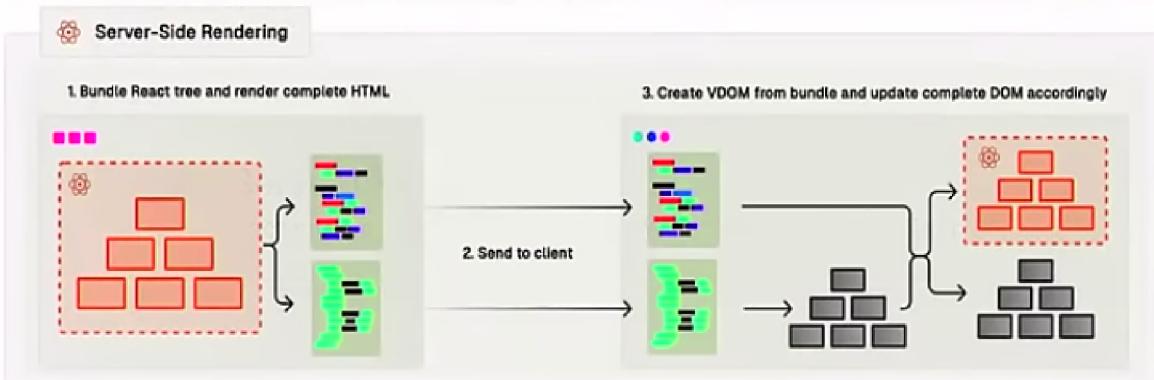
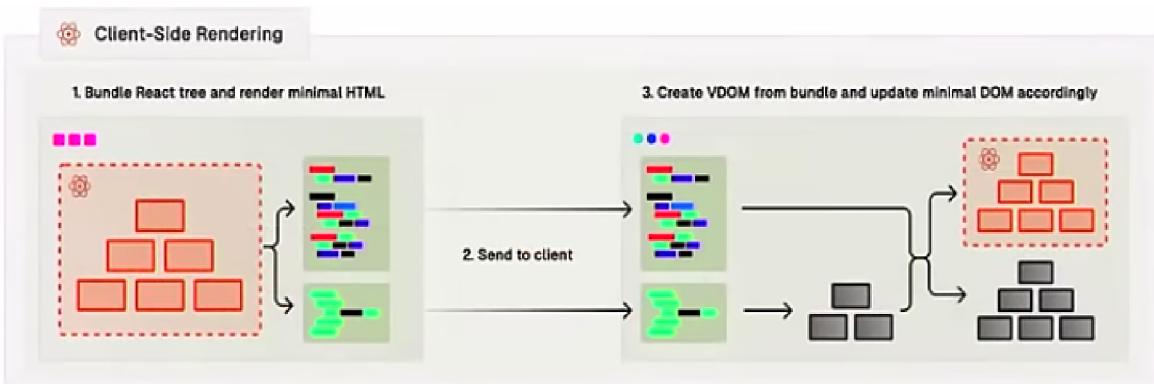


