Server-side Rendering in Vue.js

What exactly is

Sever-side Rendering?

Actually, nothing new.

It’s been done for years.

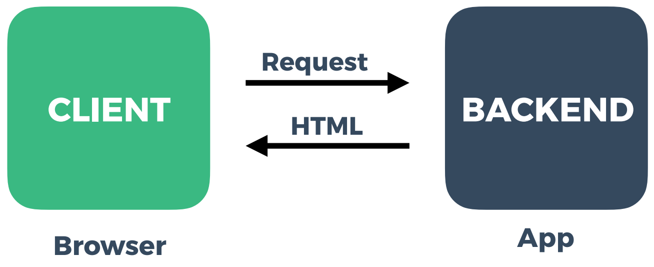
Ruby on Rails

Django

Laravel

And hundreds of other backend frameworks

Classic SSR



CLIENT

Browser

Request

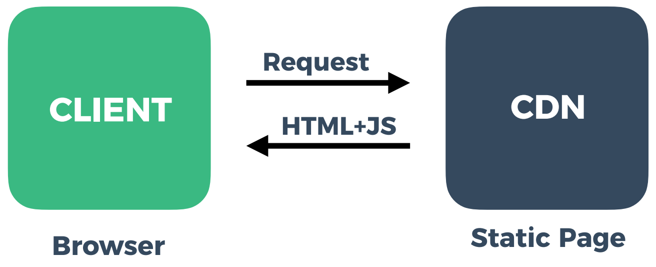
HTML

BACKEND

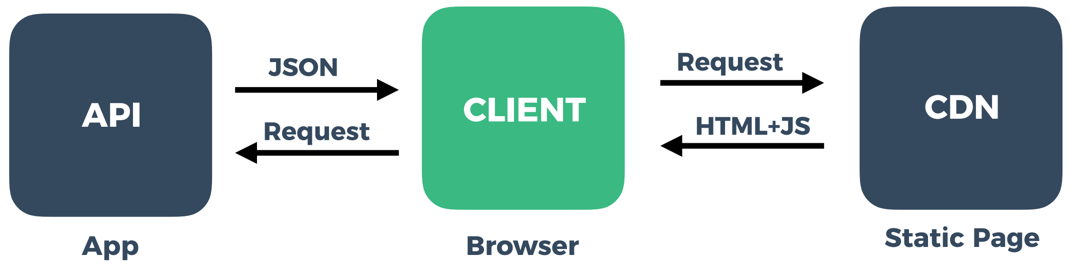
App

But we want Single Page Apps.

Typical SPA



Typical SPA



API

JSON

Request

CLIENT

Request

HTML+JS

CDN

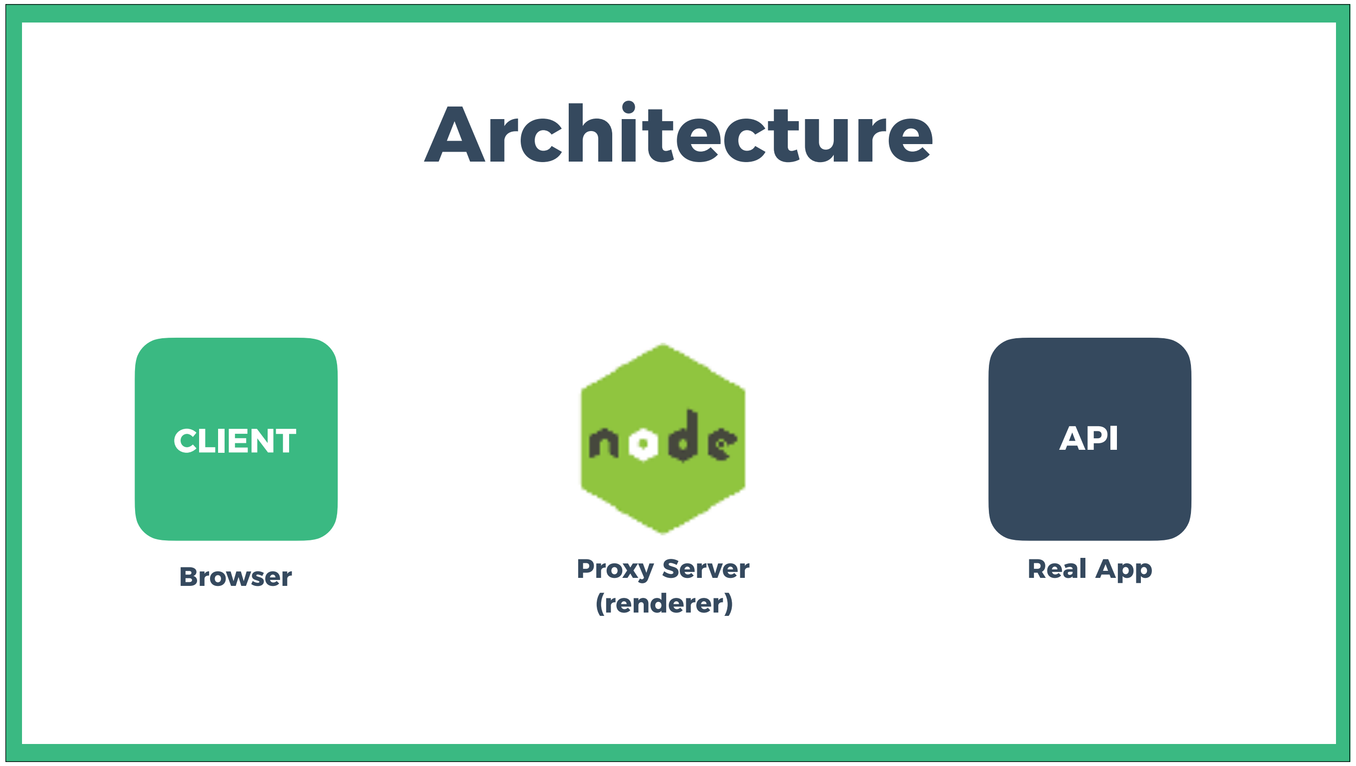
App

Browser

Static Page

Once JavaScript has been parsed and executed.

Server-side rendering to the rescue



Architecture

CLIENT

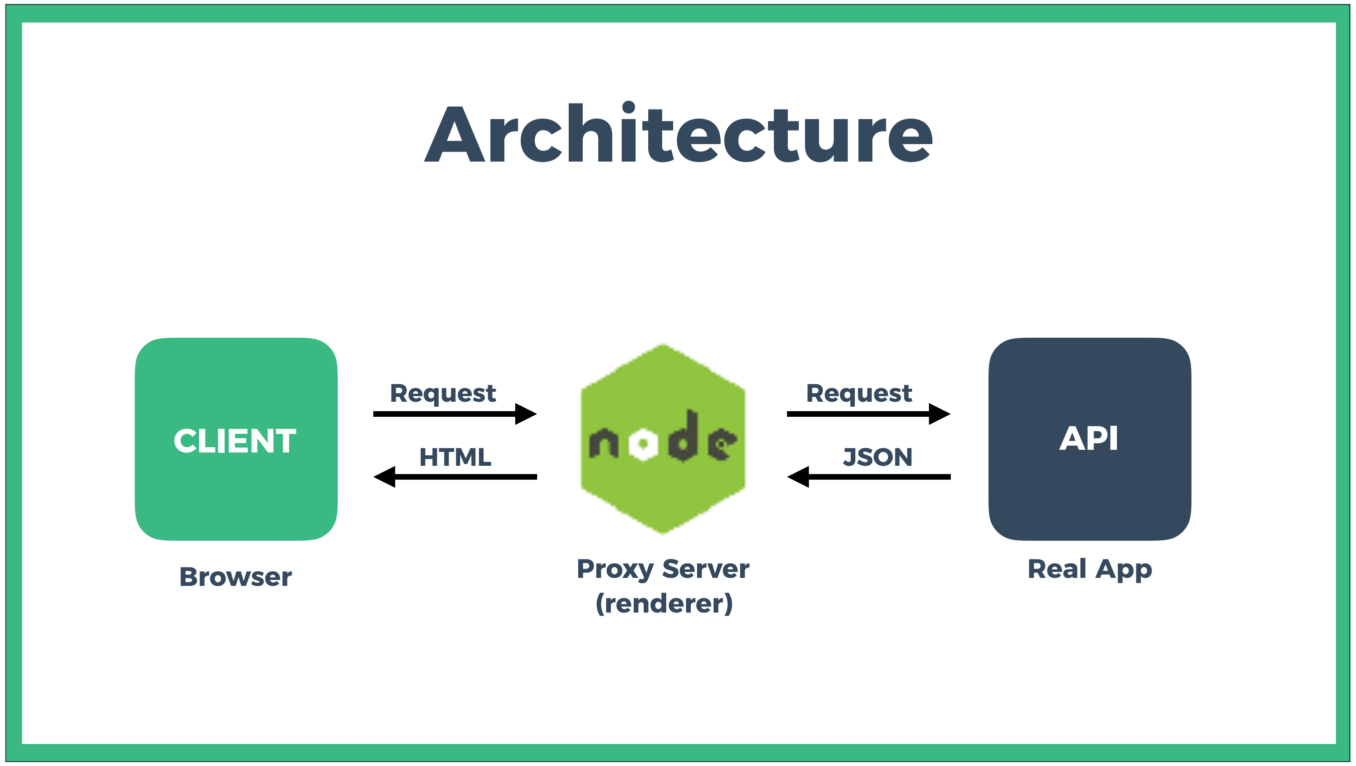
Browser

Proxy Server
(renderer)

API

Real App

To make it work we need 3 elements.
A browser (client), a proxy Node.js app and the actual backend API.



Architecture

CLIENT

Browser

Request

HTML

Proxy Server
(renderer)

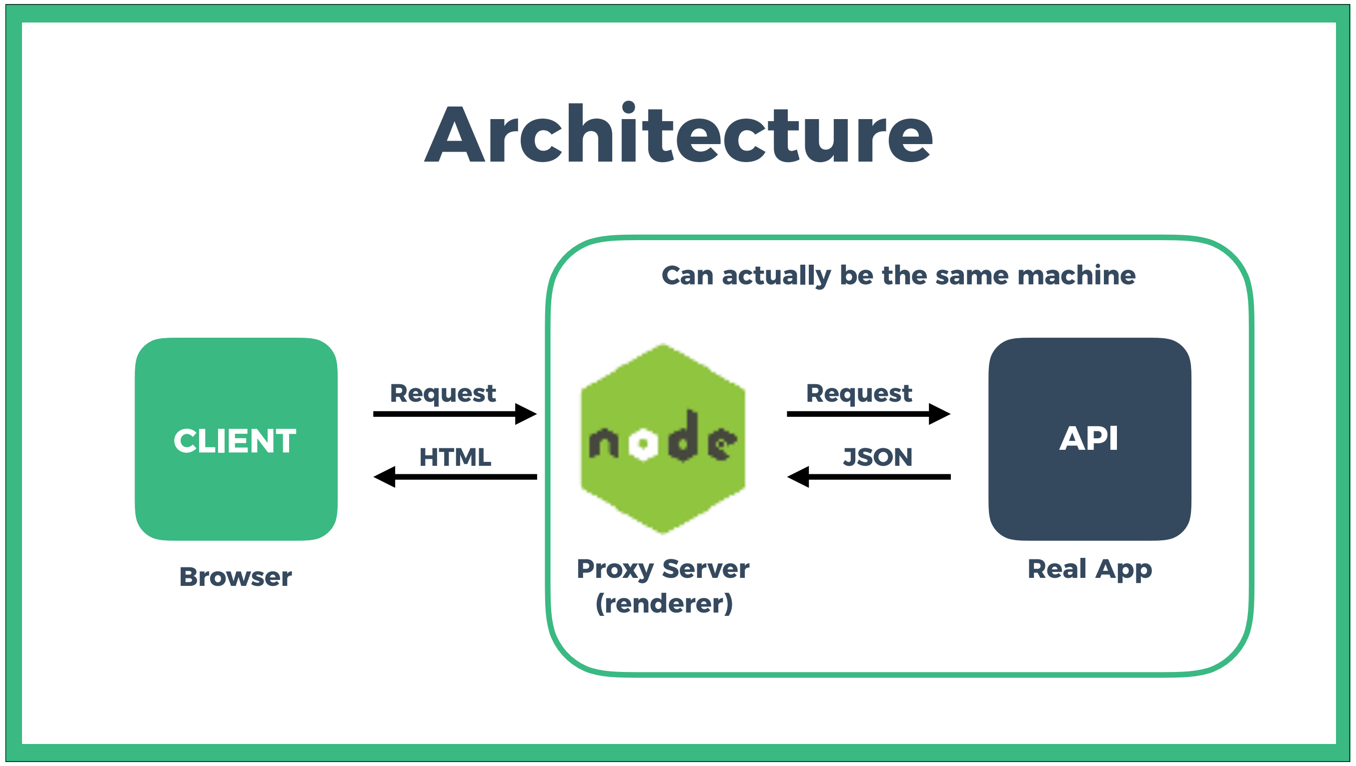
Request

JSON

API

Real App

As the name suggests, the idea is to return a fully rendered HTML page to the client. Without any spinners.



