SolidStart

Build Full-Stack Applications with SolidJS

Sinan Polat

SolidStart: Build Full-Stack Applications with SolidJS

Sinan Polat

SolidStart: Build Full-Stack Applications with SolidJS by Sinan Polat

Table of Contents

CHAPTER 1

Introduction

SolidJS has quickly established itself as a compelling option for building modern, reactive web applications. Its fine-grained reactivity system delivers performance that rivals hand-optimized code, while its developer experience remains approachable for those familiar with component-driven frameworks.

Within the broader SolidJS ecosystem, however, certain topics stand out for their depth, complexity, and importance in production scenarios. Chief among these are:

- **Server-Side Rendering (SSR)**: rendering components on the server to improve perceived performance, enable SEO, and support progressive enhancement.
- **Routing (Solid Router)**: defining and managing application navigation in a way that integrates seamlessly with Solid's reactivity model.
- **Application Framework (SolidStart)**: a meta-framework for building full-stack applications with SolidJS, combining SSR, routing, and data-loading strategies into a coherent developer workflow.

These subjects require careful treatment not only because they involve more moving parts than client-only applications, but also because they form the foundation of how applications are structured, delivered, and scaled.

The book you are reading brings together selected chapters from my larger work: *SolidJS: The Complete Guide*. While the parent book spans the entire SolidJS landscape, this volume narrows its focus to SolidStart and its surrounding topics. Together, these chapters provide a practical, production-oriented guide for developers already familiar with the fundamentals of SolidJS. Because of their depth and significance, they have been published here as a dedicated volume.

In the chapters ahead, we will begin with server-side rendering, move on to routing with Solid Router, and finally explore SolidStart, which unifies these concepts into a coherent framework for building full-stack applications.

If you're ready to move beyond the fundamentals and see how SolidJS supports production-grade development, you're in the right place.

Requirements

This book is not an introduction to SolidJS itself. It assumes familiarity with the following:

- SolidJS fundamentals: signals, stores, components, and JSX usage.
- Reactivity: understanding how SolidJS tracks dependencies and updates the DOM.
- TypeScript: while not strictly required, examples and best practices in this book use TypeScript for type safety and clarity.
- Basic web development concepts: including HTML, CSS, and JavaScript modules.

Beyond This Book

This volume is deliberately focused. By concentrating on SSR, routing, and SolidStart, it gives you the tools to build production-ready SolidJS applications.

If you want the full picture of SolidJS—starting from its foundations and building up to advanced application architecture—you'll find it in the parent book, *SolidJS: The Complete Guide*. There you'll discover:

- A thorough introduction to SolidJS fundamentals.
- The inner workings of its reactivity model.
- Component composition patterns and lifecycle strategies.
- State management and performance tuning.
- Step-by-step examples that build from beginner-friendly apps to advanced architectures.

Code Examples

All code examples used in this book are available on GitHub:

https://github.com/solid-courses/solidjs-the-complete-guide

Navigate to the `examples` folder to find the material for the chapters in this volume. The examples for this book begin with `ch33`, corresponding to the first chapter included here.

You are encouraged to download, run, and experiment with the code as you follow along in the text. Hands-on practice will help reinforce the concepts and give you a clearer sense of how SolidJS applications come together in real projects.

In general, you are free to use the example code provided with this book in your own programs and documentation. Permission is not required unless you intend to reproduce a substantial portion of the code. For example, you may use several snippets in your programs without restriction, and you may also cite the book or quote short examples when answering a question. However, selling or redistributing the examples requires prior permission, as does incorporating a significant amount of code into your product's documentation.

Contact and Feedback

To share feedback, suggestions, or corrections, please visit the repository:

https://github.com/solid-courses/solidjs-the-complete-guide

You can open an issue to provide your input. Your contributions will help improve both this book and the learning experience of future readers.

CHAPTER 33

Server Side Rendering

Up until now, we've been building client-side–only applications—code that runs entirely in the browser. The server's job has been little more than handing over some HTML, JavaScript, and CSS, and exposing an API for our client code to call. Everything else—fetching data, building UI, rendering components—has happened in the user's machine—every click, every data fetch, every DOM update handled by the browser alone.

Applications built this way are known as Single Page Applications (SPAs). Even if the user navigates to different "pages," they're really just interacting with different views in the same application shell. SPAs have their strengths: they enable smooth, app-like transitions, and they keep the server out of the rendering process once the app has loaded. But they also leave a lot of untapped potential on the server side—and in some cases, they make life harder than it needs to be.

Issues With Single Page Applications

While SPAs offer a smooth, app-like feel, they also come with trade-offs. By relying entirely on the client for rendering, you inherit a set of limitations, performance bottlenecks, and risks:

SEO limitations

In the age of the internet, success often hinges on discoverability—and discoverability is directly tied to how well your site performs in search rankings.

Search engines index the HTML they receive from the server, and in a client-rendered SPA, that initial HTML is often little more than a hollow shell—a `<div>` awaiting JavaScript to bring it to life. While some crawlers can execute JavaScript to reconstruct the full page, this behavior is neither guaranteed nor consistent.

It can break if your app depends on APIs that behave differently in a headless crawling environment, or if certain browser features are unavailable. Even when it works, the extra execution step delays content discovery, making it harder for your pages to be indexed promptly.

Worse still, every second the page remains visually empty before meaningful content appears can signal poor performance to search engines—negatively impacting both ranking and visibility.

Slow perceived load

When a user requests a page in a SPA, the following has to happen before they see anything:

- The browser downloads the JavaScript bundle.
- The JavaScript executes, bootstrapping the app.
- The app fetches data from APIs.
- The app renders the UI.

