

Routing and Route Protection in Server-Rendered Vue Apps Using Nuxt.js

BY **CHRIS NWAMBA** ON JANUARY 25, 2018

Nuxt, Server Side Rendering, Vue

<EDITOR_INTRO>

This tutorial assumes basic knowledge of Vue. If you haven't worked with it before, then you may want to check out [this CSS-Tricks guide](#) on getting started.

</EDITOR_INTRO>

You might have had some experience trying to render an app built with Vue on a server. The concept and implementation details of Server-Side Rendering (SSR) are challenging for beginners as well as experienced developers. The challenges get more daunting when you have to do things like data fetching, routing and protecting authenticated routes. This article will walk you through how to overcome these challenges with Nuxt.js.

What You will Learn

The title might have limited the scope of this article because you're going to learn more than only routing and route protection. Here is a summarized list of what this article covers:

- Why Server-Side Rendering?
- Server-Side Rendering and SEO
- Setting up a Nuxt.js project from scratch
- Custom layouts
- Webpacked and static global assets
- Implicit routing and automatic code splitting
- Nested and parameterized routes
- Protecting routes with middleware

You can get the code samples [from Github](#).

Why Should I Render to a Server?

HEY!

If you already know why you should server-render and just want to learn about routing or route protection, then you can jump to [Setting Up a Nuxt.js App from Scratch](#) section.

SSR, also referred to as Universal Rendering or Isomorphic Rendering, is a concept that sprung lately from the JavaScript ecosystem to help mitigate the downsides of JavaScript frameworks.

When we had no JS frameworks or UI libraries like Angular, React, and Vue, the de facto way of building websites was to send an HTML (accompanied with some styles and JS) string as a response from a server which is then parsed and rendered by the browser. This means your views were server-rendered. The most we could do after the page was rendered was to begin the dirty job of manipulating its content using JavaScript or jQuery.

Interactive user interfaces were such nightmares to build using these patterns. In addition to the amount of work you had to do with the DOM via JS, you still needed to do the dirty jobs of poking the DOM, traversing it, and forcing contents and features into it. Worse still, this led to a lot of bad code and poor performing (slow) UIs.

The JavaScript frameworks introduced a few concepts like virtual DOM and declarative APIs which made it faster and more fun to work with the DOM. The problem with them is that the views are entirely controlled with JavaScript. You can say they are JavaScript-rendered. The implication is that unlike the previous era where views were server-rendered by default, JavaScript is required and you have to wait for it before your users see anything.

Here is what you should take away from this long talk:

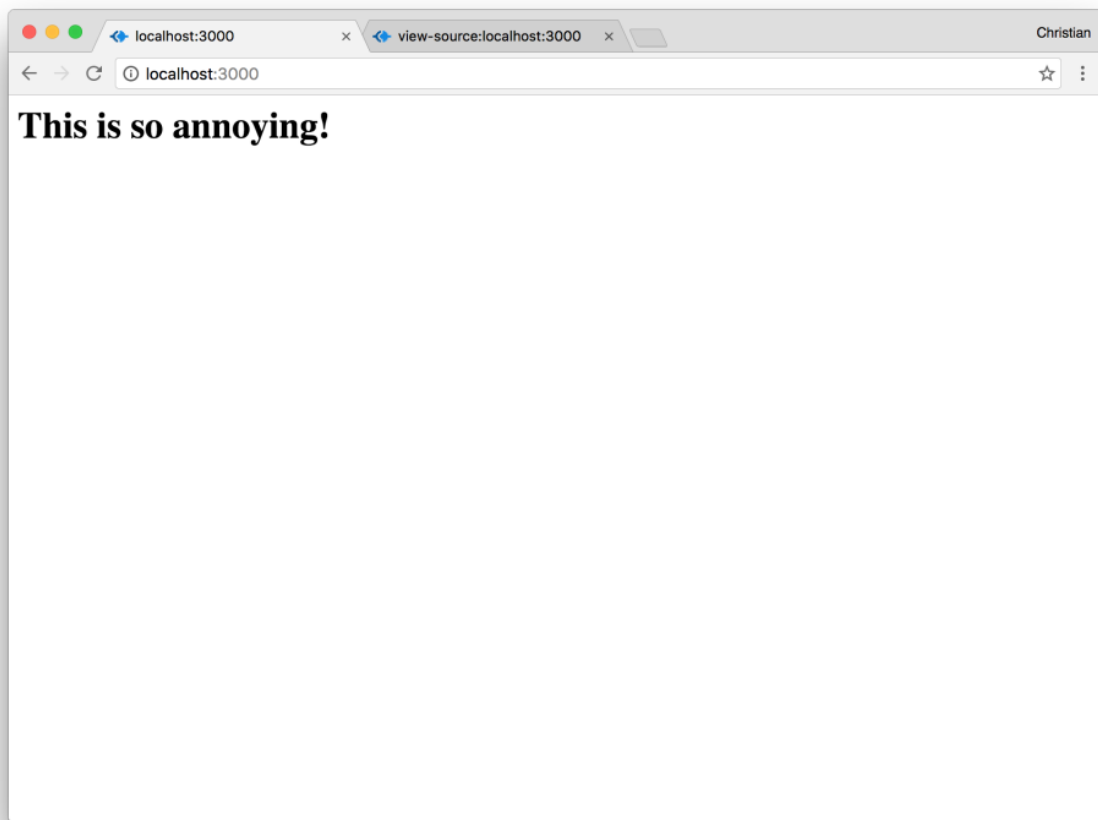
1. **Server-rendered apps** are faster because they do not rely on JavaScript to start painting the browser with content.
2. **JavaScript-rendered apps** are preferred for better user experience. Unfortunately, this is only after JavaScript has been parsed and compiled.

We want the speed of the server-rendered app first paint to improve and create a better JS-rendered user experience. This is where the concept of SSR for JavaScript frameworks comes in.

