

[← Back to Blog](#)

September 03, 2018

# Nuxt.js – the First Encounter

by Jscrambler

Tags: *Vue.js, Web Development, Nuxt.js*

## What is Nuxt.js?

Let's start with what [Nuxt.js](#) isn't. It is not another big front-end framework built to compete with the big three.

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. [I understand](#)

Nuxt.js can't work without Vue.js. And we can't replace Vue.js here for the other framework. A quick definition for Nuxt.js could be then – **a framework for creating complex Vue.js apps easier.**

But do we really need it? It seems that Vue.js is quite simple itself. There are tons of tutorials, it's built for correcting React and Angular mistakes and is very easy to learn. It has a lot of features out of the box and tons of external libraries — Vuex, vue-router, axios, vee-validate, just to name a few. It seems Vue.js can be quite powerful. And... it is. So why do we really need Nuxt?

Because, while Vue.js may be so easy, Nuxt.js makes it even easier.

It sets you up with a great file structure, improves the routing mechanism, enables server-side-rendering and allows you to create universal apps. You can do all of this yourself, but... it always takes time and sometimes requires really good skills. That's what Nuxt is for — to create a great complex and well-working app without wasting your time for preparations. With Nuxt.js, you just generate the template and that's it. You can start developing your app!

## Nuxt.js features



NUXT

# Universal Vue.js Applications

Before we dig deeper into Nuxt features, it's worth mentioning that it's up to us to decide which of them we're going to use in our project. This is possible thanks to `create-nuxt-app`. This command line script allows us to quickly create new Nuxt.js based apps according to our wishes. If you're familiar with the MERN stack and mern.io boilerplate, it's equivalent to `mern-cli`.

To create a new project with Nuxt.js CLI, you simply type `create-nuxt-app <your-app-title>` to start the process. Then, you can configure your app template as you wish.

```
varenthein@CGStudio MINGW64 ~/Desktop/Nuxt (master)
$ create-nuxt-app test-app
> Generating Nuxt.js project in C:\Users\Varenthein\Desktop\Nuxt\test-app
? Project name (test-app)
? Project name test-app
? Project description (My cat's meow Nuxt.js project) My test-app
? Project description My test-app
? Use a custom server framework (Use arrow keys)
? Use a custom server framework none
? Use a custom UI framework (Use arrow keys)
? Use a custom UI framework none
? Choose rendering mode (Use arrow keys)
? Choose rendering mode universal
? Use axios module (Use arrow keys)
? Use axios module no
? Use eslint (Use arrow keys)
? Use eslint no
? Author name (Varenthein)
? Author name Varenthein
? Choose a package manager (Use arrow keys)
? Choose a package manager npm
Initialized empty Git repository in C:\Users\Varenthein\Desktop\Nuxt\test-app\.git/
npm WARN deprecated postcss-csnext@3.1.0: 'postcss-csnext' has been deprecated in favor of 'postcss-preset-env'. Read more at https://moox.io/blog/deprecating-cssnext/
npm WARN deprecated bfj-node4@5.3.1: Switch to the 'bfj' package for fixes and new features!
```

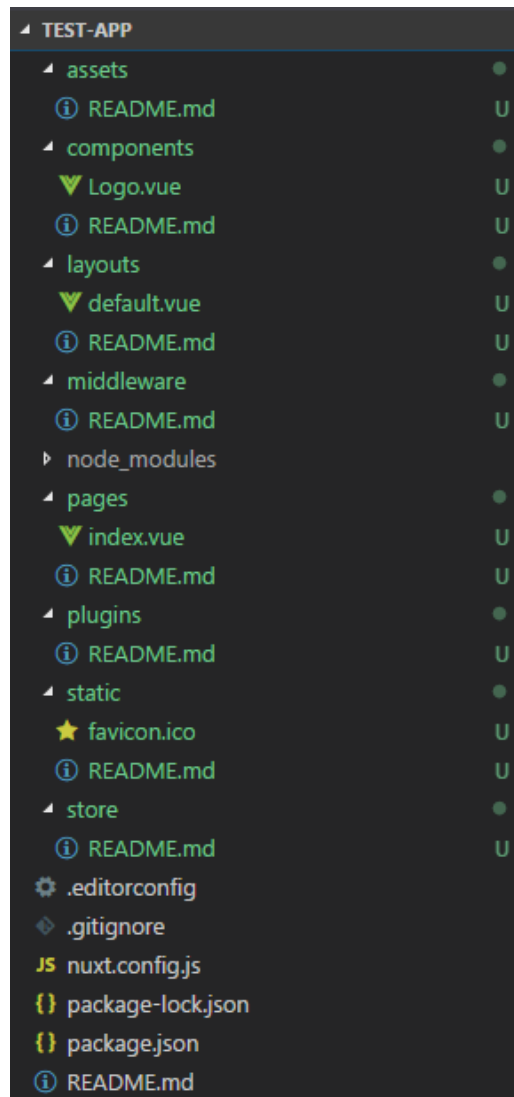
We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. **I understand**