Server API

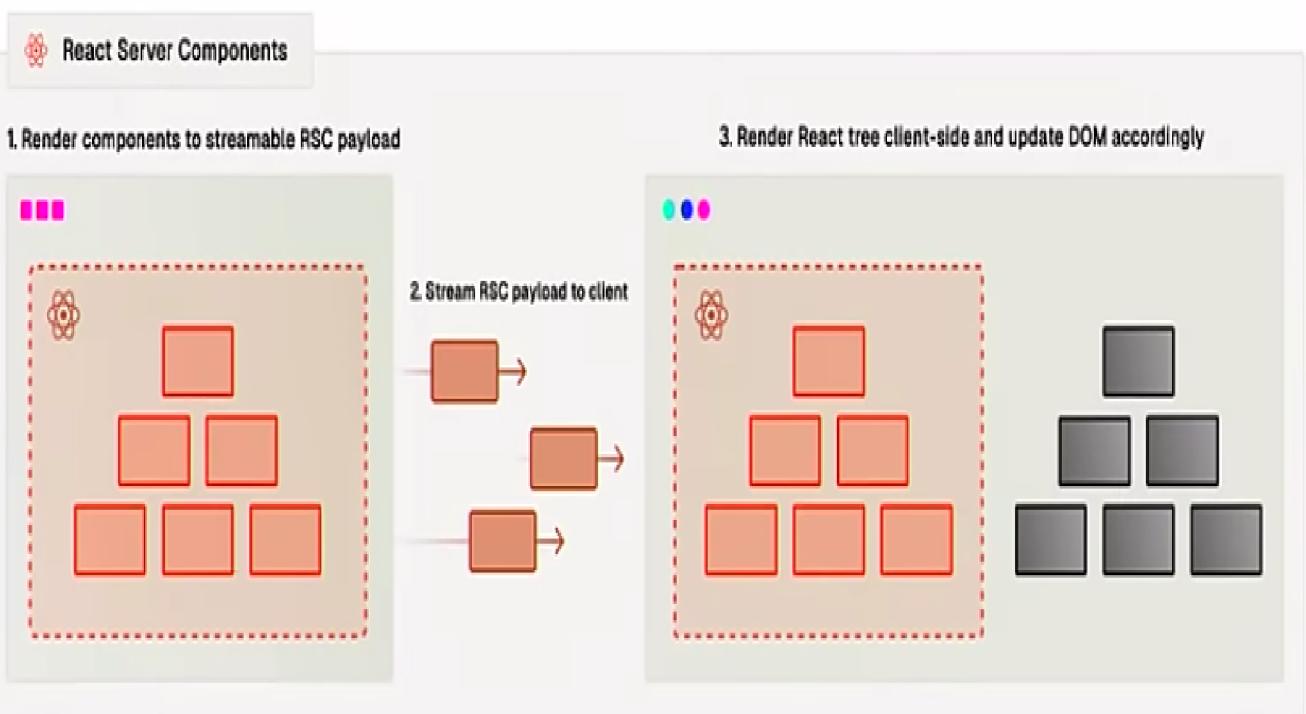
React Server Components

Render React components in an environment separate from the client

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.

```
async function Posts() {
  return (
    <ul>
      <li>Post 1</li>
      <li>Post 2</li>
    </ul>
  )
}
```

RSC Payload

```
...
[$$, "ul", null, { "children": [
  $$, "li", null, { "children": "Post 1" }],
  $$, "li", null, { "children": "Post 2" }]
}]]
```

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.

```
import db from "./database";

async function Posts() {
  const posts = await db.posts.findMany();

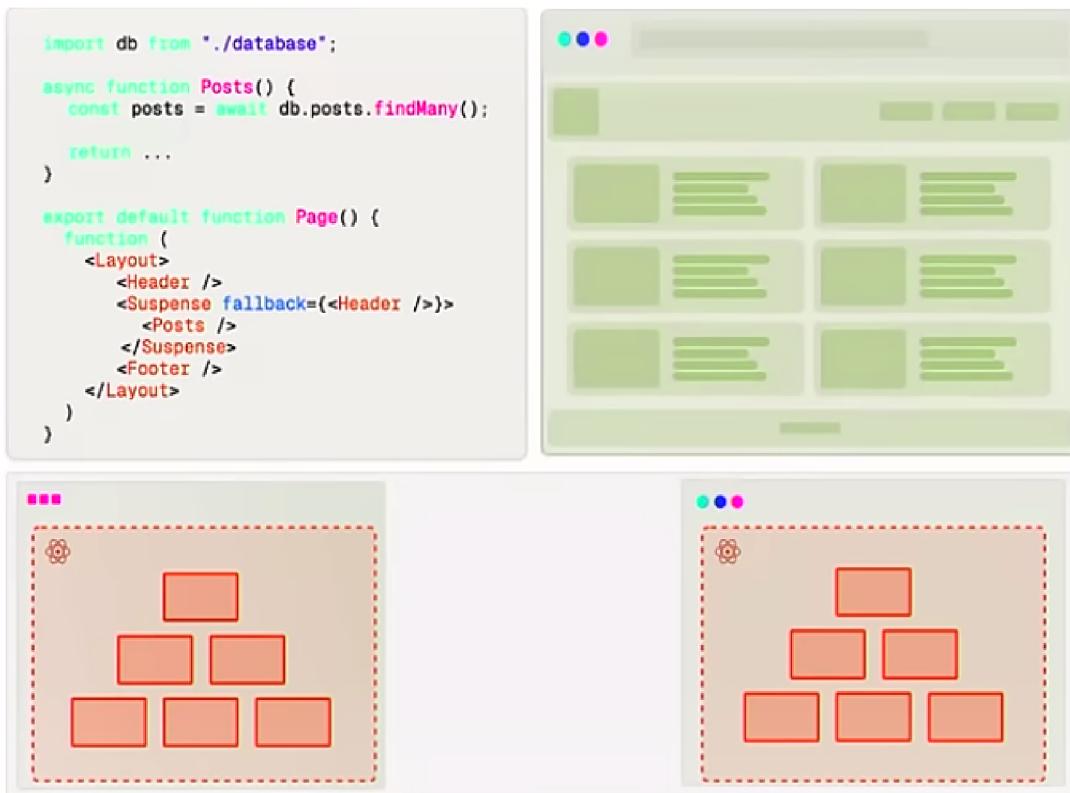
  return (
    <ul>
      {posts.map(post =>
        <li>{post.title}</li>
      )}
    </ul>
  )
}
```