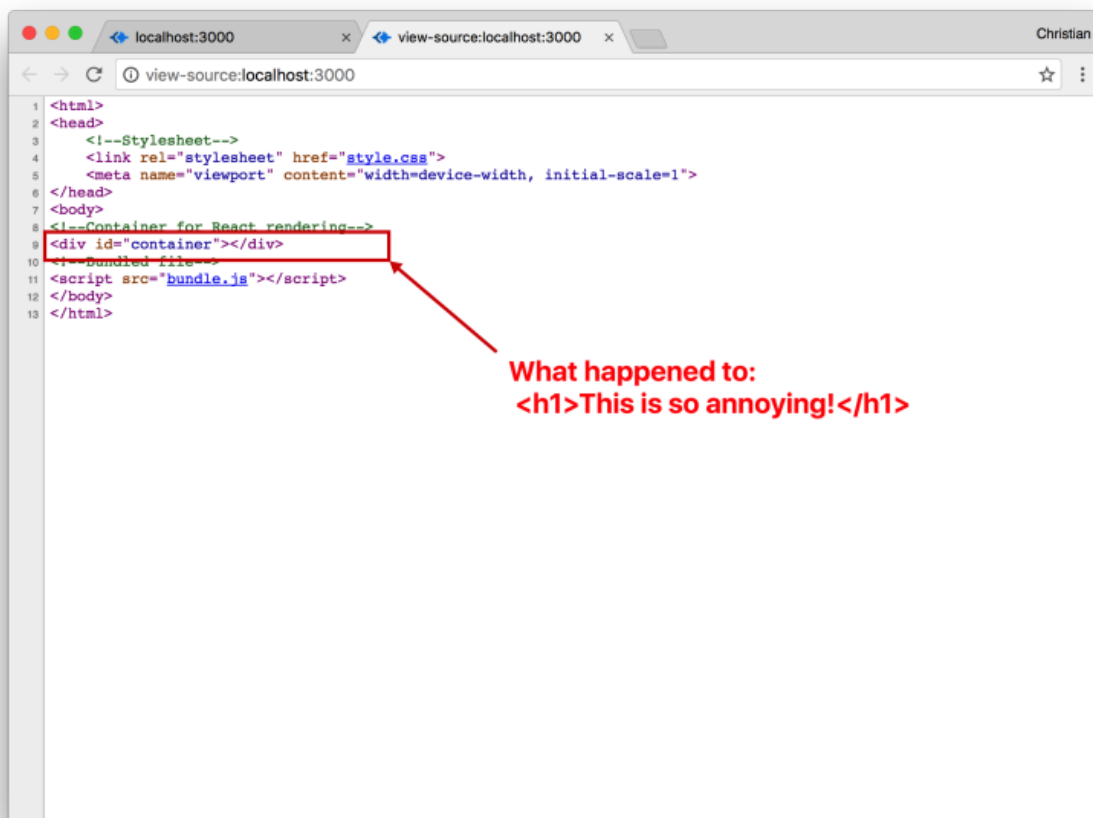
SEO Problems

Another big problem that hits you when building apps with Vue is how to make them SEO friendly. For now, web crawlers don't seek content to index in JavaScript. They just know about HTML. This is not the case for server-rendered apps because they already respond with the HTML that the crawler needs.

This is how things could go wrong:



The image above shows a simple front end app with some text. In all its simplicity, inspect the page source and you would be disappointed to find out that the text is not in the page source:



Nuxt.js for Server-Rendered Vue Apps

[Sarah Drasner](#) wrote a great post on [what Nuxt.js is and why you should use it](#). She also showed off some of the amazing things you can do with this tool like page routing and page transitions. Nuxt.js is a tool in the Vue ecosystem that you can use to build server-rendered apps from scratch without being bothered by the underlying complexities of rendering a JavaScript app to a server.

Nuxt.js is an option to what [Vue already offers](#). It builds upon the Vue SSR and routing libraries to expose a seamless platform for your own apps. Nuxt.js boils down to one thing: to **simplify** your experience as a developer building SSR apps with Vue.

We already did a lot of talking (which they say is cheap); now let's get our hands dirty.

Setting Up a Nuxt.js App from Scratch

You can quickly scaffold a new project using the [Vue CLI](#) tool by running the following command:

Command Line

```
vue init nuxt-community/starter-template <project-name>
```

But that's not the deal, and we want to get our hands dirty. This way, you would learn the underlying processes that powers the engine of a Nuxt project.

Start by creating an empty folder on your computer, open your terminal to point to this folder, and run the following command to start a new node project:

Command Line

```
npm init -y
```

OR

```
yarn init -y
```

This will generate a `package.json` file that looks like this:

JSON

```
{
  "name": "nuxt-shop",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT"
}
```

The `name` property is the same as the name of the folder you working in.

Install the Nuxt.js library via npm:

Command Line

```
npm install --save nuxt
```

OR

```
yarn add nuxt
```

Then configure a npm script to launch nuxt build process in the `package.json` file:

```
"scripts": {
  "dev": "nuxt"
}
```

You can then start-up by running the command you just created:

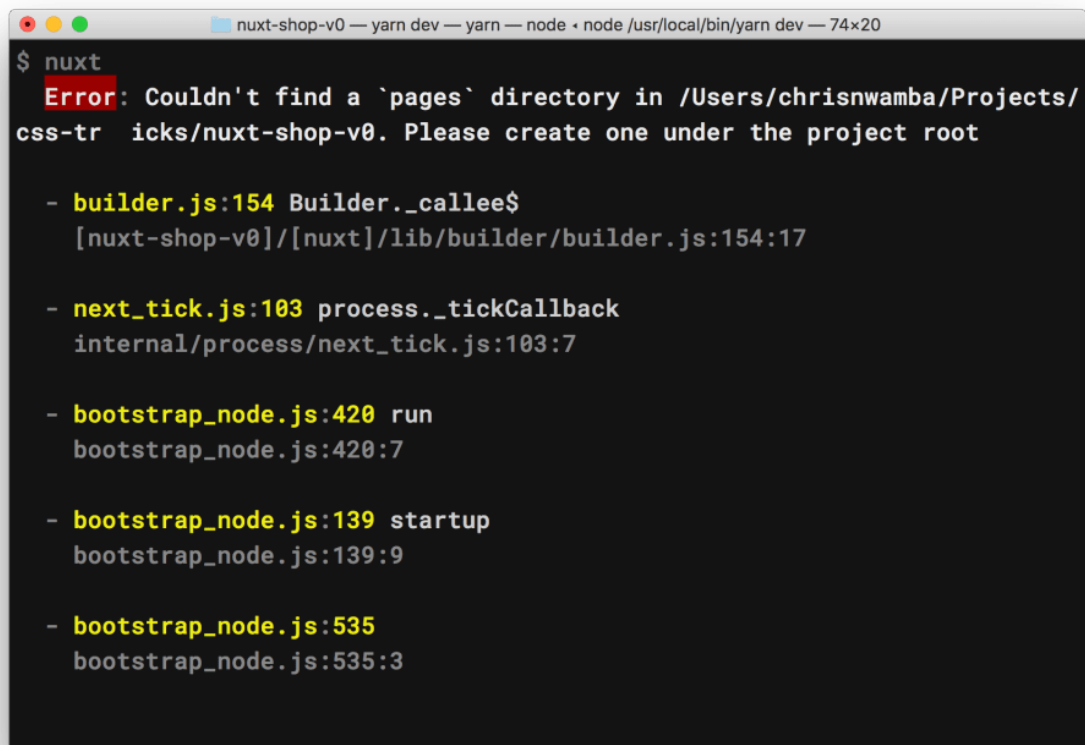
Command Line

```
npm run dev
```

```
# OR
```

```
yarn dev
```

