Mastering TanStack Start: A Complete Guide to Full-Stack React Development

This book provides an in-depth guide through the TanStack Start documentation, offering insights and practical examples for mastering full-stack React development with TanStack Router and Start. It's a valuable resource for developers looking to leverage the full potential of TanStack's powerful type-safe APIs, SSR, and streaming capabilities.

Table of Contents

Getting Started with TanStack Start

- Overview
- Getting Started
- Quick Start
- Build from Scratch
- Learn the Basics

Core Features and Functionality

- Server Functions
- Static Server Functions
- Middleware
- Server Routes
- SPA Mode

Hosting and Integration

- Hosting
- Authentication
- Databases
- Observability
- Static Prerendering
- Path Aliases
- Tailwind CSS Integration
- Migrate from Next.js

Examples and Practical Use Cases

- Basic Setup
- Basic + React Query
- Basic + Clerk Auth
- Basic + DIY Auth
- Basic + Supabase Auth
- Trellaux + Convex

TanStack Start Overview | TanStack Start React Docs

Contents

- Router or Start?
- How does it work?
- When should I use it?
- When might I not want to use it?
- How is TanStack Start funded?
- Ready to get started?

TanStack Start Overview

TanStack Start is a full-stack React framework powered by TanStack Router. It provides full-document SSR, streaming, server functions, bundling, and more. Thanks to Vite, it's ready to develop and deploy to any hosting provider or runtime you want!

Router or Start?

TanStack Router is a powerful, type-safe, and full-featured routing system for React applications. It is designed to handle the beefiest of full-stack routing requirements with ease. TanStack Start builds on top of Router's type system to provide type-safe full-stack APIs that keep you in the fast lane.

What you get with TanStack Router:

- 100% inferred TypeScript support
- Typesafe navigation
- Nested Routing and pathless layout routes
- Built-in Route Loaders w/ SWR Caching
- Designed for client-side data caches (TanStack Query, SWR, etc.)
- Automatic route prefetching
- Asynchronous route elements and error boundaries
- File-based Route Generation
- Typesafe JSON-first Search Params state management APIs
- Path and Search Parameter Schema Validation
- Search Param Navigation APIs
- Custom Search Param parser/serializer support
- Search param middleware
- Route matching/loading middleware

What you get with TanStack Start:

- Full-document SSR
- Streaming
- Server Functions / RPCs
- Bundling
- Deployment
- Full-Stack Type Safety

Summary:
Use TanStack Router for client-side routing and TanStack Start for full-stack routing.

How does it work?

TanStack Start uses Vite to bundle and deploy your application and empowers amazing features like:

- Provide a unified API for SSR, streaming, and hydration
- Extract server-only code from your client-side code (e.g., server functions)
- Bundle your application for deployment to any hosting provider

When should I use it?

TanStack Start is perfect if you want to build a full-stack React application with:

- Full-document SSR & Hydration
- Streaming
- Server Functions / RPCs
- Full-Stack Type Safety
- Robust Routing
- Rich Client-Side Interactivity

When might I not want to use it?

It's not suitable if:

- Your site will be 100% static
- Your goal is a server-rendered site with zero JS or minimal client-side interactivity
- You're seeking a React-Server-Component-first framework (though support for RSCs is coming soon!)

How is TanStack Start funded?

TanStack collaborates with partners to offer the best developer experience:

- **Clerk:** The best authentication solution for modern web apps, integrating deeply with TanStack Start. Learn more
- **Netlify:** Leading hosting platform. Learn more
- **Neon:** Serverless Postgres database, ideal for full-stack apps. Learn more
- **Convex:** Seamless real-time database backend. Learn more
- **Sentry:** Monitoring and error tracking. Learn more

Ready to get started?

Proceed to the next page to learn how to install TanStack Start and create your first app!

Edit on GitHub

On this page

- Router or Start?
- How does it work?
- When should I use it?
- When might I not want to use it?

- How is TanStack Start funded?
- Ready to get started?

Additional Links

- Discord
- Getting Started

Our Partners

- Clerk: Modern web authentication solutions.
- Netlify: Web hosting platform.
- Neon: Serverless Postgres.
- Convex: Real-time database.
- Sentry: Observability platform.

More Resources

- TanStack Table: Supercharge your tables or build a datagrid for multiple frontend frameworks.
- TanStack DB: Collections, live queries, and optimistic mutations for reactive UI.

Subscribe to Bytes

Your weekly dose of JavaScript news, delivered every Monday to over 100,000 devs, for free.

No spam. Unsubscribe at *any* time.

Getting Started | TanStack Start React Docs

TanStack Start v0
Auto
Framework

React
Version
Latest
Search...
+ K
Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Overview
  react
- Getting Started
  react
- Quick Start
  react
- Build from Scratch
  react
- Learn the Basics
  react
- Server Functions
  react
- Static Server Functions
  react
- Middleware
  react
- Server Routes
  react
- SPA Mode
  react
- Hosting
  react
- Authentication
  react
- Databases
  react
- Observability
  react
- Static Prerendering
  react
- Path Aliases
  react
- Tailwind CSS Integration

react
- Migrate from Next.js
react

Examples

- Basic
react
- Basic + React Query
react
- Basic + Clerk Auth
react
- Basic + DIY Auth
react
- Basic + Supabase Auth
react
- Trellaux + Convex
react
- Trellaux
react
- WorkOS
react
- Material UI
react

Tutorials

- Reading and Writing a File
react

Overview

Quick Start

Our Partners

- Official Deployment Partner
- Netlify
- Neon
- Convex
- Sentry

TanStack Forms

Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit and Svelte. Learn More

TanStack Config

The build and publish utilities used by all of our projects. Use it if you dare! Learn More

---

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

No spam. Unsubscribe at *any* time.

Quick Start | TanStack Start React Docs

On this page

- Impatient?
- Examples
- Stackblitz
- Quick Deploy
- Manual Deploy
- Other Router Examples

Quick Start

Impatient?

If you're impatient, you can clone and run the Basic example right away with the following commands:

```
npx gitpick TanStack/router/tree/main/examples/react/start-basic start-basic
cd start-basic
npm install
npm run dev
```

You can replace `start-basic` with the slug of any other example from the list below.

Once you've cloned the example you want, head back to the Learn the Basics guide to learn how to use TanStack Start!

Examples

TanStack Start has lots of examples to get you started. Pick one below:

- Basic (`start-basic`)
- Basic + Auth (`start-basic-auth`)
- Counter (`start-counter`)
- Basic + React Query (`start-basic-react-query`)
- Clerk Auth (`start-clerk-basic`)
- Convex + Trellaux (`start-convex-trellaux`)
- Supabase (`start-supabase-basic`)
- Trellaux (`start-trellaux`)
- WorkOS (`start-workos`)
- Material UI (`start-material-ui`)

Stackblitz

Each example has an embedded Stackblitz preview to help you find a good starting point.

Quick Deploy

Click the **Deploy to Netlify** button on any example's page to clone and deploy it directly.

Manual Deploy

To clone and deploy an example elsewhere, run:

```
npx gitpick TanStack/router/tree/main/examples/react/EXAMPLE_SLUG my-new-project
cd my-new-project
npm install
npm run dev
```

Replace `EXAMPLE_SLUG` with the specific example slug.

After deploying, return to the Learn the Basics to learn how to use TanStack Start!

Other Router Examples

While not Start-specific, these examples help understand how TanStack Router works:

- Quickstart (file-based)
- Basic (file-based)
- Kitchen Sink (file-based)
- Kitchen Sink + React Query
- Location Masking
- Authenticated Routes
- Scroll Restoration
- Deferred Data
- Navigation Blocking
- View Transitions
- With tRPC
- With tRPC + React Query

Navigation Links

- Get Started
- Build from Scratch

Our Partners

- Clerk (Official Deployment Partner)
- Netlify
- Neon
- Convex
- Sentry

Resources

- TanStack Form
  Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.
  Learn More

- TanStack Config
  The build and publish utilities used by all of our projects. Use it if you dare!
  Learn More

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

No spam. Unsubscribe at *any* time.


Build a Project from Scratch | TanStack Start React Docs

Link to GitHub edit

On this page

- TypeScript Configuration
- Install Dependencies
- Update Configuration Files
- Add the Basic Templating
- The Router Configuration
- The Root of Your Application
- Writing Your First Route

Note

If you chose to quick start with an example or cloned project, you can skip this guide and move to the Learn the Basics.

So you want to build a TanStack Start project from scratch?

This guide will help you build a **very** basic TanStack Start web application. Together, we will use TanStack Start to:

- Serve an index page...
- Which displays a counter...
- With a button to increment the counter persistently.

Here is what that will look like

Let's create a new project directory and initialize it.

```
mkdir myApp
cd myApp
npm init -y
```

Note:
> We use `npm` in all of these examples, but you can use your package manager of choice instead.

TypeScript Configuration

We highly recommend using TypeScript with TanStack Start. Create a `tsconfig.json` file with at least the following settings:

```
{
  "compilerOptions": {
    "jsx": "react-jsx",
    "moduleResolution": "Bundler",
    "module": "ESNext",
    "target": "ES2022",
    "skipLibCheck": true,
    "strictNullChecks": true
  }
}
```

Note:
> Enabling `verbatimModuleSyntax` can result in server bundles leaking into client bundles. It is recommended to keep this option disabled.

Install Dependencies

TanStack Start is powered by Vite and TanStack Router, and requires them as dependencies.

To install them, run:

```
npm i @tanstack/react-start @tanstack/react-router vite
```

You'll also need React:

```
npm i react react-dom
```

And some TypeScript packages:

```
npm i -D typescript @types/react @types/react-dom vite-tsconfig-paths
```

Update Configuration Files

Update your `package.json` to use Vite's CLI and set `"type": "module"`:

```
{
  // ...
  "type": "module",
  "scripts": {
    "dev": "vite dev",
    "build": "vite build"
  }
}
```

Then, configure TanStack Start's Vite plugin in `vite.config.ts`:

```
// vite.config.ts
import { defineConfig } from 'vite'
import tsConfigPaths from 'vite-tsconfig-paths'
import { tanstackStart } from '@tanstack/react-start/plugin/vite'

export default defineConfig({
  server: {
    port: 3000,
  },
```

```
  plugins: [tsConfigPaths(), tanstackStart()],
})
```

Add the Basic Templating

There are 2 required files for TanStack Start usage:

1. The router configuration
2. The root of your application

Once configured, your file tree should look like:

```
.
├── src/
│   ├── routes/
│   │   └── __root.tsx
│   ├── router.tsx
│   ├── routeTree.gen.ts
├── .gitignore
├── vite.config.ts
├── package.json
└── tsconfig.json
```

The Router Configuration

This file dictates the behavior of TanStack Router used within Start. You can configure preloading, data caching, etc.

Note:
You won't have `routeTree.gen.ts` yet; it will be generated when you run TanStack Start.

```
// src/router.tsx
import { createRouter as createTanStackRouter } from '@tanstack/react-router'
import { routeTree } from './routeTree.gen'

export function createRouter() {
  const router = createTanStackRouter({
    routeTree,
    scrollRestoration: true,
  })

  return router
}

declare module '@tanstack/react-router' {
  interface Register {
    router: ReturnType<typeof createRouter>
  }
}
```

The Root of Your Application

Create the root component that wraps all routes:

```
// src/routes/__root.tsx
/// <reference types="vite/client" />
import type { ReactNode } from 'react'
import {
  Outlet,
  createRootRoute,
  HeadContent,
  Scripts,
} from '@tanstack/react-router'

export const Route = createRootRoute({
  head: () => ({
    meta: [
      { charSet: 'utf-8' },
      { name: 'viewport', content: 'width=device-width, initial-scale=1' },
      { title: 'TanStack Start Starter' },
    ],
  }),
  component: RootComponent,
})

function RootComponent() {
  return (
    <RootDocument>
      <Outlet />
    </RootDocument>
  )
}

function RootDocument({ children }: Readonly<{ children: ReactNode }>) {
  return (
    <html>
      <head>
        <HeadContent />
      </head>
      <body>
        {children}
        <Scripts />
      </body>
    </html>
  )
}
```

Writing Your First Route

Create a route in `src/routes/index.tsx`:

```
// src/routes/index.tsx
import * as fs from 'node:fs'
import { createFileRoute, useRouter } from '@tanstack/react-router'
import { createServerFn } from '@tanstack/react-start'

const filePath = 'count.txt'
```

```
async function readCount() {
  return parseInt(
    await fs.promises.readFile(filePath, 'utf-8').catch(() => '0'),
  )
}

const getCount = createServerFn({
  method: 'GET',
}).handler(() => {
  return readCount()
})

const updateCount = createServerFn({ method: 'POST' })
  .validator((d: number) => d)
  .handler(async ({ data }) => {
    const count = await readCount()
    await fs.promises.writeFile(filePath, `${count + data}`)
  })

export const Route = createFileRoute('/')({
  component: Home,
  loader: async () => await getCount(),
})

function Home() {
  const router = useRouter()
  const state = Route.useLoaderData()

  return (
    <button
      type="button"
      onClick={() => {
        updateCount({ data: 1 }).then(() => {
          router.invalidate()
        })
      }}
    >
      Add 1 to {state}?
    </button>
  )
}
```

Final notes

That's it! 🤯 You've set up a TanStack Start project and created your first route. 🎉

Run `npm run dev` to start the server and open `http://localhost:3000` to see your app.