Architecture

Can actually be the same machine

CLIENT

Browser

Request

HTML

Proxy Server (renderer)

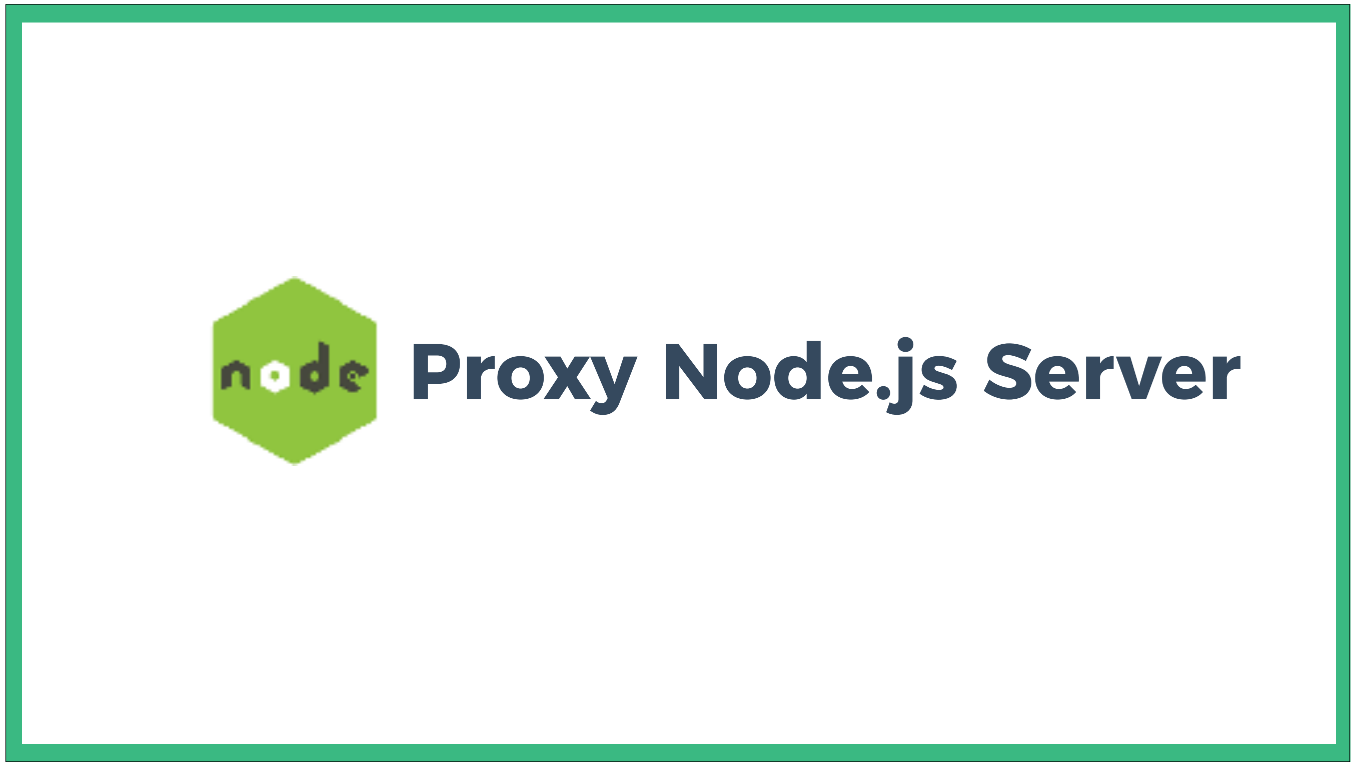
Request

JSON

API

Real App

There is nothing preventing you from having it all on the same machine. This will actually reduce the latency which is a good thing. But then you’re usually stuck with using JS for your backend.

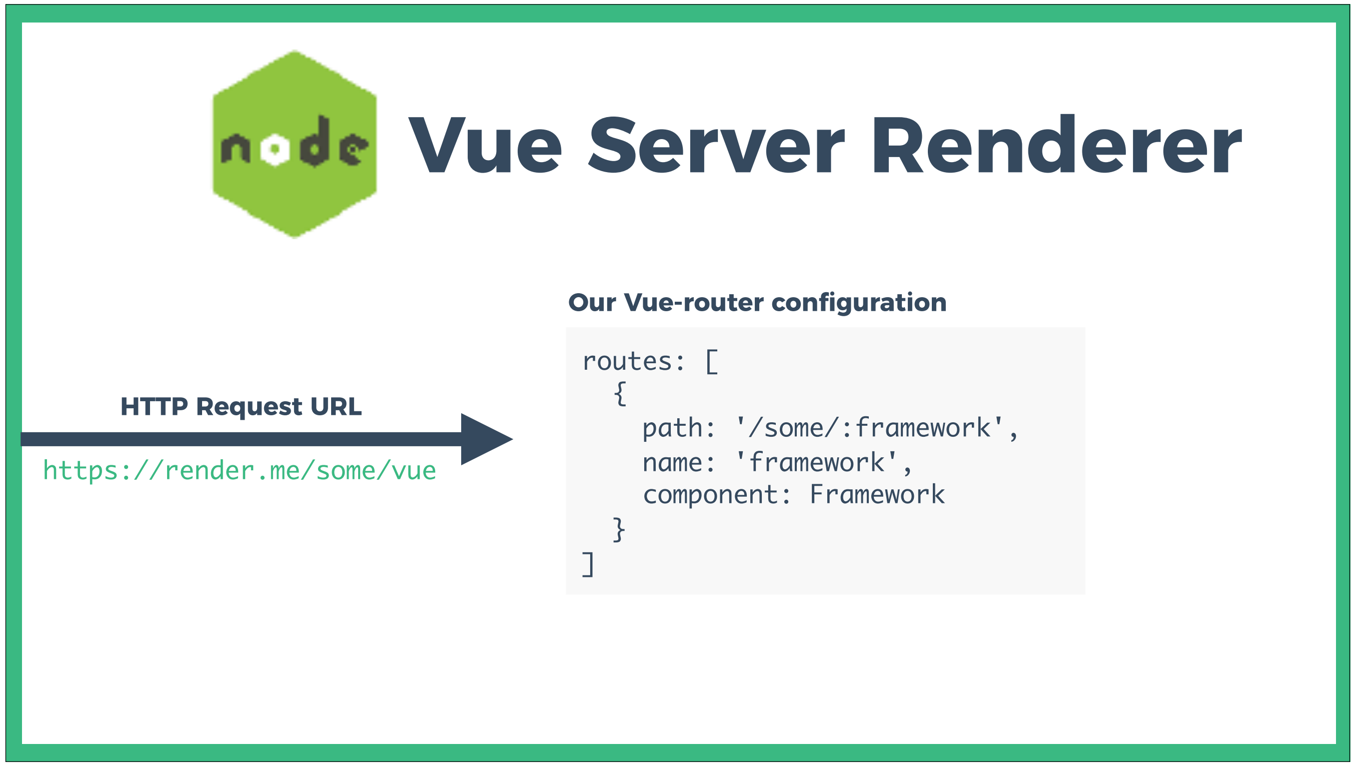


Proxy Node.js Server

The client and the API are pretty straightforward. Let’s look into the Node app though.

Vue Server Renderer

Usually it runs a minimal http server, for example Express.js that will use the vue-server-renderer as the rendering engine. It requires some configuration, although we will skip that.



Vue Server Renderer

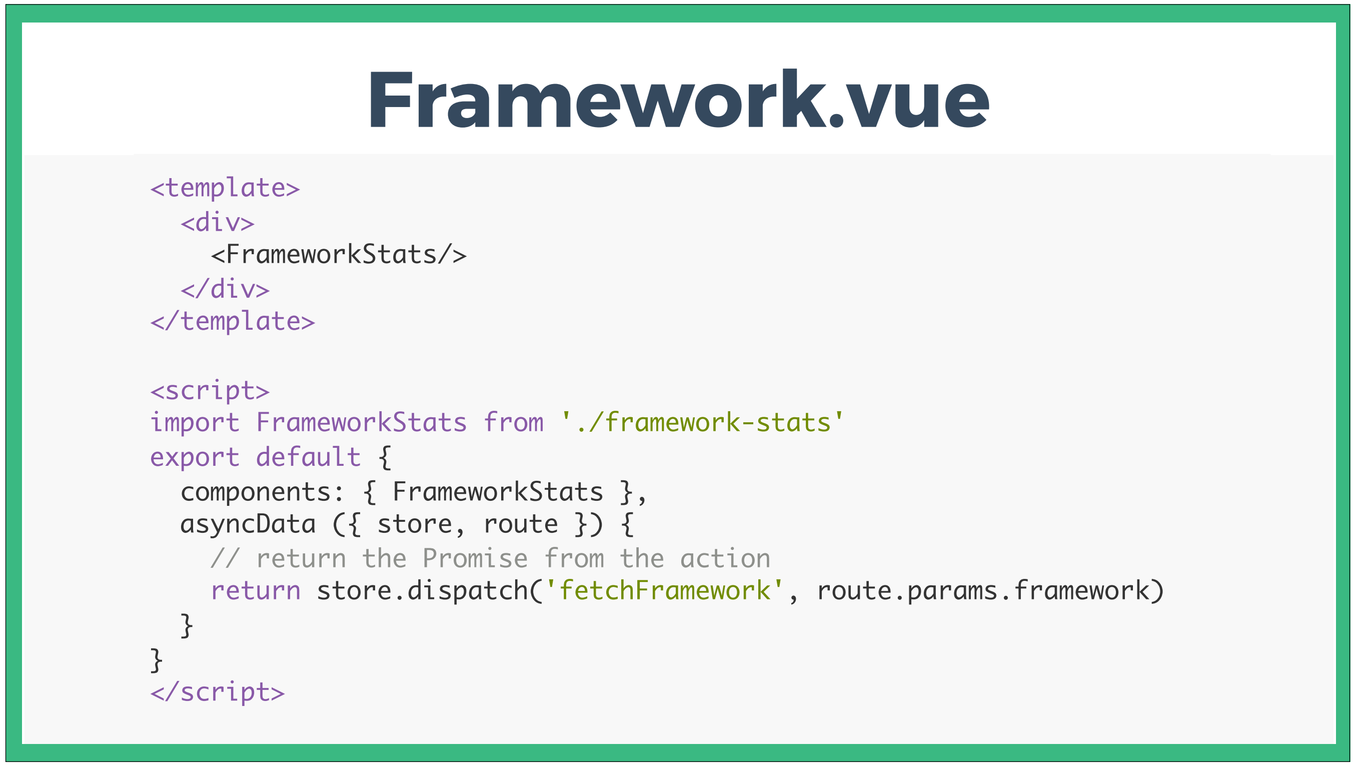
HTTP Request URL

https://render.me/some/vue

Our Vue-router configuration

```
routes: [
  {
    path: '/some/:framework',
    name: 'framework',
    component: Framework
  }
]
```

