by 

What Does "use client" Do?

April 25, 2025

Pay what you like Watch on YouTube

React Server Components (in?)famously has no API surface. It's an entire programming paradigm largely stemming from two directives:

- `'use client'`
- `'use server'`

I'd like to make a bold claim that their invention belongs in the same category as structured programming (`if` / `while`), first-class functions, and `async` / `await` . In other words, I expect them to survive past React and to become common sense.

The server *needs* to send code to the client (by sending a `<script>`). The client *needs* to talk back to the server (by doing a `fetch`). The `'use client'` and `'use server'` directives abstract over those, offering a first-class, typed, and statically analyzable way to pass control to a piece of your codebase on another computer:

- **`'use client'` is a typed `<script>` .**
- **`'use server'` is a typed `fetch()` .**

Together, these directives let you express the client/server boundary *within* the module system. They let you model a client/server application *as a single program spanning the two machines* without losing sight of the reality of the

network and serialization gap. That, in turn, allows seamless composition across the network.

Even if you never plan to use React Server Components, I think you should learn about these directives and how they work anyway. They're not even about React.

They are about the module system.

`'use server'`

First, let's look at `'use server'`.

Suppose you're writing a backend server that has some API routes:

```
async function likePost(postId) {
  const userId = getCurrentUser();
  await db.likes.create({ postId, userId });
  const count = await db.likes.count({ where: { postId } });
  return { likes: count };
}

async function unlikePost(postId) {
  const userId = getCurrentUser();
  await db.likes.destroy({ where: { postId, userId } });
  const count = await db.likes.count({ where: { postId } });
  return { likes: count };
}
```

```
app.post('/api/like', async (req, res) => {
  const { postId } = req.body;
  const json = await likePost(postId);
  res.json(json);
});

app.post('/api/unlike', async (req, res) => {
  const { postId } = req.body;
  const json = await unlikePost(postId);
  res.json(json);
});
```

Then you have some frontend code that calls these API routes:

```
document.getElementById('likeButton').onclick = async function() {
  const postId = this.dataset.postId;
  if (this.classList.contains('liked')) {
    const response = await fetch('/api/unlike', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ postId })
    });
    const { likes } = await response.json();
    this.classList.remove('liked');
    this.textContent = likes + ' Likes';
  } else {
    const response = await fetch('/api/like', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ postId, userId })
    });
    const { likes } = await response.json();
    this.classList.add('liked');
    this.textContent = likes + ' Likes';
  }
});
```

(For simplicity, this example doesn't try to handle race conditions and errors.)

This code is all dandy and fine, but it is "stringly-typed". What we're trying to do is to *call a function on another computer*. However, since the backend and the frontend are two separate programs, we have no way to express that other than a `fetch`.

Now imagine we thought about the frontend and the backend as *a single program split between two machines*. How would we express the fact that a piece of code wants to call another piece of code? What is the most direct way to express that?

If we set aside our preconceived notions about how the backend and the frontend "should" be built for a moment, we can remember that all we're *really*

trying to say is that we want to *call* `likePost` and `unlikePost` from our frontend code:

```
import { likePost, unlikePost } from './backend'; // This doesn't work :(

document.getElementById('likeButton').onclick = async function() {
  const postId = this.dataset.postId;
  if (this.classList.contains('liked')) {
    const { likes } = await unlikePost(postId);
    this.classList.remove('liked');
    this.textContent = likes + ' Likes';
  } else {
    const { likes } = await likePost(postId);
    this.classList.add('liked');
    this.textContent = likes + ' Likes';
  }
};
```

The problem is, of course, `likePost` and `unlikePost` cannot actually execute on the frontend. We can't literally import their implementations *into* the frontend. Importing the backend directly from the frontend is by definition *meaningless.*

However, suppose that there was a way to annotate the `likePost` and `unlikePost` functions as being *exported from the server* at the module level:

```
'use server'; // Mark all exports as "callable" from the frontend

export async function likePost(postId) {
  const userId = getCurrentUser();
  await db.likes.create({ postId, userId });
  const count = await db.likes.count({ where: { postId } });
  return { likes: count };
}

export async function unlikePost(postId) {
  const userId = getCurrentUser();
  await db.likes.destroy({ where: { postId, userId } });
  const count = await db.likes.count({ where: { postId } });
  return { likes: count };
}
```