Want to deploy? Check out the hosting guide.

Learn the Basics | TanStack Start React Docs

Navigation

- Dependencies
- It all "Starts" with the Router
- Route Generation
- The Server Entry Point (Optional)
- The Client Entry Point (Optional)
- The Root of Your Application
- Routes
- Navigation
- Server Functions (RPCs)
- Mutations
- Data Loading

Dependencies

This guide will help you learn the basics behind how TanStack Start works, regardless of how you set up your project.

TanStack Start is powered by Vite and TanStack Router.

- **TanStack Router:** A router for building web applications.
- **Vite:** A build tool for bundling your application.

It all "Starts" with the Router

This is the file that will dictate the behavior of TanStack Router used within Start. Here, you can configure everything from the default preloading functionality to caching staleness.

```
// app/router.tsx
import { createRouter as createTanStackRouter } from '@tanstack/react-router'
import { routeTree } from './routeTree.gen'

export function createRouter() {
  const router = createTanStackRouter({
    routeTree,
    scrollRestoration: true,
  })

  return router
}

declare module '@tanstack/react-router' {
  interface Register {
    router: ReturnType<typeof createRouter>
```

```
  }
}
```

- Notice the `scrollRestoration` property. This is used to restore the scroll position of the page when navigating between routes.

Route Generation

The `routeTree.gen.ts` file is generated when you run TanStack Start (via `npm run dev` or `npm run start`) for the first time. It contains the generated route tree and utility types that make TanStack Start fully type-safe.

The Server Entry Point (Optional)

Note: The server entry point is **optional** out of the box. If not provided, TanStack Start will automatically handle the server entry point for you using the default setup below:

```
// src/server.ts
import {
  createStartHandler,
  defaultStreamHandler,
} from '@tanstack/react-start/server'
import { createRouter } from './router'

export default createStartHandler({
  createRouter,
})(defaultStreamHandler)
```

Whether statically generating or serving dynamically, this `server.ts` file is the entry point for SSR.

- It's important that a new router is created for each request to keep data unique.
- `defaultStreamHandler` streams HTML to the client, enabling streaming SSR.

The Client Entry Point (Optional)

Note: The client entry point is **optional**. If not provided, TanStack Start manages it automatically.

To hydrate on the client and enable client-side routing:

```
// app/client.tsx
import { StartClient } from '@tanstack/react-start'
import { StrictMode } from 'react'
import { hydrateRoot } from 'react-dom/client'
import { createRouter } from './router'

const router = createRouter()

hydrateRoot(
  document,
  <StrictMode>
    <StartClient router={router} />
  </StrictMode>,
)
```

This setup hydrates the app after server render, enabling client-side routing once the initial HTML is loaded.

The Root of Your Application

The `__root` route wraps all app routes and functions as the application shell. It is always rendered and is ideal for global layout logic.

```
// app/routes/__root.tsx
import {
  Outlet,
  createRootRoute,
  HeadContent,
  Scripts,
} from '@tanstack/react-router'
import type { ReactNode } from 'react'

export const Route = createRootRoute({
  head: () => ({
    meta: [
      { charSet: 'utf-8' },
      { name: 'viewport', content: 'width=device-width, initial-scale=1' },
      { title: 'TanStack Start Starter' },
    ],
  }),
  component: RootComponent,
})

function RootComponent() {
  return (
    <RootDocument>
      <Outlet />
    </RootDocument>
  )
}

function RootDocument({ children }: Readonly<{ children: ReactNode }>) {
  return (
    <html>
      <head>
        <HeadContent />
      </head>
      <body>
        {children}
        <Scripts />
      </body>
    </html>
  )
}
```

Notes:

- This layout may evolve when SPA mode is introduced.
- `<Scripts />` loads all client-side JavaScript.

Routes

Routes are defined with `createFileRoute`, are code-split, lazy-loaded, and can coordinate data fetching using loaders:

```
// app/routes/index.tsx
import * as fs from 'node:fs'
import { createFileRoute, useRouter } from '@tanstack/react-router'
import { createServerFn } from '@tanstack/react-start'

const filePath = 'count.txt'

async function readCount() {
  return parseInt(
    await fs.promises.readFile(filePath, 'utf-8').catch(() => '0'),
  )
}

const getCount = createServerFn({ method: 'GET' }).handler(() => readCount())

const updateCount = createServerFn({ method: 'POST' })
  .validator((d: number) => d)
  .handler(async ({ data }) => {
    const count = await readCount()
    await fs.promises.writeFile(filePath, `${count + data}`)
  })

export const Route = createFileRoute('/')({
  component: Home,
  loader: async () => await getCount(),
})

function Home() {
  const router = useRouter()
  const state = Route.useLoaderData()

  return (
    <button
      type="button"
      onClick={() => {
        updateCount({ data: 1 }).then(() => {
          router.invalidate()
        })
      }}
    >
      Add 1 to {state}?
    </button>
  )
}
```

Navigation

TanStack Start is built on TanStack Router. Features include:

- `<Link to="/path">` for navigation
- `useNavigate()` for imperative routing
- `useRouter()` to access router instance
- Reactive hooks that update on router state changes

Example of using `<Link>`:

```
import { Link } from '@tanstack/react-router'

function Home() {
  return <Link to="/about">About</Link>
}
```

For more info, refer to the navigation guide.

Server Functions (RPCs)

Server functions, created with `createServerFn`, can be called from SSR and client:

```
import { createServerFn } from '@tanstack/react-start'
import * as fs from 'node:fs'
import { z } from 'zod'

const getUserById = createServerFn({ method: 'GET' })
  .validator(z.string())
  .handler(async ({ data }) => {
    // server-side logic here
    return db.query.users.findFirst({ where: { id: data } })
  })

// usage
const user = await getUserById({ data: '1' })
```

More details: Server functions guide.

Mutations

Mutations are performed with server functions, optionally invalidating cached data:

```
import { createServerFn } from '@tanstack/react-start'
import { z } from 'zod'
import { dbUpdateUser } from '...'

const UserSchema = z.object({ id: z.string(), name: z.string() })
export type User = z.infer<typeof UserSchema>

export const updateUser = createServerFn({ method: 'POST' })
  .validator(UserSchema)
  .handler(({ data }) => dbUpdateUser(data))

// Usage example with React Query and Router:
import { useQueryClient } from '@tanstack/react-query'
import { useRouter } from '@tanstack/react-router'
import { useServerFn } from '@tanstack/react-start'
```

```
import { updateUser, type User } from '...'

function MyComponent() {
  const router = useRouter()
  const queryClient = useQueryClient()
  const _updateUser = useServerFn(updateUser)

  const handleClick = async () => {
    await _updateUser({ data: { id: '1', name: 'John' } })
    router.invalidate()
    queryClient.invalidateQueries({ queryKey: ['users', 'updateUser', '1'] })
  }

  return <button onClick={handleClick}>Click Me</button>
}
```

Learn more: Mutations guide.

Data Loading

Data loaders fetch data during SSR and client-side rendering, and are used via `loader` functions:

```
// Example route with data loader
export const Route = createFileRoute('/example')({
  component: ExampleComponent,
  loader: async () => {
    // fetch data here
  },
})
```

Features:

- Executed on server and client
- Supports server-side logic via server functions
- Data is cached and revalidated automatically

More info: Data loading guide.

Edit on GitHub

Server Functions

What are Server Functions?

Server functions allow you to specify logic that can be invoked almost anywhere (even the client), but run **only** on the server. In fact, they are not so different from an API Route, but with a few key differences:

- They do not have a stable public URL.
- They can be called from anywhere in your application, including loaders, hooks, components, server routes, etc.

However, they are similar to regular API Routes in that:

- They have access to the request context, allowing you to read headers, set cookies, and more.
- They can access sensitive information, such as environment variables, without exposing them to the client.
- They can be used to perform any kind of server-side logic, such as fetching data from a database, sending emails, or interacting with other services.
- They can return any value, including primitives, JSON-serializable objects, and even raw Response objects.
- They can throw errors, including redirects and notFounds, which can be handled automatically by the router.

  How are server functions different from "React Server Functions"?
  TanStack Server Functions are not tied to a specific front-end framework, and can be used with any front-end framework or none at all.
  TanStack Server Functions are backed by standard HTTP requests and can be called as often as you like without suffering from serial-execution bottlenecks.

How do they work?

Server functions can be defined anywhere in your application, but must be defined at the top level of a file. They can be called throughout your application, including loaders, hooks, etc. Traditionally, this pattern is known as a Remote Procedure Call (RPC), but due to the isomorphic nature of these functions, we refer to them as server functions.

- On the server bundle, server functions logic is left alone. Nothing needed since they are already in the correct place.
- On the client, server functions will be removed; they exist only on the server. Any calls to the server function on the client will be replaced with a `fetch` request to the server to execute that function and send the response back to the client.

Server Function Middleware

Server functions can use middleware to share logic, context, common operations, prerequisites, and more. To learn more about middleware, see the Middleware guide.

Defining Server Functions

Server functions are defined with the `createServerFn` function from the `@tanstack/react-start` package. This function optionally takes an `options` argument for configuration like HTTP method and response type, and allows chaining to define the body, input validation, middleware, etc.

Example:

```
// getServerTime.ts
import { createServerFn } from '@tanstack/react-start'

export const getServerTime = createServerFn().handler(async () => {
  // Wait for 1 second
  await new Promise((resolve) => setTimeout(resolve, 1000))
  // Return the current time
  return new Date().toISOString()
})
```

Configuration Options

When creating a server function, you can provide options to customize behavior:

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

export const getData = createServerFn({
  method: 'GET', // HTTP method
  response: 'data', // Response mode
}).handler(async () => {
  // Implementation
})
```

Available Options:

method

Specifies the HTTP method:

`method?: 'GET' | 'POST'`

Defaults to `'GET'` if not specified.

response

Controls how responses are processed:

`response?: 'data' | 'full' | 'raw'`

Options:

- `'data'`: Parses JSON responses, returns just data (default).
- `'full'`: Returns response object with data, error, etc.
- `'raw'`: Returns the raw `Response` object, for streaming, headers, etc.

Where can I call server functions?

- From server-side code
- From client-side code
- From other server functions

Warning:
Server functions cannot be called from API Routes. For sharing logic between server functions and API Routes, extract the core logic into utility functions.

Accepting Parameters

Server functions accept a single parameter of various types:

- Basic JavaScript types:
  - `string`
  - `number`
  - `boolean`
  - `null`
  - `Array`
  - `Object`
- `FormData`
- `ReadableStream` (of above types)
- `Promise` (of above types)

Example:

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

export const greet = createServerFn({
  method: 'GET',
})
.validator((data: string) => data)
.handler(async (ctx) => {
  return `Hello, ${ctx.data}!`
})

// Usage:
greet({ data: 'John' })
```

Runtime Input Validation / Type Safety

You can configure validation to ensure correct input at runtime via `.validator()`. This improves safety and error messaging.

Validators accept the input data and return the processed value, which is passed to the handler.

Example with basic validation:

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

type Person = { name: string }

export const greet = createServerFn({ method: 'GET' })
  .validator((person: unknown): Person => {
    if (typeof person !== 'object' || person === null) {
      throw new Error('Person must be an object')
    }
    if ('name' in person && typeof person.name !== 'string') {
      throw new Error('Person.name must be a string')
    }
    return person as Person
  })
  .handler(async ({ data }) => {
```

```
    return `Hello, ${data.name}!`
  })
```

Using a Validation Library (e.g., Zod):

```
import { createServerFn } from '@tanstack/react-start'
import { z } from 'zod'

const PersonSchema = z.object({ name: z.string() })

export const greet = createServerFn({ method: 'GET' })
  .validator((person: unknown) => PersonSchema.parse(person))
  .handler(async (ctx) => {
    return `Hello, ${ctx.data.name}!`
  })

greet({ data: { name: 'John' } })
```

Type Safety

Since server functions cross network boundaries, validating input/output at runtime is crucial to prevent errors and security issues. The return type of `.validator()` dictates the data passed to the handler.

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

type Person = { name: string }

export const greet = createServerFn({ method: 'GET' })
  .validator((person: unknown): Person => {
    if (typeof person !== 'object' || person === null) {
      throw new Error('Person must be an object')
    }
    if ('name' in person && typeof person.name !== 'string') {
      throw new Error('Person.name must be a string')
    }
    return person as Person
  })
  .handler(async ({ data }) => {
    return `Hello, ${data.name}!`
  })

// Usage:
greet({ data: { name: 'John' } }) // OK
greet({ data: { name: 123 } }) // Error: Argument of type '{ name: number; }' is not a
```

Inference

Server functions infer their input and output types based on `.validator()` and `.handler()`. Validators can be transformed, enabling complex data transformations.

