React Server Components Your Way

by Manuel Schiller, Tanner Linsley, and Jack Herrington on Apr 13, 2026.

At TanStack, we have always strived to build tools that cover the 90% use case with ease, but still give you the flexibility to break out of the box for advanced use cases. Why? Because we know that when things get serious, **you know what's best for your application and deserve the freedom to take control**.

That's always been the TanStack philosophy, and we're happy that you've trusted us to take our time to deliver that same experience with React Server Components.

What are server components?

documentation for a few minutes to get acquainted.

Why do RSCs matter?

RSCs are a necessary primitive for moving heavy or expensive rendering logic off the client and onto the server.

Especially static or infrequently changing content that can be cached consistently and granularly.

Markdown parsers, syntax highlighters, date formatting libraries, search indexing, content transforms, etc. are all great use cases, but certainly not the only ones.

They are a *very powerful primitive*.

The RSC Status Quo

Most people now think of RSCs in a server-first way: the server owns the tree, `'use client'` marks the interactive parts, and the framework conventions decide how the whole thing fits together.

That model can be compelling. It makes streaming, server rendering, and colocated server-side work feel built in from the start.

But it also turns RSCs from a useful primitive into the thing your whole app has to orbit. The framework ends up owning how RSCs are created, where they render, how interactive boundaries are defined, and how UI gets recomposed when data or user actions change.

That is the part we kept getting hung up on. We do not think you should have to buy into that whole model up front just to get value out of RSCs.

A Different RSC Model

the first place?

In TanStack Start, the core idea is that RSCs are **just streams of data** that you can fetch, cache, and render on your terms at any time on the client instead of a server-owned component tree. That one shift makes them far more composable without changing anything fundamental about how they work.

With TanStack Start, RSCs are just React Flight streams. That sounds almost too obvious to say out loud, but that's exactly the point. We did not want them wrapped in a black-box convention with special rules, APIs, and network effects that change everything about the framework.

We wanted RSCs to behave more like any other piece of server data, which means **nothing special should need to happen in a framework to support them** outside of simply supporting streaming as a first-class citizen.

What does this mean in practice? You can:

- Create and return them from anywhere on the server (server functions, API routes)
- Decode them wherever you want (both SSR and on the client)
- Cache them however you want. They're just streams of bytes. Immediately compatible with all existing tools.

What do they look like?

Here is an RSC in TanStack Start:

> *"Naturally, to cut down on server sync logic, we'll use TanStack Query to manage it!"*

tsx

```tsx
import { createServerFn } from '@tanstack/react-start'
import {
  createFromReadableStream,
```

```
// Create a server function
const getGreeting = createServerFn().handler(async () => {
  // Create an RSC readable stream
  return renderToReadableStream(
    // Return JSX
    <h1>Hello from the server</h1>,
  )
})

function Greeting() {
  const query = useQuery({
    queryKey: ['greeting'],
    queryFn: async () =>
      // Create a renderable element from the stream
      createFromReadableStream(
        // Call our server function to get the stream
        await getGreeting(),
      ),
  })

  // Render!
  return <>{query.data}</>
}
```

The Primitive APIs

At the primitive level, the API surface is intentionally small:

- **renderToReadableStream** renders React elements to a Flight stream on the server.
- **createFromReadableStream** decodes a Flight stream on the client or during SSR.
- **createFromFetch** decodes directly from a fetch response when that shape is more convenient.

Under the hood, the story is straightforward: React renders to a Flight stream on the server, and the client decodes that stream back into a React element tree.

There is no secret extra protocol here. That is the primitive. Standard Flight streams in, standard React elements out.

decision through.

Caching

Speaking of unnecessary framework convention: caching is not something new to reinvent, and **RSCs are no exception**.

When RSCs become "just data", the caching story gets a lot simpler. And because these are just granular streams delivered plainly over HTTP and handled transparently during rendering, they are not only easy to cache on the client, but also anywhere along the way on the server: in memory, in a database, behind a CDN, or wherever else your architecture already caches bytes, responses, or data.

This equally applies to **caching layers you likely already know on the client** as well, instead of requiring novel approaches and mental model shifts.

Let me explain.

Query: Fine-Grained Control

TanStack Query illustrates this so well. It does not need a special "RSC mode". Once the RSC payload is part of an async query, you still get explicit cache keys, `staleTime`, background refetching, and the rest of Query's toolbox. For static content, just set `staleTime: Infinity` and you are done.

tsx

```tsx
import { createServerFn } from '@tanstack/react-start'
import {
  createFromReadableStream,
  renderToReadableStream,
} from '@tanstack/react-start/rsc'

const getGreeting = createServerFn().handler(async () => {
  return renderToReadableStream(<h1>Hello from the server</h1>)
})
```

```
  const { data } = useSuspenseQuery({
    queryKey: ['greeting-rsc', postId],
    queryFn: async () => ({
      Greeting: await createFromReadableStream(await getGreeting()),
    }),
    staleTime: 5 * 60 * 1000,
  })

  return <>{data.Greeting}</>
}
```

Router: Automatic Route-Based Caching

TanStack Router is even cooler. Because it natively supports streams, RSC payloads in route loaders are, again, "just data".

