Server-Side Rendering (SSR)

Overview

What is SSR?

Vue.js is a framework for building client-side applications. By default, Vue components produce and manipulate DOM in the browser as output. However, it is also possible to render the same components into HTML strings on the server, send them directly to the browser, and finally "hydrate" the static markup into a fully interactive app on the client.

A server-rendered Vue.js app can also be considered "isomorphic" or "universal", in the sense that the majority of your app's code runs on both the server **and** the client.

Why SSR?

Compared to a client-side Single-Page Application (SPA), the advantage of SSR primarily lies in:

- **Faster time-to-content**: this is more prominent on slow internet or slow devices. Server-rendered markup doesn't need to wait until all JavaScript has been downloaded and executed to be displayed, so your user will see a fully-rendered page sooner. In addition, data fetching is done on the server-side for the initial visit, which likely has a faster connection to your database than the client. This generally results in improved Core Web Vitals metrics, better user experience, and can be critical for applications where time-to-content is directly associated with conversion rate.
- **Unified mental model**: you get to use the same language and the same declarative, component-oriented mental model for developing your entire app, instead of jumping back and forth between a backend templating system and a frontend framework.
- **Better SEO**: the search engine crawlers will directly see the fully rendered page.

> ⓘ TIP

spinner, then fetches content via Ajax, the crawler will not wait for you to finish. This means if you have content fetched asynchronously on pages where SEO is important, SSR might be necessary.

There are also some trade-offs to consider when using SSR:

- Development constraints. Browser-specific code can only be used inside certain lifecycle hooks; some external libraries may need special treatment to be able to run in a server-rendered app.
- More involved build setup and deployment requirements. Unlike a fully static SPA that can be deployed on any static file server, a server-rendered app requires an environment where a Node.js server can run.
- More server-side load. Rendering a full app in Node.js is going to be more CPU-intensive than just serving static files, so if you expect high traffic, be prepared for corresponding server load and wisely employ caching strategies.

Before using SSR for your app, the first question you should ask is whether you actually need it. It mostly depends on how important time-to-content is for your app. For example, if you are building an internal dashboard where an extra few hundred milliseconds on initial load doesn't matter that much, SSR would be an overkill. However, in cases where time-to-content is absolutely critical, SSR can help you achieve the best possible initial load performance.

SSR vs. SSG

Static Site Generation (SSG), also referred to as pre-rendering, is another popular technique for building fast websites. If the data needed to server-render a page is the same for every user, then instead of rendering the page every time a request comes in, we can render it only once, ahead of time, during the build process. Pre-rendered pages are generated and served as static HTML files.

SSG retains the same performance characteristics of SSR apps: it provides great time-to-content performance. At the same time, it is cheaper and easier to deploy than SSR apps because the output is static HTML and assets. The keyword here is **static**: SSG can only be applied to pages providing static data, i.e. data that is known at build time and can not change between requests. Every time the data changes, a new deployment is needed.

If you're only investigating SSR to improve the SEO of a handful of marketing pages (e.g. `/`, `/about`, `/contact`, etc.), then you probably want SSG instead of SSR. SSG is also great for content-based websites such as documentation sites or blogs. In fact, this website you are reading right now is statically generated using VitePress, a Vue-powered static site generator.

Rendering an App

Let's take a look at the most bare-bones example of Vue SSR in action.

1. Create a new directory and `cd` into it
2. Run `npm init -y`
3. Add `"type": "module"` in `package.json` so that Node.js runs in ES modules mode.
4. Run `npm install vue`
5. Create an `example.js` file:

```js
// this runs in Node.js on the server.
import { createSSRApp } from 'vue'
// Vue's server-rendering API is exposed under `vue/server-renderer`.
import { renderToString } from 'vue/server-renderer'

const app = createSSRApp({
  data: () => ({ count: 1 }),
  template: `<button @click="count++">{{ count }}</button>`
})

renderToString(app).then((html) => {
  console.log(html)
})
```

Then run:

```sh
> node example.js
```

It should print the following to the command line:

```
<button>1</button>
```

`renderToString()` takes a Vue app instance and returns a Promise that resolves to the rendered HTML of the app. It is also possible to stream rendering using the Node.js Stream API or Web Streams API. Check out the SSR API Reference for full details.

We can then move the Vue SSR code into a server request handler, which wraps the application markup with the full page HTML. We will be using `express` for the next steps:

- Run `npm install express`
- Create the following `server.js` file:

```
import { renderToString } from 'vue/server-renderer'

const server = express()

server.get('/', (req, res) => {
  const app = createSSRApp({
    data: () => ({ count: 1 }),
    template: `<button @click="count++">{{ count }}</button>`
  })

  renderToString(app).then((html) => {
    res.send(`
    <!DOCTYPE html>
    <html>
      <head>
        <title>Vue SSR Example</title>
      </head>
      <body>
        <div id="app">${html}</div>
      </body>
    </html>
    `)
  })
})

server.listen(3000, () => {
  console.log('ready')
})
```

Finally, run `node server.js` and visit `http://localhost:3000`. You should see the page working with the button.