It's OK to watch the build fail. This is because Nuxt.js looks into a `pages` folder for contents which it wills serve to the browser. At this point, this folder does not exist:



```
nuxt-shop-v0 — yarn dev — yarn — node • node /usr/local/bin/yarn dev — 74x20
$ nuxt
Error: Couldn't find a `pages` directory in /Users/chrisnwamba/Projects/css-tricks/nuxt-shop-v0. Please create one under the project root

- builder.js:154 Builder._callee$
  [nuxt-shop-v0]/[nuxt]/lib/builder/builder.js:154:17

- next_tick.js:103 process._tickCallback
  internal/process/next_tick.js:103:7

- bootstrap_node.js:420 run
  bootstrap_node.js:420:7

- bootstrap_node.js:139 startup
  bootstrap_node.js:139:9

- bootstrap_node.js:535
  bootstrap_node.js:535:3
```

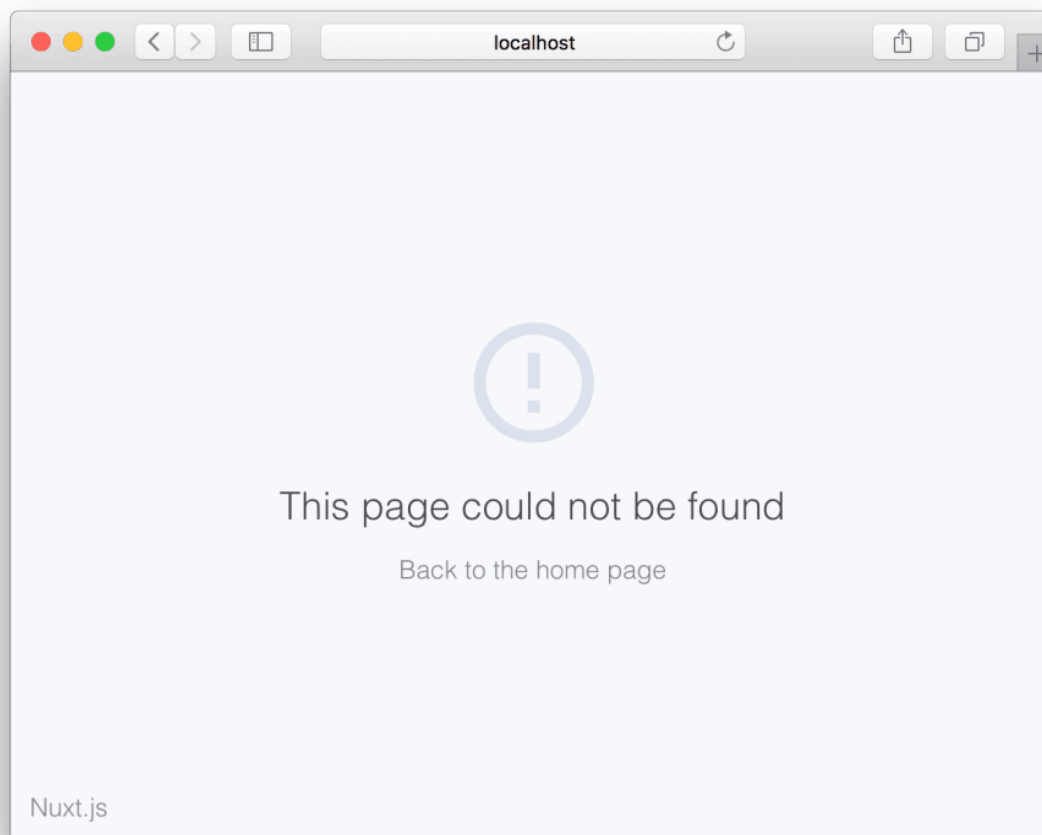
Exit the build process then create a `pages` folder in the root of your project and try running once more. This time you should get a successful build:

```
nuxt-shop-v0 — yarn dev — yarn — node • node /usr/local/bin/yarn dev — 74x20
Chriss-MacBook-Pro ~ /projects/css-tricks/nuxt-shop-v0:
[1546] yarn dev
yarn dev v0.17.10
$ nuxt
  nuxt:build App root: /Users/chrisnwamba/Projects/css-tricks/nuxt-shop-v0
+0ms
  nuxt:build Generating /Users/chrisnwamba/Projects/css-tricks/nuxt-shop-v0/.nuxt files... +1ms
  nuxt:build Generating files... +5ms
  nuxt:build Generating routes... +0ms
  nuxt:build Building files... +20ms
  nuxt:build Adding webpack middleware... +98ms
Build completed in 4.356s

DONE Compiled successfully in 4360ms 3:20:37 PM

OPEN http://localhost:3000
```

The app launches on Port 3000 but you get a 404 when you try to access it:



Nuxt.js maps page routes to file names in the `pages` folder. This implies that if you had a file named `index.vue` and another `about.vue` in the `pages` folder, they will resolve to `/` and `/about`, respectively. Right now, `/` is throwing a 404 because, `index.vue` does not exist in the `pages` folder.

Create the `index.vue` file with this dead simple snippet:

HTML

```
<template>
  <h1>Greetings from Vue + Nuxt</h1>
</template>
```

Now, restart the server and the 404 should be replaced with an index route showing the greetings message:



Project-Wide Layout and Assets

Before we get deep into routing, let's take some time to discuss how to structure your project in such a way that you have a reusable layout as sharing global assets on all pages. Let's start with the global assets. We need these two assets in our project:

1. Favicon
2. Base Styles

Nuxt.js provides two root folder options (depending on what you're doing) for managing assets:

1. **assets:** Files here are webpacked (bundled and transformed by webpack). Files like your CSS, global JS, LESS, SASS, images, should be here.

2. **static:** Files here don't go through webpack. They are served to the browser as is. Makes sense for `robot.txt` , favicons, Github CNAME file, etc.