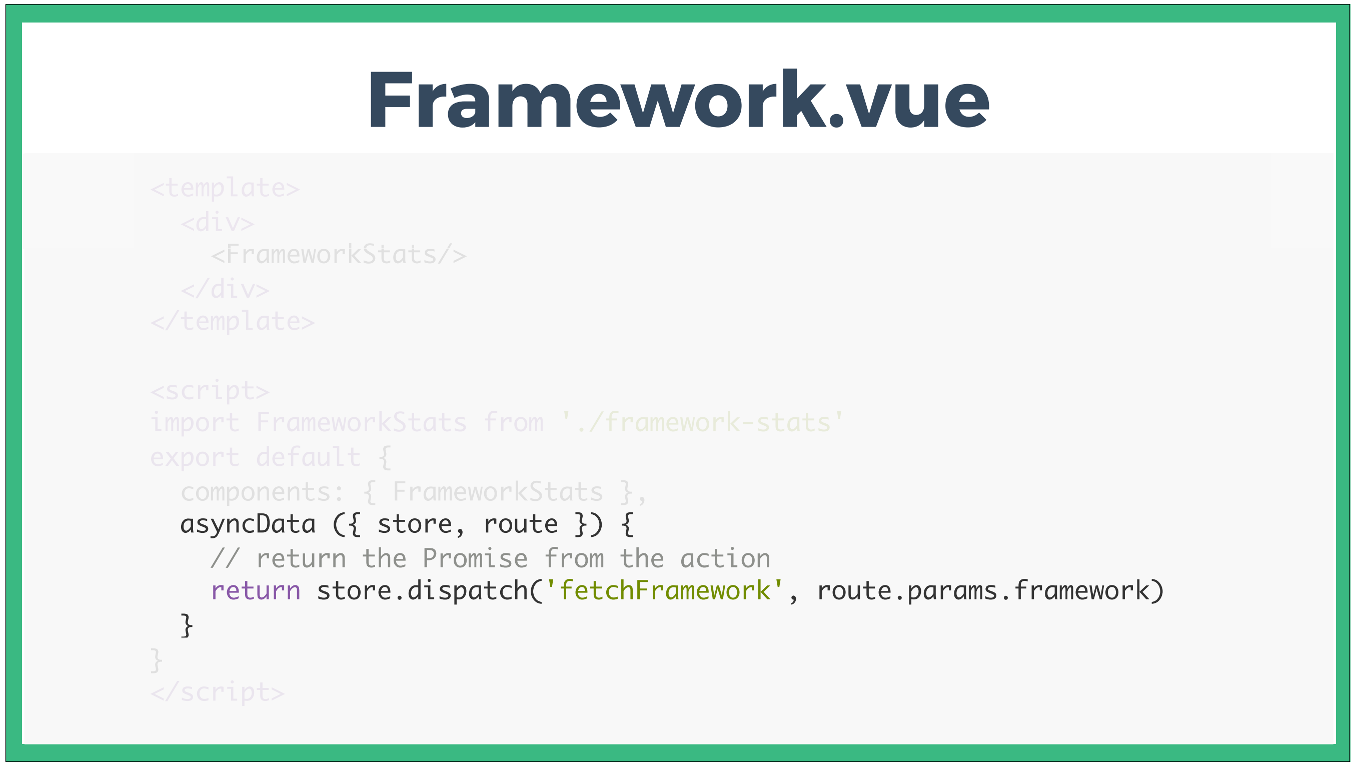
Let’s say we get an request for the some/vue page. What our app should be doing now is to locate all route components involved in rendering the page. In this case it’s just the Framework component and the Index component. However it’s just the Framework components that has to do some API calls.
Let’s look into that component.



Framework.vue

```
<template>
  <div>
    <FrameworkStats/>
  </div>
</template>

<script>
import FrameworkStats from './framework-stats'
export default {
  components: { FrameworkStats },
  asyncData ({ store, route }) {
    // return the Promise from the action
    return store.dispatch('fetchFramework', route.params.framework)
  }
}
</script>
```

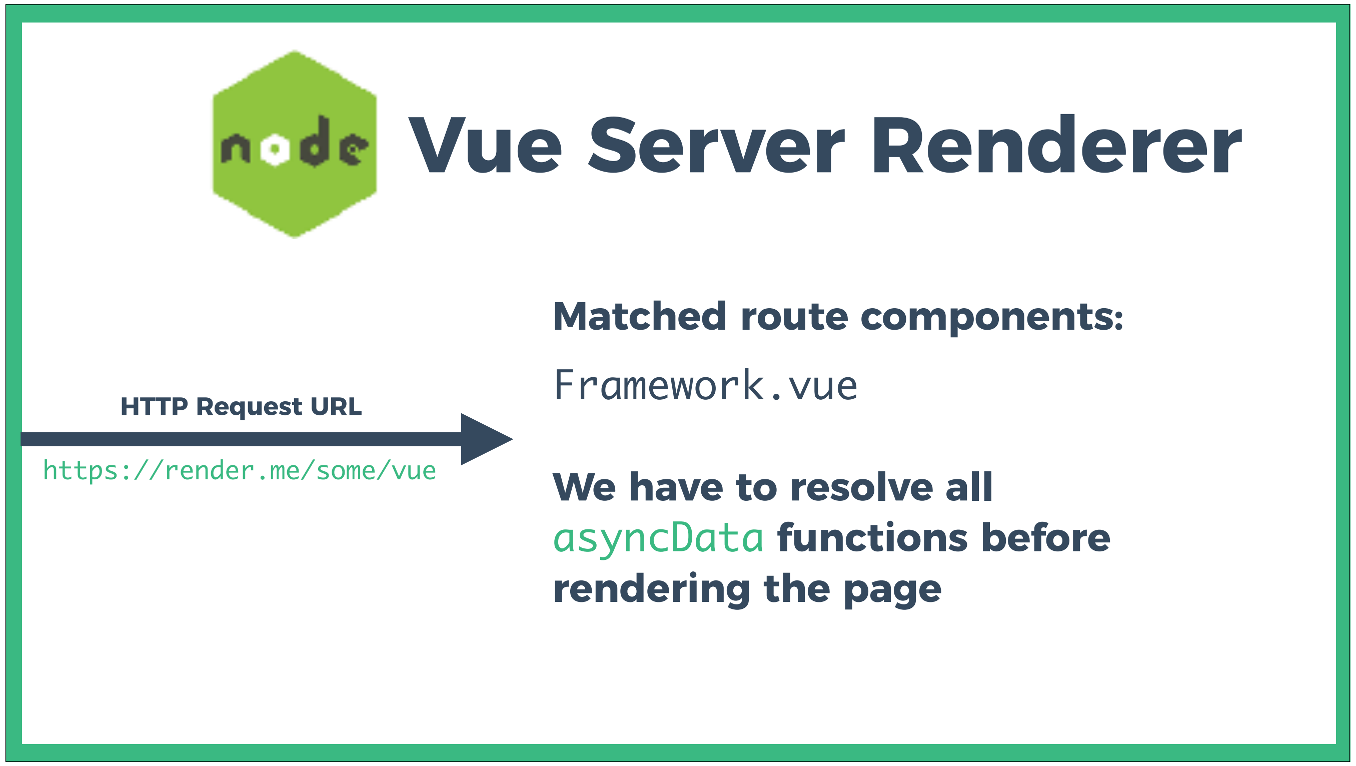
Hopefully you did your home work and got familiar with how Vue components look like. Nothing unusual here besides asyncData.



Framework.vue

```
<template>
  <div>
    <FrameworkStats/>
  </div>
</template>

<script>
import FrameworkStats from './framework-stats'
export default {
  components: { FrameworkStats },
  asyncData ({ store, route }) {
    // return the Promise from the action
    return store.dispatch('fetchFramework', route.params.framework)
  }
}
</script>
```

This is new and is used specifically for SSR and is the place where you will be doing all your async requests. You can call Vuex actions here or make direct calls to the API. You can even use async/await. The important thing is that it has to return a promise which can also resolve to an object that will be assigned to the data model. The promise can also resolve to data that will be added to the components data model.



Vue Server Renderer

HTTP Request URL

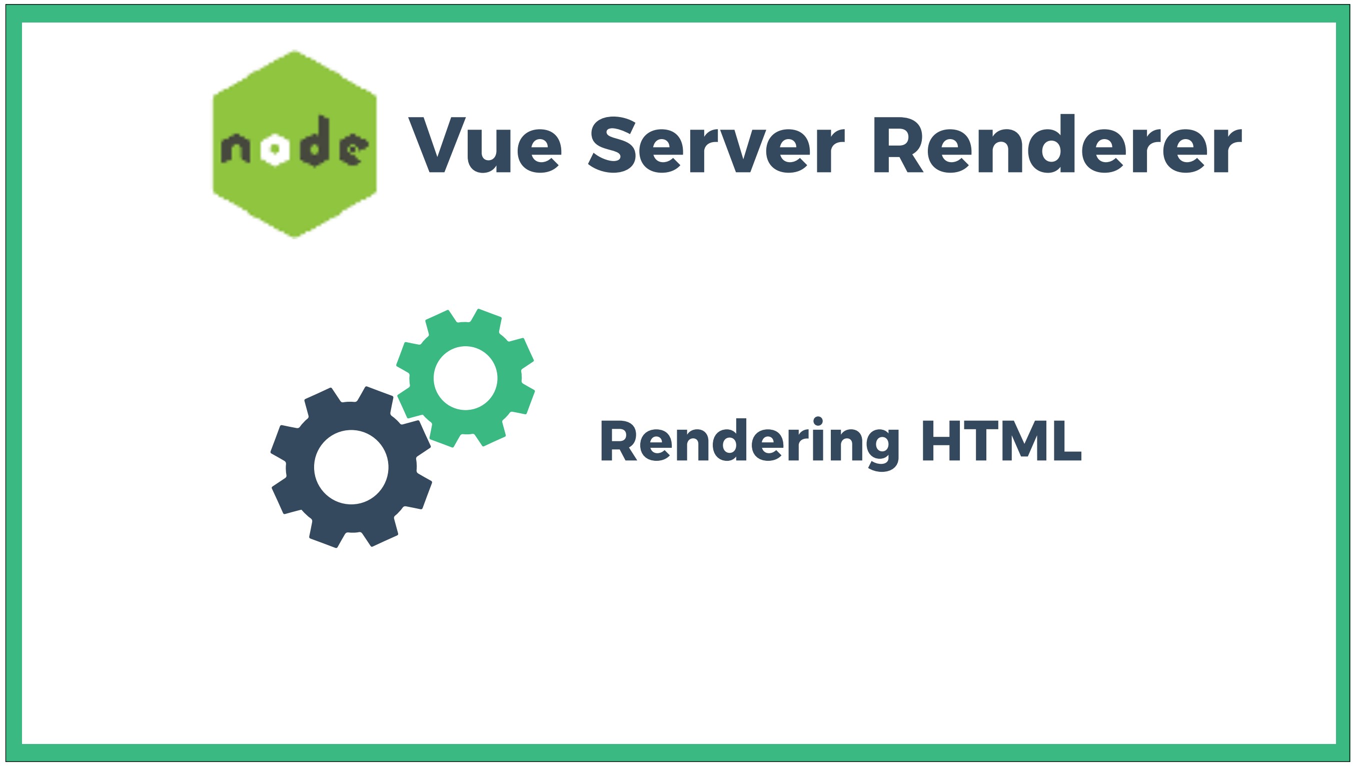
`https://render.me/some/vue`

Matched route components:

`Framework.vue`

We have to resolve all `asyncData` functions before rendering the page

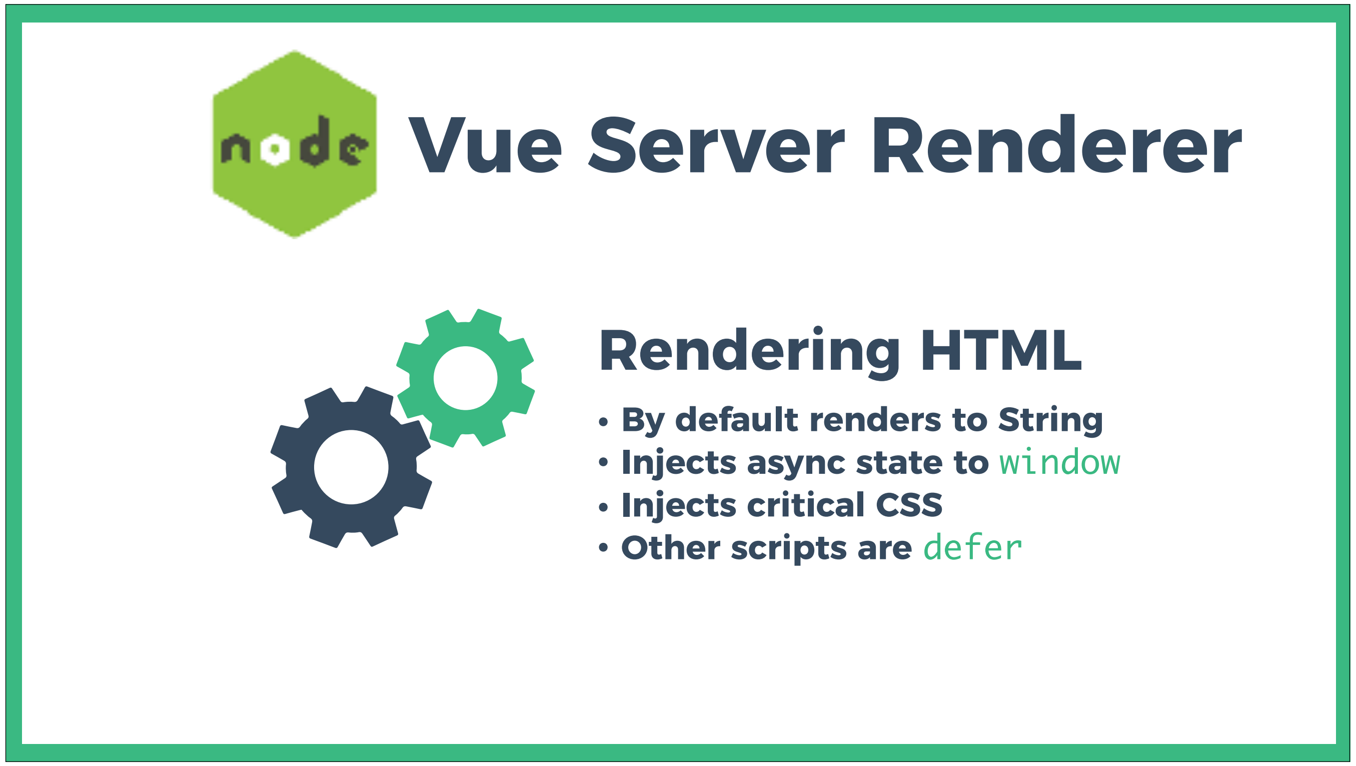
We know the the insides of the component. We have to go through all those asyncData functions and resolve those before we start rendering the App.