Try it on StackBlitz

Client Hydration

If you click the button, you'll notice the number doesn't change. The HTML is completely static on the client since we are not loading Vue in the browser.

To make the client-side app interactive, Vue needs to perform the **hydration** step. During hydration, it creates the same Vue application that was run on the server, matches each component to the DOM nodes it should control, and attaches DOM event listeners.

To mount an app in hydration mode, we need to use `createSSRApp()` instead of `createApp()`:

```
// this runs in the browser.
import { createSSRApp } from 'vue'

const app = createSSRApp({
```

```
// mounting an SSR app on the client assumes
// the HTML was pre-rendered and will perform
// hydration instead of mounting new DOM nodes.
app.mount('#app')
```

Code Structure

Notice how we need to reuse the same app implementation as on the server. This is where we need to start thinking about code structure in an SSR app - how do we share the same application code between the server and the client?

Here we will demonstrate the most bare-bones setup. First, let's split the app creation logic into a dedicated file, `app.js`:

JS app.js

```js
// (shared between server and client)
import { createSSRApp } from 'vue'

export function createApp() {
  return createSSRApp({
    data: () => ({ count: 1 }),
    template: `<button @click="count++">{{ count }}</button>`
  })
}
```

This file and its dependencies are shared between the server and the client - we call them **universal code**. There are a number of things you need to pay attention to when writing universal code, as we will discuss below.

Our client entry imports the universal code, creates the app, and performs the mount:

JS client.js

```js
import { createApp } from './app.js'

createApp().mount('#app')
```

And the server uses the same app creation logic in the request handler:

JS server.js

```js
// (irrelevant code omitted)
import { createApp } from './app.js'

server.get('/', (req, res) => {
```

```
    // ...
  })
})
```

In addition, in order to load the client files in the browser, we also need to:

1. Serve client files by adding `server.use(express.static('.'))` in `server.js`.
2. Load the client entry by adding `<script type="module" src="/client.js"></script>` to the HTML shell.
3. Support usage like `import * from 'vue'` in the browser by adding an Import Map to the HTML shell.

Try the completed example on StackBlitz. The button is now interactive!

Higher Level Solutions

Moving from the example to a production-ready SSR app involves a lot more. We will need to:

- Support Vue SFCs and other build step requirements. In fact, we will need to coordinate two builds for the same app: one for the client, and one for the server.

  > ⓘ TIP
  >
  > Vue components are compiled differently when used for SSR - templates are compiled into string concatenations instead of Virtual DOM render functions for more efficient rendering performance.

- In the server request handler, render the HTML with the correct client-side asset links and optimal resource hints. We may also need to switch between SSR and SSG mode, or even mix both in the same app.

- Manage routing, data fetching, and state management stores in a universal manner.

A complete implementation would be quite complex and depends on the build toolchain you have chosen to work with. Therefore, we highly recommend going with a higher-level, opinionated solution that abstracts away the complexity for you. Below we will introduce a few recommended SSR solutions in the Vue ecosystem.

Nuxt

Nuxt is a higher-level framework built on top of the Vue ecosystem which provides a streamlined development experience for writing universal Vue applications. Better yet, you

Quasar

Quasar is a complete Vue-based solution that allows you to target SPA, SSR, PWA, mobile app, desktop app, and browser extension all using one codebase. It not only handles the build setup, but also provides a full collection of Material Design compliant UI components.

Vite SSR

Vite provides built-in support for Vue server-side rendering, but it is intentionally low-level. If you wish to go directly with Vite, check out vite-plugin-ssr, a community plugin that abstracts away many challenging details for you.

You can also find an example Vue + Vite SSR project using manual setup here, which can serve as a base to build upon. Note this is only recommended if you are experienced with SSR / build tools and really want to have complete control over the higher-level architecture.

Writing SSR-friendly Code

Regardless of your build setup or higher-level framework choice, there are some principles that apply in all Vue SSR applications.

Reactivity on the Server

During SSR, each request URL maps to a desired state of our application. There is no user interaction and no DOM updates, so reactivity is unnecessary on the server. By default, reactivity is disabled during SSR for better performance.

Component Lifecycle Hooks

Since there are no dynamic updates, lifecycle hooks such as `onMounted` or `onUpdated` will **NOT** be called during SSR and will only be executed on the client.

You should avoid code that produces side effects that need cleanup in `setup()` or the root scope of `<script setup>`. An example of such side effects is setting up timers with `setInterval`. In client-side only code we may setup a timer and then tear it down in `onBeforeUnmount` or `onUnmounted`. However, because the unmount hooks will never be called during SSR, the timers will stay around forever. To avoid this, move your side-effect code into `onMounted` instead.

Universal code cannot assume access to platform-specific APIs, so if your code directly uses browser-only globals like `window` or `document`, they will throw errors when executed in Node.js, and vice-versa.

For tasks that are shared between server and client but with different platform APIs, it's recommended to wrap the platform-specific implementations inside a universal API, or use libraries that do this for you. For example, you can use `node-fetch` to use the same fetch API on both server and client.