You can await them, stream them (yes, stream a stream), and naturally cache the result in the router cache just like any other loader output.

tsx

```tsx
const getGreeting = createServerFn().handler(async () => {
  return renderToReadableStream(<h1>Hello from the server</h1>)
})

export const Route = createFileRoute('/hello')({
  loader: async () => ({
    greeting: getGreeting(),
  }),
  component: function HelloPage() {
    const { greeting } = Route.useLoaderData()
    return <>{greeting}</>
  },
})
```

> *"Navigate from `/posts/abc` to `/posts/xyz` and the loader runs again. Navigate back to `/posts/abc` and Router can serve the cached result instantly. That snappy back-button experience falls out of the same loader caching model you are already using."*

Because GET server functions are still just HTTP under the hood, you can also cache the response itself at the CDN layer.

tsx

```tsx
import { createServerFn } from '@tanstack/react-start'
import { renderToReadableStream } from '@tanstack/react-start/rsc'
import { setResponseHeaders } from '@tanstack/react-start/server'

const getGreeting = createServerFn({ method: 'GET' }).handler(async () => {
  setResponseHeaders(
    new Headers({
      'Cache-Control': 'public, max-age=0, must-revalidate',
      'Netlify-CDN-Cache-Control':
        'public, max-age=300, durable, stale-while-revalidate=300',
    }),
  )

  return renderToReadableStream(<h1>Hello from the server</h1>)
})
```

That is the same pattern we use here for blog and docs content. Browser cache rules can stay conservative while the CDN caches the server function response much more aggressively.

With Start, RSCs fit into the same data workflows you already use.

Security: One-Way Data Flow

You've probably seen recent CVEs around RSC stacks.

We intentionally do not support `'use server'` actions, both because of existing attack vectors and because they can create highly implicit network boundaries.

TanStack Start requires explicit RPCs via `createServerFn`. The client-server boundary is deliberate, with hardened serialization, validation, and middleware semantics that encourage treating all user input as untrusted by default.

patched.

The Full Spectrum

With RSCs as primitives, TanStack Start covers every frontend use case. And we mean *every* one:

- **Fully Interactive**
  No server components at all. Client-first, SPA-style. RSCs are an optimization you add when helpful, not a paradigm you're forced to build around. This is where most "apps" already live today.

- **Hybrid**
  Server components for static shells, data-heavy regions, or SEO-critical content, with client components where interactivity matters. Mixed "app" and "site" projects (product + marketing) fit this especially well. In practice, this tends to help the "site" side more often than the "app" side.

- **Mostly Static**
  Predominantly static content, parsed and rendered server-side as RSCs, but still a powerful and hydrated SPA with bits of client interactivity sprinkled in where needed (e.g. comments, search, dynamic widgets). Think blogs, docs, marketing pages.

- **Fully Static**
  Pre-render everything at build time. Just ship HTML. You might not even need RSCs here if you never need to hydrate!

One framework. One mental model. The entire spectrum.
You don't have to choose "interactive framework" or "static framework" or "RSC framework."
You choose patterns **per-route, per-component, per-use-case**. The architecture supports all of it. Because, again, you know what's best for your app.

We did not want to make the case with vibes, so we migrated the content-heavy parts of tanstack.com and measured it.

The result was exactly what we hoped for, and also more limited than the hype would suggest.

The best pages got meaningfully smaller:

- **Blog post pages** dropped about **153 KB gzipped** from the client JS graph.
- **Docs pages** dropped about **153 KB gzipped**.
- **Docs example pages** dropped about **40 KB gzipped**.

And the real-world numbers moved with them:

- `/blog/react-server-components` went from **52 → 74** in Lighthouse.
  - Total Blocking Time dropped from **1,200ms → 260ms**.
  - Transfer size dropped from **1,101 KiB → 785 KiB**.
- `/router/latest/docs/overview` went from **78 → 81**.
  - Total Blocking Time dropped from **280ms → 200ms**.
  - Transfer size dropped from **917 KiB → 777 KiB**.

That is the point. **Heavy client work stopped shipping to the client.** Markdown parsing went away. Syntax highlighting went away. The browser got less JavaScript and did less work. As a side effect, we also got to delete the old client markdown and highlighting path instead of carrying two versions of the same rendering logic around.

But RSCs are not a universal coupon code for performance. Some landing pages were basically flat, and a few were slightly worse. Pages that are already dominated by interactive UI shell do not automatically get faster just because you threaded a server component into the tree somewhere.

That is the tradeoff:

pages. That is the sweet spot.

- **RSCs are less obviously useful when the page is already mostly client state and interaction.** Dashboards, builders, long-lived app sessions, and some landing pages can be flat or mixed unless you are removing real client-side work.

That is why we think they matter. Not because every route should become a server component. Because when you use them where they fit, the payoff is measurable and not subtle.

Introducing Composite Components

Everything above stands on its own. If all TanStack Start did was treat RSCs as fetchable, cacheable, renderable data, we would already think that was a better foundation for RSCs.

