Rendering Modes

Nuxt supports different rendering modes, universal rendering, client-side rendering but also offers hybrid-rendering and the possibility to render your application on CDN Edge Servers.

Both the browser and server can interpret JavaScript code to turn Vue.js components into HTML elements. This step is called **rendering**. Nuxt supports both **universal** and **client-side** rendering. The two approaches have benefits and downsides that we will cover.

By default, Nuxt uses **universal rendering** to provide better user experience, performance and to optimize search engine indexing, but you can switch rendering modes in one line of configuration.

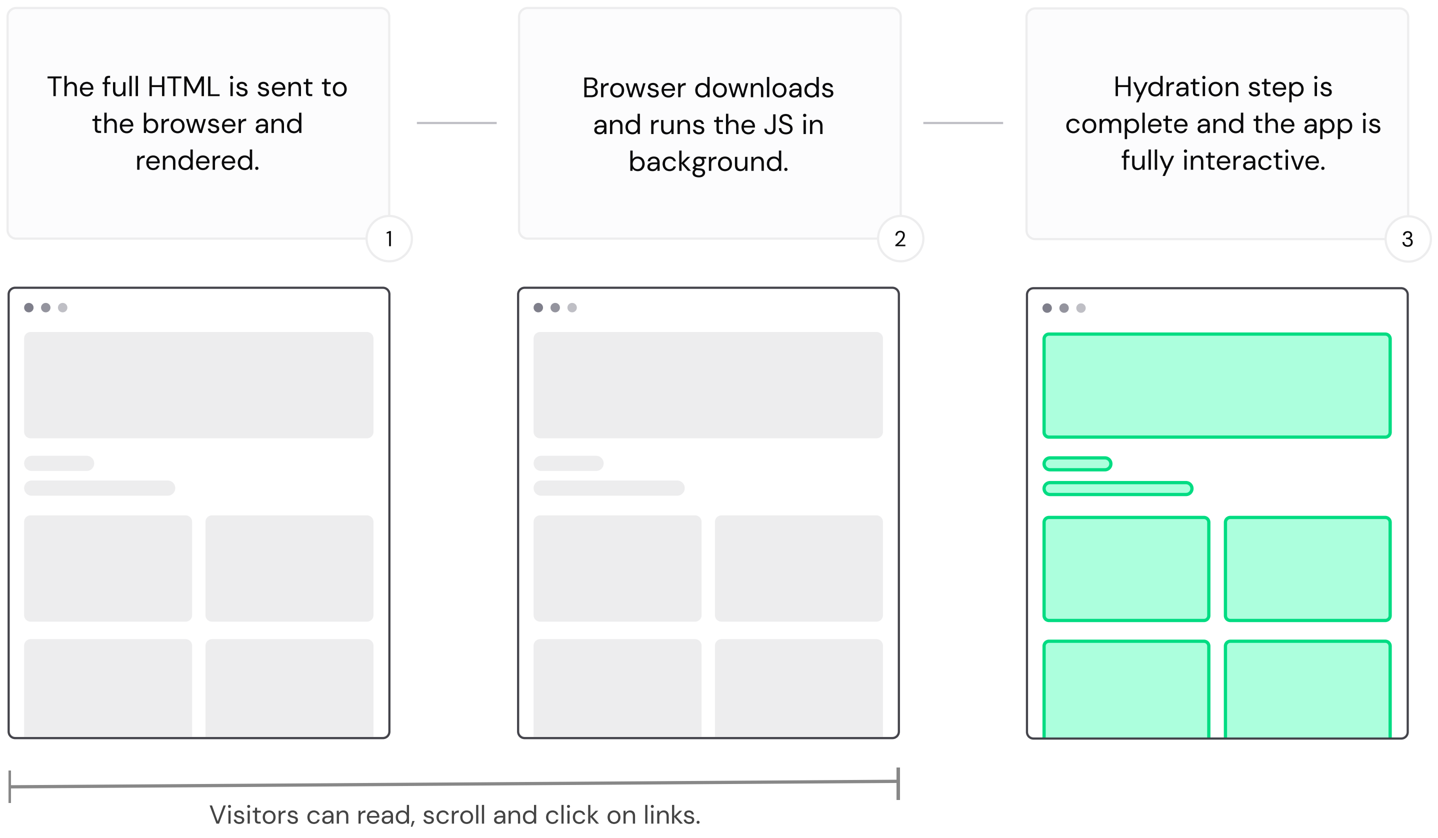
Universal Rendering

When the browser requests a URL with universal (server-side + client-side) rendering enabled, the server returns a fully rendered HTML page to the browser. Whether the page has been generated in advance and cached or is rendered on the fly, at some point, Nuxt has run the JavaScript (Vue.js) code in a server environment, producing an HTML document. Users immediately get the content of our application, contrary to client-side rendering. This step is similar to traditional **server-side rendering** performed by PHP or Ruby applications.

To not lose the benefits of the client-side rendering method, such as dynamic interfaces and pages transitions, the Client (browser) loads the JavaScript code that runs on the Server in the background once the HTML document has been downloaded. The browser interprets it again (hence **Universal rendering**) and Vue.js takes control of the document and enables interactivity.

Making a static page interactive in the browser is called "Hydration."