Vue Server Renderer

Rendering HTML

Now that we have all the required data, we can process with rendering the page. Vue uses some smart optimisations here like component-based caching, avoiding using virtual DOM when possible and using string concatenation instead. Using templates helps with the latter!



Vue Server Renderer

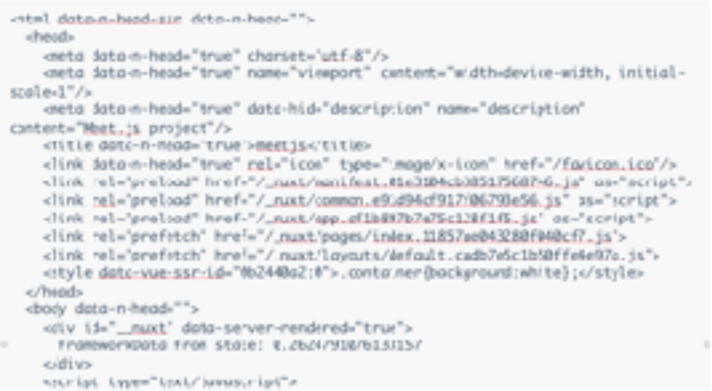
Rendering HTML

- By default renders to String
- Injects async state to `window`
- Injects critical CSS
- Other scripts are `defer`

The output is a HTML string. All async component and Vuex store data is injected into window through an injected script tag.
So are all the styles needed to render the page. All other scripts including Vue are added with the defer attribute.
This prevents them from blocking the rendering.
Fun fact: when on a slow connection it might happen that that some elements (like buttons) won’t be interactive because the scripts have not yet loaded.

CLIENT

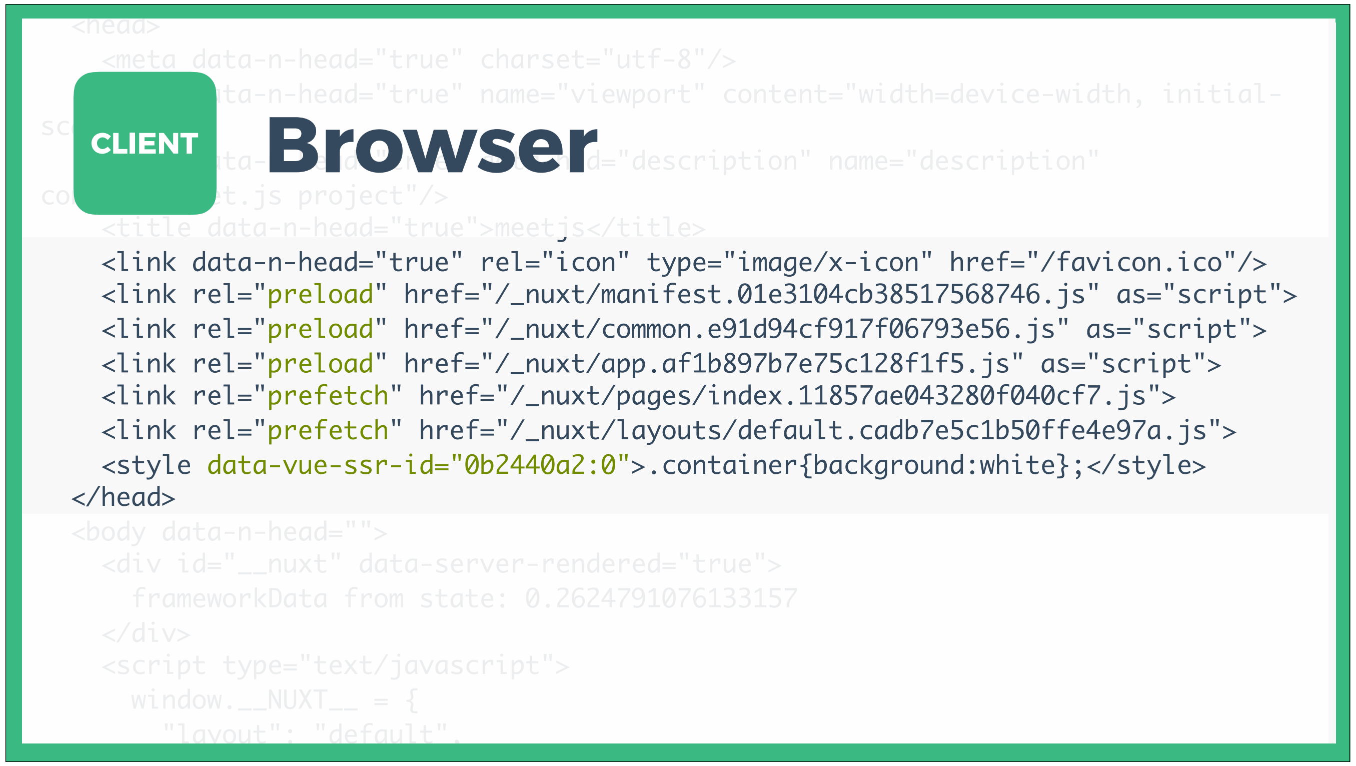
Browser



```
<html data-n-head-ssr data-n-head="">
  <head>
    <meta data-n-head="true" charset="utf-8"/>
    <meta data-n-head="true" name="viewport" content="width=device-width, initial-
scale=1"/>
    <meta data-n-head="true" data-hid="description" name="description"
content="Meet.js project"/>
    <title data-n-head="true">meetjs</title>
    <link data-n-head="true" rel="icon" type="image/x-icon" href="/favicon.ico"/>
    <link rel="preload" href="/_nuxt/manifest.81e3104cb3851756078-6.js" as="script">
    <link rel="preload" href="/_nuxt/common.e9cd94cf9170067930e56.js" as="script">
    <link rel="preload" href="/_nuxt/app.cf1b897b7a75c128f1f5.js" as="script">
    <link rel="prefetch" href="/_nuxt/pages/index.11857ae043280f040cf7.js">
    <link rel="prefetch" href="/_nuxt/layouts/default.cadb7e5c1b50ffe4e97a.js">
    <style data-vue-ssr-id="0b2440a2:0">.container{background:white};</style>
  </head>
  <body data-n-head="">
    <div id="__nuxt" data-server-rendered="true">
      frameworkData from state: 0.2624791076153157
    </div>
    <script type="text/javascript">
```

Response (HTML page)

This is what we get. There are some additional attributes added so let’s take a closer look.

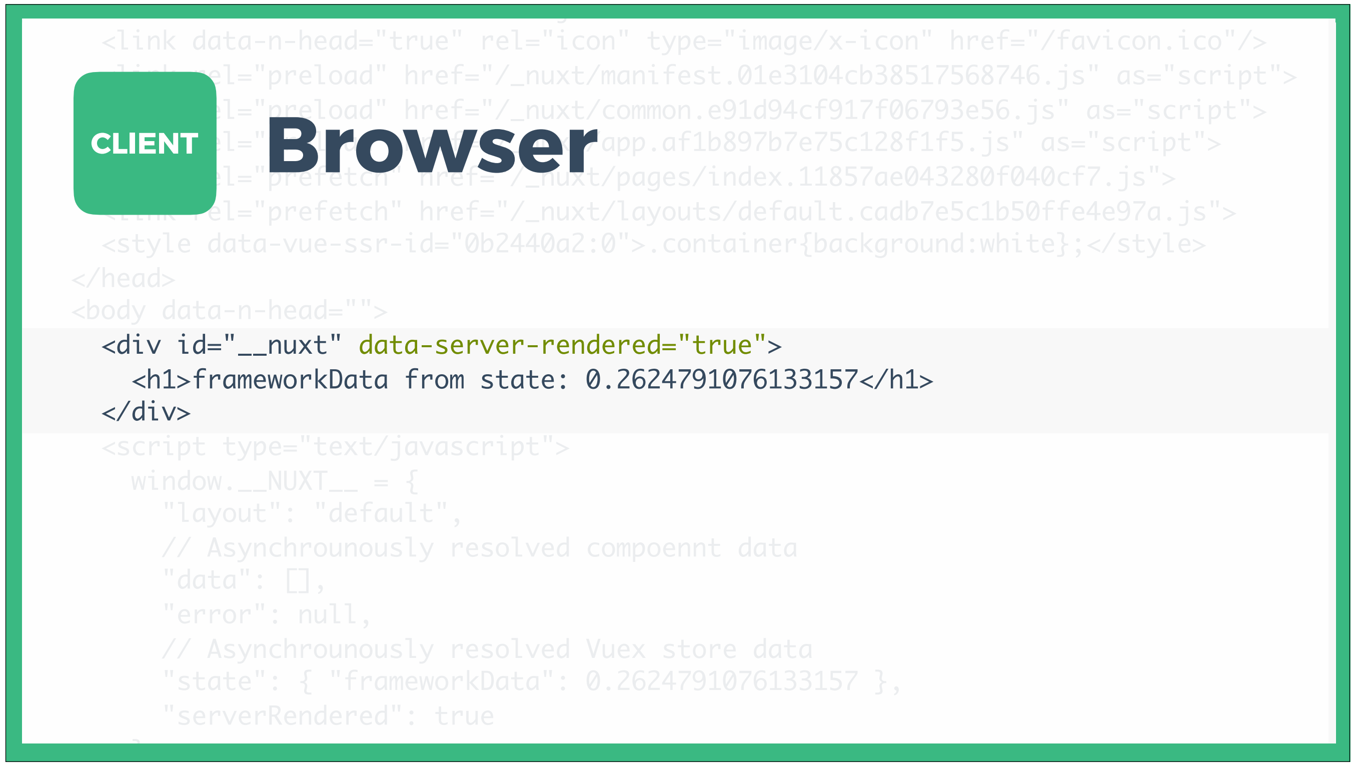


CLIENT

Browser

```
<head>
  <meta data-n-head="true" charset="utf-8"/>
       ta-n-head="true" name="viewport" content="width=device-width, initial-
sc
       ta-                              d="description" name="description"
co                t.js project"/>
  <title data-n-head="true">meetjs</title>
  <link data-n-head="true" rel="icon" type="image/x-icon" href="/favicon.ico"/>
  <link rel="preload" href="/_nuxt/manifest.01e3104cb38517568746.js" as="script">
  <link rel="preload" href="/_nuxt/common.e91d94cf917f06793e56.js" as="script">
  <link rel="preload" href="/_nuxt/app.af1b897b7e75c128f1f5.js" as="script">
  <link rel="prefetch" href="/_nuxt/pages/index.11857ae043280f040cf7.js">
  <link rel="prefetch" href="/_nuxt/layouts/default.cadb7e5c1b50ffe4e97a.js">
  <style data-vue-ssr-id="0b2440a2:0">.container{background:white};</style>
</head>
<body data-n-head="">
  <div id="__nuxt" data-server-rendered="true">
    frameworkData from state: 0.2624791076133157
  </div>
  <script type="text/javascript">
    window.__NUXT__ = {
      "layout": "default",
```

Additionally to scripts having the defer attribute, they are also added as link tags and marked for preload.
We can also see the critical CSS being injected in the head.