We could then automate setting up the HTTP endpoints behind the scenes. And now that we have an opt-in syntax for exporting functions over the network, we could assign *meaning* to importing them from the frontend code — `import`ing them could simply give us `async` functions that perform those HTTP calls:

```
import { likePost, unlikePost } from './backend';

document.getElementById('likeButton').onclick = async function() {
  const postId = this.dataset.postId;
  if (this.classList.contains('liked')) {
    const { likes } = await unlikePost(postId); // HTTP call
    this.classList.remove('liked');
    this.textContent = likes + ' Likes';
  } else {
    const { likes } = await likePost(postId); // HTTP call
    this.classList.add('liked');
    this.textContent = likes + ' Likes';
  }
};
```

That's precisely what the `'use server'` directive is.

This is not a new idea—RPC has been around for decades. This is just a specific flavor of RPC for client-server applications where the server code can designate some functions as "server exports" (`'use server'`). Importing `likePost` from the server code works the same as a normal `import` , but importing `likePost` *from the client code* gives you an `async` function that performs the HTTP call.

Have another look at this pair of files:

```
'use server'; // Mark all exports as "callable" from the frontend

export async function likePost(postId) {
  const userId = getCurrentUser();
  await db.likes.create({ postId, userId });
  const count = await db.likes.count({ where: { postId } });
  return { likes: count };
}
```

```
export async function unlikePost(postId) {
  const userId = getCurrentUser();
  await db.likes.destroy({ where: { postId, userId } });
  const count = await db.likes.count({ where: { postId } });
  return { likes: count };
}
```

```
import { likePost, unlikePost } from './backend';

document.getElementById('likeButton').onclick = async function() {
  const postId = this.dataset.postId;
  if (this.classList.contains('liked')) {
    const { likes } = await unlikePost(postId); // HTTP call
    this.classList.remove('liked');
    this.textContent = likes + ' Likes';
  } else {
    const { likes } = await likePost(postId); // HTTP call
    this.classList.add('liked');
    this.textContent = likes + ' Likes';
  }
};
```

You may have objections—yes, it doesn't allow multiple consumers of the API (unless they're within the same codebase); yes, it requires some thought as to versioning and deployment; yes, it is more implicit than writing a `fetch`.

But if you adopt the view that the backend and a frontend are *a single program split across two computers*, you can't really "unsee" it. There is now a direct and visceral connection between the two modules. You can add types to narrow down their contract (and enforce that their types are serializable). You can use "Find All References" to see which functions from the server are used on the client. Unused endpoints can be automatically flagged and/or eliminated with dead code analysis.

Most importantly, you can now create self-contained abstractions that fully encapsulate both sides—a "frontend" attached to its corresponding "backend" piece. You don't need to worry about an explosion of API routes—the server/client split can be as modular as your abstractions. There is no global naming scheme; you organize the code using `export` and `import`, wherever you need them.

The `'use server'` directive makes the connection between the server and the client *syntactic*. It is no longer a matter of convention—it's *in* your module system.

It opens a *door* to the server.

`'use client'`

Now suppose that you want to pass some information from the backend to the frontend code. For example, you might render some HTML with a `<script>`:

```
app.get('/posts/:postId', async (req, res) => {
  const { postId } = req.params;
  const userId = getCurrentUser();
  const likeCount = await db.likes.count({ where: { postId } });
  const isLiked = await db.likes.count({ where: { postId, userId } }) > 0;
  const html = `<html>
    <body>
      <button
        id="likeButton"
        className="${isLiked ? 'liked' : ''}"
        data-postid="${Number(postId)}">
        ${likeCount} Likes
      </button>
      <script src="./frontend.js></script>
    </body>
  </html>`;
  res.text(html);
});
```

The browser will load that `<script>` which will attach the interactive logic:

```
document.getElementById('likeButton').onclick = async function() {
  const postId = this.dataset.postId;
  if (this.classList.contains('liked')) {
    // ...
  } else {
    // ...
  }
};
```

This works but leaves a few things to be desired.