In our case, our favicon belongs to `static` while the base style goes to the `assets` folder. Hence, create the two folders and add `base.css` in `/assets/css/base.css` . Also [download this favicon file](#) and put it in the `static` folder. We need `normalize.css` but we can install it via npm rather than putting it in `assets` :

Command Line

```
yarn add normalize.css
```

Finally, tell Nuxt.js about all these assets in a config file. This config file should live in the root of your project as `nuxt.config.js` :

JS

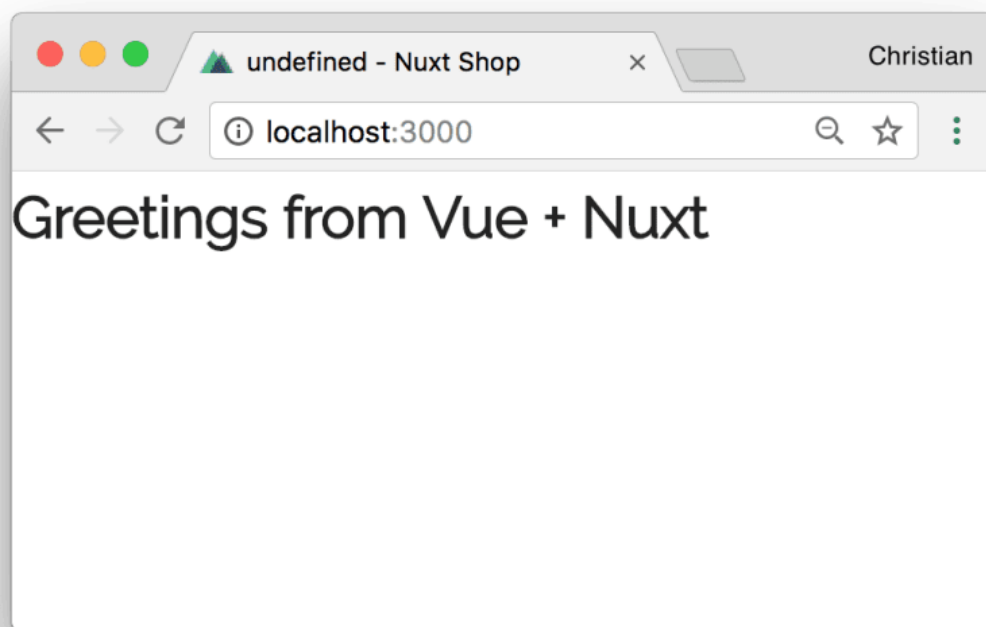
```
module.exports = {
  head: {
    titleTemplate: '%s - Nuxt Shop',
    meta: [
      { charset: 'utf-8' },
      { name: 'viewport', content: 'width=device-width, initial-scale=1' },
      { hid: 'description', name: 'description', content: 'Nuxt online shop' }
    ],
    link: [
      {
        rel: 'stylesheet',
        href: 'https://fonts.googleapis.com/css?family=Raleway'
      },
      { rel: 'icon', type: 'image/x-icon', href: 'https://cdn.css-tricks.com/fav'
    ]
  },
  css: ['normalize.css', '@/assets/css/base.css']
};
```

We just defined our title template, page meta information, fonts, favicon and all our styles. Nuxt.js will automatically include them all in the head of our pages.

Add this in the `base.css` file and let's see if everything works as expected:

```
html, body, #__nuxt {  
  height: 100%;  
}  
  
html {  
  font-size: 62.5%;  
}  
  
body {  
  font-size: 1.5em;  
  line-height: 1.6;  
  font-weight: 400;  
  font-family: 'Raleway', 'HelveticaNeue', 'Helvetica Neue', Helvetica, Arial, sans-serif;  
  color: #222;  
}
```

You should see that the font of the greeting message has changed to reflect the CSS:



Now we can talk about layout. Nuxt.js already has a default layout you can customize. Create a `layouts` folder on the root and add a `default.vue` file in it with the following layout content:

HTML

```
<template>
  <div class="main">
    <app-nav></app-nav>
    <!-- Mount the page content here -->
    <nuxt/>

  </div>
</template>
<style>
/* You can get the component styles from the Github repository for this demo */
</style>

<script>
import nav from '@components/nav';
export default {
  components: {
    'app-nav': nav
  }
};
</script>
```

HEY!

I am omitting all the styles in the `style` tag but you can get them from the code repository. I omitted them for brevity.

The layout file is also a component but wraps the `nuxt` component. Everything in the this file is shared among all other pages while each page content replaces the `nuxt` component. Speaking of shared contents, the `app-nav` component in the file should show a simple navigation.

Add the `nav` component by creating a `components` folder and adding a `nav.vue` file in it:

HTML

```
<template>
  <nav>
    <div class="logo">
      <app-h1 is-brand="true">Nuxt Shop</app-h1>
```

```

    </div>
    <div class="menu">
      <ul>
        <li>
          <nuxt-link to="/">Home</nuxt-link>
        </li>
        <li>
          <nuxt-link to="/about">About</nuxt-link>
        </li>
      </ul>
    </div>
  </nav>
</template>
<style>
/* You can get the component styles from the Github repository for this demo */
</style>
<script>
import h1 from './h1';
export default {
  components: {
    'app-h1': h1
  }
}
</script>

```

The component shows brand text and two links. Notice that for Nuxt to handle routing appropriately, we are not using the `<a>` tag but the `<nuxt-link>` component. The brand text is rendered using a reusable `<h1>` component that wraps and extends a `<h1>` tag. This component is in `components/h1.vue` :