For browser-only APIs, the common approach is to lazily access them inside client-only lifecycle hooks such as `onMounted`.

Note that if a third-party library is not written with universal usage in mind, it could be tricky to integrate it into a server-rendered app. You *might* be able to get it working by mocking some of the globals, but it would be hacky and may interfere with the environment detection code of other libraries.

Cross-Request State Pollution

In the State Management chapter, we introduced a simple state management pattern using Reactivity APIs. In an SSR context, this pattern requires some additional adjustments.

The pattern declares shared state in a JavaScript module's root scope. This makes them **singletons** - i.e. there is only one instance of the reactive object throughout the entire lifecycle of our application. This works as expected in a pure client-side Vue application, since the modules in our application are initialized fresh for each browser page visit.

However, in an SSR context, the application modules are typically initialized only once on the server, when the server boots up. The same module instances will be reused across multiple server requests, and so will our singleton state objects. If we mutate the shared singleton state with data specific to one user, it can be accidentally leaked to a request from another user. We call this **cross-request state pollution.**

We can technically re-initialize all the JavaScript modules on each request, just like we do in browsers. However, initializing JavaScript modules can be costly, so this would significantly affect server performance.

The recommended solution is to create a new instance of the entire application - including the router and global stores - on each request. Then, instead of directly importing it in our components, we provide the shared state using app-level provide and inject it in components that need it:

app.js

```js
// (shared between server and client)
import { createSSRApp } from 'vue'
```

```
// called on each request
export function createApp() {
  const app = createSSRApp(/* ... */)
  // create new instance of store per request
  const store = createStore(/* ... */)
  // provide store at the app level
  app.provide('store', store)
  // also expose store for hydration purposes
  return { app, store }
}
```

State Management libraries like Pinia are designed with this in mind. Consult Pinia's SSR guide for more details.

Hydration Mismatch

If the DOM structure of the pre-rendered HTML does not match the expected output of the client-side app, there will be a hydration mismatch error. Hydration mismatch is most commonly introduced by the following causes:

1. The template contains invalid HTML nesting structure, and the rendered HTML got "corrected" by the browser's native HTML parsing behavior. For example, a common gotcha is that `<div>` cannot be placed inside `<p>`:

   ```html
   <p><div>hi</div></p>
   ```

   If we produce this in our server-rendered HTML, the browser will terminate the first `<p>` when `<div>` is encountered and parse it into the following DOM structure:

   ```html
   <p></p>
   <div>hi</div>
   <p></p>
   ```

2. The data used during render contains randomly generated values. Since the same application will run twice - once on the server, and once on the client - the random values are not guaranteed to be the same between the two runs. There are two ways to avoid random-value-induced mismatches:

   1. Use `v-if` + `onMounted` to render the part that depends on random values only on the client. Your framework may also have built-in features to make this easier, for example the `<ClientOnly>` component in VitePress.

   2. Use a random number generator library that supports generating with seeds, and guarantee the server run and the client run are using the same seed (e.g. by including the seed in serialized state and retrieving it on the client).

the timezone during the client run are not always the same, and we may not reliably know the user's timezone during the server run. In such cases, the local time conversion should also be performed as a client-only operation.

When Vue encounters a hydration mismatch, it will attempt to automatically recover and adjust the pre-rendered DOM to match the client-side state. This will lead to some rendering performance loss due to incorrect nodes being discarded and new nodes being mounted, but in most cases, the app should continue to work as expected. That said, it is still best to eliminate hydration mismatches during development.

Suppressing Hydration Mismatches 3.5+

In Vue 3.5+, it is possible to selectively suppress inevitable hydration mismatches by using the `data-allow-mismatch` attribute.

Custom Directives

Since most custom directives involve direct DOM manipulation, they are ignored during SSR. However, if you want to specify how a custom directive should be rendered (i.e. what attributes it should add to the rendered element), you can use the `getSSRProps` directive hook:

```js
const myDirective = {
  mounted(el, binding) {
    // client-side implementation:
    // directly update the DOM
    el.id = binding.value
  },
  getSSRProps(binding) {
    // server-side implementation:
    // return the props to be rendered.
    // getSSRProps only receives the directive binding.
    return {
      id: binding.value
    }
  }
}
```

Teleports

Teleports require special handling during SSR. If the rendered app contains Teleports, the teleported content will not be part of the rendered string. An easier solution is to conditionally render the Teleport on mount.

If you do need to hydrate teleported content, they are exposed under the `teleports` property of the ssr context object:

```
console.log(ctx.teleports) // { '#teleported': 'teleported content' }
```

You need to inject the teleport markup into the correct location in your final page HTML similar to how you need to inject the main app markup.

> ⓘ TIP
>
> Avoid targeting `body` when using Teleports and SSR together - usually, `<body>` will contain other server-rendered content which makes it impossible for Teleports to determine the correct starting location for hydration.
>
> Instead, prefer a dedicated container, e.g. `<div id="teleported"></div>` which contains only teleported content.

Edit this page on GitHub

Previous
Testing

Next
Production Deployment