For one, you probably don't want the frontend logic to be "global"—ideally, it should be possible to render multiple Like buttons, each receiving its own data and maintaining its own local state. It would also be nice to unify the display logic between the template in the HTML and the interactive JavaScript event handlers.

We know how to solve these problems. That's what component libraries are for! Let's reimplement the frontend logic as a declarative `LikeButton` component:

```
function LikeButton({ postId, likeCount, isLiked }) {
  function handleClick() {
    // ...
  }

  return (
    <button className={isLiked ? 'liked' : ''}>
      {likeCount} Likes
    </button>
  );
}
```

For simplicity, let's temporarily drop down to purely client-side rendering. With purely client-side rendering, our server code's job is just to pass the initial props:

```
app.get('/posts/:postId', async (req, res) => {
  const { postId } = req.params;
```

```
  const userId = getCurrentUser();
  const likeCount = await db.likes.count({ where: { postId } });
  const isLiked = await db.likes.count({ where: { postId, userId } }) > 0;
  const html = `<html>
    <body>
      <script src="./frontend.js></script>
      <script>
        const output = LikeButton(${JSON.stringify({
          postId,
          likeCount,
          isLiked
        })});
        render(document.body, output);
      </script>
    </body>
  </html>`;
  res.text(html);
});
```

Then the `LikeButton` can appear on the page with these props:

```
function LikeButton({ postId, likeCount, isLiked }) {
  function handleClick() {
    // ...
  }

  return (
    <button className={isLiked ? 'liked' : ''}>
      {likeCount} Likes
    </button>
  );
}
```

This makes sense, and is in fact exactly how React used to be integrated in server-rendered applications before the advent of client-side routing. You'd need to write a `<script>` to the page with your client-side code, and you would write another `<script>` with the inline data (i.e. the initial props) needed by that code.

Let's entertain the shape of this code for a little bit longer. There's something curious happening: the backend code clearly wants to *pass information* to the frontend code. However, the act of passing information is again *stringly-typed!*

What's going on here?

```
app.get('/posts/:postId', async (req, res) => {
  // ...
  const html = `<html>
    <body>
      <script src="./frontend.js></script>
      <script>
        const output = LikeButton(${JSON.stringify({
          postId,
          likeCount,
          isLiked
        })});
        render(document.body, output);
      </script>
    </body>
  </html>`;
  res.text(html);
});
```

What we seem to be saying is: have the browser load `frontend.js`, then find the `LikeButton` function in that file, and then pass this JSON to that function.

So what if could *just say that?*

```
import { LikeButton } from './frontend';

app.get('/posts/:postId', async (req, res) => {
  // ...
  const jsx = (
    <html>
      <body>
        <LikeButton
          postId={postId}
          likeCount={likeCount}
          isLiked={isLiked}
        />
      </body>
    </html>
  );
  // ...
});
```

```
'use client'; // Mark all exports as "renderable" from the backend
```

```
export function LikeButton({ postId, likeCount, isLiked }) {
  function handleClick() {
    // ...
  }

  return (
    <button className={isLiked ? 'liked' : ''}>
      {likeCount} Likes
    </button>
  );
}
```

We're taking a conceptual leap there but stick with me. What we're saying is, these are still two separate runtime environments—the backend and the frontend—but we're looking at them as a *single program* rather than as two separate programs.

This is why we set up a *syntactic connection* between the place that passes the information (the backend) and the function that needs to receive it (the frontend). And the most natural way to express that connection is, again, a plain `import`.

Note how, here too, importing from a file decorated with `'use client'` from the backend doesn't give us the `LikeButton` function itself. Instead, it gives a *client reference*—something that we can turn into a `<script>` tag under the hood later.

Let's see how this works.