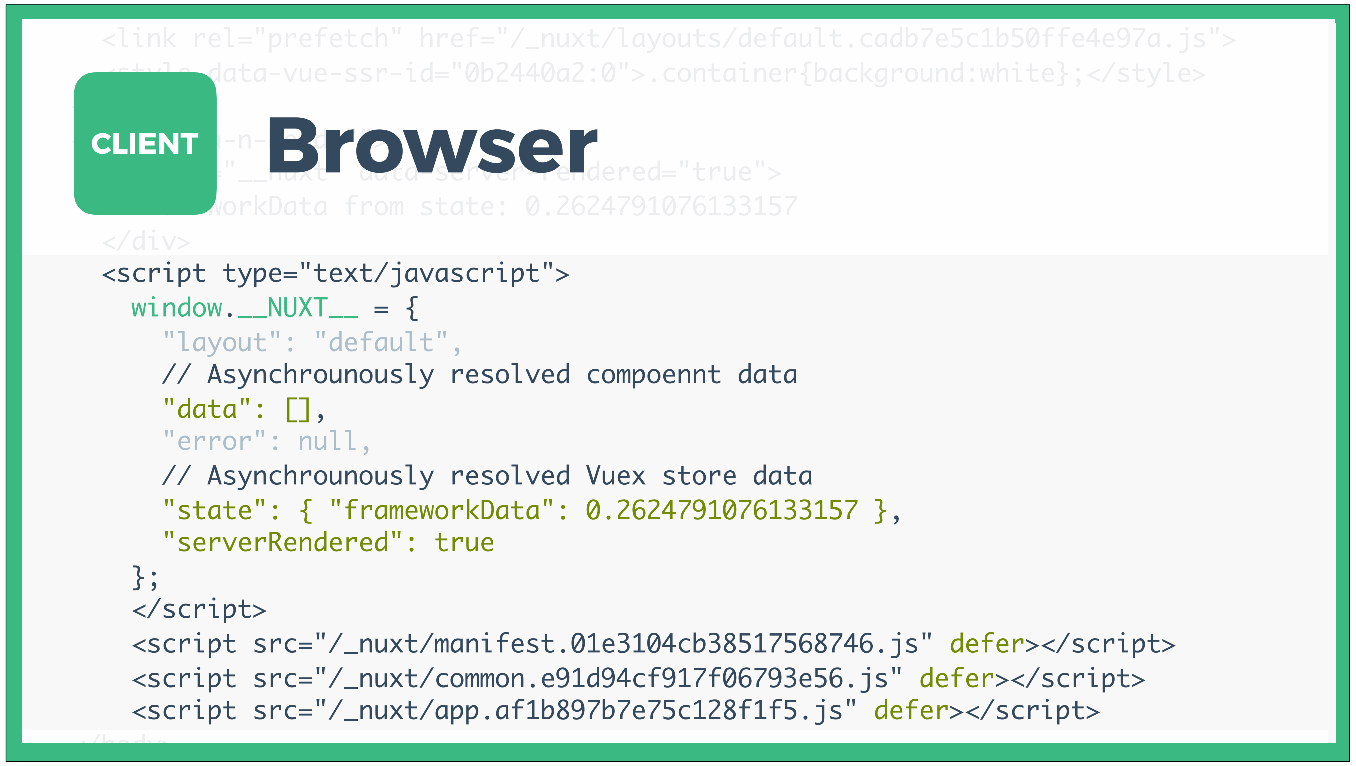


CLIENT

Browser

```
  <link data-n-head="true" rel="icon" type="image/x-icon" href="/favicon.ico"/>
       el="preload" href="/_nuxt/manifest.01e3104cb38517568746.js" as="script">
       l="preload" href="/_nuxt/common.e91d94cf917f06793e56.js" as="script">
       l="                     app.af1b897b7e75c128f1f5.js" as="script">
       l="prefetch" href="/_nuxt/pages/index.11857ae043280f040cf7.js">
  <link rel="prefetch" href="/_nuxt/layouts/default.cadb7e5c1b50ffe4e97a.js">
  <style data-vue-ssr-id="0b2440a2:0">.container{background:white};</style>
</head>
<body data-n-head="">
  <div id="__nuxt" data-server-rendered="true">
    <h1>frameworkData from state: 0.2624791076133157</h1>
  </div>
  <script type="text/javascript">
    window.__NUXT__ = {
      "layout": "default",
      // Asynchrounously resolved compoennt data
      "data": [],
      "error": null,
      // Asynchrounously resolved Vuex store data
      "state": { "frameworkData": 0.2624791076133157 },
      "serverRendered": true
```

This is app content. The header is what our component has rendered. What’s important here is the ‘data-server-rendered=“true”’ attribute. It indicates that the content of the node has been rendered server side. We will get back to this later.

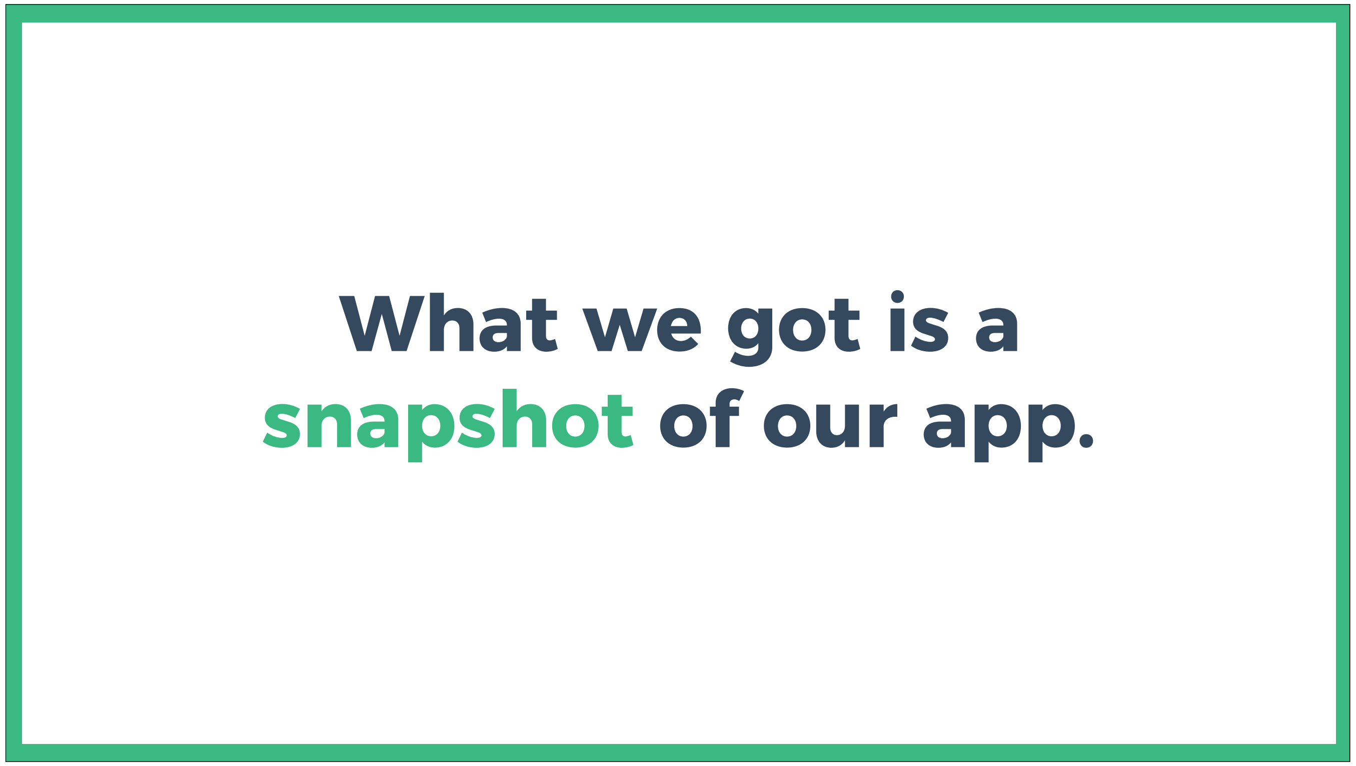


CLIENT

Browser

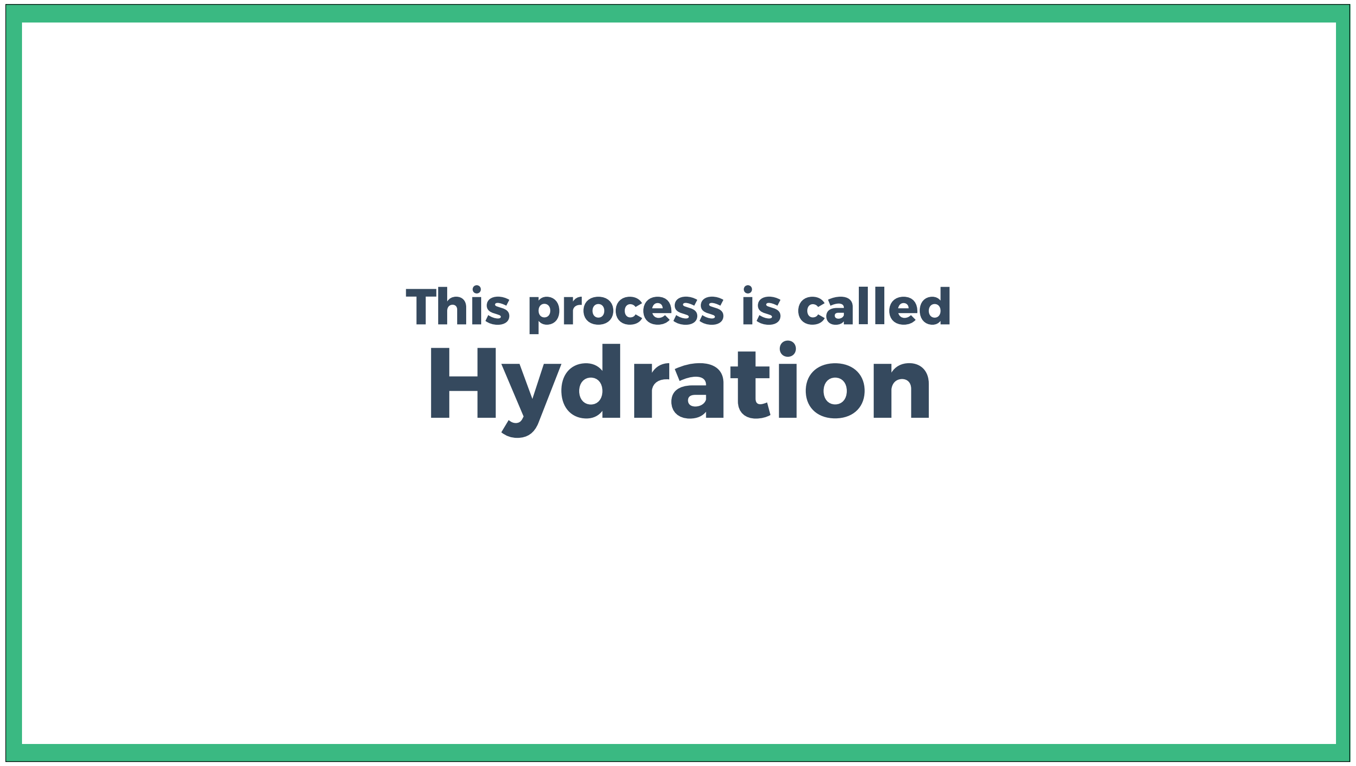
```
<link rel="prefetch" href="/_nuxt/layouts/default.cadb7e5c1b50ffe4e97a.js">
<style data-vue-ssr-id="0b2440a2:0">.container{background:white};</style>

     ="__nuxt" data-server-rendered="true">
     workData from state: 0.2624791076133157
</div>
<script type="text/javascript">
  window.__NUXT__ = {
    "layout": "default",
    // Asynchrounously resolved compoennt data
    "data": [],
    "error": null,
    // Asynchrounously resolved Vuex store data
    "state": { "frameworkData": 0.2624791076133157 },
    "serverRendered": true
  };
  </script>
  <script src="/_nuxt/manifest.01e3104cb38517568746.js" defer></script>
  <script src="/_nuxt/common.e91d94cf917f06793e56.js" defer></script>
  <script src="/_nuxt/app.af1b897b7e75c128f1f5.js" defer></script>
```

And this how our vue-server-renderer injects the state it reached during rendering. By having this data, we can later restore the state of our application.
You can also see the ‘defer’ attributes on all other script tags.



What we got is a **snapshot** of our app.