Example with Zod:

```
import { createServerFn } from '@tanstack/react-start'
import { z } from 'zod'
```

```
const transactionSchema = z.object({
  amount: z.string().transform((val) => parseInt(val, 10)),
})

const createTransaction = createServerFn()
  .validator(transactionSchema)
  .handler(({ data }) => {
    return data.amount // Returns a number
  })

createTransaction({ data: { amount: '123' } }) // Accepts string, outputs number
```

Non-Validated Inference

You can also skip validation and provide explicit types via an identity function for input/output:

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

type Person = { name: string }

export const greet = createServerFn({ method: 'GET' })
  .validator((d: Person) => d)
  .handler(async ({ data }) => {
    return `Hello, ${data.name}!`
  })

// Usage:
greet({ data: { name: 'John' } })
```

JSON Parameters

Server functions can accept JSON-serializable objects:

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

type Person = { name: string; age: number }

export const greet = createServerFn({ method: 'GET' })
  .validator((data: Person) => data)
  .handler(async ({ data }) => {
    return `Hello, ${data.name}! You are ${data.age} years old.`
  })

greet({ data: { name: 'John', age: 34 } })
```

FormData Parameters

Server functions can accept `FormData` objects, validating and parsing fields:

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

export const greetUser = createServerFn({ method: 'POST' })
  .validator((data) => {
```

```
    if (!(data instanceof FormData)) {
      throw new Error('Invalid form data')
    }
    const name = data.get('name')
    const age = data.get('age')
    if (!name || !age) {
      throw new Error('Name and age are required')
    }
    return {
      name: name.toString(),
      age: parseInt(age.toString(), 10),
    }
  })
  .handler(async ({ data: { name, age } }) => {
    return `Hello, ${name}! You are ${age} years old.`
  })

// Usage example with form submission:
function Test() {
  return (
    <form
      onSubmit={async (event) => {
        event.preventDefault()
        const formData = new FormData(event.currentTarget)
        const response = await greetUser({ data: formData })
        console.log(response)
      }}
    >
      <input name="name" />
      <input name="age" />
      <button type="submit">Submit</button>
    </form>
  )
}
```

Server Function Context

Access request context within server functions via utilities from `@tanstack/react-start/server`. For example:

```
import { createServerFn } from '@tanstack/react-start'
import { getWebRequest } from '@tanstack/react-start/server'

export const getServerTime = createServerFn({ method: 'GET' }).handler(
  async () => {
    const request = getWebRequest()
    console.log(request.method) // e.g., GET
    console.log(request.headers.get('User-Agent'))
  },
)
```

Other context utilities include:

- `getHeaders()`
- `getHeader('Header-Name')`

- Setting headers, cookies, response status, etc.

Returning Values

Server functions can return:

- Primitives
- JSON-serializable objects
- Redirect or notFound errors (can be thrown)
- Raw Response objects

Returning primitives and JSON:

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

export const getServerTime = createServerFn({ method: 'GET' }).handler(async () => {
  return new Date().toISOString()
})

export const getServerData = createServerFn({ method: 'GET' }).handler(async () => {
  return { message: 'Hello, World!' }
})
```

Respond with custom headers:

```
import { createServerFn } from '@tanstack/react-start'
import { setHeader } from '@tanstack/react-start/server'

export const getServerTime = createServerFn({ method: 'GET' }).handler(async () => {
  setHeader('X-Custom-Header', 'value')
  return new Date().toISOString()
})
```

Respond with custom status code:

```
import { createServerFn } from '@tanstack/react-start'
import { setResponseStatus } from '@tanstack/react-start/server'

export const getServerTime = createServerFn({ method: 'GET' }).handler(async () => {
  setResponseStatus(201)
  return new Date().toISOString()
})
```

Return raw Response object:

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

export const getServerTime = createServerFn({
  method: 'GET',
  response: 'raw',
}).handler(async () => {
  return fetch('https://example.com/time.txt')
})
```

This option is useful for:

- Streaming APIs
- Server-sent events
- Long-polling
- Custom content types/binary data

Throwing Errors

Aside from redirects and notFound, server functions can throw custom errors:

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

export const doStuff = createServerFn({ method: 'GET' }).handler(async () => {
  throw new Error('Something went wrong!')
})

// Usage:
try {
  await doStuff()
} catch (error) {
  console.error(error) // logs the error object
}
```

Cancellation

Server functions support cancellation with `AbortSignal`:

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

export const abortableServerFn = createServerFn().handler(async ({ signal }) => {
  return new Promise((resolve, reject) => {
    if (signal.aborted) {
      return reject(new Error('Aborted before start'))
    }
    const timerId = setTimeout(() => {
      resolve('Result after delay')
    }, 1000)
    const onAbort = () => {
      clearTimeout(timerId)
      reject(new Error('Aborted'))
    }
    signal.addEventListener('abort', onAbort)
  })
})
```

Usage example:

```
function Test() {
  const controller = new AbortController()
  const promise = abortableServerFn({ signal: controller.signal })
  setTimeout(() => controller.abort(), 500)
  try {
    const result = await promise
    console.log(result) // never reached if aborted
```

```
  } catch (error) {
    console.error(error) // "Aborted"
  }
}
```

Calling server functions from within route lifecycles

You can call server functions in route loaders, beforeLoad, or other router-controlled APIs, which handle errors, redirects, etc.:

```
import { getServerTime } from './getServerTime'

export const Route = createFileRoute('/time')({
  loader: async () => {
    const time = await getServerTime()
    return { time }
  },
})
```

Calling server functions from hooks and components

Use the `useServerFn` hook from `@tanstack/react-start`:

```
import { useServerFn } from '@tanstack/react-start'
import { useQuery } from '@tanstack/react-query'
import { getServerTime } from './getServerTime'

export function Time() {
  const getTime = useServerFn(getServerTime)
  const timeQuery = useQuery({
    queryKey: 'time',
    queryFn: () => getTime(),
  })
}
```

Calling server functions anywhere else

Redirects and notFound errors thrown will be automatically handled only when called from:

- Route lifecycles
- Components using `useServerFn`

Other locations require manual error handling.

Redirects

Server functions can throw a `redirect` error to redirect the user:

```
import { redirect } from '@tanstack/react-router'
import { createServerFn } from '@tanstack/react-start'
```

```
export const doStuff = createServerFn({ method: 'GET' }).handler(async () => {
  throw redirect({ to: '/' })
})
```

Options include:

- Path Params, Search Params, Hash
- `status` code (e.g., 301, 302)

External redirect example:

```
import { redirect } from '@tanstack/react-router'
import { createServerFn } from '@tanstack/react-start'

export const auth = createServerFn({ method: 'GET' }).handler(async () => {
  throw redirect({ href: 'https://authprovider.com/login' })
})
```

> ⚠️ Do not use `sendRedirect` from `@tanstack/react-start/server`, as it sends a full page redirect, not suitable for SPA navigation.

Redirect Headers

Set custom headers on redirect:

```
import { redirect } from '@tanstack/react-router'
import { createServerFn } from '@tanstack/react-start'

export const doStuff = createServerFn({ method: 'GET' }).handler(async () => {
  throw redirect({
    to: '/',
    headers: {
      'X-Custom-Header': 'value',
    },
  })
})
```

Not Found

Throw `notFound()` inside route loaders or beforeLoad to signal resource not found:

```
import { notFound } from '@tanstack/react-router'
import { createServerFn } from '@tanstack/react-start'

const getStuff = createServerFn({ method: 'GET' }).handler(async () => {
  if (Math.random() < 0.5) throw notFound()
  return { stuff: 'stuff' }
})

// Usage in route:
export const Route = createFileRoute('/stuff')({
  loader: async () => {
    const stuff = await getStuff()
    return { stuff }
```

```
  },
})
```

Handling Errors

Errors other than redirect or notFound are serialized and sent as JSON with a 500 status code. Handle them gracefully:

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

export const doStuff = createServerFn({ method: 'GET' }).handler(async () => {
  undefined.foo() // throws error
})

try {
  await doStuff()
} catch (error) {
  // handle error
}
```

No-JS Server Functions

Without JavaScript, server functions are invoked via `<form>` submission with the `action` attribute pointing to the server function URL.

Example:

```
const myFn = createServerFn({ method: 'POST' }).validator((formData) => {
  if (!(formData instanceof FormData)) throw new Error('Invalid form data')
  const name = formData.get('name')
  if (!name) throw new Error('Name required')
  return name
})
console.info(myFn.url)

function MyComponent() {
  return (
    <form action={myFn.url} method="POST">
      <input name="name" defaultValue="John" />
      <button type="submit">Submit</button>
    </form>
  )
}
```

Passing arguments via form inputs:

```
<form
  action={myFn.url}
  method="POST"
  encType="multipart/form-data"
>
  <input name="age" defaultValue="34" />
  <button type="submit">Click me!</button>
</form>
```

Server Function Return Value (non-JS)

When JavaScript is disabled, the server function can respond with a redirect or page refresh, e.g., via HTTP response with `Location`.

```
// Example of reloading page:
return new Response('ok', { status: 301, headers: { Location: '/' } })
```

Summary

Server functions enable powerful server-side logic accessible from anywhere, with flexible validation, streaming, and error handling. They are designed to work seamlessly in both client and server environments, supporting a wide range of use cases, including static generation and fallback behaviors when JavaScript is disabled.

Edit on GitHub

Static Server Functions | TanStack Start React Docs

On this page

- What are Static Server Functions?
- Customizing the Server Functions Static Cache

What are Static Server Functions?

Static server functions are server functions that are executed at build time and cached as static assets when using prerendering/static-generation. They can be set to "static" mode by passing the `type: 'static'` option to `createServerFn`:

```
const myServerFn = createServerFn({ type: 'static' }).handler(async () => {
  return 'Hello, world!'
})
```

This pattern goes as follows:

- **Build-time**
  - During build-time prerendering, a server function with `type: 'static'` is executed.
  - The result is cached with your build output as a static JSON file under a derived key (function ID + params/payload hash).
  - The result is returned as normal during prerendering/static-generation and used to prerender the page.
- **Runtime**
  - Initially, the prerendered page's HTML is served and the server function data is embedded in the HTML.
  - When the client mounts, the embedded server function data is hydrated.
  - For future client-side invocations, the server function is replaced with a fetch call to the static JSON file.

Customizing the Server Functions Static Cache

By default, the static server function cache implementation stores and retrieves static data in the build output directory via Node's `fs` module and fetches the data at runtime using a `fetch` call to the same static file.

This interface can be customized by importing and calling the `createServerFnStaticCache` function to create a custom cache implementation and then calling `setServerFnStaticCache` to set it:

```
import {
  createServerFnStaticCache,
  setServerFnStaticCache,
} from '@tanstack/react-start/client'

const myCustomStaticCache = createServerFnStaticCache({
```

```
  setItem: async (ctx, data) => {
    // Store the static data in your custom cache
  },
  getItem: async (ctx) => {
    // Retrieve the static data from your custom cache
  },
  fetchItem: async (ctx) => {
    // During runtime, fetch the static data from your custom cache
  },
})

setServerFnStaticCache(myCustomStaticCache)
```

Edit on GitHub

On this page

- What are Static Server Functions?
- Customizing the Server Functions Static Cache

Middleware

What is Server Function Middleware?

Middleware allows you to customize the behavior of server functions created with `createServerFn` with things like shared validation, context, and much more. Middleware can even depend on other middleware to create a chain of operations that are executed hierarchically and in order.

What kinds of things can I do with Middleware in my Server Functions?

- **Authentication**: Verify a user's identity before executing a server function.
- **Authorization**: Check if a user has the necessary permissions to execute a server function.
- **Logging**: Log requests, responses, and errors.
- **Observability**: Collect metrics, traces, and logs.
- **Provide Context**: Attach data to the request object for use in other middleware or server functions.
- **Error Handling**: Handle errors in a consistent way.
- And many more! The possibilities are up to you!