This JSX:

```
import { LikeButton } from './frontend'; // "/src/frontend.js#LikeButton"

// ...
<html>
  <body>
    <LikeButton
      postId={42}
      likeCount={8}
      isLiked={true}
    />
  </body>
</html>
```

produces this JSON:

```
{
  type: "html",
  props: {
    children: {
      type: "body",
      props: {
        children: {
          type: "/src/frontend.js#LikeButton", // A client reference!
          props: {
            postId: 42
            likeCount: 8
            isLiked: true
          }
        }
      }
    }
  }
}
```

And this information—this *client reference*—lets us generate the `<script>` tags that load the code from the right file and call the right function under the hood:

```
<script src="./frontend.js"></script>
<script>
  const output = LikeButton({
    postId: 42,
    likeCount: 8,
    isLiked: true
```

```
  });
  // ...
</script>
```

In fact, we also have enough information that we can run the same function on the server to pregenerate the initial HTML, which we lost with client rendering:

```
<!-- Optional: Initial HTML -->
<button class="liked">
  8 Likes
</button>

<!-- Interactivity -->
<script src="./frontend.js"></script>
<script>
  const output = LikeButton({
    postId: 42,
    likeCount: 8,
    isLiked: true
  });
  // ...
</script>
```

Prerendering the initial HTML is optional, but it works using the same primitives.

Now that you know how it *works*, look over this code one more time:

```
import { LikeButton } from './frontend'; // "/src/frontend.js#LikeButton"

app.get('/posts/:postId', async (req, res) => {
  // ...
  const jsx = (
    <html>
      <body>
        <LikeButton
          postId={postId}
          likeCount={likeCount}
          isLiked={isLiked}
        />
      </body>
    </html>
  );
  // ...
```

```
});

'use client'; // Mark all exports as "renderable" from the backend

export function LikeButton({ postId, likeCount, isLiked }) {
  function handleClick() {
    // ...
  }

  return (
    <button className={isLiked ? 'liked' : ''}>
      {likeCount} Likes
    </button>
  );
}
```

If you set aside your existing notions of how the backend and the frontend code should interact, you'll see that there's something special happening here.

The backend code *references* the frontend code by using an `import` with `'use client'`. In other words, it expresses a direct connection *within the module system* between the part of the program that *sends* the `<script>` and the part of the program that lives *within* that `<script>`. Since there is a direct connection, it can be typechecked, you can use "Find All References", and all tooling is aware of it.

Like `'use server'` before it, `'use client'` makes the connection between the server and the client *syntactic*. Whereas `'use server'` opens a door from the client to the server, `'use client'` opens a door from the server to the client.

It's like two worlds with two doors between them.

Two Worlds, Two Doors

This is why `'use client'` and `'use server'` should not be seen as ways to "mark" code as being "on the client" or "on the server". That is not what they

do.

Rather, they let you *open the door* from one environment to the other:

- **`'use client'` exports client functions *to* the server.** Under the hood, the backend code sees them as references like `'/src/frontend.js#LikeButton'`. They can be rendered as JSX tags and will ultimately turn into `<script>` tags. (You can optionally pre-run those scripts on the server to get their initial HTML.)
- **`'use server'` exports server functions *to* the client.** Under the hood, the frontend sees them as `async` functions that call the backend via HTTP.

These directives express the network gap *within* your module system. They let you describe a client/server application as a *single program spanning two environments*.

They acknowledge and fully embrace the fact that these environments don't share any execution context—this is why neither `import` executes any code. Instead, they only let one side *refer* to code on the other side—and pass information to it.

Together, they let you "weave" the two sides of your program by creating and composing reusable abstractions with logic from both sides. But I think the pattern extends beyond React and even beyond JavaScript. Really, this is just RPC at the module system level with a mirror twin for sending more code to the client.

The server and the client are two sides of a single program. They're separated by time and space so they can't share the execution context and directly `import` each other. The directives "open the doors" across time and space: the server can *render* the client as a `<script>`; the client can *talk back* to the server via `fetch()`. But `import` is the most direct way to express that, so the directives let you use it.

Makes sense, doesn't it?

P.S.

Here's a little architectural diagram that you can use for your slides:

Pay what you like

Discuss on Bluesky · Watch on YouTube · Edit on GitHub