We just have to restart it now

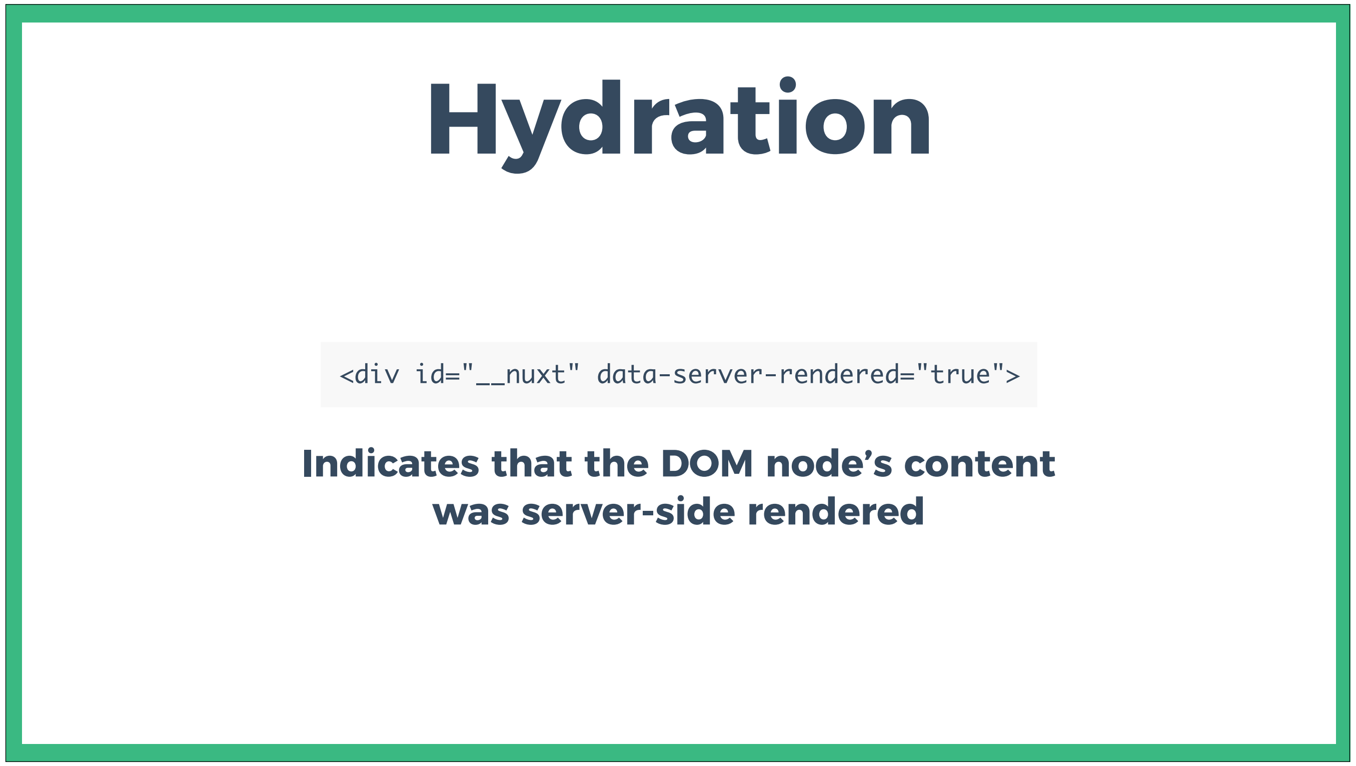


This process is called

Hydration

Hydration

Love this Back to the Future gif.



Hydration

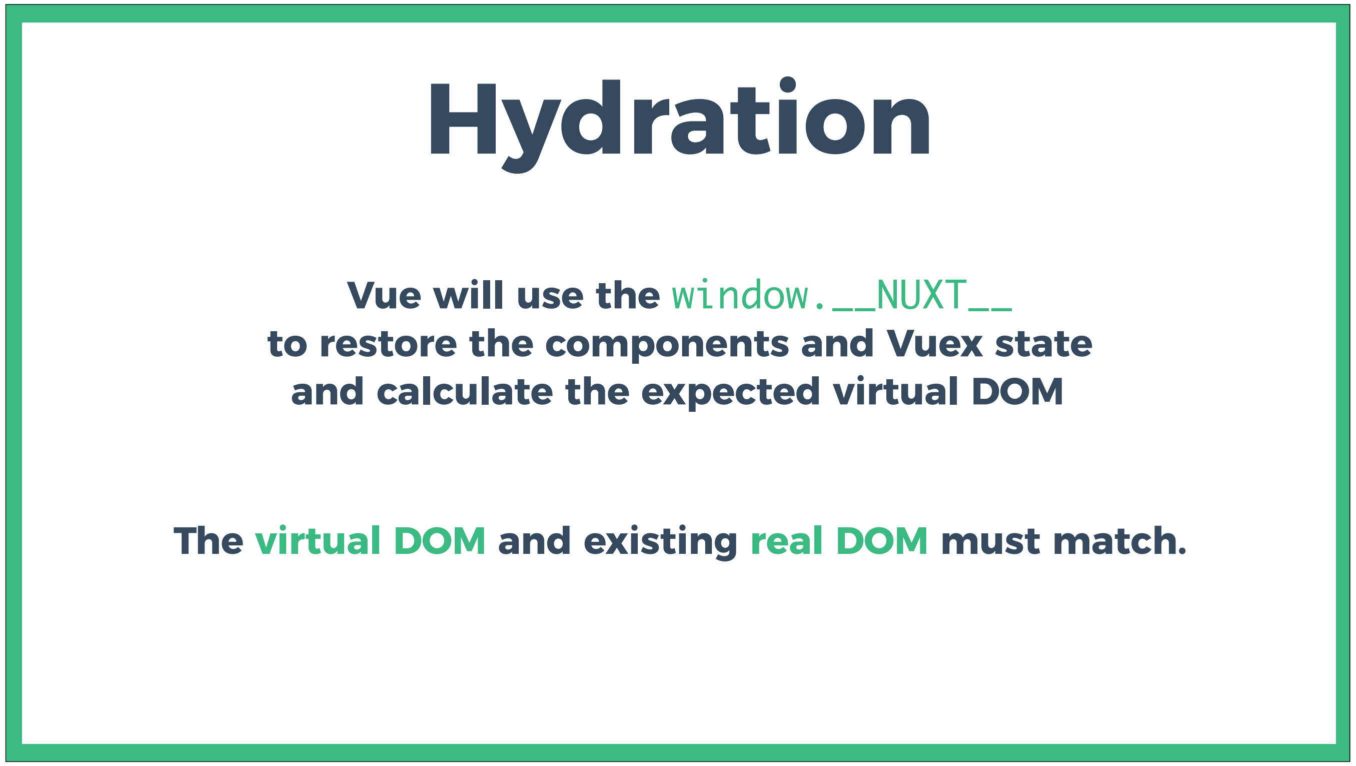
```
<div id="__nuxt" data-server-rendered="true">
```

Indicates that the DOM node’s content was server-side rendered

Once Vue is loaded and tries to mount on “#__nuxt” element, the added attribute will make it work in the hydration mode.

Hydration

**Vue will use the `window.__NUXT__`
to restore the components and Vuex states
and calculate the expected virtual DOM**

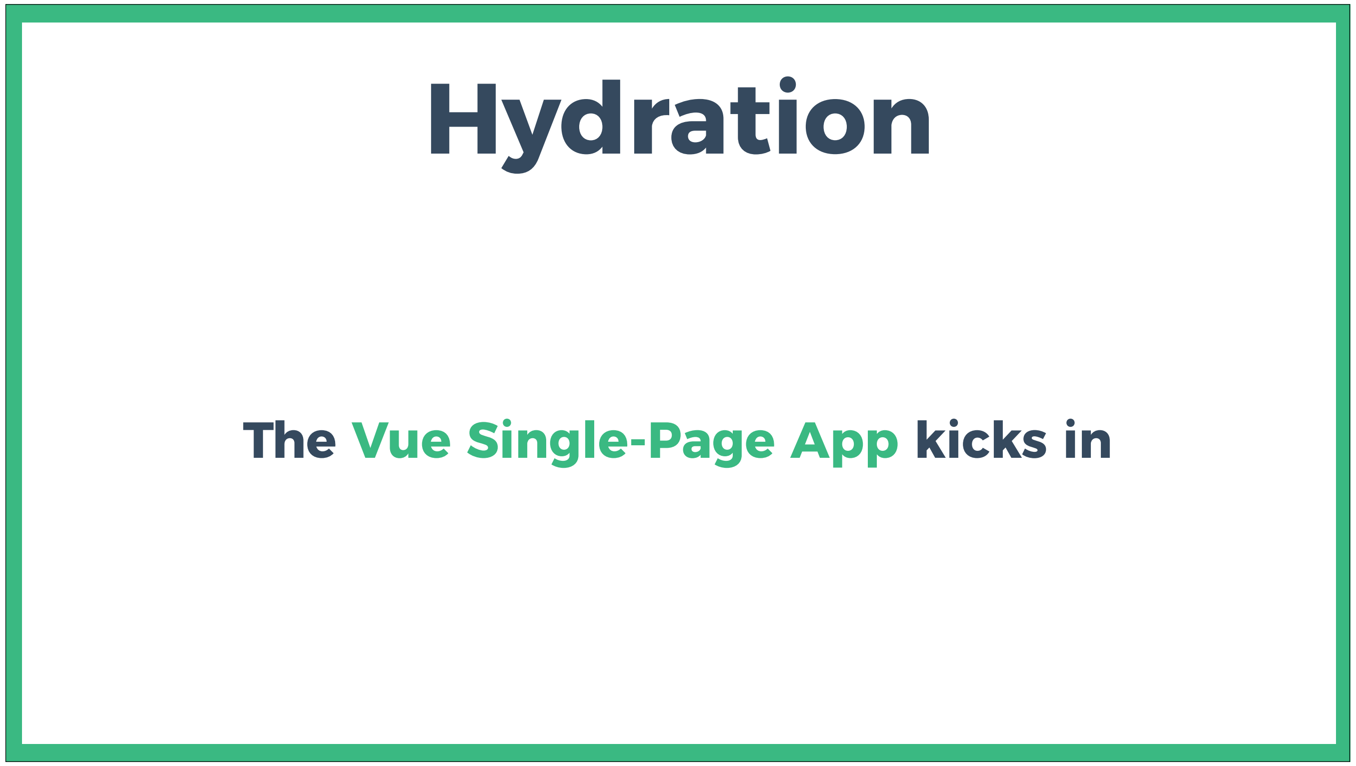


Hydration

**Vue will use the `window.__NUXT__`
to restore the components and Vuex state
and calculate the expected virtual DOM**

The virtual DOM and existing real DOM must match.

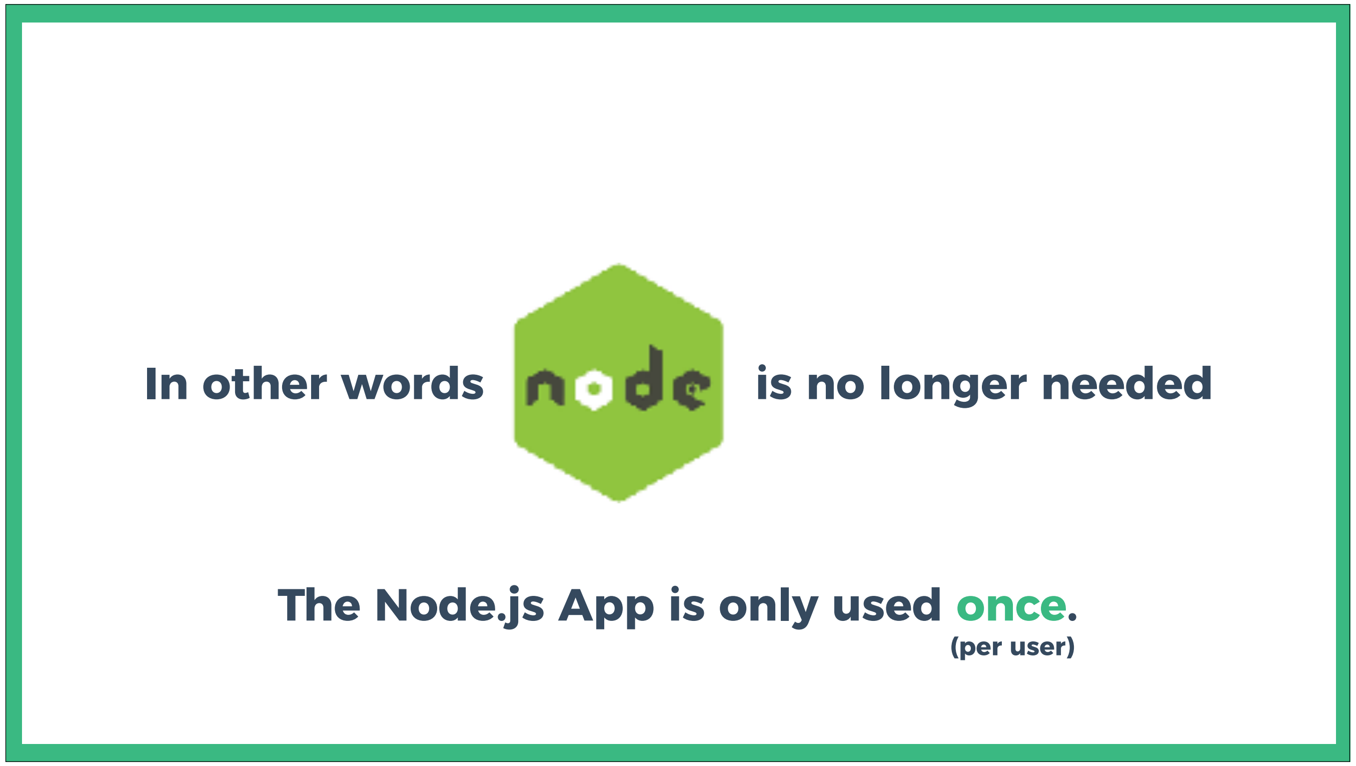
Otherwise Vue will bail out on hydrating and replace the existing DOM. This might cause a flash on the page.
Funny story – watch out for obfuscations services. Those can trigger unexpected content flashing. Same goes to non-determinant things like using Math.random inside your computed properties etc.