linting or package manager. After that, you click enter and... that's it. Quick and easy.

Even though this is not really a time-consuming step, it's an important one. Here, we can really decide what features we want Nuxt.js to prepare. Of course, if you want, you can choose not to include them now and set them up manually later — but using the configurator is surely easier.

Now it's time to dig deeper into the most important Nuxt.js features

## Folder structure



We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. **I understand**

where to put them? Where place the store? What to do with routing? Here, it's already done for you.

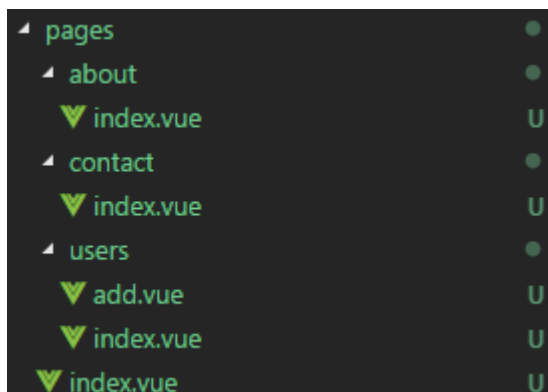
A couple of things are different than in the standard Vue.js boilerplate, but it all makes sense. If you are disturbed about the amount of changes, read further. A lot of changes are described there.

## Routing by files and folders structure

The file structure organization made out of the box is a pretty nice feature, but it's not really breathtaking. After all, you could do a similar one yourself once and then use it as the boilerplate. However, the second feature will be surely more engaging.

Nuxt.js allows routing creation by... naming files and folders in the "pages" folder. Every .vue file is considered a new route. You don't have to create a routing table yourself. Nesting your files and folders is enough.

In order not to leave you without an example, we'll set that up now.



The file structure above will automatically generate:

```
router: {  
  routes: [  

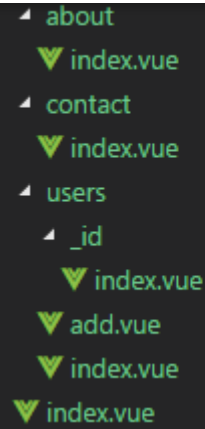
```

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. [I understand](#)

```
    component: 'pages/index.vue'
  },
  {
    name: 'about',
    path: '/about',
    component: 'pages/about/index.vue'
  },
  {
    name: 'contact',
    path: '/contact',
    component: 'pages/contact/index.vue'
  },
  {
    name: 'user',
    path: '/user',
    component: 'pages/user/index.vue'
  },
  {
    name: 'user-add',
    path: '/user/add',
    component: 'pages/user/add.vue'
  }
]
```

Obviously, you could've done it yourself — but nevertheless, it's a pretty handy feature.

We can even generate dynamic routes with this.



```
└─ about
  └─ index.vue
└─ contact
  └─ index.vue
└─ users
  └─ _id
    └─ index.vue
  └─ add.vue
  └─ index.vue
  └─ index.vue
```

The structure above will automatically generate:

```
router: {
  routes: [
    {
      name: 'index',
      path: '/',
      component: 'pages/index.vue'
    },
    {
      name: 'about',
      path: '/about',
      component: 'pages/about/index.vue'
    },
    {
      name: 'contact',
      path: '/contact',
      component: 'pages/contact/index.vue'
    },
    {
      name: 'user',
      path: '/user',
```

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. **I understand**

```
    path: '/user/:id?',
    component: 'pages/user/_id/index.vue'
  },
  {
    name: 'user-add',
    path: '/user/add',
    component: 'pages/user/add.vue'
  }
]
```

As you can see, it is also intuitive. The only thing we have to do is to properly name the folder.