Defining Middleware for Server Functions

Middleware is defined using the `createMiddleware` function. This function returns a **Middleware** object that can be used to continue customizing the middleware with methods like `middleware`, `validator`, `server`, and `client`.

```
import { createMiddleware } from '@tanstack/react-start'

const loggingMiddleware = createMiddleware({ type: 'function' }).server(
  async ({ next, data }) => {
    console.log('Request received:', data)
    const result = await next()
    console.log('Response processed:', result)
    return result
  },
)
```

Using Middleware in Your Server Functions

Once you've defined your middleware, you can use it in combination with the `createServerFn` function to customize the behavior of your server functions.

```
import { createServerFn } from '@tanstack/react-start'
import { loggingMiddleware } from './middleware'

const fn = createServerFn()
  .middleware([loggingMiddleware])
  .handler(async () => {
    // ...
  })
```

Middleware Methods

Several methods are available to customize the middleware. If you are (hopefully) using TypeScript, the order of these methods is enforced by the type system to ensure maximum inference and type safety.

- **middleware**: Add a middleware to the chain.
- **validator**: Modify the data object before it is passed to this middleware and any nested middleware.
- **server**: Define server-side logic that the middleware will execute before any nested middleware and ultimately a server function, and also provide the result to the next middleware.
- **client**: Define client-side logic that the middleware will execute before any nested middleware and ultimately the client-side RPC function (or the server-side function), and also provide the result to the next middleware.

The `middleware` method

The `middleware` method is used to dependency middleware to the chain that will be executed **before** the current middleware. Just call the `middleware` method with an array of middleware objects.

```
import { createMiddleware } from '@tanstack/react-start'

const loggingMiddleware = createMiddleware({ type: 'function' }).middleware([
  authMiddleware,
  loggingMiddleware,
])
```

Type-safe context and payload validation are also inherited from parent middlewares!

The `validator` method

The `validator` method is used to modify the data object before it is passed to this middleware, nested middleware, and ultimately the server function. This method should receive a function that takes the data object and returns a validated (and optionally modified) data object. It's common to use a validation library like `zod` to do this. Here's an example:

```
import { createMiddleware } from '@tanstack/react-start'
import { zodValidator } from '@tanstack/zod-adapter'
import { z } from 'zod'

const mySchema = z.object({
  workspaceId: z.string(),
})

const workspaceMiddleware = createMiddleware({ type: 'function' })
  .validator(zodValidator(mySchema))
  .server(({ next, data }) => {
    console.log('Workspace ID:', data.workspaceId)
    return next()
  })
```

The `server` method

The `server` method is used to define **server-side** logic that the middleware will execute both before and after any nested middleware and ultimately a server function. This method receives an object with the following properties:

- **next**: A function that, when called, will execute the next middleware in the chain.
- **data**: The data object that was passed to the server function.
- **context**: An object that stores data from parent middleware. It can be extended with additional data that will be passed to child middleware.

Returning the required result from `next`

The `next` function is used to execute the next middleware in the chain. **You must await and return (or return directly) the result of the `next` function** for the chain to continue executing.

```
import { createMiddleware } from '@tanstack/react-start'

const loggingMiddleware = createMiddleware({ type: 'function' }).server(
  async ({ next }) => {
    console.log('Request received')
    const result = await next()
    console.log('Response processed')
    return result
  },
)
```

Providing context to the next middleware via `next`

The `next` function can be optionally called with an object that has a `context` property with an object value. Whatever properties you pass to this `context` value will be merged into the parent `context` and provided to the next middleware.

```
import { createMiddleware } from '@tanstack/react-start'

const awesomeMiddleware = createMiddleware({ type: 'function' }).server(
  ({ next }) => {
    return next({
      context: {
        isAwesome: Math.random() > 0.5,
      },
    })
  },
)

const loggingMiddleware = createMiddleware({ type: 'function' })
  .middleware([awesomeMiddleware])
  .server(async ({ next, context }) => {
    console.log('Is awesome?', context.isAwesome)
    return next()
  })
```

Client-Side Logic

Despite server functions being mostly server-side bound operations, there is still plenty of client-side logic surrounding the outgoing RPC request from the client. This means that we can also define client-side logic in middleware that will execute on the client side around any nested middleware and ultimately the RPC function and its response to the client.

Client-side Payload Validation

By default, middleware validation is only performed on the server to keep the client bundle size small. However, you may also choose to validate data on the client side by passing the `validateClient: true` option to the `createMiddleware` function. This will cause the data to be validated on the client side before being sent to the server, potentially saving a round trip.

> **Why can't I pass a different validation schema for the client?**
> The client-side validation schema is derived from the server-side schema. This is because the client-side validation schema is used to validate the data before it is sent to the server. If the client-side schema were different from the server-side schema, the server would receive data that it did not expect, which could lead to unexpected behavior.

```
import { createMiddleware } from '@tanstack/react-start'
import { zodValidator } from '@tanstack/zod-adapter'
import { z } from 'zod'

const workspaceMiddleware = createMiddleware({ validateClient: true })
  .validator(zodValidator(mySchema))
  .server(({ next, data }) => {
    console.log('Workspace ID:', data.workspaceId)
    return next()
  })
```

The `client` method

Client middleware logic is defined using the `client` method on a `Middleware` object. This method is used to define client-side logic that the middleware will execute both before and after any nested middleware and ultimately the client-side RPC function (or the server-side function if you're doing SSR or calling this function from another server function).

Client-side middleware logic shares much of the same API as logic created with the `server` method, but it is executed on the client side. This includes:

- Requiring the `next` function to be called to continue the chain.
- The ability to provide context to the next client middleware via the `next` function.
- The ability to modify the data object before it is passed to the next client middleware.

Similar to the `server` function, it also receives an object with the following properties:

- **next**: A function that, when called, will execute the next client middleware in the chain.
- **data**: The data object that was passed to the client function.
- **context**: An object that stores data from parent middleware. It can be extended with additional data that will be passed to child middleware.

```
const loggingMiddleware = createMiddleware({ type: 'function' }).client(
  async ({ next }) => {
    console.log('Request sent')
    const result = await next()
    console.log('Response received')
    return result
  },
)
```

Sending client context to the server

Client context is NOT sent to the server by default since this could end up unintentionally sending large payloads to the server. If you need to send client context to the server, you must call the `next` function with

a `sendContext` property and object to transmit any data to the server. Any properties passed to `sendContext` will be merged, serialized, and sent to the server along with the data and will be available on the normal context object of any nested server middleware.

```
const requestLogger = createMiddleware({ type: 'function' })
  .client(async ({ next, context }) => {
    return next({
      sendContext: {
        // Send the workspace ID to the server
        workspaceId: context.workspaceId,
      },
    })
  })
  .server(async ({ next, data, context }) => {
    // Woah! We have the workspace ID from the client!
    console.log('Workspace ID:', context.workspaceId)
    return next()
  })
```

Client-Sent Context Security

You may have noticed that while client-sent context is type-safe, it is not required to be validated at runtime. If you pass dynamic user-generated data via context, that could pose a security concern, so **if you are sending dynamic data from the client to the server via context, you should validate it in the server-side middleware before using it.** Here's an example:

```
import { zodValidator } from '@tanstack/zod-adapter'
import { z } from 'zod'

const requestLogger = createMiddleware({ type: 'function' })
  .client(async ({ next, context }) => {
    return next({
      sendContext: {
        workspaceId: context.workspaceId,
      },
    })
  })
  .server(async ({ next, data, context }) => {
    // Validate the workspace ID before using it
    const workspaceId = zodValidator(z.number()).parse(context.workspaceId)
    console.log('Workspace ID:', workspaceId)
    return next()
  })
```

Sending server context to the client

Similar to sending client context to the server, you can also send server context to the client by calling the `next` function with a `sendContext` property and object to transmit any data to the client. Any properties passed to `sendContext` will be merged, serialized, and sent to the client along with the response and will be available on the normal context object of any nested client middleware. The returned object of calling `next` in `client` contains the context sent from server to the client and is type-safe. Middleware is able to infer the context sent from the server to the client from previous middleware chained from the `middleware` function.

Warning
The return type of `next` in `client` can only be inferred from middleware known in the current middleware chain. Therefore, the most accurate return type of `next` is in middleware at the end of the middleware chain.

```
const serverTimer = createMiddleware({ type: 'function' }).server(
  async ({ next }) => {
    return next({
      sendContext: {
        // Send the current time to the client
        timeFromServer: new Date(),
      },
    })
  },
)

const requestLogger = createMiddleware({ type: 'function' })
  .middleware([serverTimer])
  .client(async ({ next }) => {
    const result = await next()
    // Woah! We have the time from the server!
    console.log('Time from the server:', result.context.timeFromServer)

    return result
  })
```

Reading/Modifying the Server Response

Middleware that uses the `server` method executes in the same context as server functions, so you can follow the exact same Server Function Context Utilities to read and modify anything about the request headers, status codes, etc.

Modifying the Client Request

Middleware that uses the `client` method executes in a **completely different client-side context** than server functions, so you can't use the same utilities to read and modify the request. However, you can still modify the request by returning additional properties when calling the `next` function. Currently supported properties are:

- `headers`: An object containing headers to be added to the request.

Here's an example of adding an `Authorization` header to any request using this middleware:

```
import { getToken } from 'my-auth-library'

const authMiddleware = createMiddleware({ type: 'function' }).client(
  async ({ next }) => {
    return next({
      headers: {
        Authorization: `Bearer ${getToken()}`,
      },
    })
  },
)
```

Using Middleware

Middleware can be used in two different ways:

- **Global Middleware**: Middleware that should be executed for every request.
- **Server Function Middleware**: Middleware that should be executed for a specific server function.

Global Middleware

Global middleware runs automatically for every server function in your application. This is useful for functionality like authentication, logging, and monitoring that should apply to all requests.

To use global middleware, create a `global-middleware.ts` file in your project (typically at `app/global-middleware.ts`). This file runs in both client and server environments and is where you register global middleware.

```
// app/global-middleware.ts
import { registerGlobalMiddleware } from '@tanstack/react-start'
import { authMiddleware } from './middleware'

registerGlobalMiddleware({
  middleware: [authMiddleware],
})
```

Global Middleware Type Safety

Global middleware types are inherently **detached** from server functions themselves. This means that if a global middleware supplies additional context to server functions or other server function-specific middleware, the types will not be automatically passed through to the server function or other server function-specific middleware.

```
// app/global-middleware.ts
registerGlobalMiddleware({
  middleware: [authMiddleware],
})
```

```
// authMiddleware.ts
const authMiddleware = createMiddleware({ type: 'function' }).server(
  ({ next, context }) => {
    console.log(context.user) // <-- This will not be typed!
    // ...
  },
)
```

To solve this, add the global middleware you are trying to reference to the server function's middleware array. **The global middleware will be deduped to a single entry (the global instance), and your server function will receive the correct types.**

```
import { authMiddleware } from './authMiddleware'

const fn = createServerFn()
  .middleware([authMiddleware])
  .handler(async ({ context }) => {
    console.log(context.user)
    // ...
  })
```

Middleware Execution Order

Middleware is executed dependency-first, starting with global middleware, followed by server function middleware. The following example will log the following in this order:

- `globalMiddleware1`
- `globalMiddleware2`
- `a`
- `b`
- `c`
- `d`

```
const globalMiddleware1 = createMiddleware({ type: 'function' }).server(
  async ({ next }) => {
    console.log('globalMiddleware1')
    return next()
  },
)

const globalMiddleware2 = createMiddleware({ type: 'function' }).server(
  async ({ next }) => {
    console.log('globalMiddleware2')
    return next()
  },
)

registerGlobalMiddleware({
  middleware: [globalMiddleware1, globalMiddleware2],
})

const a = createMiddleware({ type: 'function' }).server(async ({ next }) => {
  console.log('a')
  return next()
})

const b = createMiddleware({ type: 'function' })
  .middleware([a])
  .server(async ({ next }) => {
    console.log('b')
    return next()
  })

const c = createMiddleware({ type: 'function' })
  .middleware()
  .server(async ({ next }) => {
    console.log('c')
    return next()
  })

const d = createMiddleware({ type: 'function' })
  .middleware([b, c])
  .server(async () => {
    console.log('d')
  })

const fn = createServerFn()
```

```
  .middleware([d])
  .server(async () => {
    console.log('fn')
  })
```

Environment Tree Shaking

Middleware functionality is tree-shaken based on the environment for each bundle produced.

- On the server, nothing is tree-shaken, so all code used in middleware will be included in the server bundle.
- On the client, all server-specific code is removed from the client bundle. This means any code used in the `server` method is always removed from the client bundle. If `validateClient` is set to `true`, the client-side validation code will be included in the client bundle, otherwise `data` validation code will also be removed.