HTML

```

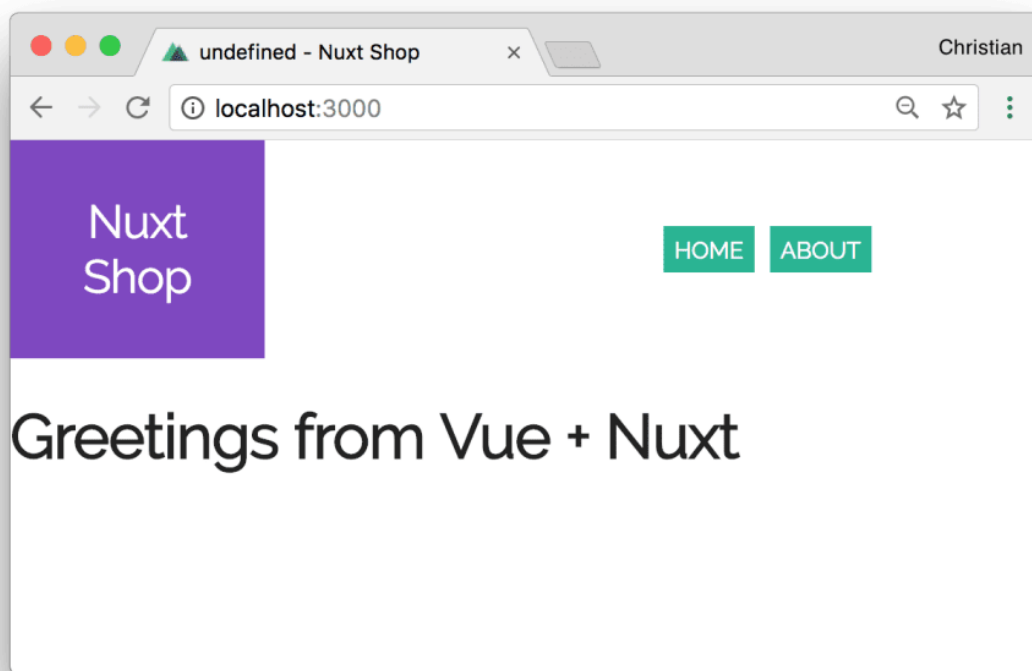
<template>
  <h1 :class="{brand: isBrand}">
    <slot></slot>
  </h1>
</template>
<style>
/* You can get the component styles

```

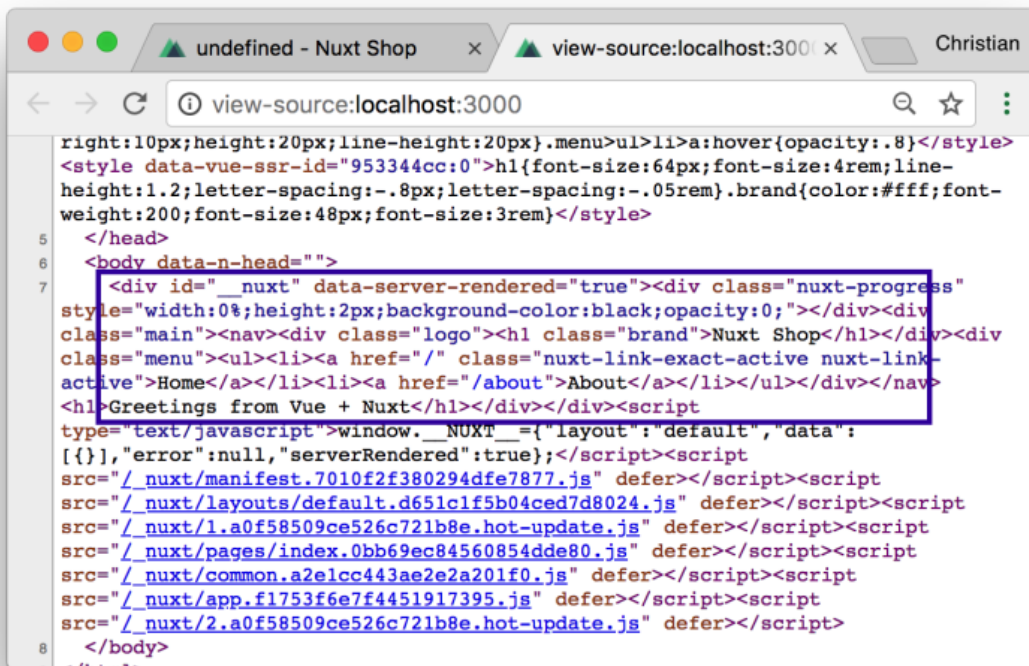
from the Github repository for this demo

```
*/  
</style>  
<script>  
export default {  
  props: ['isBrand']  
}  
</script>
```

This is the output of the index page with the layout and these components added:



When you inspect the output, you should see the contents are rendered to the server:



Implicit Routing and Automatic Code Splitting

As mentioned earlier, Nuxt.js uses its file system to generate routes. All the files in the `pages` directory are mapped to a URL on the server. So, if I had this kind of directory structure:

Directory

```
pages/  
--| product/  
----| index.vue  
----| new.vue  
--| index.vue  
--| about.vue
```

...then I would automatically get a Vue router object with the following structure:

JS

```
router: {  
  routes: [  
    {  
      name: 'index',
```

```

    path: '/',
    component: 'pages/index.vue'
  },
  {
    name: 'about',
    path: '/about',
    component: 'pages/about.vue'
  },
  {
    name: 'product',
    path: '/product',
    component: 'pages/product/index.vue'
  },
  {
    name: 'product-new',
    path: '/product/new',
    component: 'pages/product/new.vue'
  }
]
}

```

This is what I prefer to refer to as *implicit routing*.

On the other hand, each of these pages are not bundled in one `bundle.js`. This would be the expectation when using webpack. In plain Vue projects, this is what we get and we would manually split the code for each route into their own files. With Nuxt.js, you get this out of the box and it's referred to as automatic code splitting.

You can see this whole thing in action when you add another file in the `pages` folder. Name this file, `about.vue` with the following content:

HTML

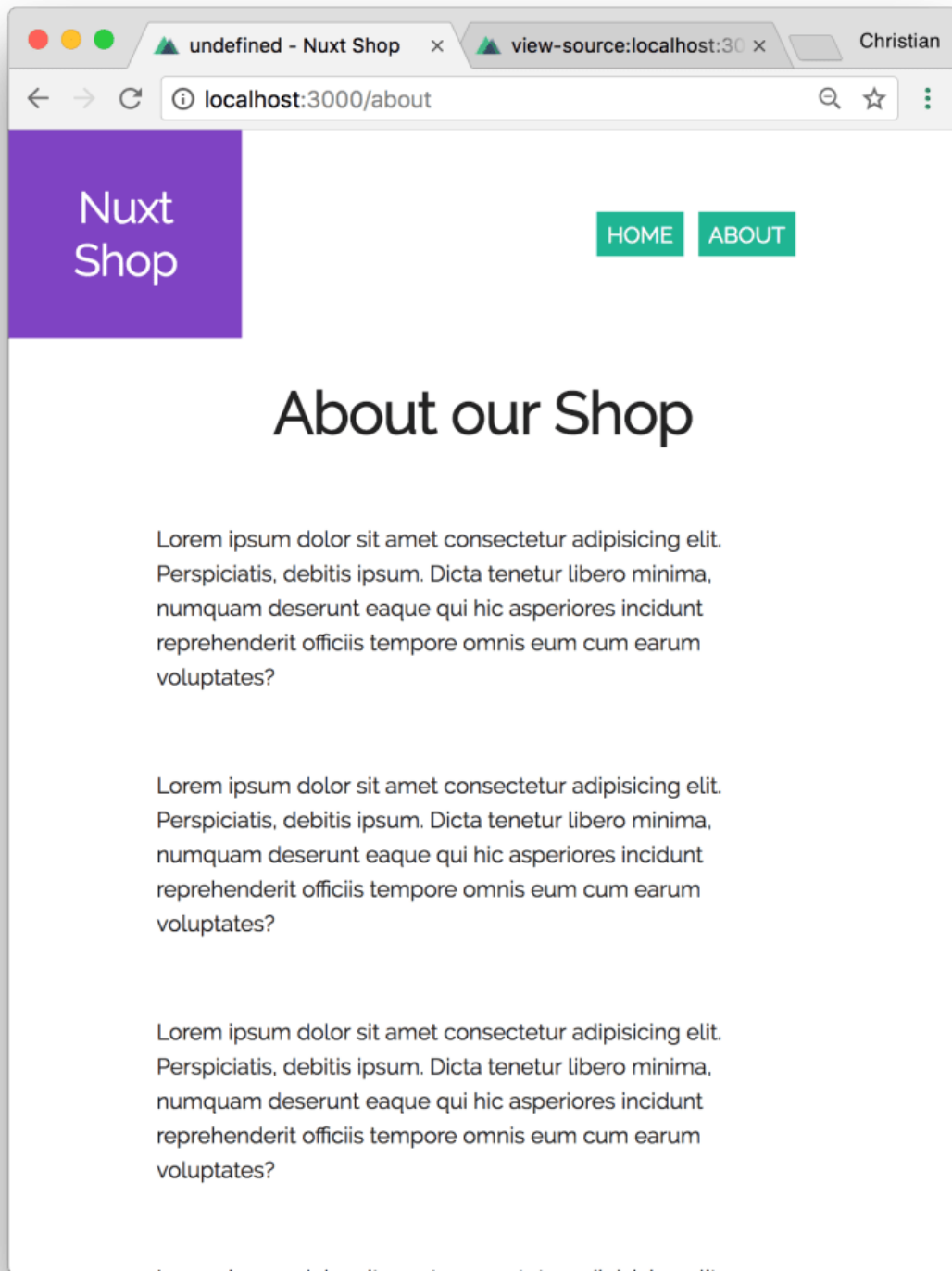
```

<template>
  <div>
    <app-h1>About our Shop</app-h1>
    <p class="about">Lorem ipsum dolor sit amet consectetur adipisicing ...</p>
    <p class="about">Lorem ipsum dolor sit amet consectetur adipisicing ...</p>
    <p class="about">Lorem ipsum dolor sit amet consectetur adipisicing ...</p>
    <p class="about">Lorem ipsum dolor sit amet consectetur adipisicing ...</p>
  </div>
</template>

```

```
...
</div>
</template>
<style>
...
</style>
<script>
import h1 from '@components/h1';
export default {
  components: {
    'app-h1': h1
  }
};
</script>
```

Now click on the **About** link in the navigation bar and it should take you to `/about` with the page content looking like this:



A look at the Network tab in DevTools will show you that no `pages/index.[hash].js` file was loaded, rather, a `pages/about.[hash].js` :

Name	Status	Type	Initiator	Size	Time	Waterfall	40.00 s	▲1.
manifest.17938c621...	200	script	about	30.9...	407 ...			
common.a2e1cc443...	200	script	about	291 ...	408 ...			
app.079b889b91560...	200	script	about	369 ...	409 ...			
default.d651c1f5b04...	200	script	about	30.4...	409 ...			
about.f5d6398dbed...	200	script	about	22.0...	409 ...			
5.15f5d29e05db9b3...	200	script	about	(fro...	407 ...			

You should take out one thing from this: Routes === Pages . Therefore, you’re free to use them interchangeably in the server-side rendering world.

Data Fetching

This is where the game changes a bit. In plain Vue apps, we would usually wait for the component to load, then make a HTTP request in the `created` lifecycle method. Unfortunately, when you are also rendering to the server, the server is ready way before the component is ready. Therefore, if you stick to the `created` method, you can’t render fetched data to the server because it’s already too late.

For this reason, Nuxt.js exposes another instance method like `created` called `asyncData` . This method has access to two contexts: the client and the server. Therefore, when you make request in this method and return a data payload, the payload is automatically attached to the Vue instance.

Let’s see an example. Create a `services` folder in the root and add a `data.js` file to it. We are going to simulate data fetching by requesting data from this file:

JS

```
export default [
  {
    id: 1,
    price: 4,
    title: 'Drinks',
    imgUrl: 'http://res.cloudinary.com/christekh/image/upload/v1515183358/pro3_t',
  },
  {
    id: 2,
    price: 3,
    title: 'Home',
  },
]
```

```
    imgUrl: 'http://res.cloudinary.com/christekkh/image/upload/v1515183358/pro2_9
  },
  // Truncated for brevity. See repo for full code.
]
```

Next, update the index page to consume this file:

HTML

```
<template>
  <div>
    <app-banner></app-banner>
    <div class="cta">
      <app-button>Start Shopping</app-button>
    </div>
    <app-product-list :products="products"></app-product-list>
  </div>
</template>

<style>
...
</style>

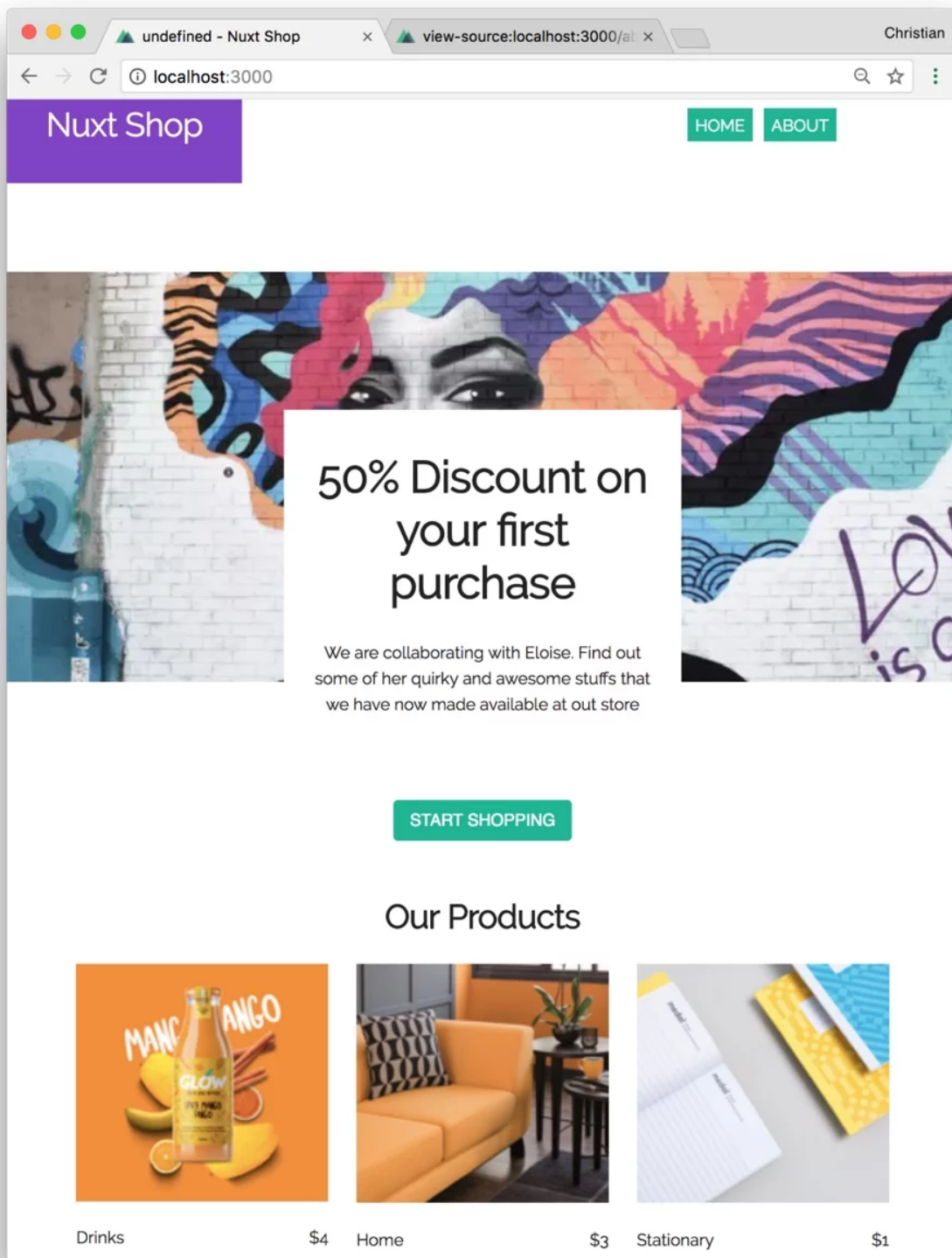
<script>
import h1 from '@components/h1';
import banner from '@components/banner';
import button from '@components/button';
import productList from '@components/product-list';
import data from '@services/data';
export default {
  asyncData(ctx, callback) {
    setTimeout(() => {
      callback(null, { products: data });
    }, 2000);
  },
  components: {
    'app-h1': h1,
    'app-banner': banner,
    'app-button': button,
    'app-product-list': productList
  }
}
```

```
    }  
  };  
</script>
```

Ignore the imported components and focus on the `asyncData` method for now. I am simulating an async operation with `setTimeout` and fetching data after two seconds. The callback method is called with the data you want to expose to the component.

Now back to the imported components. You have already seen the `<h1>` component. I have created few more to serve as UI components for our app. All these components live in the `components` directory and you can get the code for them from the Github repo. Rest assured that they contain mostly HTML and CSS so you should be fine understanding what they do.

This is what the output should look like:



Guess what? The fetched data is still rendered to the server!

Parameterized (Dynamic) Routes

Sometimes the data you show in your page views are determined by the state of the routes. A common pattern in web apps is to have a dynamic parameter in a URL. This parameter is used to query data or a database for a given resource. The parameters can come in this form:

`https://example.com/product/2`

The value `2` in the URL can be `3` or `4` or any value. The most important thing is that your app would fetch that value and run a query against a dataset to retrieve relative information.

In Nuxt.js, you have the following structure in the `pages` folder:

Directory

```
pages/  
--| product/  
-----| _id.vue
```

This resolves to:

JS

```
router: {  
  routes: [  
    {  
      name: 'product-id',  
      path: '/product/:id?',  
      component: 'pages/product/_id.vue'  
    }  
  ]  
}
```

To see how that works out, create a `product` folder in the `pages` directory and add a `_id.vue` file to it:

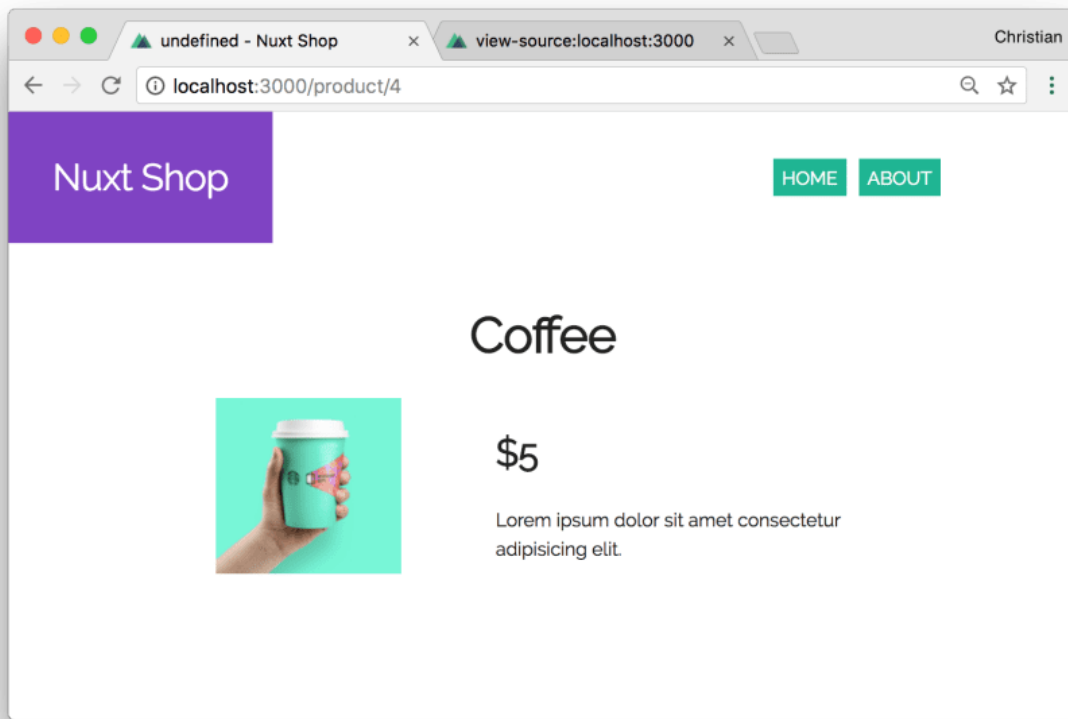
HTML

```
<template>  
  <div class="product-page">  
    <app-h1>{{product.title}}</app-h1>  
    <div class="product-sale">  
      <div class="image">  
          
      </div>  
      <div class="description">  
        <app-h2>{{{product.price}}}</app-h2>  
        <p>Lorem ipsum dolor sit amet consectetur adipisicing elit.</p>  
      </div>  
    </div>  
  </template>
```

```
    </div>
  </div>
</template>
<style>

</style>
<script>
import h1 from '@components/h1';
import h2 from '@components/h2';
import data from '@services/data';
export default {
  asyncData({ params }, callback) {
    setTimeout(() => {
      callback(null, {product: data.find(v => v.id === parseInt(params.id))})
    }, 2000)
  },
  components: {
    'app-h1': h1,
    'app-h2': h2
  },
};
</script>
```