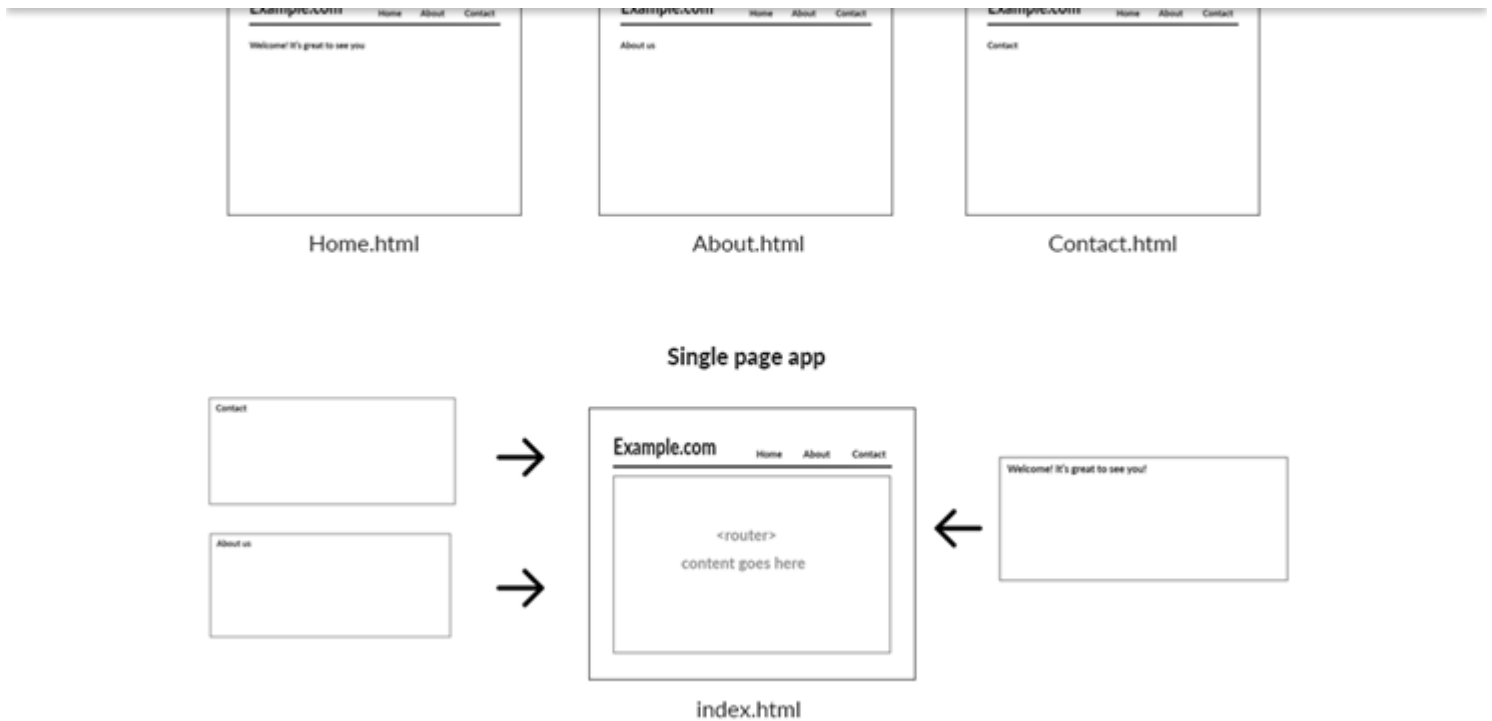
## Server-side rendering

While learning modern frameworks, one of the things that everybody always likes is an easy creation of single-page-apps (SPAs).

The idea is simple — we want to provide a user with the desktop app experience. Our app loads all needed resources once or dynamically rewrites the data when a user goes through pages. It goes away from the old idea of loading entire new pages from a server every time we click on the link.

Below you can see the difference:





In the traditional approach, every click on a link loads a new page. Of course, they are very similar — only the content below the header is different. But even then, they are reloaded completely. Even the header (which is always the same) is reloaded.