Hydration

The Vue Single-Page App kicks in

Anyway, once Vue connects with the DOM the app behaves just like a regular SPA.
All the async requests are done straight to the API.



In other words node is no longer needed

The Node.js App is only used **once**.
(per user)

Unless someone decides to refresh the page of course.

Complicated?

Complicated?

It doesn’t have to be

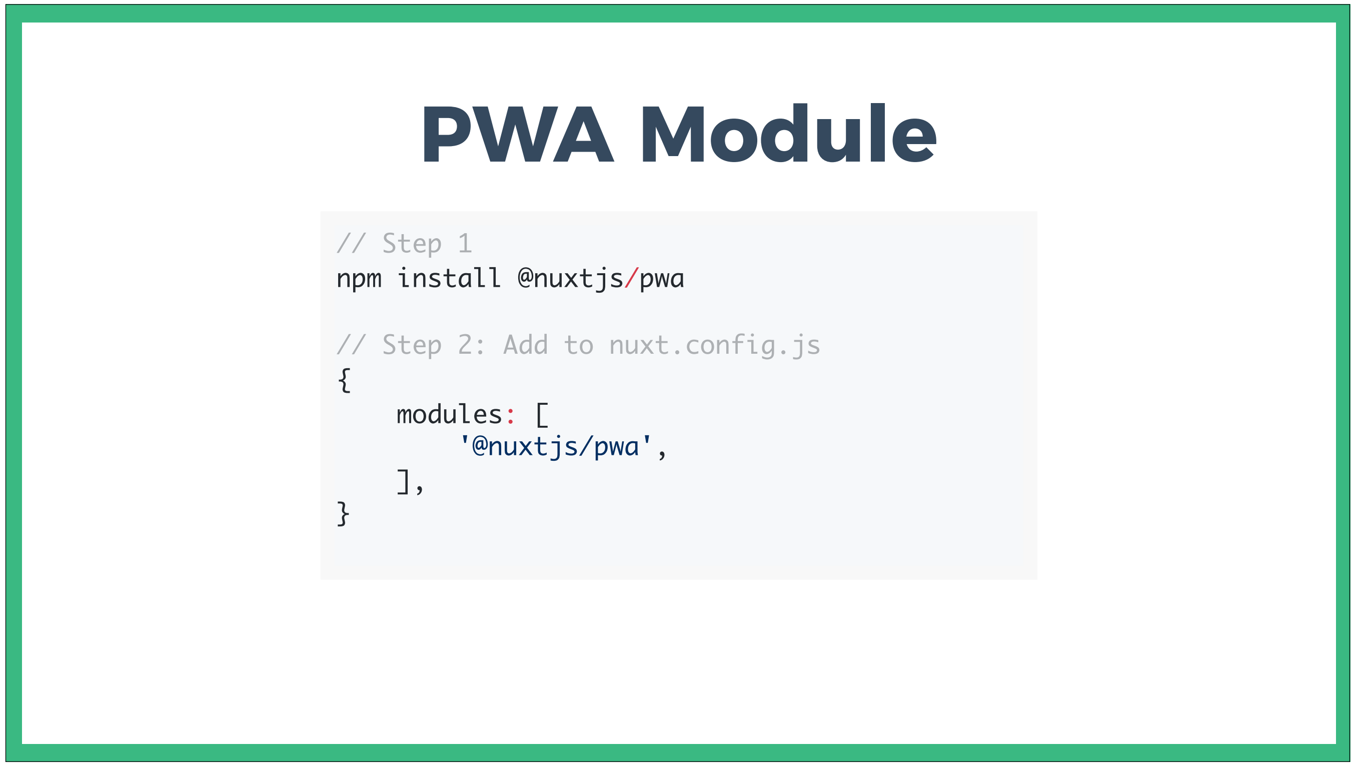
Created by the Chopin brothers and originally inspired by Next.js for React.

Nuxt.js

- **Vue Server Renderer + Server App**
- **Integrated meta tags management (`vue-meta`)**
- **Automatic code splitting**
- **Routing based on file structure (inside `/pages`)**
- **No config required (scaffolded by `vue-cli`)**
- **Extendable with modular architecture**

Nuxt.js Modules

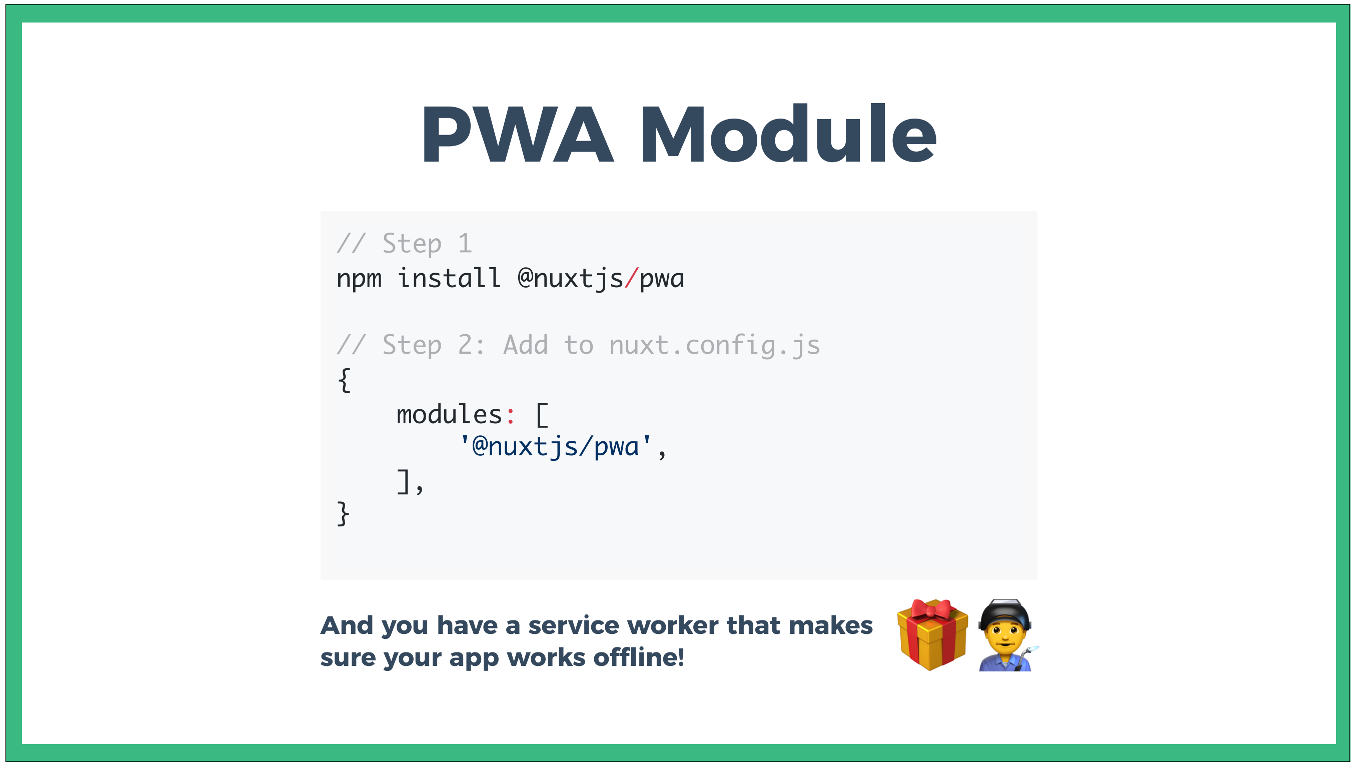
- Progressive Web App
- Sitemaps
- Google Analytics
- OAuth
- Vue-Apollo
- And more at awesome-nuxt



PWA Module

```
// Step 1
npm install @nuxtjs/pwa

// Step 2: Add to nuxt.config.js
{
    modules: [
        '@nuxtjs/pwa',
    ],
}
```

It uses Workbox and you can be further configured.



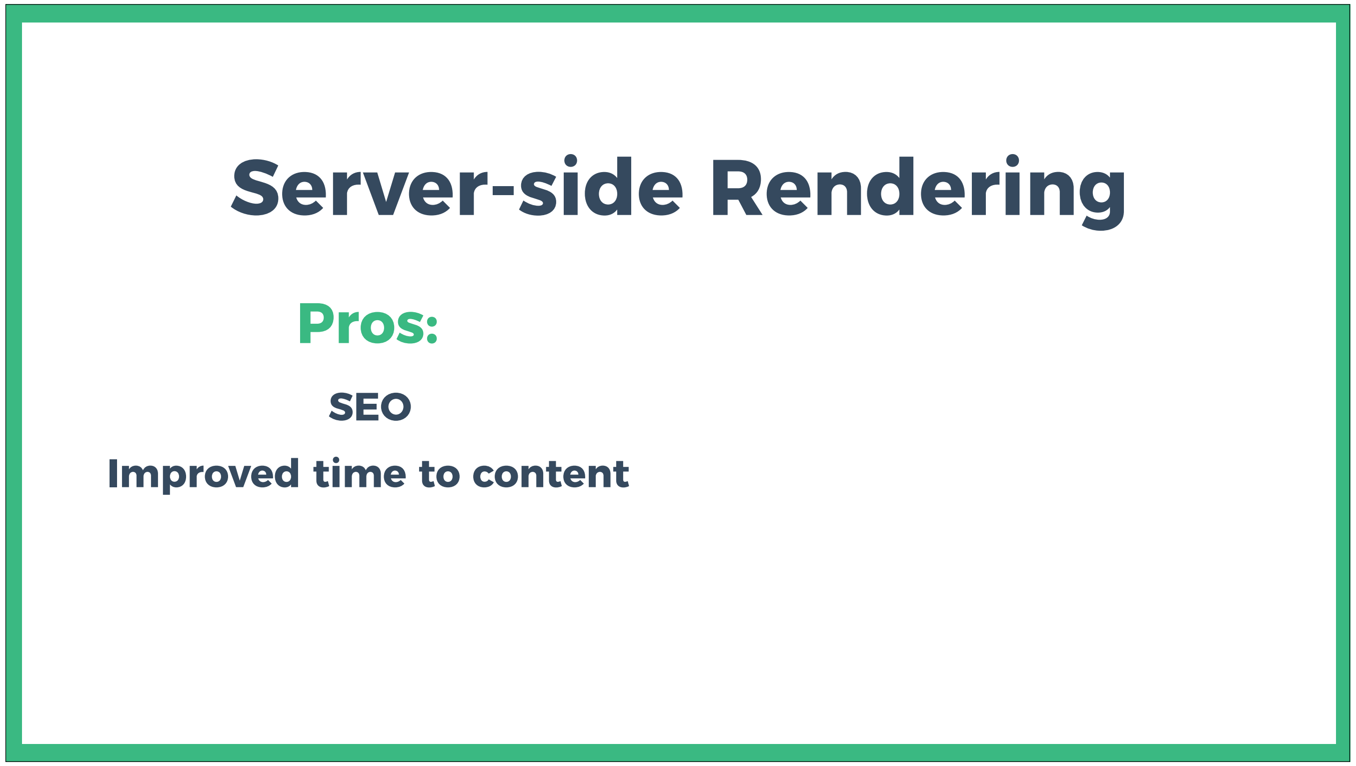
PWA Module

```
// Step 1
npm install @nuxtjs/pwa

// Step 2: Add to nuxt.config.js
{
    modules: [
        '@nuxtjs/pwa',
    ],
}
```

And you have a service worker that makes sure your app works offline! 🎁 👨‍🏭

It uses Workbox and you can be further configured.