Universal rendering allows a Nuxt application to provide quick page load times while preserving the benefits of client-side rendering. Furthermore, as the content is already present in the HTML document, crawlers can index it without overhead.


The full HTML is sent to the browser and rendered.
1
Browser downloads and runs the JS in background.
2
Hydration step is complete and the app is fully interactive.
3
Visitors can read, scroll and click on links.

Benefits of server-side rendering:

- **Performance**: Users can get immediate access to the page's content because browsers can display static content much faster than JavaScript-generated one. At the same time, Nuxt preserves the interactivity of a web application when the hydration process happens.

- **Search Engine Optimization**: Universal rendering delivers the entire HTML content of the page to the browser as a classic server application. Web crawlers can directly index the page's content, which makes Universal rendering a great choice for any content that you want to index quickly.

Downsides of server-side rendering:

- **Development constraints:** Server and browser environments don't provide the same APIs, and it can be tricky to write code that can run on both sides seamlessly. Fortunately, Nuxt provides guidelines and specific variables to help you determine where a piece of code is executed.

- **Cost:** A server needs to be running in order to render pages on the fly. This adds a monthly cost like any traditional server. However, the server calls are highly reduced thanks to universal rendering with the browser taking over on client-side navigation. A cost reduction is possible by leveraging edge-side-rendering.