RSC Payload

```
...
[$$, "ul", null, { "children": [
  $$, "li", null, { "children": "Post 1" }],
  $$, "li", null, { "children": "Post 2" }]
}]]
```

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.

```
"use client"

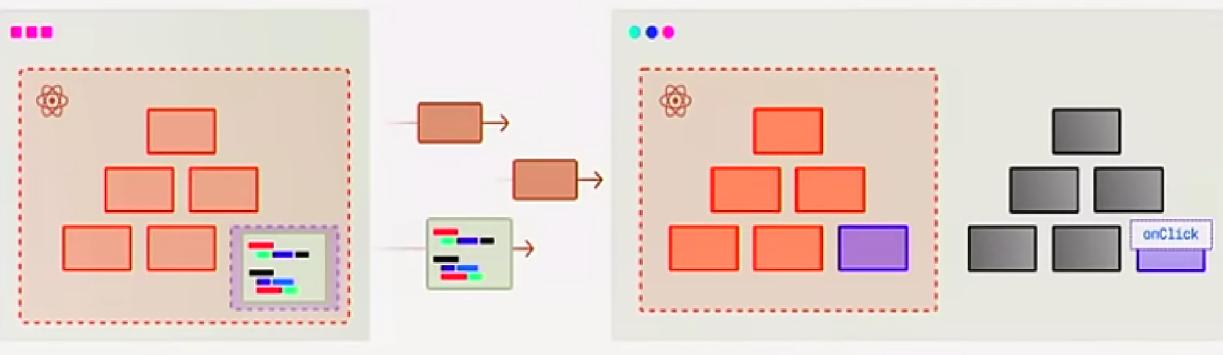
function Component() {
  const [state, setState] = useState();
  useEffect(() => {...});

  return (
    <button onClick={() => { ... }}>
      ...
    </button>
  )
}
```

```
function ServerComponent() {
  return (
    <div>
      <h1>Welcome!</h1>
      <ClientComponent />
    </div>
  )
}
```

```
"use client"

function ClientComponent() {
  return (
    <button onClick={() => { ... }}>...</button>
  )
}
```