That’s fancy and all, but why should I care?



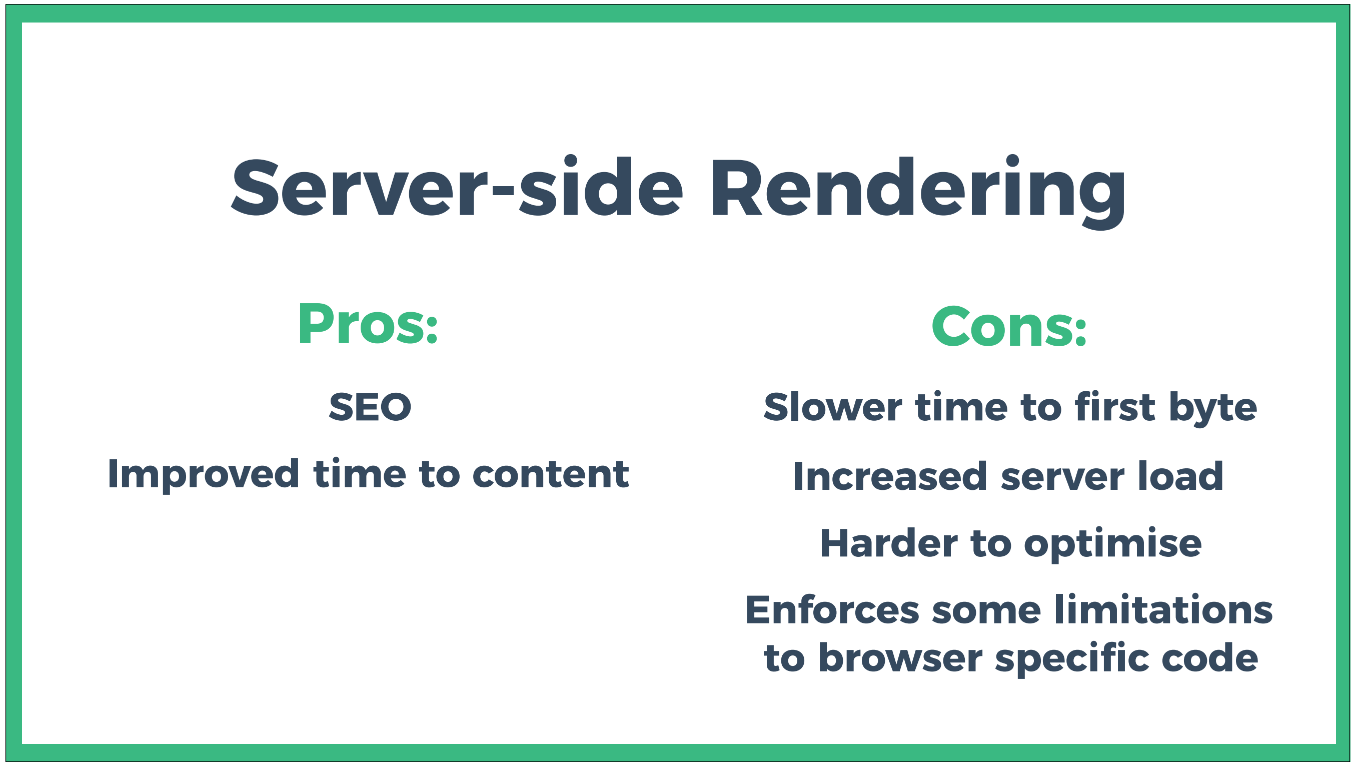
Server-side Rendering

Pros:

SEO

Improved time to content

SEO – this is pretty obvious. If you need it for your product you have to use SSR or some of it’s alternatives.
Time to content – there’s been cases where improving it resulted in much higher conversion rate.



Server-side Rendering

Pros:

SEO

Improved time to content

Cons:

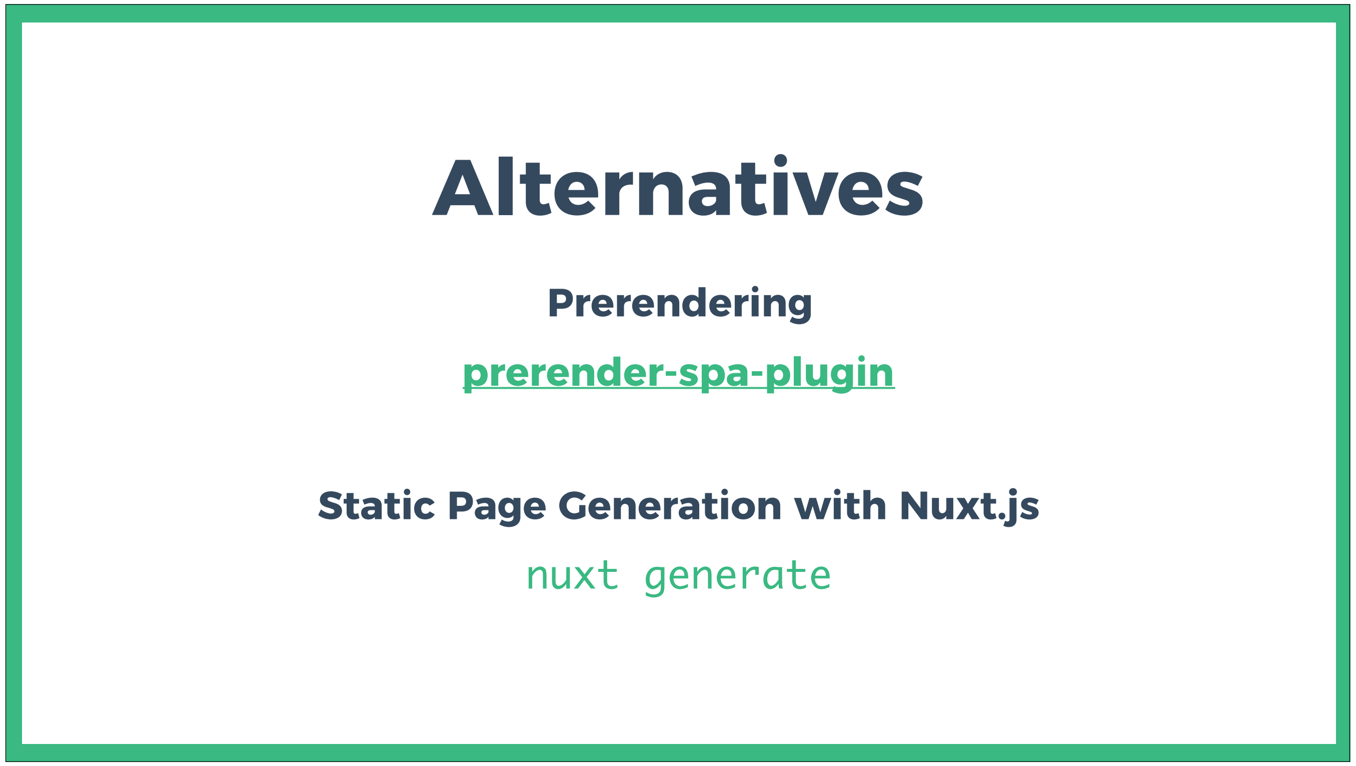
Slower time to first byte

Increased server load

Harder to optimise

Enforces some limitations to browser specific code

1. Because the proxy app has to make a call to the API and back. Can partially solve this with request caching.
2. Compared to serving a static app from CDN
3. To optimise and maintain – hence we got another block in our architecture that we need to take care of.
4. The window object is not existing in the Node env. Worth remembering – the mounted lifecycle hook is not called during SSR. Move your window/browser dependant code calls there.



Alternatives

Prerendering

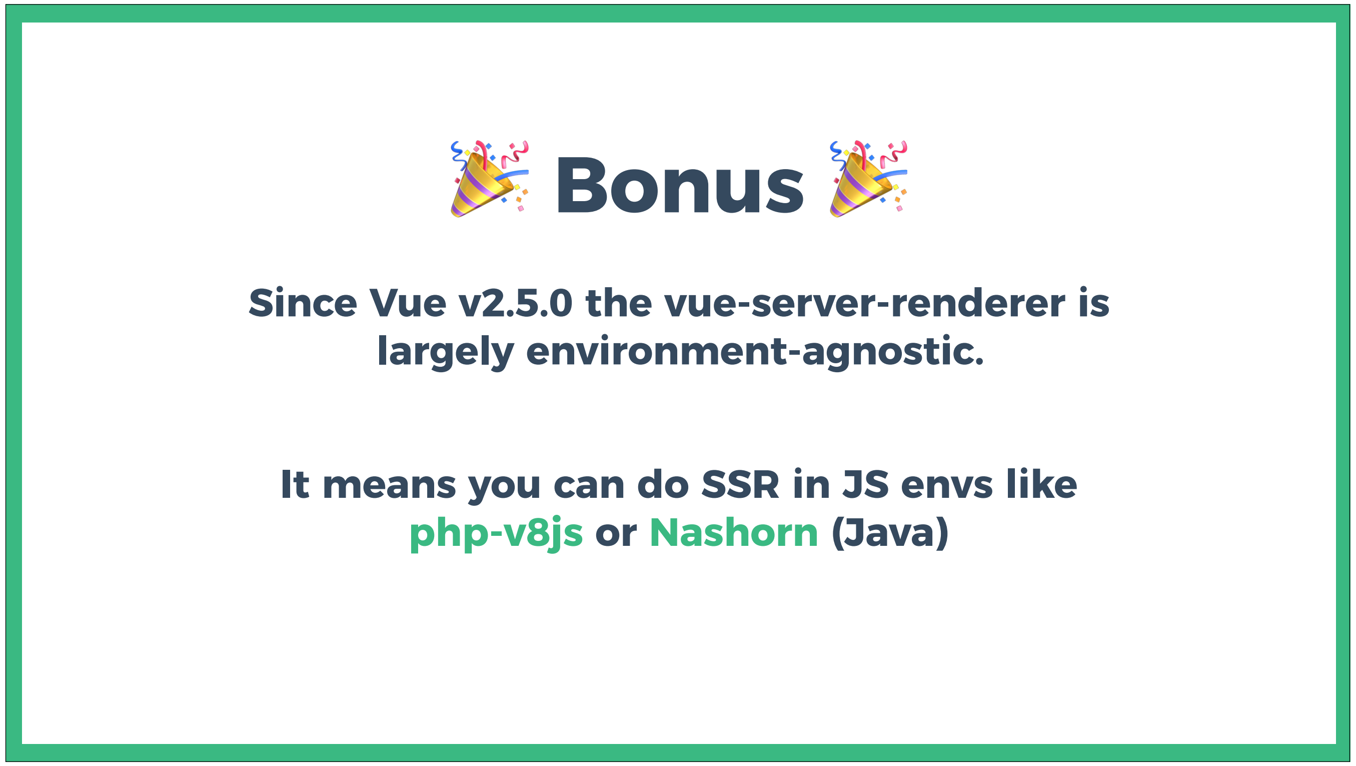
prerender-spa-plugin

Static Page Generation with Nuxt.js

```
nuxt generate
```

Prerendering is a well known solution. Actually the author of prerender-spa-plugin is Chris Fritz from the Vue core team.

You know Jekyll and Hexo? Nuxt.js can do something similar but with the additional benefit of your page being a SPA after it loads. Really powerful if you combine it with headless CMS like Contentful or GraphCMS, or even Wordpress API. All of those solutions can call a webhook to rebuild your page when the content changes (Netlfy has such webhooks).



🎉 Bonus 🎉

Since Vue v2.5.0 the vue-server-renderer is largely environment-agnostic.

It means you can do SSR in JS envs like php-v8js or Nashorn (Java)

So you don’t actually need a Node.js proxy app!

Thank you!

Damian Dulisz

@DamianDulisz

@shentao

Consultant

Vue Core Team

Open-source Developer

Questions?