In the SPA approach, we load the base page once (in this example, it's `index.html`). Then, when we click any link, the script only reloads the content — the header and other parts of the site don't reload.

Dynamic loading could be done using Ajax, but also by loading every subpage once and then showing appropriate content when the time comes.

In theory, this is great, but making it work by yourself could be sometimes overwhelming. This is not an issue if we use a complete framework, as it is quite easy to manage it with special packages (like vue-router for Vue.js). But is there a catch?

Yes, there is — and you can mainly see it when you try your best to make your site **SEO-friendly**.

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. **I understand**

`id="root"></div>`. Of course, those robots can wait for an app to render its view, but what about routing? As we said before, in SPA, clicking on a link doesn't trigger a page reload. So, the best option for the indexing robot would be... waiting. And, if a subpage loads through Ajax, it can take long.

Of course, SPA websites have been on the market for so long that search engines learned how to work with them. But they are still not the best at it. So, if you try to build a really SEO-friendly page, you should consider not using it.

What should you use then? The idea of making every subpage a single page (with header and footer) with Vue.js rendering seems really weird. And that's where Nuxt.js comes with its **server-side-rendering** feature.

SSR allows us to prerender views on the server before a user enters a link. A short description, but that's what it is. So, what's the difference?

In SPA, the app user types a link and connects to a server. The server then returns a file (typically index.html), but, as we said before, it's just a template that looks like this:

```
<DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <title>Example APP</title>
  </head>
  <body>
    <div id="app"></div>
    <!-- built files will be auto injected here →
    <script type="text/javascript"></script>
  </body>
```

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. [I understand](#)

And even though it's generated so fast that you can't spot this, its underlying impact is clear. So, how it would look with SSR?

With SSR, the proper view would be generated on the fly on the server before being returned to the client. So yes, the user would wait for this second or two for the server response, but would get an already rendered view. You can spot the real difference in the page source.

That's how it would look like now:

```
<!DOCTYPE html>
<html data-n-head-ssr data-n-head="">
<head>
  <meta data-n-head="true" charset="utf-8"/><meta data-n-head="true" name="vi
</head>
<body data-n-head="">
  <div data-server-rendered="true" id="__nuxt"><div class="nuxt-progress" sty
  <h1 class="title">
    test
  </h1>
  <h2 class="subtitle">
    My stunning Nuxt.js project
  </h2>
  <div class="links"><a href="https://nuxtjs.org/" target="_blank" class="bu
</body>
</html>
```

Now, what we see is not an empty div, but a fully rendered view. And that's a lot easier to handle for indexing robots.

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. **I understand**

from `index.html` to e.g. `index.html#/about/` only gives you the illusion that you've changed the page, even if you actually didn't. The `#` symbol is not a decoration — its role is literally to stop you on the page. SSR approaches this matter differently, more similar to the traditional way, which we've covered earlier.

In an SSR app, when you click on a link, you actually reload your page. So the link changes. For instance, if you click such a link `<a href="/about">About</a>` the link changes from `/` to `/about`. And — what's even more important — the page really reloads itself. That way, the indexing robot doesn't stay on the same page and doesn't have to guess if any data will come. Again, it receives a pre-rendered view, so it knows what to index just as the page has loaded.

It's pretty great. And the best thing is, you can still develop your app the same as with SPA. You don't have to split your app in any way — it's all handled by Nuxt.js, which makes it work differently.

The only thing left to explain is: how does this work? Everybody knows that Vue.js is a purely front-end framework, but Nuxt.js has greater possibilities.

By enabling universal page rendering mode (you could do this in the CLI), you allow Nuxt.js to work on the back-end. As so, it is able to provide us the pre-rendering feature. But it's important — it's just for this. Nuxt.js is not a competitor of JavaScript back-end frameworks. It uses server only for enhancing speed or the way the front-end app is loaded.

## AsyncData

The Server-side-rendering feature gives us assurance that a page will be visible only when it's already rendered. But what if a given component should also load some data after it's loaded?

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. [I understand](#)

```
<template>
  <section class="container">
    <div>

      <ul>
        <li v-for="user in users">
          {{ user.name }}
        </li>
      </ul>

    </div>
  </section>
</template>

<script>
export default {
  data: function() {
    return {
      users: []
    }
  },
  methods: {
    loadUsers() {
      //small function to load data from the server
    }
  },
  created() {
    this.loadUsers()
  }
}
</script>
```

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. **I understand**

component and the page are already rendered doesn't use the whole advantage we've got from SSR.