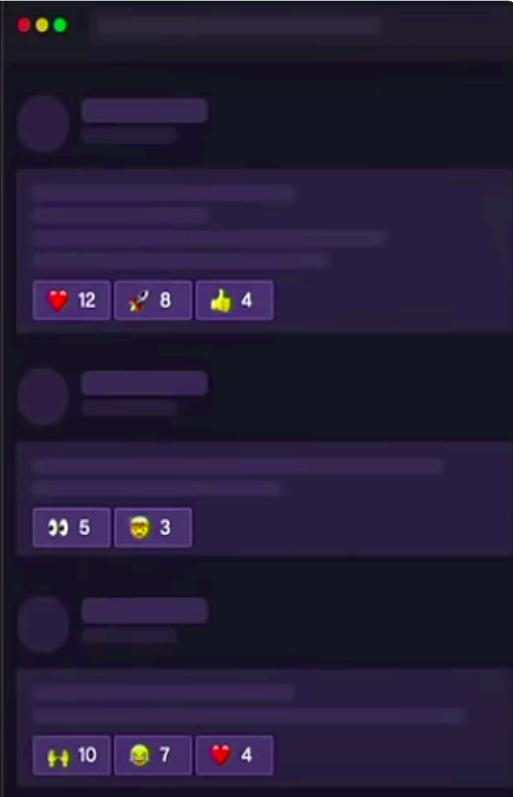
Because we are actually shipping the component to the client and not just the return value (as server components do), we can't fetch data the same way that we could with a server component. First, we could fetch the data in the server component and then pass it as a prop to the client component.

```
async function Comment() {
  const comment = await db.post.getComment();
  const reactions = await db.comment.getReactions(comment.id);

  return (
    <div>
      <p>{comment.text}</p>
      <Reactions reactions={reactions} />
    </div>
  )
}
```

```
"use client"

function Reactions({ reactions }) {
  return (
    <div>
      {reactions.map(reaction => <Reaction {...reaction} />)}
    </div>
  )
}
```



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

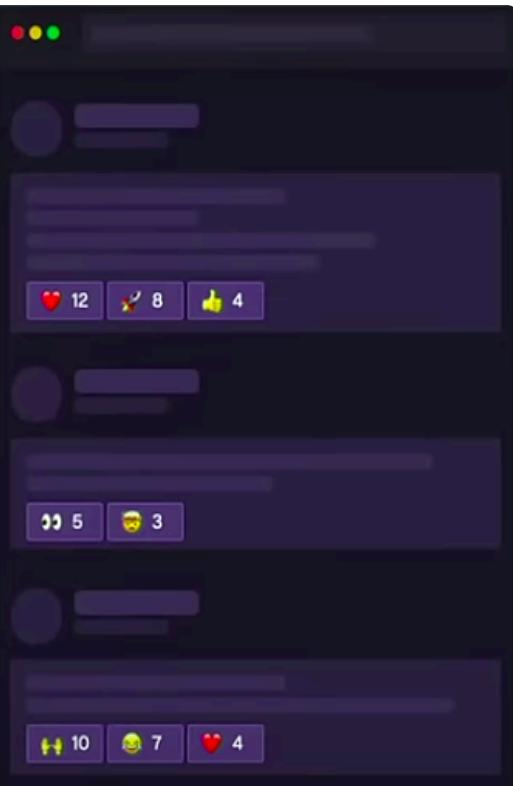
```
async function Comment() {
  const comment = await db.post.getComment();
  const reactionsPromise = db.comment.getReactions(comment.id);

  return (
    <div>
      <p>{comment.text}</p>
      <Reactions reactionsPromise={reactionsPromise} />
    </div>
  )
}
```

```
"use client"

function Reactions({ reactionsPromise }) {
  const reactions = use(reactionsPromise);

  return (
    <div>
      {reactions.map(reaction => <Reaction {...reaction} />)}
    </div>
  )
}
```



```
const resolvedData = use(promise);
```

```
const context = use(Context);
```

Client API

use

API that lets you read the value of a Promise or context

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

```
async function action() {  
  "use server";  
  ...  
}
```

Server API

Server Actions

Allow Client Components to call async functions on the server

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.

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

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

```
"use client"

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

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

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

```
"use client"

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

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

```
"use client"

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

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

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

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

Improvements

ref as a prop

Access `ref` as a prop for function components

```
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

```
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

```
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