What's important is the `asyncData` again. We are simulating an async request with `setTimeout`. The request uses the `id` received via the context object's `params` to query our dataset for the first matching id. The rest is just the component rendering the `product`.



Protecting Routes With Middleware

It won't take too long before you start realizing that you need to secure some of your website's contents from unauthorized users. Yes, the data source might be secured (which is important) but user experience demands that you prevent users from accessing unauthorized contents. You can do this by showing a friendly walk-away error or redirecting them to a login page.

In Nuxt.js, you can use a middleware to protect your pages (and in turn your contents). A middleware is a piece of logic that is executed before a route is accessed. This logic can prevent the route from being accessed entirely (probably with redirections).

Create a `middleware` folder in the root of the project and add an `auth.js` file:

JS

```
export default function (ctx) {  
  if(!isAuth()) {  
    return ctx.redirect('/login')  
  }  
}  
  
function isAuth() {  
  // Check if user session exists somehow
```



```
    return false;
  }
}
```

The middleware checks if a method, `isAuth`, returns false. If that is the case, it implies that the user is not authenticated and would redirect the user to a login page. The `isAuth` method just returns false by default for test purposes. Usually, you would check a session to see if the user is logged in.

HEY!

Don't rely on `localStorage` because the server does not know that it exists.

You can use this middleware to protect pages by adding it as value to the `middleware` instance property. You can add it to the `_id.vue` file we just created:

JS

```
export default {
  asyncData({ params }, callback) {
    setTimeout(() => {
      callback(null, {product: data.find(v => v.id === parseInt(params.id))})
    }, 2000)
  },
  components: {
    //...
  },
  middleware: 'auth'
};
```

This automatically shuts this page out every single time we access it. This is because the `isAuth` method is always returning `false`.

Long Story, Short

I can safely assume that you have learned what SSR is and why you should be interested in using it. You also learned some fundamental concepts like routing, layouts, security, as well as async data fetching. There is more to it, though. You should dig into the [Nuxt.js guide](#) for more features and use cases. If you're working on a React project and need this kind of tool, then I think you should try [Next.js](#).

Related



Simple Server Side Rendering, Routing, and Page Transitions with Nuxt.js

A bit of a wordy title, huh? What is server side rendering? What does it have to do with routing and page transitions? What



Intro to Vue.js: Rendering, Directives, and Events

If I was going to sum up my experiences with Vue in a sentence, I'd probably say something like "it's just so reasonable" or

announced some amazing developer resources for creating Progressive Web

Comments

Sarah Drasner

JANUARY 25, 2018

You did a great job with this post. It's so thorough and a fantastic explanation. Nice work!

Chris Nwamba

JANUARY 28, 2018

Thank you so much, Sarah. Glad you loved it!



Sebastien Chopin (@Atinux)

JANUARY 26, 2018

Great post! You are explaining perfectly what Nuxt.js does under the hood while keeping it simple to everyone :)

McDave

JANUARY 27, 2018

EXCELLENT intro. Thank you.

<GerardSans /> (@gerardsans)

JANUARY 29, 2018

Cool! Thanks Chris! Going to mention Nuxt in my coming Vue talk =)

Raymond Camden

JANUARY 29, 2018

So one thing I've been struggling with is what can be done server side. As an example, when I first tried to use Nuxt, I switched out Axios to request-promise. In my mind, Nuxt was rendering everything on the server, so I should be able to use any Node module I wanted. (You can see the issue I raised here: <https://github.com/nuxt/nuxtjs.org/issues/106>). So given that you need to be isomorphic, how would a user session work? As you said, you can't use localStorage as it's being rendered on the client, so... how would you handle that? Also, doesn't that break down if the Nuxt app is generated statically?

Luke

JANUARY 30, 2018

Thank you for the great write up! Are you aware of any tutorials on deploying a Nuxt app to a production environment, lets say, digital ocean?

DomesticWBAccounting (@dwba)

FEBRUARY 7, 2018

Another great post Chris – Many thanks!

In your introduction you emphasised that you would cover much more than just SSR and so you did – very useful ...

In particular, I realised I had misunderstood the Nuxt documentation in regard to Props and Vuex with store. I hadn't been able to get Props to work in my Nuxt prototyping and incorrectly (I now see from your code) concluded from the nuxt.org examples that vuex store was the only way to communicate down the component hierarchy – sometimes rather heavy overhead for the sort of transfers you needed in your h1, h2, button etc components. So glad that Props can be used but I think, not for s, only component 'calls'. Having said that, Vuex and store and the new modules capabilities appear very powerful but quite complex/tricky to use.

That would be a wonderful subject for a new post by one of you Nuxt experts as the nuxt.org shopping-cart example, for me, lacks detailed explanation.

A couple of other related issues as the possibilities widen using Nuxt are first, having nuxt-linked to a new page and navigated up and down the SPA hierarchy on that page, how do you programmatically return to the original page, and secondly, without wanting a single monolithic Nuxt program for a hierarchy of projects/modules, is there a way for example in the Nuxt components and pages folders to 'point' to other drives, folders and files on your machine, so not necessarily directly containing them in the Nuxt project folder structure?

Thanks again, John P Christchurch Dorset UK