But we kept pulling on one question: what if the server did not need to decide every client-shaped part of the UI at all?

That led us to create something entirely new: **Composite Components**.

`use client` still works the same way in TanStack Start when the server intentionally wants to render a client component. `use server` does not. Start uses explicit Server Functions instead.

Composite Components are not a replacement for `use client`. They solve a similar composition problem from the opposite direction. Instead of the server deciding which client component renders where, the server can leave join points open and let the client own the tree and decide what fills them.

That is the part that feels genuinely new to us.

Slots Inside One Component

A Composite Component can render server UI while exposing **slots** for client content. Slots use plain React patterns you already know:

- render props (like `renderPostActions`)

Because the client owns the component tree, the components you pass into slots are regular client components. No `'use client'` directive required. The server positions them as opaque placeholders but can't inspect, clone, or transform them. That is the point: the server can ask for "something goes here" without needing to know what that something is.

Server

tsx

```tsx
import { createCompositeComponent } from '@tanstack/react-start/rsc'

const getPost = createServerFn().handler(async ({ data }) => {
  const post = await db.posts.get(data.postId)

  const src = await createCompositeComponent(
    (props: {
      children?: React.ReactNode
      renderPostActions?: (data: {
        postId: string
        authorId: string
      }) => React.ReactNode
    }) => (
      <article>
        <h1>{post.title}</h1>
        <p>{post.body}</p>

        {/* Server renders this link directly */}
        <Link to="/posts/$postId" params={{ postId: post.nextPostId }}>
          Next Post
        </Link>

        {/* Slot: server requests client UI here */}
        <footer>
          {props.renderPostActions?.({
            postId: post.id,
            authorId: post.authorId,
          })}
        </footer>

        {/* Slot: client fills this with children */}
        {props.children}
      </article>
```

```
  return { src }
})
```

Client

tsx

```tsx
import { CompositeComponent } from '@tanstack/react-start/rsc'

function PostPage({ postId }) {
  const { data } = useSuspenseQuery({
    queryKey: ['post', postId],
    queryFn: () => getPost({ data: { postId } }),
  })

  return (
    <CompositeComponent
      src={data.src}
      renderPostActions={({ postId, authorId }) => (
        // Full client interactivity: hooks, state, context
        <PostActions postId={postId} authorId={authorId} />
      )}
    >
      <Comments postId={postId} />
    </CompositeComponent>
  )
}
```

The server renders the `<Link>` directly and leaves join points for the client:

- A `renderPostActions` slot for `<PostActions>` with server-provided arguments
- A `children` slot for `<Comments>`

Slot names are just props. `renderPostActions` is example code, not special API syntax.

Since a Composite Component is still data, the client can also treat it as a building block:

- Interleave multiple fragments in a new tree
- Wrap them in client providers or layouts

- Reorder or swap them based on client state

Same mental model as normal React composition. The difference is that the server no longer has to decide every interesting part of the tree ahead of time.

Current Status: Experimental

RSC support is experimental in TanStack Start RC and will remain experimental into early v1.

Serialization: This release uses React's native Flight protocol. TanStack Start's usual serialization features aren't available within server components for now.

API surface: The current helpers are stable enough to use, but expect refinements while the feature is experimental. The docs will stay current as the APIs evolve.

If you hit rough edges, open an issue or join the Discord.

FAQ

We get questions. Here are answers.

How does this compare to Next.js App Router?

Next.js App Router is server-first: your component tree lives on the server by default, and you opt into client interactivity with `'use client'`.

TanStack Start is **isomorphic-first**: your tree lives wherever makes sense. At the base level, RSC output can be fetched, cached, and rendered where it makes sense instead of owning the whole tree. When you want to go further, Composite Components let the client assemble the final tree instead of just accepting a server-owned one.

Can I use this with Next.js or Remix?

Do I have to use RSCs?

Nope. RSCs are completely opt-in. You can build fully client-side routes (including `ssr: false`), use traditional SSR without server components, or go fully static.

They are another tool in the box, not the new mandatory center of gravity.

Where should I look for the full technical docs?

The Server Components docs. They cover setup, helper APIs, examples, constraints, and the low-level details this post intentionally skips.

What about security?

See the Security: One-Way Data Flow section above. The short version: TanStack Start's architecture doesn't parse Flight data from the client, so recent CVEs affecting other RSC frameworks don't apply here.

Your RSCs, Your Way

We started this post with a simple idea: you know what's best for your application architecture. That is why we built TanStack Start's RSC model to stay flexible instead of prescriptive.

Too much of the ecosystem treats RSCs like they need to become your app architecture. We think they work better as a primitive. And when you want to go further, Composite Components open up a composition model that most RSC systems do not even try to offer. Want a fully interactive SPA? Go for it. Want to sprinkle in server components for heavy lifting? Easy. Want to go full static? That works too. The architecture supports all of it because *your* app isn't one-size-fits-all, and your framework shouldn't be either.

- Server Components Docs
- GitHub
- Discord

Let's build something amazing together.

Edit on GitHub