If we would use SSR here, we would receive an already pre-rendered component. An indexing robot would see a structured page and a list, but the list would be empty. Surely the data would be loaded after a while, but that was the issue with not using pre-rendering. With SSR, we want to completely pre-render our page before it's shown to a user.

The answer is `asyncData`. It works really similar to a basic `data` block (Vue.js standard), but it's prepared on the server. So, its goal is the same (to prepare some data for the component), but it will be done before the component is rendered. Thus, when it's rendered, a user sees a whole component with data — not only a structure with the hope that the data will come eventually. Of course, loading it on the server means that we have to wait a little longer for the page to be displayed. However, thanks to that, a user sees a completely ready page when it's finally ready.

It works pretty easily. In our example, we could use it like this:

```
<template>
  <section class="container">
    <div>

      <ul>
        <li v-for="user in users">
          {{ user.name }}
        </li>
      </ul>

    </div>
```

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. [I understand](#)

```
export default {
  asyncData() {
    return axios.get('http://localhost:8000/users')
      .then(res => {
        return { users: res.data }
      })
  }
}
```

</script>

The key difference is that now we do not set an empty users array. Instead, with `asyncData`, we tell Nuxt to load the data just as a view is pre-rendered on the server. So, we will get a prepared users array when the page is loaded.

When it comes to using this data, it's not really different from `data`. After all, when the view is ready, it's simply merged with basic data. That's quite an important fact.

`asyncData` is used **only** on the server. Then, the data that has been loaded is merged to a normal `data()` and... that's all. We benefit from having it, but `asyncData` finishes its duty just before the page is fully loaded and the view renders. By this, we have to understand that we cannot use any part of the component in `asyncData` — because it's not rendered yet when `asyncData` works.

For better understating, we'll leave you with the schema form the official [Nuxt.js docs](#).

## Layouts

To achieve the same base layout for every subpage in Vue.js, we mostly use the first rendered component – `App.vue`.

```
<template>
```

```
... ..
```

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. **I understand**

```
<page-header />

<!-- ROUTER -->
<router-view />

<!-- page-footer -->
<page-footer />

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

It is the first component that is loaded and it can create the base layout for the rest of our application.

In the example above, `<page-header>` is the component responsible for rendering a header and navigation; `<page-footer>` renders a footer; `<router-view>` is responsible for loading a proper component (depending on the link).

In Nuxt.js, such a file doesn't even exist. There is no component that is superior to others, like `App.vue` was here. Instead, we use layouts, whose role is the same. However, besides default layouts (similar to `App.vue` in context before) we can make other layouts and use them whenever we want.

The default layout (located in `layouts/default.vue`) is the one which is rendered as the base template. It looks like this at first:

```
<template>
  <div>
    <nuxt />
  </div>
```

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. [I understand](#)



```
...  
</style>
```

`<nuxt>` is the component responsible for rendering a proper subpage (with a role similar to `<router-view>` in `App.vue`). The rest of the code is just a place for default structure which will be rendered always before and/or after the loaded subpage.

To make this example in Nuxt.js easier to compare with a previous example, we've made some changes, adding `<page-header>` and `<page-footer>` components.

```
<template>  
  <div>  
  
    <!-- HEADER -->  
    <page-header />  
  
    <nuxt />  
  
    <!-- page-footer -->  
    <page-footer />  
  
  </div>  
</template>  
  
<style>  
  ...  
</style>
```

As you can see, the idea is really similar. Only the execution is a bit different. It is now more clear than having one superior component.

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. **I understand**

For instance, besides the default layout (mandatory) which is, let's say, bright, you can have also a dark one. It could be named a `dark.vue`. It would obviously be located in the `layouts` folder. Now, depending on the context, you could choose to render a given page with the dark layout or the default one. It's easy to do. If you want to use a different layout than the default one, you only have to set the `layout` property in the component.