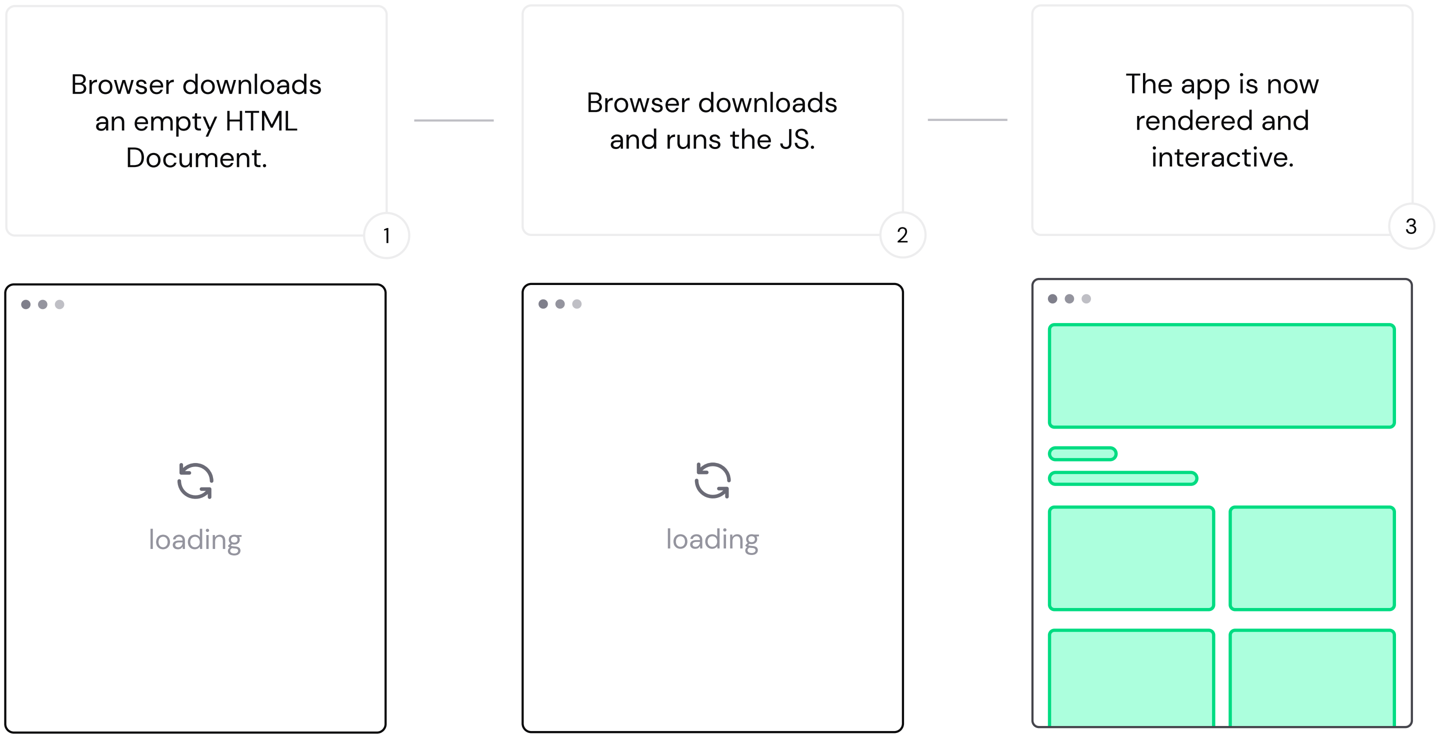
Universal rendering is very versatile and can fit almost any use case, and is especially appropriate for any content-oriented websites: **blogs, marketing websites, portfolios, e-commerce sites, and marketplaces.**

For more examples about writing Vue code without hydration mismatch, see the Vue docs.

When importing a library that relies on browser APIs and has side effects, make sure the component importing it is only called client-side. Bundlers do not treeshake imports of modules containing side effects.

Client-Side Rendering

Out of the box, a traditional Vue.js application is rendered in the browser (or **client**). Then, Vue.js generates HTML elements after the browser downloads and parses all the JavaScript code containing the instructions to create the current interface.


Browser downloads an empty HTML Document.
1
Browser downloads and runs the JS.
2
The app is now rendered and interactive.
3
loading
loading

Benefits of client-side rendering:

- **Development speed**: When working entirely on the client-side, we don't have to worry about the server compatibility of the code, for example, by using browser-only APIs like the `window` object.
- **Cheaper:** Running a server adds a cost of infrastructure as you would need to run on a platform that supports JavaScript. We can host Client-only applications on any static server with HTML, CSS, and JavaScript files.

- **Offline:** Because code entirely runs in the browser, it can nicely keep working while the internet is unavailable.

Downsides of client-side rendering:

- **Performance**: The user has to wait for the browser to download, parse and run JavaScript files. Depending on the network for the download part and the user's device for the parsing and execution, this can take some time and impact the user's experience.
- **Search Engine Optimization**: Indexing and updating the content delivered via client-side rendering takes more time than with a server-rendered HTML document. This is related to the performance drawback we discussed, as search engine crawlers won't wait for the interface to be fully rendered on their first try to index the page. Your content will take more time to show and update in search results pages with pure client-side rendering.

Client-side rendering is a good choice for heavily interactive **web applications** that don't need indexing or whose users visit frequently. It can leverage browser caching to skip the download phase on subsequent visits, such as **SaaS, back-office applications, or online games**.

You can enable client-side only rendering with Nuxt in your `nuxt.config.ts` :

nuxt.config.ts

```ts
export default defineNuxtConfig({
  ssr: false
})
```

If you do use `ssr: false` , you should also place an HTML file in `~/app/spa-loading-template.html` with some HTML you would like to use to render a loading screen that will be rendered until your app is hydrated.

👉 Read more in Docs > API > Configuration > Nuxt Config #spaloadingtemplate.

