Static site generation (SSG)

In architecture, we mentioned that the theme is run in Webpack. But beware: that doesn't mean it always has access to browser globals! The theme is built twice:

- During **server-side rendering**, the theme is compiled in a sandbox called React DOM Server. You can see this as a "headless browser", where there is no `window` or `document`, only React. SSR produces static HTML pages.
- During **client-side rendering**, the theme is compiled to JavaScript that gets eventually executed in the browser, so it has access to browser variables.

SSR OR SSG?

Server-side rendering and *static site generation* can be different concepts, but we use them interchangeably.

Strictly speaking, Docusaurus is a static site generator, because there's no server-side runtime—we statically render to HTML files that are deployed on a CDN, instead of dynamically pre-rendering on each request. This differs from the working model of Next.js.

Therefore, while you probably know not to access Node globals like `process` (or can we?) or the `'fs'` module, you can't freely access browser globals either.

```
import React from 'react';

export default function WhereAmI() {
  return <span>{window.location.href}</span>;
}
```

This looks like idiomatic React, but if you run `docusaurus build`, you will get an error:

```
ReferenceError: window is not defined
```

This is because during server-side rendering, the Docusaurus app isn't actually run in browser, and it doesn't know what `window` is.

▪ What about `process.env.NODE_ENV`?

Understanding SSR

React is not just a dynamic UI runtime—it's also a templating engine. Because Docusaurus sites mostly contain static contents, it should be able to work without any JavaScript (which React runs in), but only plain HTML/CSS. And that's what server-side rendering offers: statically rendering your React code into HTML, without any dynamic content. An HTML file has no concept of client state (it's purely markup), hence it shouldn't rely on browser APIs.

These HTML files are the first to arrive at the user's browser screen when a URL is visited (see routing). Afterwards, the browser fetches and runs other JS code to provide the "dynamic" parts of your site—anything implemented with JavaScript. However, before that, the main content of your page is already visible, allowing faster loading.

In CSR-only apps, all DOM elements are generated on client side with React, and the HTML file only ever contains one root element for React to mount DOM to; in SSR, React is already facing a fully built HTML page, and it only needs to correlate the DOM elements with the virtual DOM in its model. This step is called "hydration". After React has hydrated the static markup, the app starts to work as any normal React app.

Note that Docusaurus is ultimately a single-page application, so static site generation is only an optimization (*progressive enhancement*, as it's called), but our functionality does not fully depend on those HTML files. This is contrary to site generators like Jekyll and Docusaurus v1, where all files are statically transformed to markup, and interactiveness is added through external

JavaScript linked with `<script>` tags. If you inspect the build output, you will still see JS assets under `build/assets/js`, which are, really, the core of Docusaurus.

Escape hatches

If you want to render any dynamic content on your screen that relies on the browser API to be functional at all, for example:

- Our live codeblock, which runs in the browser's JS runtime
- Our themed image that detects the user's color scheme to display different images
- The JSON viewer of our debug panel which uses the `window` global for styling

You may need to escape from SSR since static HTML can't display anything useful without knowing the client state.

⚠ CAUTION

It is important for the first client-side render to produce the exact same DOM structure as server-side rendering, otherwise, React will correlate virtual DOM with the wrong DOM elements.

Therefore, the naïve attempt of `if (typeof window !== 'undefined) {/* render something */}` won't work appropriately as a browser vs. server detection, because the first client render would instantly render different markup from the server-generated one.

You can read more about this pitfall in The Perils of Rehydration.

We provide several more reliable ways to escape SSR.

`<BrowserOnly>`

If you need to render some component in browser only (for example, because the component relies on browser specifics to be functional at all), one common approach is to wrap your component with `<BrowserOnly>` to make sure it's invisible during SSR and only rendered in CSR.

```
import BrowserOnly from '@docusaurus/BrowserOnly';

function MyComponent(props) {
  return (
    <BrowserOnly fallback={<div>Loading...</div>}>
      {() => {
        const LibComponent =
          require('some-lib-that-accesses-window').LibComponent;
        return <LibComponent {...props} />;
      }}
    </BrowserOnly>
  );
}
```

It's important to realize that the children of `<BrowserOnly>` is not a JSX element, but a function that *returns* an element. This is a design decision. Consider this code:

```
import BrowserOnly from '@docusaurus/BrowserOnly';

function MyComponent() {
  return (
    <BrowserOnly>
      {/* DON'T DO THIS - doesn't actually work */}
      <span>page url = {window.location.href}</span>
    </BrowserOnly>
  );
}
```

While you may expect that `BrowserOnly` hides away the children during server-side rendering, it actually can't. When the React renderer tries to render this JSX tree, it does see the `{window.location.href}` variable as a node of this tree and tries to render it, although it's actually not used! Using a function ensures that we only let the renderer see the browser-only component when it's needed.

`useIsBrowser`

You can also use the `useIsBrowser()` hook to test if the component is currently in a browser environment. It returns `false` in SSR and `true` is CSR, after first client render. Use this hook if you only need to perform certain conditional operations on client-side, but not render an entirely different UI.

```
import useIsBrowser from '@docusaurus/useIsBrowser';

function MyComponent() {
  const isBrowser = useIsBrowser();
  const location = isBrowser ? window.location.href : 'fetching location...';
  return <span>{location}</span>;
}
```

`useEffect`

Lastly, you can put your logic in `useEffect()` to delay its execution until after first CSR. This is most appropriate if you are only performing side-effects but don't *get* data from the client state.

```
function MyComponent() {
  useEffect(() => {
    // Only logged in the browser console; nothing is logged during server-side rendering
    console.log("I'm now in the browser");
  }, []);
  return <span>Some content...</span>;
}
```

`ExecutionEnvironment`

The `ExecutionEnvironment` namespace contains several values, and `canUseDOM` is an effective way to detect browser environment.

Beware that it essentially checked `typeof window !== 'undefined'` under the hood, so you should not use it for rendering-related logic, but only imperative code, like reacting to user input by sending web requests, or dynamically importing libraries, where DOM isn't updated at all.

a-client-module.js
```
import ExecutionEnvironment from '@docusaurus/ExecutionEnvironment';

if (ExecutionEnvironment.canUseDOM) {
  document.title = "I'm loaded!";
}
```

Edit this page
*Last updated on **Mar 23, 2023** by **Sébastien Lorber***

Previous
« Routing

Next
Client architecture »

Learn

Introduction

Installation

Migration from v1 to v2

Community

Stack Overflow

Feature Requests

Discord

Help

More

Blog

Changelog

GitHub

Twitter

Legal

Privacy

Terms

Data Policy

Cookie Policy