That's several steps before the user sees anything useful. In contrast, with server-side rendering (SSR), the server can send a fully-rendered HTML document immediately. The browser can start displaying meaningful content as soon as it arrives—no waiting for the JavaScript to finish running.

More complexity, more points of failure

With client-only rendering, the browser is in charge of everything. That means your carefully designed app is at the mercy of:

- Slow CPUs or low-memory devices
- Flaky networks
- Browser quirks
- Misbehaving browser extensions
- Users with JavaScript disabled (rare, but not unheard of)

Any one of these can break the experience in ways that are hard to predict and even harder to debug. SSR gives you a chance to deliver a usable, meaningful page before the client code even starts running.

Underused server capabilities

Servers are powerful machines. They can prepare content in advance, cache results, aggregate data from multiple sources, or render pages ahead of time. With SSR, you can leverage these capabilities to reduce client work, shorten round trips, and improve time-to-first-meaningful-paint. Even without complex caching strategies, rendering content on the server generally means faster initial display.

Security considerations

In SPAs, more logic and data handling happen in the browser, where you have less control. This can expose a larger attack surface. With SSR, sensitive logic can stay on the server, where you can protect it using tried-and-tested server-side security practices. You still need to secure your client-side code, but you can reduce its responsibilities and limit its exposure.

Accessibility improvements

A server-rendered application can work even with JavaScript disabled. That's not the goal in most modern apps, but it's a side effect of sending fully-formed HTML to the browser. For users relying on assistive technologies—or older devices and browsers—this means your content is immediately available and correctly structured from the moment it arrives.

SSR: Visible Content From the First Byte

Server-side rendering (SSR) addresses these shortcomings head-on by sending the user—and search engines—a fully rendered page right from the start. Instead of shipping an empty shell and leaving the browser to assemble the UI, the server prepares the HTML in advance, complete with real content.

With SSR, the server renders the initial HTML for the requested page and sends it directly to the browser. The browser displays this content immediately, and then the client-side JavaScript "hydrates" it—attaching event listeners, restoring state, and making it interactive. From that point on, navigation can happen entirely on the client, just like in a SPA. You get the best of both worlds: fast initial load and smooth subsequent navigation.

When that HTML arrives in the browser, the client-side JavaScript "hydrates" it—reconstructing the component state that existed on the server, attaching event listeners, and wiring up reactivity so the UI can update dynamically. Hydration ensures that the server-rendered markup becomes a live, interactive application without having to rebuild the DOM from scratch.

Solid embraces this model with an SSR pipeline that's deliberately simple yet flexible. At its core, Solid renders your components to HTML strings or streams directly from the server, then ships them to the browser along with the minimal JavaScript needed to take over. This approach provides a fast initial render, better SEO, and a more resilient user experience.

Before we look at the specific rendering strategies Solid offers, it's worth noting that these same mechanisms power Solid Start, Solid's official full-stack framework for building server-rendered applications. By understanding how SSR works at this lower level, you'll gain the insight needed to customize, troubleshoot, and extend your own apps—while also laying the groundwork to take full advantage of Solid Start when we explore it later in the book.

With that in mind, let's examine the three rendering strategies Solid provides for SSR applications and see how each one balances speed, flexibility, and support for asynchronous data.

Three Rendering Approaches for SSR Applications

To support different performance and data-fetching needs, Solid provides three rendering functions —`renderToString`, `renderToStringAsync`, and `renderToStream`. Each handles asynchronous data and `Suspense` boundaries differently, allowing you to choose the right trade-off between speed, completeness, and interactivity.

Let's break down how each of these works and when you might use them.

`renderToString` - Synchronous HTML Generation

The `renderToString` function is a synchronous function that renders an application into an HTML string by concatenating the output of all components into a single value, which can then be sent to the client in one response. There is no streaming and no support for asynchronous data—any asynchronous values must be resolved later on the client side after hydration. In other words, if your components use Suspense, their fallback content will be rendered, and any associated data fetching will happen in the browser once the client-side application takes over.

```
import { renderToString } from "solid-js/web";
const app = renderToString(() => <App />);
```

If the returned string from `renderToString` forms a complete HTML document (with `<head>`, `<body>`, and `<title>`), it can be sent directly to the client:

```
const app = express();

app.get('/', (req, res) => {
  try {
    const app = renderToString(() => <App />);
    res.send(app);
  } catch (error) {
    res.send('Failed to render!');
  }
});
```

If the output is not a complete HTML document, you need to wrap it in a template that includes the missing elements. Otherwise, the browser will not interpret it correctly:

```
app.get('/', (req, res) => {
  const app = renderToString(() => <App />);
```

```
  const html = `
  <html lang="en">
    <head>
      <title>My SSR App</title>
      <meta charset="UTF-8" />
    </head>
    <body>${app}</body>
  </html>
`;

  res.send(html);
});
```

`renderToStringAsync` - Asynchronous HTML Generation

The `renderToStringAsync` function is an asynchronous function that renders the application into an HTML string, but unlike `renderToString`, it waits for all asynchronous resources to resolve before producing the output. Because it returns a promise, you can use `await` when calling it.

```
import { renderToStringAsync } from "solid-js/web";

app.get('/', async (req, res) => {
  try {
    const app = await renderToStringAsync(() => <App />);
    res.send(app);
  } catch (error) {
    res.send('Failed to render!');
  }
});
```

This ensures the user gets a fully populated page on the first load—ideal for SEO-sensitive content and scenarios where data freshness matters. The trade-off is a slower time-to-first-byte (TTFB), since the server waits until everything is ready before sending any HTML.

`renderToStream` - Streaming with Progressive Rendering

The `renderToStream` function takes advantage of HTTP streaming to send HTML to the client in chunks. It starts by sending the static parts of the page (often called the "shell") immediately, then flushes the content of Suspense boundaries as their data resolves.