```
<template>
  <div>
    <h1>Hello world</h1>
  </div>
</template>

<script>
  export default {
    layout: 'dark'
  }
</script>
```

It's a really handy feature and definitely simple to use.

## Plugins

The last feature we'll look into is [plugins](#).

## External packages

There are tons of external packages for Nuxt.js, some of which are really helpful. Most projects use at least one or two of them. One of the most used is axios.

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. [I understand](#)

component and that's it. For `axios`, it would look like this:

```
<template>
  <div>
    <h1>Hello world</h1>
  </div>
</template>

<script>

import axios from 'axios'
export default {
  async asyncData ({ params }) {
    let { data } = await axios.get('https://localhost:8000/users')
    return { users: data }
  }
}
</script>
```

While it's simple to import axios, it generates one problem — it will be present in every page bundle.

Nuxt.js allows us to handle such a situation. If you set a proper key in `build.vendor` (`nuxt.config.js` file), it will be bundled only once.

```
module.exports = {
  build: {
    vendor: ['axios']
  }
}
```

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. **I understand**

folder and then it's ready.

For instance, if we would like to use the `VueNotifications` plugin, we should create a proper file in the plugins folder:

```
import Vue from 'vue'
import VueNotifications from 'vue-notifications'

Vue.use(VueNotifications)
```

... and, after that, everything should work.

Additionally, we can include it in the vendor bundle for better caching. To do this, we should update our `nuxt.config.js` and add `vue-notifications` in the vendor bundle.

```
module.exports = {
  build: {
    vendor: ['vue-notifications']
  },
  plugins: ['~/plugins/vue-notifications']
}
```

## Do you always need it?

Nuxt.js is such a good tool that we would like to use it in every project. We can, but we don't always need to.

If you create a small project, for which you don't care about SEO and your file structure is not overwhelming, it's not needed. The same goes for projects with uncomplicated

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. **I understand**

Another situation is when you already have your own well working boilerplate. If it does the same as Nuxt, learning a new tool and switching to it is not necessary.

However, if you want to create a big app, with good SEO or you have trouble handling complex apps with just Vue, we really recommend you try Nuxt.js.

## Conclusion

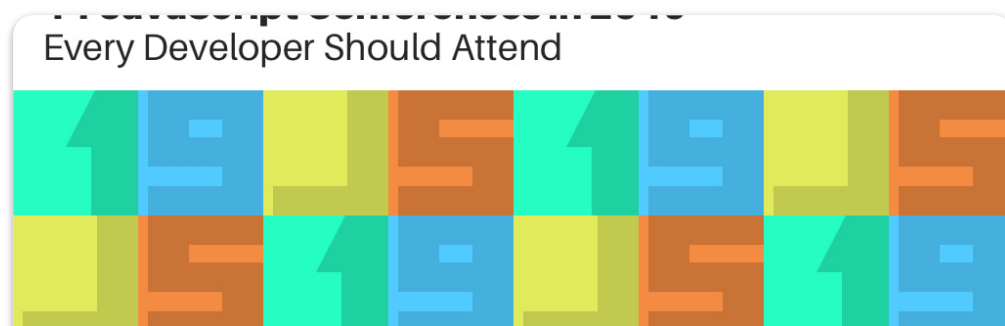
[Nuxt.js](#) is a convenient tool that allows you to create really advanced apps with a small amount of work.

If you're a [Vue.js](#) fan and want to create more than a simple to-do App, you will eventually need it. It makes developing bigger apps really easy. Do you always need it? No, but in most cases, you eventually will.

Share this Blog Post



## Recommended Posts



We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. [I understand](#)

## Developer Should Attend

If you're working with JavaScript, here are some must-attend conferences to engage with the JS community in 2019. Time to learn, connect, and build!



## GENERATE A STATIC MARKDOWN BLOG USING VUELOG

October 03, 2017

### Generate a Static Markdown Blog Using Vuelog

Vuelog is an impressive Vue.js project utilizing some of the latest technologies. In this tutorial, we are going to go through cloning and setting up a static blogging application with Vuelog.

[About Us](#)

[Contact Us](#)

[Help Center](#)

[Careers](#)

[Privacy & Security](#)

[Changelog](#)

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. [I understand](#)



[PRODUCTS](#)

[SOLUTIONS](#)

[DOCS](#)

[BLOG](#)

[TRY JSCRAMBLER FOR FRE](#)

---

Copyright © Jscrambler 2019 *All Rights Reserved*

We use cookies to give you the best and most relevant experience. By using Jscrambler you are complying with this. **I understand**