Edit on GitHub

Server Routes

Server Routes and App Routes

Because server routes can be defined in the same directory as your app routes, you can even use the same file for both!

```
// routes/hello.tsx

export const ServerRoute = createServerFileRoute().methods({
  POST: async ({ request }) => {
    const body = await request.json()
    return new Response(JSON.stringify({ message: `Hello, ${body.name}!` }))
  },
})

export const Route = createFileRoute('/hello')({
  component: HelloComponent,
})

function HelloComponent() {
  const [reply, setReply] = useState('')

  return (
    <div>
      <button
        onClick={() => {
          fetch('/hello', {
            method: 'POST',
            headers: {
              'Content-Type': 'application/json',
            },
            body: JSON.stringify({ name: 'Tanner' }),
          })
            .then((res) => res.json())
            .then((data) => setReply(data.message))
        }}
      >
        Say Hello
      </button>
    </div>
  )
}
```

File Route Conventions

Server routes in TanStack Start follow the same file-based routing conventions as TanStack Router. Examples:

- `/routes/users.ts` creates an API route at `/users`
- `/routes/users.index.ts` also creates `/users` (errors if duplicate methods)
- `/routes/users/$id.ts` creates `/users/$id`
- `/routes/users/$id/posts.ts` creates `/users/$id/posts`

- `/routes/users.$id.posts.ts` creates `/users/$id/posts`
- `/routes/api/file/$.ts` creates `/api/file/$`
- `/routes/my-script[.]js.ts` creates `/my-script.js`

Unique Route Paths

Each route can only have one handler file. For example, `routes/users.ts` (path `/users`) can't have other files like:

- `/routes/users.index.ts`
- `/routes/users.ts`
- `/routes/users/index.ts`

—they all resolve to the same route and cause errors.

Escaped Matching

Server routes can match escaped characters. For example, a file named:

`routes/users[.]json.ts`

creates an API route at `/users[.]json`.

Pathless Layout Routes and Break-out Routes

Supported for similar middleware functionality:

- Pathless layout routes add middleware to a group
- Break-out routes "break out" of parent middleware

Nested Directories vs File-names

File naming is flexible—you can organize by directories or filenames as makes sense. Read more in the TanStack Router File-based Routing Guide.

Handling Server Route Requests

Handled via Start's `createStartHandler` in your `server.ts`.

```
// server.ts
import {
  createStartHandler,
  defaultStreamHandler,
} from '@tanstack/react-start/server'
import { createRouter } from './router'

export default createStartHandler({
  createRouter,
})(defaultStreamHandler)
```

The handler matches requests to routes and runs middleware/handlers.

Custom server handler example:

```
// server.ts
import { createStartHandler } from '@tanstack/react-start/server'

export default defineHandler((event) => {
  const startHandler = createStartHandler({
    createRouter,
  })(defaultStreamHandler)

  return startHandler(event)
})
```

Defining a Server Route

Create by exporting a `ServerRoute` from a route file, using `createServerFileRoute()`. You can:

- Add route middleware
- Define handlers for HTTP methods

```
// routes/hello.ts
export const ServerRoute = createServerFileRoute().methods({
  GET: async ({ request }) => {
    return new Response('Hello, World! from ' + request.url)
  },
})
```

Defining a Server Route Handler

Two options:

1. **Direct handler function:**

```
// routes/hello.ts
export const ServerRoute = createServerFileRoute().methods({
  GET: async ({ request }) => {
    return new Response('Hello, World! from ' + request.url)
  },
})
```

2. **Using `handler()` method for advanced cases:**

```
// routes/hello.ts
export const ServerRoute = createServerFileRoute().methods((api) => ({
  GET: api.middleware([loggerMiddleware]).handler(async ({ request }) => {
    return new Response('Hello, World! from ' + request.url)
  }),
}))
```

Handler Context

Each HTTP method handler receives an object with:

- `request`: The incoming Request object (MDN Docs)
- `params`: Dynamic route parameters, e.g., for `/users/$id`, params = `{ id: '123' }`
- `context`: Shared data between middleware

Returns `Response`, `Promise<Response>`, or helpers like `json()`.

Dynamic Path Params

Example for route `/users/$id`:

```
// routes/users/$id.ts
export const ServerRoute = createServerFileRoute().methods({
  GET: async ({ params }) => {
    const { id } = params
    return new Response(`User ID: ${id}`)
  },
})
```

Visit `/users/123` => response: `User ID: 123`

Multiple parameters:

```
// routes/users/$id/posts/$postId.ts
export const ServerRoute = createServerFileRoute().methods({
  GET: async ({ params }) => {
    const { id, postId } = params
    return new Response(`User ID: ${id}, Post ID: ${postId}`)
  },
})
```

Visit `/users/123/posts/456` for `User ID: 123, Post ID: 456`.

Wildcard/Splat Param

Supported by appending `$` at the end of the path:

```
// routes/file/$.ts
export const ServerRoute = createServerFileRoute().methods({
  GET: async ({ params }) => {
    const { _splat } = params
    return new Response(`File: ${_splat}`)
  },
})
```

Visit `/file/hello.txt` => `File: hello.txt`

Handling requests with a body

Add handlers for methods like `POST`, `PUT`, `PATCH`, `DELETE`. Access body with `request.json()`:

```
// routes/hello.ts
export const ServerRoute = createServerFileRoute().methods({
  POST: async ({ request }) => {
    const body = await request.json()
    return new Response(`Hello, ${body.name}!`)
  },
})
```

Post example with JSON body `{ "name": "Tanner" }` returns `"Hello, Tanner!"`.

Responding with JSON

Pattern:

```
// routes/hello.ts
export const ServerRoute = createServerFileRoute().methods({
  GET: async ({ request }) => {
    return new Response(JSON.stringify({ message: 'Hello, World!' }), {
      headers: {
        'Content-Type': 'application/json',
      },
    })
  },
})
```

Using the `json()` helper function

Easier:

```
// routes/hello.ts
import { json } from '@tanstack/react-start'

export const ServerRoute = createServerFileRoute().methods({
  GET: async ({ request }) => {
    return json({ message: 'Hello, World!' })
  },
})
```

Responding with a status code

Options:

1. Pass `status` in Response constructor:

```
// routes/hello.ts
export const ServerRoute = createServerFileRoute().methods({
  GET: async ({ request, params }) => {
    const user = await findUser(params.id)
    if (!user) {
      return new Response('User not found', { status: 404 })
    }
    return json(user)
  },
})
```

2. Use `setResponseStatus()` helper:

```
// routes/hello.ts
import { setResponseStatus } from '@tanstack/react-start/server'

export const ServerRoute = createServerFileRoute().methods({
  GET: async ({ request, params }) => {
    const user = await findUser(params.id)
    if (!user) {
      setResponseStatus(404)
      return new Response('User not found')
    }
    return json(user)
```

```
  },
})
```

Setting headers in the response

Option 1: Pass headers object:

```
// routes/hello.ts
return new Response('Hello, World!', {
  headers: {
    'Content-Type': 'text/plain',
  },
})
```

Option 2: Use `setHeaders()` helper:

```
// routes/hello.ts
import { setHeaders } from '@tanstack/react-start/server'

export const ServerRoute = createServerFileRoute().methods({
  GET: async ({ request }) => {
    setHeaders({ 'Content-Type': 'text/plain' })
    return new Response('Hello, World!')
  },
})
```

Edit on GitHub

On this page

- Server Routes and App Routes
- File Route Conventions
- Unique Route Paths
- Escaped Matching
- Pathless Layout Routes and Break-out Routes
- Nested Directories vs File-names
- Handling Server Route Requests
- Defining a Server Route
- Defining a Server Route Handler
- Providing a handler function directly to the method
- Providing a handler function via the method builder object
- Handler Context
- Dynamic Path Params
- Wildcard/Splat Param
- Handling requests with a body
- Responding with JSON
- Using the `json()` helper function
- Responding with a status code
- Setting headers in the response

Additional content available via links and navigation.

SPA mode

Overview

React

On this page

- What the heck is SPA mode?
- Why use Start without SSR?
- Benefits of SPA mode
- Caveats of SPA mode
- How does it work?
- Configuring SPA mode
- Use Necessary Redirects
- Basic Redirects Example
- Allowing Server Functions and Server Routes
- Shell Mask Path
- Prerendering Options
- Customized rendering in SPA mode
- Dynamic Data in your Shell

What the heck is SPA mode?

For applications that do not require SSR for either SEO, crawlers, or performance reasons, it may be desirable to ship static HTML to your users containing the "shell" of your application (or even prerendered HTML for specific routes) that contain the necessary *html*, *head*, and *body* tags to bootstrap your application only on the client.

Why use Start without SSR?

No SSR doesn't mean giving up server-side features!
SPA modes actually pair very nicely with server-side features like server functions and/or server routes or even other external APIs. It *simply* means that the initial document will not contain the fully rendered HTML of your application until it has been rendered on the client using JavaScript.

Benefits of SPA mode

- **Easier to deploy** - A CDN that can serve static assets is all you need.
- **Cheaper** to host - CDNs are cheap compared to Lambda functions or long-running processes.
- **Client-side Only is simpler** - No SSR means less to go wrong with hydration, rendering, and routing.

Caveats of SPA mode

- **Slower time to full content** - Time to full content is longer since all JS must download and execute before anything below the shell can be rendered.

- **Less SEO friendly** - Robots, crawlers and link unfurlers *may* have a harder time indexing your application unless they are configured to execute JS and your application can render within a reasonable amount of time.

How does it work?

After enabling the SPA mode, running a Start build will have an additional prerendering step afterwards to generate the shell. This is done by:

- **Prerendering** your application's **root route only**
- Where your application would normally render your matched routes, your router's configured **pending fallback component is rendered instead.**
- The resulting HTML is stored to a static HTML page called **/_shell.html** (configurable)
- Default rewrites are configured to redirect all 404 requests to the SPA mode shell

Note
Other routes may also be prerendered and it is recommended to prerender as much as you can in SPA mode, but this is not required for SPA mode to work.

Configuring SPA mode

To configure SPA mode, add the following options to your Start plugin:

```
// vite.config.ts
export default defineConfig({
  plugins: [
    TanStackStart({
      spa: {
        enabled: true,
      },
    }),
  ],
})
```

Use Necessary Redirects

Deploying a purely client-side SPA to a host or CDN often requires the use of redirects to ensure URLs are properly rewritten to the SPA shell. The priorities are:

1. Serve static assets if they exist (e.g., `/about.html`)
2. Optionally route specific subpaths to dynamic server handlers (e.g., `/api/**`)
3. Rewrite all 404 requests to the SPA shell (e.g., `/shell.html`)

Basic Redirects Example

Using Netlify's `_redirects` file:

```
# Catch all other 404 requests and rewrite to the SPA shell
/* /_shell.html 200
```

Allowing Server Functions and Server Routes

Example `_redirects`:

```
# Route requests to server functions
/serverFn/* /_serverFn/:splat 200

# Route requests to API routes
/api/* /api/:splat 200

# Catch all other requests to shell
/* /_shell.html 200
```

Shell Mask Path

The default pathname used to generate the SPA shell is `/`. This is called the **shell mask path**. Since matched routes are not included, it is mostly irrelevant, but it is configurable.

```
// vite.config.ts
export default defineConfig({
  plugins: [
    tanstackStart({
      spa: {
        maskPath: '/app',
      },
    }),
  ],
})
```

Note:
It's recommended to keep the default value `/`.

Prerendering Options

The prerender option configures the prerender behavior of the SPA shell, accepting options like in the prerendering guide.

Default options:

- `outputPath`: `/_shell.html`
- `crawlLinks`: `false`
- `retryCount`: `0`

Override defaults:

```
// vite.config.ts
export default defineConfig({
  plugins: [
    TanStackStart({
      spa: {
        prerender: {
          outputPath: '/custom-shell',
          crawlLinks: true,
          retryCount: 3,
        },
      },
    }),
  ],
})
```

Customized rendering in SPA mode

You can customize the HTML output of the SPA shell to:

- Provide generic head tags for SPA routes
- Use a custom pending fallback component
- Change shell's HTML, CSS, and JS

Use the `isShell` boolean on the router:

```
// src/routes/root.tsx
export default function Root() {
  const isShell = useRouter().isShell

  if (isShell) console.log('Rendering the shell!')
}
```

Note: After hydration, the router navigates to the first route and `isShell` becomes false, which may produce flashes of unstyled content if not handled properly.

Dynamic Data in your Shell

The shell is prerendered using SSR, so any loaders or server functions on the **Root Route** run during prerender and data is included.

Example:

```
// src/routes/__root.tsx
export const RootRoute = createRootRoute({
  loader: async () => {
    return {
      name: 'Tanner',
    }
  },
  component: Root,
})

export default function Root() {
  const { name } = useLoaderData()

  return (
    <html>
      <body>
        <h1>Hello, {name}!</h1>
        <Outlet />
      </body>
    </html>
  )
}
```

Edit on GitHub

Edit this page on GitHub

Hosting

Introduction

Hosting is the process of deploying your application to the internet so that users can access it. This is a critical part of any web development project, ensuring your application is available to the world. TanStack Start is built on Vite, a powerful dev/build platform that allows us to make it possible to deploy your application to any hosting provider.