Hybrid Rendering

Hybrid rendering allows different caching rules per route using **Route Rules** and decides how the server should respond to a new request on a given URL.

Previously every route/page of a Nuxt application and server must use the same rendering mode, universal or client-side. In various cases, some pages could be generated at build time, while others should be client-side rendered. For example, think of a content website with an admin section. Every content page should be primarily static and generated once, but the admin section requires registration and behaves more like a dynamic application.

Nuxt 3 includes route rules and hybrid rendering support. Using route rules you can define rules for a group of nuxt routes, change rendering mode or assign a cache strategy based on route!

Nuxt server will automatically register corresponding middleware and wrap routes with cache handlers using Nitro caching layer.

Example:

nuxt.config.ts

```
export default defineNuxtConfig({
  routeRules: {
    // Homepage pre-rendered at build time
    '/': { prerender: true },
    // Product page generated on-demand, revalidates in background
    '/products/**': { swr: 3600 },
    // Blog post generated on-demand once until next deploy
    '/blog/**': { isr: true },
    // Admin dashboard renders only on client-side
    '/admin/**': { ssr: false },
    // Add cors headers on API routes
    '/api/**': { cors: true },
    // Redirects legacy urls
    '/old-page': { redirect: '/new-page' }
  }
})
```

The different properties you can use are the following:

- `redirect: string` - Define server-side redirects.
- `ssr: boolean` - Disables server-side rendering for sections of your app and make them SPA-only with `ssr: false`
- `cors: boolean` - Automatically adds cors headers with `cors: true` - you can customize the output by overriding with `headers`
- `headers: object` - Add specific headers to sections of your site - for example, your assets
- `swr: number|boolean` - Add cache headers to the server response and cache it on the server or reverse proxy for a configurable TTL (time to live). The `node-server` preset of Nitro is able to cache the full response. When the TTL expired, the cached response will be sent while the page will be regenerated in the background. If true is used, a `stale-while-revalidate` header is added without a MaxAge.

- `isr: number|boolean` - The behavior is the same as `swr` except that we are able to add the response to the CDN cache on platforms that support this (currently Netlify or Vercel). If `true` is used, the content persists until the next deploy inside the CDN.
- `prerender:boolean` - Prerenders routes at build time and includes them in your build as static assets
- `experimentalNoScripts: boolean` - Disables rendering of Nuxt scripts and JS resource hints for sections of your site.

Whenever possible, route rules will be automatically applied to the deployment platform's native rules for optimal performances (Netlify and Vercel are currently supported).

> Note that Hybrid Rendering is not available when using `nuxt generate`.

Examples:

- Nuxt + Vercel integration with hybrid rendering

Edge-Side Rendering

Edge-Side Rendering (ESR) is a powerful feature introduced in Nuxt 3 that allows the rendering of your Nuxt application closer to your users via edge servers of a Content Delivery Network (CDN). By leveraging ESR, you can ensure improved performance and reduced latency, thereby providing an enhanced user experience.

With ESR, the rendering process is pushed to the 'edge' of the network - the CDN's edge servers. Note that ESR is more a deployment target than an actual rendering mode.

When a request for a page is made, instead of going all the way to the original server, it's intercepted by the nearest edge server. This server generates the HTML for the page and sends it back to the user. This process minimizes the physical distance the data has to travel, **reducing latency and loading the page faster**.

Edge-side rendering is possible thanks to Nitro, the server engine that powers Nuxt 3. It offers cross-platform support for Node.js, Deno, Cloudflare Workers, and more.

The current platforms where you can leverage ESR are:

- Cloudflare Pages with zero configuration using the git integration and the `nuxt build` command
- Lagon using the `NITRO_PRESET=lagon npx nuxt build` command
- Vercel Edge Functions using the `nuxt build` command and `NITRO_PRESET=vercel-edge` environment variable

- Netlify Edge Functions using the `nuxt build` command and `NITRO_PRESET=netlify-edge` environment variable

Note that **Hybrid Rendering** can be used when using Edge-Side Rendering with route rules.

You can explore open source examples deployed on some of the platform mentioned above:

- Nuxt Todos Edge: A todos application with user authentication, SSR and SQLite.
- Atinotes: An editable website with universal rendering.

Edit on Github