What should I use?

TanStack Start is **designed to work with any hosting provider**, so if you already have a hosting provider in mind, you can deploy your application there using the full-stack APIs provided by TanStack Start.

However, since hosting is one of the most crucial aspects of your application's performance, reliability, and scalability, we highly recommend using our official hosting partner Netlify.

What is Netlify?

Netlify is a leading hosting platform that provides a fast, secure, and reliable environment for deploying your web applications. With Netlify, you can deploy your TanStack Start application in just a few clicks and benefit from features like a global edge network, automatic scaling, and seamless integrations with GitHub and GitLab. Netlify is designed to make your development process as smooth as possible, from local development to production deployment.

Learn more

- To learn more about Netlify, visit the Netlify website
- To sign up, visit the Netlify dashboard

Deployment

Warning

The page is still a work in progress. We'll keep updating this page with guides on deployment to different hosting providers soon!

When deploying a TanStack Start application, the `target` value in the TanStack Start Vite plugin inside `vite.config.ts` determines the deployment target. The deployment target can be set to one of the following:

- netlify: Deploy to Netlify
- vercel: Deploy to Vercel
- cloudflare-pages: Deploy to Cloudflare Pages
- node-server: Deploy to a Node.js server
- bun: Deploy to a Bun server

... and more to come!

Once you've chosen a deployment target, follow the guidelines below to deploy your application to your preferred hosting provider.

Netlify

Set the `target` value to `'netlify'` in the TanStack Start Vite plugin inside `vite.config.ts`:

```
// vite.config.ts
import { tanstackStart } from '@tanstack/react-start/plugin/vite'
import { defineConfig } from 'vite'

export default defineConfig({
  plugins: [tanstackStart({ target: 'netlify' })],
})
```

Deploy your application to Netlify using their one-click deployment process, and you're all set!

Vercel

Set the `target` value to `'vercel'` in the TanStack Start Vite plugin inside `vite.config.ts`:

```
// vite.config.ts
import { tanstackStart } from '@tanstack/react-start/plugin/vite'
import { defineConfig } from 'vite'

export default defineConfig({
  plugins: [tanstackStart({ target: 'vercel' })],
})
```

Deploy to Vercel with their one-click process.

Cloudflare Pages

Deploying to Cloudflare Pages requires a few extra steps:

1. Install `unenv`

```
npm install unenv
```

2. Update `app.config.ts`

Set `server.preset` to `'cloudflare-pages'` and `server.unenv` to `cloudflare` from `unenv`:

```
// app.config.ts
import { defineConfig } from '@tanstack/react-start/config'
import { cloudflare } from 'unenv'

export default defineConfig({
  server: {
    preset: 'cloudflare-pages',
    unenv: cloudflare,
  },
})
```

3. Add `wrangler.toml` file

```
# wrangler.toml
name = "your-cloudflare-project-name"
pages_build_output_dir = "./dist"
compatibility_flags = ["nodejs_compat"]
compatibility_date = "2024-11-13"
```

Follow the deployment instructions for Cloudflare Pages.

Node.js

Set `server.preset` to `'node-server'` in `app.config.ts`:

```
// app.config.ts
import { defineConfig } from '@tanstack/react-start/config'

export default defineConfig({
  server: {
    preset: 'node-server',
  },
})

// Or use the `--preset` flag during build:
// npm run build --preset node-server
```

Build and run your application:

```
npm run build
node .output/server/index.mjs
```

Bun

Note: Bun deployment guidelines work only with React 19; for React 18, see Node.js docs.

Upgrade `react` and `react-dom` to v19:

```
npm install react@rc react-dom@rc
```

Set `server.preset` to `'bun'`:

```
// app.config.ts
import { defineConfig } from '@tanstack/react-start/config'

export default defineConfig({
  server: {
    preset: 'bun',
  },
})

// Or with CLI flag:
// npm run build --preset bun
```

Build and deploy:

```
bun run build
bun run .output/server/index.mjs
```

Start your Bun server:

```
bun run .output/server/index.mjs
```

Authentication | TanStack Start React Docs

TanStackStart v0
Auto
Framework

React
Version
Latest
Search...
+ K
Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Overview
  react
- Getting Started
  react
- Quick Start
  react
- Build from Scratch
  react
- Learn the Basics
  react
- Server Functions
  react
- Static Server Functions
  react
- Middleware
  react
- Server Routes
  react
- SPA Mode
  react
- Hosting
  react
- Authentication
  react
- Databases
  react
- Observability
  react
- Static Prerendering
  react
- Path Aliases
  react
- Tailwind CSS Integration

react
- Migrate from Next.js
react

Examples

- Basic
react
- Basic + React Query
react
- Basic + Clerk Auth
react
- Basic + DIY Auth
react
- Basic + Supabase Auth
react
- Trellaux + Convex
react
- Trellaux
react
- WorkOS
react
- Material UI
react

Tutorials

- Reading and Writing a File
react

TanStackStart v0
Auto
Search...
+ K
Framework

React
Version
Latest
Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Overview
react
- Getting Started
react
- Quick Start
react
- Build from Scratch
react
- Learn the Basics
react
- Server Functions
react
- Static Server Functions

react
- Middleware
  react
- Server Routes
  react
- SPA Mode
  react
- Hosting
  react
- Authentication
  react
- Databases
  react
- Observability
  react
- Static Prerendering
  react
- Path Aliases
  react
- Tailwind CSS Integration
  react
- Migrate from Next.js
  react

Examples

- Basic
  react
- Basic + React Query
  react
- Basic + Clerk Auth
  react
- Basic + DIY Auth
  react
- Basic + Supabase Auth
  react
- Trellaux + Convex
  react
- Trellaux
  react
- WorkOS
  react
- Material UI
  react

Tutorials

- Reading and Writing a File
  react

On this page

- What should I use?
- What is Clerk?
- Documentation & APIs

Authentication

Authentication is the process of verifying the identity of a user. This is a critical part of any application that requires users to log in or access protected resources. TanStack Start provides the necessary full-stack APIs to implement authentication in your application.

What should I use?

TanStack Start is **designed to work with any authentication provider**, so if you already have an authentication provider or strategy in mind, you can use either find an existing example or implement your own authentication logic using the full-stack APIs provided by TanStack Start.

That said, authentication is not something to be taken lightly. After much vetting, usage and reviewing on our end, we highly recommend using Clerk for the best possible authentication experience. Clerk provides a full suite of authentication APIs and UI components that make it easy to implement authentication in your application and provide a seamless user experience.

What is Clerk?

Clerk is a modern authentication platform that provides a full suite of authentication APIs and UI components to help you implement authentication in your application. Clerk is designed to be easy to use and provides a seamless user experience. With Clerk, you can implement authentication in your application in minutes and provide your users with a secure and reliable authentication experience.

- To learn more about Clerk, visit the Clerk website
- To sign up, visit the Clerk dashboard
- To get started with Clerk, check out our official Start + Clerk examples!

Documentation & APIs

Documentation for implementing your own authentication logic with TanStack Start is coming soon! In the meantime, you can check out any of the -auth prefixed examples for a starting point.

Edit on GitHub

On this page

- What should I use?
- What is Clerk?
- Documentation & APIs

No spam. Unsubscribe at *any* time.

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

Subscribe

No spam. Unsubscribe at *any* time.

Databases | TanStack Start React Docs

On this page

- What should I use?
- How simple is it to use a database with TanStack Start?
- Recommended Database Providers
- What is Neon?
- What is Convex?
- Documentation & APIs

What should I use?

TanStack Start is **designed to work with any database provider**, so if you already have a preferred database system, you can integrate it with TanStack Start using the provided full-stack APIs. Whether you're working with SQL, NoSQL, or other types of databases, TanStack Start can handle your needs.

How simple is it to use a database with TanStack Start?

Using a database with TanStack Start is as simple as calling into your database's adapter/client/driver/service from a TanStack Start server function or server route.

Here's an abstract example of how you might connect with a database and read/write to it:

```
import { createServerFn } from '@tanstack/react-start';

const db = createMyDatabaseClient();

export const getUser = createServerFn(async ({ ctx }) => {
  const user = await db.getUser(ctx.userId);
  return user;
});

export const createUser = createServerFn(async ({ ctx, input }) => {
  const user = await db.createUser(input);
  return user;
});
```

This is obviously contrived, but it demonstrates that you can use literally any database provider with TanStack Start as long as you can call into it from a server function or server route.

Recommended Database Providers

While TanStack Start is designed to work with any database provider, we highly recommend considering one of our vetted partner database providers Neon or Convex. They have been vetted by TanStack to match our quality, openness, and performance standards and are both excellent choices for your database needs.

What is Neon?

Neon is a fully managed serverless PostgreSQL with a generous free tier. It separates storage and compute to offer autoscaling, branching, and bottomless storage. With Neon, you get all the power and reliability of PostgreSQL combined with modern cloud capabilities, making it perfect for TanStack Start applications.

Key features that make Neon stand out:

- Serverless PostgreSQL that scales automatically
- Database branching for development and testing
- Built-in connection pooling
- Point-in-time restore
- Web-based SQL editor
- Bottomless storage

To learn more about Neon, visit the Neon website.

To sign up, visit the Neon dashboard.

What is Convex?

Convex is a powerful, serverless database platform that simplifies the process of managing your application's data. With Convex, you can build full-stack applications without the need to manually manage database servers or write complex queries. Convex provides a real-time, scalable, and transactional data backend that seamlessly integrates with TanStack Start, making it an excellent choice for modern web applications.

Convex's declarative data model and automatic conflict resolution ensure that your application remains consistent and responsive, even at scale. It's designed to be developer-friendly, with a focus on simplicity and productivity.

Features:

- To learn more about Convex, visit the Convex website.
- To sign up, visit the Convex dashboard.

Documentation & APIs

Documentation for integrating different databases with TanStack Start is coming soon! In the meantime, keep an eye on our examples and guide to learn how to fully leverage your data layer across your TanStack Start application.

Edit on GitHub

Observability

On this page

- What should I use?
- What is Sentry?
- Documentation & APIs

What should I use?

TanStack Start is **designed to work with any observability tool**, so you can integrate your preferred solution using the full-stack APIs provided by TanStack Start. Whether you need logging, tracing, or error monitoring, TanStack Start is flexible enough to meet your observability needs.

However, for the best observability experience, we highly recommend using Sentry. Sentry is a powerful, full-featured observability platform that provides real-time insights into your application's performance and error tracking.

What is Sentry?

Sentry is a leading observability platform that helps developers monitor and fix crashes in real-time. With Sentry, you can track errors, performance issues, and trends across your entire stack, from the frontend to the backend. Sentry integrates seamlessly with TanStack Start, enabling you to identify and resolve issues faster, maintain a high level of performance, and deliver a better experience to your users.

Sentry's comprehensive dashboards, alerting capabilities, and in-depth error analysis tools make it an invaluable resource for any development team looking to maintain control over their application's health in production.

- To learn more about Sentry, visit the Sentry website
- To sign up, visit the Sentry dashboard

Documentation & APIs

Documentation for integrating different observability tools with TanStack Start is coming soon! Stay tuned for more examples and guides on how to use Sentry effectively with your TanStack Start projects.

Static Prerendering | TanStack Start React Docs

On this page

- Prerendering

Static Prerendering

Static prerendering is the process of generating static HTML files for your application. This can be useful for either improving the performance of your application, as it allows you to serve pre-rendered HTML files to users without having to generate them on the fly, or for deploying static sites to platforms that do not support server-side rendering.

Prerendering

TanStack Start can prerender your application to static HTML files, which can then be served to users without having to generate them on the fly. To prerender your application, you can add the `server.prerender` option to your `app.config.js` file:

```
// app.config.js

import { defineConfig } from '@tanstack/react-start/config'

export default defineConfig({
  server: {
    prerender: {
      routes: ['/'],
      crawlLinks: true,
    },
  },
})
```

Edit on GitHub

Additional Links and Sections

- Observability
- Path Aliases

Our Partners

- Official Deployment Partner
- Clerk
- Neon
- Convex
- Sentry

TanStack Products

- TanStack Form
  Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.
  Learn More

- TanStack Config
  The build and publish utilities used by all of our projects. Use it if you dare!
  Learn More

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

No spam. Unsubscribe at *any* time.

Path Aliases

TanStack Start React Docs

Path Aliases

Path aliases are a useful feature of TypeScript that allows you to define a shortcut for a path that could be distant in your project's directory structure. This can help you avoid long relative imports in your code and make it easier to refactor your project's structure. This is especially useful for avoiding long relative imports in your code.

By default, TanStack Start does not include path aliases. However, you can easily add them to your project by updating your `tsconfig.json` file in the root of your project and adding the following configuration:

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "~/*": ["./src/*"]
    }
  }
}
```

In this example, we've defined the path alias `~/*` that maps to the `./src/*` directory. This means that you can now import files from the `src` directory using the `~` prefix.

After updating your `tsconfig.json` file, you'll need to install the `vite-tsconfig-paths` plugin to enable path aliases in your TanStack Start project. You can do this by running:

```
npm install -D vite-tsconfig-paths
```

Now, update your `app.config.ts` to include the following:

```
// app.config.ts
import { defineConfig } from '@tanstack/react-start/config'
import viteTsConfigPaths from 'vite-tsconfig-paths'

export default defineConfig({
  vite: {
    plugins: [
      // this is the plugin that enables path aliases
      viteTsConfigPaths({
        projects: ['./tsconfig.json'],
      }),
    ],
  },
})
```

Once this configuration is complete, you'll be able to import files using the path alias like so:

```
// app/routes/posts/$postId/edit.tsx
import { Input } from '~/components/ui/input'
```

```
// instead of
import { Input } from '../../../components/ui/input'
```

Edit on GitHub

TanStack Start React Docs

Navigation

TanStack
[**Start** v0](/start)

Auto

Framework

React

Version

Latest

Search...

+ K

Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- [Overview](/start/latest/docs/framework/react/overview) - react
- [Getting Started](/start/latest/docs/framework/react/getting-started) - react
- [Quick Start](/start/latest/docs/framework/react/quick-start) - react
- [Build from Scratch](/start/latest/docs/framework/react/build-from-scratch) - react
- [Learn the Basics](/start/latest/docs/framework/react/learn-the-basics) - react
- [Server Functions](/start/latest/docs/framework/react/server-functions) - react
- [Static Server Functions](/start/latest/docs/framework/react/static-server-functions) - react
- [Middleware](/start/latest/docs/framework/react/middleware) - react
- [Server Routes](/start/latest/docs/framework/react/server-routes) - react
- [SPA Mode](/start/latest/docs/framework/react/spa-mode) - react
- [Hosting](/start/latest/docs/framework/react/hosting) - react
- [Authentication](/start/latest/docs/framework/react/authentication) - react
- [Databases](/start/latest/docs/framework/react/databases) - react
- [Observability](/start/latest/docs/framework/react/observability) - react
- [Static Prerendering](/start/latest/docs/framework/react/static-prerendering) - react
- [Path Aliases](/start/latest/docs/framework/react/path-aliases) - react
- [Tailwind CSS Integration](/start/latest/docs/framework/react/tailwind-integration) - react
- [Migrate from Next.js](/start/latest/docs/framework/react/migrate-from-next-js) - react

---

Examples

- [Basic](/start/latest/docs/framework/react/examples/start-basic) - react
- [Basic + React Query](/start/latest/docs/framework/react/examples/start-basic-react-query) - react
- [Basic + Clerk Auth](/start/latest/docs/framework/react/examples/start-clerk-basic) - react
- [Basic + DIY Auth](/start/latest/docs/framework/react/examples/start-basic-auth) - react
- [Basic + Supabase Auth](/start/latest/docs/framework/react/examples/start-supabase-basic) - react
- [Trellaux + Convex](/start/latest/docs/framework/react/examples/start-convex-trellaux) - react
- [Trellaux](/start/latest/docs/framework/react/examples/start-trellaux) - react
- [WorkOS](/start/latest/docs/framework/react/examples/start-workos) - react
- [Material UI](/start/latest/docs/framework/react/examples/start-material-ui) - react

---

Tutorials

- [Reading and Writing a File](/start/latest/docs/framework/react/reading-writing-file) - react

---

Error Page

404 Not Found

The page you are looking for does not exist.

Go back [Start Over](/)

Our Partners

- Clerk{:target="_blank"}
- Netlify{:target="_blank"}
  Official Deployment Partner
- Neon{:target="_blank"}
- Convex{:target="_blank"}
- Sentry{:target="_blank"}

TanStack Links

Form

Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.
Learn More

Config

The build and publish utilities used by all of our projects. Use it if you dare!
Learn More

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

No spam. Unsubscribe at *any* time.

Subscribe

TanStack Start React Docs

TanStack Start v0
Auto
Framework

React
Version
Latest
Search...
+ K
Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Overview
  react
- Getting Started
  react
- Quick Start
  react
- Build from Scratch
  react
- Learn the Basics
  react
- Server Functions
  react
- Static Server Functions
  react
- Middleware
  react
- Server Routes
  react
- SPA Mode
  react
- Hosting
  react
- Authentication
  react
- Databases
  react
- Observability
  react
- Static Prerendering
  react
- Path Aliases
  react
- Tailwind CSS Integration
  react
- Migrate from Next.js

react

Examples

- Basic
  react
- Basic + React Query
  react
- Basic + Clerk Auth
  react
- Basic + DIY Auth
  react
- Basic + Supabase Auth
  react
- Trellaux + Convex
  react
- Trellaux
  react
- WorkOS
  react
- Material UI
  react

Tutorials

- Reading and Writing a File
  react

---

404 Not Found

The page you are looking for does not exist.

Go backStart Over

Home

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

Subscribe

No spam. Unsubscribe at *any* time.

React TanStack Start Basic Example | TanStack Start Docs

Home

Overview

Framework: React
Version: Latest

Search and Menu

- Search...
- K
- Menu:
  - Home
  - Frameworks
  - Contributors
  - GitHub
  - Discord

Getting Started

- Overview
- Getting Started
- Quick Start
- Build from Scratch
- Learn the Basics
- Server Functions
- Static Server Functions
- Middleware
- Server Routes
- SPA Mode
- Hosting
- Authentication
- Databases
- Observability
- Static Prerendering
- Path Aliases
- Tailwind CSS Integration
- Migrate from Next.js

Examples

- Basic
- Basic + React Query
- Basic + Clerk Auth
- Basic + DIY Auth

- Basic + Supabase Auth
- Trellaux + Convex
- Trellaux
- WorkOS
- Material UI

Tutorials

- Reading and Writing a File

React Example: Basic

Github
CodeSandbox

Code Explorer | Code | Interactive Sandbox

Files Overview

- `.vscode`
- `src`
  - `main.tsx`
  - `posts.lazy.tsx`
  - `posts.ts`
  - `styles.css`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `postcss.config.mjs`
- `tailwind.config.mjs`
- `tsconfig.json`
- `vite.config.js`

```
import ReactDOM from 'react-dom/client'
import {
  ErrorComponent,
  Link,
  Outlet,
  RouterProvider,
  createRootRoute,
  createRoute,
  createRouter,
} from '@tanstack/react-router'
import { TanStackRouterDevtools } from '@tanstack/react-router-devtools'
import { NotFoundError, fetchPost, fetchPosts } from './posts'
import type { ErrorComponentProps } from '@tanstack/react-router'
import './styles.css'

const rootRoute = createRootRoute({
  component: RootComponent,
  notFoundComponent: () => {
    return (
      <div>
```

```
        <p>This is the notFoundComponent configured on root route</p>
        <Link to="/">Start Over</Link>
      </div>
    )
  },
})

function RootComponent() {
  return (
    <>
      <div className="p-2 flex gap-2 text-lg border-b">
        <Link
          to="/"
          activeProps={{
            className: 'font-bold',
          }}
          activeOptions={{ exact: true }}
        >
          Home
        </Link>{' '}
        <Link
          to="/posts"
          activeProps={{
            className: 'font-bold',
          }}
        >
          Posts
        </Link>{' '}
        <Link
          to="/route-a"
          activeProps={{
            className: 'font-bold',
          }}
        >
          Pathless Layout
        </Link>{' '}
        <Link
          // @ts-expect-error
          to="/this-route-does-not-exist"
          activeProps={{
            className: 'font-bold',
          }}
        >
          This Route Does Not Exist
        </Link>
      </div>
      <Outlet />
      <TanStackRouterDevtools position="bottom-right" />
    </>
  )
}
const indexRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/',
  component: IndexComponent,
```

```
})

function IndexComponent() {
  return (
    <div className="p-2">
      <h3>Welcome Home!</h3>
    </div>
  )
}

export const postsLayoutRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: 'posts',
  loader: () => fetchPosts(),
}).lazy(() => import('./posts.lazy').then((d) => d.Route))

const postsIndexRoute = createRoute({
  getParentRoute: () => postsLayoutRoute,
  path: '/',
  component: PostsIndexComponent,
})

function PostsIndexComponent() {
  return <div>Select a post.</div>
}

const postRoute = createRoute({
  getParentRoute: () => postsLayoutRoute,
  path: '$postId',
  errorComponent: PostErrorComponent,
  loader: ({ params }) => fetchPost(params.postId),
  component: PostComponent,
})

function PostErrorComponent({ error }: ErrorComponentProps) {
  if (error instanceof NotFoundError) {
    return <div>{error.message}</div>
  }

  return <ErrorComponent error={error} />
}

function PostComponent() {
  const post = postRoute.useLoaderData()

  return (
    <div className="space-y-2">
      <h4 className="text-xl font-bold">{post.title}</h4>
      <hr className="opacity-20" />
      <div className="text-sm">{post.body}</div>
    </div>
  )
}

const pathlessLayoutRoute = createRoute({
```

```
  getParentRoute: () => rootRoute,
  id: '_pathlessLayout',
  component: PathlessLayoutComponent,
})

function PathlessLayoutComponent() {
  return (
    <div className="p-2">
      <div className="border-b">I'm a pathless layout</div>
      <div>
        <Outlet />
      </div>
    </div>
  )
}

const nestedPathlessLayout2Route = createRoute({
  getParentRoute: () => pathlessLayoutRoute,
  id: '_nestedPathlessLayout',
  component: PathlessLayout2Component,
})

function PathlessLayout2Component() {
  return (
    <div>
      <div>I'm a nested pathless layout</div>
      <div className="flex gap-2 border-b">
        <Link
          to="/route-a"
          activeProps={{
            className: 'font-bold',
          }}
        >
          Go to Route A
        </Link>
        <Link
          to="/route-b"
          activeProps={{
            className: 'font-bold',
          }}
        >
          Go to Route B
        </Link>
      </div>
      <div>
        <Outlet />
      </div>
    </div>
  )
}

const pathlessLayoutARoute = createRoute({
  getParentRoute: () => nestedPathlessLayout2Route,
  path: '/route-a',
  component: PathlessLayoutAComponent,
```

```
})

function PathlessLayoutAComponent() {
  return <div>I'm route A!</div>
}

const pathlessLayoutBRoute = createRoute({
  getParentRoute: () => nestedPathlessLayout2Route,
  path: '/route-b',
  component: PathlessLayoutBComponent,
})

function PathlessLayoutBComponent() {
  return <div>I'm route B!</div>
}

const routeTree = rootRoute.addChildren([
  postsLayoutRoute.addChildren([postRoute, postsIndexRoute]),
  pathlessLayoutRoute.addChildren([
    nestedPathlessLayout2Route.addChildren([pathlessLayoutARoute, pathlessLayoutBRoute
  ]),
  indexRoute,
])

// Set up a Router instance
const router = createRouter({
  routeTree,
  defaultPreload: 'intent',
  defaultStaleTime: 5000,
  scrollRestoration: true,
})

// Register things for typesafety
declare module '@tanstack/react-router' {
  interface Register {
    router: typeof router
  }
}

const rootElement = document.getElementById('app')!

if (!rootElement.innerHTML) {
  const root = ReactDOM.createRoot(rootElement)
  root.render(<RouterProvider router={router} />)
}
```

React TanStack Start Basic React Query Example | TanStack Start Docs

Navigation

Main Content

React Example: Basic React Query

Github
CodeSandbox

Code Explorer | Interactive Sandbox

.vscode
src
  main.tsx
  posts.lazy.tsx
  posts.ts
  styles.css
.gitignore
README.md
index.html
package.json
postcss.config.mjs
tailwind.config.mjs
tsconfig.json
vite.config.js

Example Code

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import {
  ErrorComponent,
  Link,
  Outlet,
  RouterProvider,
  createRootRouteWithContext,
  createRoute,
  createRouter,
  useRouter,
} from '@tanstack/react-router'
import { TanStackRouterDevtools } from '@tanstack/react-router-devtools'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'
```

```
import {
  QueryClient,
  QueryClientProvider,
  useQueryErrorResetBoundary,
  useSuspenseQuery,
} from '@tanstack/react-query'
import { NotFoundError, postQueryOptions, postsQueryOptions } from './posts'
import type { ErrorComponentProps } from '@tanstack/react-router'
import './styles.css'

const rootRoute = createRootRouteWithContext<{ queryClient: QueryClient }>()({
  component: RootComponent,
  notFoundComponent: () => (
    <div>
      <p>This is the notFoundComponent configured on root route</p>
      <Link to="/">Start Over</Link>
    </div>
  ),
})

function RootComponent() {
  return (
    <>
      <div className="p-2 flex gap-2 text-lg">
        <Link to="/" activeProps={{ className: 'font-bold' }} activeOptions={{ exact:
        <Link to="/posts" activeProps={{ className: 'font-bold' }}>Posts</Link>
        <Link to="/route-a" activeProps={{ className: 'font-bold' }}>Pathless Layout</
        {/* @ts-expect-error */}
        <Link to="/this-route-does-not-exist" activeProps={{ className: 'font-bold' }}
      </div>
      <hr />
      <Outlet />
      <ReactQueryDevtools buttonPosition="top-right" />
      <TanStackRouterDevtools position="bottom-right" />
    </>
  )
}

const indexRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/',
  component: IndexRouteComponent,
})

function IndexRouteComponent() {
  return (
    <div className="p-2">
      <h3>Welcome Home!</h3>
    </div>
  )
}

const postsLayoutRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: 'posts',
```

```
  loader: ({ context: { queryClient } }) =>
    queryClient.ensureQueryData(postsQueryOptions),
}).lazy(() => import('./posts.lazy').then(d => d.Route))

const postsIndexRoute = createRoute({
  getParentRoute: () => postsLayoutRoute,
  path: '/',
  component: PostsIndexRouteComponent,
})

function PostsIndexRouteComponent() {
  return <div>Select a post.</div>
}

const postRoute = createRoute({
  getParentRoute: () => postsLayoutRoute,
  path: '$postId',
  errorComponent: PostErrorComponent,
  loader: ({ context: { queryClient }, params: { postId } }) =>
    queryClient.ensureQueryData(postQueryOptions(postId)),
  component: PostRouteComponent,
})

function PostErrorComponent({ error }: ErrorComponentProps) {
  const router = useRouter()
  if (error instanceof NotFoundError) {
    return <div>{error.message}</div>
  }
  const queryErrorResetBoundary = useQueryErrorResetBoundary()

  React.useEffect(() => {
    queryErrorResetBoundary.reset()
  }, [queryErrorResetBoundary])

  return (
    <div>
      <button
        onClick={() => {
          router.invalidate()
        }}
      >
        retry
      </button>
      <ErrorComponent error={error} />
    </div>
  )
}

function PostRouteComponent() {
  const { postId } = postRoute.useParams()
  const postQuery = useSuspenseQuery(postQueryOptions(postId))
  const post = postQuery.data

  return (
    <div className="space-y-2">
```

```
      <h4 className="text-xl font-bold underline">{post.title}</h4>
      <div className="text-sm">{post.body}</div>
    </div>
  )
}

const pathlessLayoutRoute = createRoute({
  getParentRoute: () => rootRoute,
  id: '_pathlessLayout',
  component: PathlessLayoutComponent,
})

function PathlessLayoutComponent() {
  return (
    <div className="p-2">
      <div className="border-b">I'm a pathless layout</div>
      <div>
        <Outlet />
      </div>
    </div>
  )
}

const nestedPathlessLayoutRoute = createRoute({
  getParentRoute: () => pathlessLayoutRoute,
  id: '_nestedPathlessLayout',
  component: Layout2Component,
})

function Layout2Component() {
  return (
    <div>
      <div>I'm a nested pathless layout</div>
      <div className="flex gap-2 border-b">
        <Link to="/route-a" activeProps={{ className: 'font-bold' }}>Go to route A</Li
        <Link to="/route-b" activeProps={{ className: 'font-bold' }}>Go to route B</Li
      </div>
      <div>
        <Outlet />
      </div>
    </div>
  )
}

const pathlessLayoutARoute = createRoute({
  getParentRoute: () => nestedPathlessLayoutRoute,
  path: '/route-a',
  component: PathlessLayoutAComponent,
})

function PathlessLayoutAComponent() {
  return <div>I'm layout A!</div>
}

const pathlessLayoutBRoute = createRoute({
```

```
  getParentRoute: () => nestedPathlessLayoutRoute,
  path: '/route-b',
  component: PathlessLayoutBComponent,
})

function PathlessLayoutBComponent() {
  return <div>I'm layout B!</div>
}

const routeTree = rootRoute.addChildren([
  postsLayoutRoute.addChildren([postRoute, postsIndexRoute]),
  pathlessLayoutRoute.addChildren([
    nestedPathlessLayoutRoute.addChildren([pathlessLayoutARoute, pathlessLayoutBRoute]
  ]),
  indexRoute,
])

const queryClient = new QueryClient()

// Set up a Router instance
const router = createRouter({
  routeTree,
  defaultPreload: 'intent',
  // Since we're using React Query, we don't want loader calls to ever be stale
  // This will ensure that the loader is always called when the route is preloaded or
  defaultPreloadStaleTime: 0,
  scrollRestoration: true,
  context: {
    queryClient,
  },
})

// Register things for typesafety
declare module '@tanstack/react-router' {
  interface Register {
    router: typeof router
  }
}

const rootElement = document.getElementById('app')!

if (!rootElement.innerHTML) {
  const root = ReactDOM.createRoot(rootElement)
  root.render(
    <QueryClientProvider client={queryClient}>
      <RouterProvider router={router} />
    </QueryClientProvider>
  )
}
```

(Note: The second large code block in the original was a duplicate, so it's omitted here for brevity.)

React TanStack Start Basic Clerk Auth Example | TanStack Start Docs

---

Navigation

TanStack | Start v0
Auto

Framework

React

Version: Latest

Search...

- K

Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Overview (react)
- Getting Started (react)
- Quick Start (react)
- Build from Scratch (react)
- Learn the Basics (react)
- Server Functions (react)
- Static Server Functions (react)
- Middleware (react)
- Server Routes (react)
- SPA Mode (react)
- Hosting (react)
- Authentication (react)
- Databases (react)
- Observability (react)
- Static Prerendering (react)
- Path Aliases (react)
- Tailwind CSS Integration (react)
- Migrate from Next.js (react)

Examples

- Basic (react)
- Basic + React Query (react)
- Basic + Clerk Auth (react)
- Basic + DIY Auth (react)
- Basic + Supabase Auth (react)
- Trellaux + Convex (react)
- Trellaux (react)
- WorkOS (react)
- Material UI (react)

Tutorials

- Reading and Writing a File (react)

Error Message

Something went wrong!

Hide Error

```
Failed to fetch repo contents for tanstack/router/main/examples/react/basic-clerk-auth
```

Try Again | Go Back

Navigation (Repeated)

Home

Our Partners

- Official Deployment Partner
- Netlify
- Neon
- Convex
- Sentry

More Links

TanStack Form
Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.
Learn More

TanStack Config
The build and publish utilities used by all of our projects. Use it if you dare!
Learn More

Subscription Section

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

No spam. Unsubscribe at any time.

Subscribe

(Repeated Subscription Section)

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

No spam. Unsubscribe at any time.

Subscribe

React TanStack Start Basic DIY Auth Example | TanStack Start Docs

---

Navigation Header

TanStack
Start v0
Auto

Filters and Search

Search...

- K

Framework Information

Framework:  React
Version: **Latest**

Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Overview - react
- Getting Started - react
- Quick Start - react
- Build from Scratch - react
- Learn the Basics - react
- Server Functions - react
- Static Server Functions - react
- Middleware - react
- Server Routes - react
- SPA Mode - react
- Hosting - react
- Authentication - react
- Databases - react
- Observability - react
- Static Prerendering - react

- Path Aliases - react
- Tailwind CSS Integration - react
- Migrate from Next.js - react

Examples

- Basic - react
- Basic + React Query - react
- Basic + Clerk Auth - react
- Basic + DIY Auth - react
- Basic + Supabase Auth - react
- Trellaux + Convex - react
- Trellaux - react
- WorkOS - react
- Material UI - react

Tutorials

- Reading and Writing a File - react

Error Message

Something went wrong!

Hide Error

```
Failed to fetch repo contents for tanstack/router/main/examples/react/basic-diy-auth:
```

Retry options

Try Again
Go Back

Navigation Footer

Home

Partners and Resources

Our Partners

- Clerk (opens in new tab)
- Netlify (opens in new tab)
- Neon (opens in new tab)
- Convex (opens in new tab)
- Sentry (opens in new tab)

Create a TanStack Form

TanStack Form
Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.

Learn More

Create a TanStack Config

TanStack Config
The build and publish utilities used by all of our projects. Use it if you dare!

Learn More

Subscribe to Bytes

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

No spam. Unsubscribe at *any* time.

Subscribe

React TanStack Start Basic Supabase Auth Example | TanStack Start Docs

TanStack Start v0
Auto
Framework

React
Version
Latest
Search...
+ K
Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Overview
  react
- Getting Started
  react
- Quick Start
  react
- Build from Scratch
  react
- Learn the Basics
  react
- Server Functions
  react
- Static Server Functions
  react
- Middleware
  react
- Server Routes
  react
- SPA Mode
  react
- Hosting
  react
- Authentication
  react
- Databases
  react
- Observability
  react
- Static Prerendering
  react

- Path Aliases
  react
- Tailwind CSS Integration
  react
- Migrate from Next.js
  react

Examples

- Basic
  react
- Basic + React Query
  react
- Basic + Clerk Auth
  react
- Basic + DIY Auth
  react
- Basic + Supabase Auth
  react
- Trellaux + Convex
  react
- Trellaux
  react
- WorkOS
  react
- Material UI
  react

Tutorials

- Reading and Writing a File
  react

TanStack Start v0
Auto
Search...
+ K
Framework

React
Version
Latest
Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Overview
  react
- Getting Started
  react
- Quick Start
  react
- Build from Scratch
  react
- Learn the Basics

react
- Server Functions
  react
- Static Server Functions
  react
- Middleware
  react
- Server Routes
  react
- SPA Mode
  react
- Hosting
  react
- Authentication
  react
- Databases
  react
- Observability
  react
- Static Prerendering
  react
- Path Aliases
  react
- Tailwind CSS Integration
  react
- Migrate from Next.js
  react

Examples

- Basic
  react
- Basic + React Query
  react
- Basic + Clerk Auth
  react
- Basic + DIY Auth
  react
- Basic + Supabase Auth
  react
- Trellaux + Convex
  react
- Trellaux
  react
- WorkOS
  react
- Material UI
  react

Tutorials

- Reading and Writing a File
  react

Something went wrong! Hide Error

```
Failed to fetch repo contents for tanstack/router/main/examples/react/basic-supabase-a
```

Try Again Go Back
Home

Our Partners
Official Deployment Partner
TanStackForm
Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit and Svelte.
Learn More
TanStackConfig
The build and publish utilities used by all of our projects. Use it if you dare!
Learn More

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

Subscribe

No spam. Unsubscribe at *any* time.

React TanStack Start Trellaux Convex Example | TanStack Start Docs

---

TanStackStart v0
Auto
Framework

React
Version
Latest
Search...
+ K
Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Overview
  react
- Getting Started
  react
- Quick Start
  react
- Build from Scratch
  react
- Learn the Basics
  react
- Server Functions
  react
- Static Server Functions
  react
- Middleware
  react
- Server Routes
  react
- SPA Mode
  react
- Hosting
  react
- Authentication
  react
- Databases
  react
- Observability
  react
- Static Prerendering
  react

- Path Aliases
  react
- Tailwind CSS Integration
  react
- Migrate from Next.js
  react

Examples

- Basic
  react
- Basic + React Query
  react
- Basic + Clerk Auth
  react
- Basic + DIY Auth
  react
- Basic + Supabase Auth
  react
- Trellaux + Convex
  react
- Trellaux
  react
- WorkOS
  react
- Material UI
  react

Tutorials

- Reading and Writing a File
  react

TanStackStart v0
Auto
Search...
+ K
Framework

React
Version
Latest
Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Overview
  react
- Getting Started
  react
- Quick Start
  react
- Build from Scratch
  react
- Learn the Basics
  react

- Server Functions
  react
- Static Server Functions
  react
- Middleware
  react
- Server Routes
  react
- SPA Mode
  react
- Hosting
  react
- Authentication
  react
- Databases
  react
- Observability
  react
- Static Prerendering
  react
- Path Aliases
  react
- Tailwind CSS Integration
  react
- Migrate from Next.js
  react

Examples

- Basic
  react
- Basic + React Query
  react
- Basic + Clerk Auth
  react
- Basic + DIY Auth
  react
- Basic + Supabase Auth
  react
- Trellaux + Convex
  react
- Trellaux
  react
- WorkOS
  react
- Material UI
  react

Tutorials

- Reading and Writing a File
  react

Something went wrong! Hide Error

```
Failed to fetch repo contents for tanstack/router/main/examples/react/trellaux-convex: Status is Not Found - 404
```

Try Again Go Back

Home

Our Partners
Official Deployment Partner
TanStackForm
Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit and Svelte.
Learn More
TanStackConfig
The build and publish utilities used by all of our projects. Use it if you dare!
Learn More

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

Subscribe

No spam. Unsubscribe at *any* time.