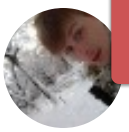


ithub or

@xenetics on Steem.

Jan 19 · 8 min read

Build social media platform with the Steem Blockchain #1—Vue.js, Storybook.js and Steem.js

Image: Fintech News

Welcome to a new series, where we will build a full blogging platform with Vue.js and Storybook, based on the Steem blockchain!

Learn how to use Storybook.js for Vue, and how to build decoupled applications using the component/container pattern. The source for this part is available here. The next part of the series, "making a post", is available here.

This series will showcase all the best practices, patterns and tools I learned building large applciations using Vue.js, TDD and the other front tools over the last two years.

In this introduction, we will:

1. Set up the Vue app
2. Configure Storybook.js for Vue
3. Fetch data from the Steem blockchain using `steem.js`
4. Display data using the container/component pattern
5. Design a `<Profile>` component in storybook

Why storybook? What is the container/component pattern?

Storybook helps further decouple presentational components (ones focused on just displaying things) and container components, which handle API calls, manipulate and store local state, and so on.

Storybook allows designers and developers to work together more effectively, by not getting in each other's way. Developers spend most of their time working on business logic, API calls, and in Vuex stores, and designers tend to focus on CSS, layouts and how things looks and flow.

This also makes your applcation more testable—presentational component tests normally assert whether the props are correct, and the markup is accurate. Container components are more about calling functions, interacting with external services and manipulating data to be passed to a presentational component. Dan Abramov explains it well here.

What is Steem?

> *Steem is a blockchain-based rewards platform for publishers to monetize content and grow community.*

So basically a distributed social network, build on blockchain technology. There are several web clients, such as steemit.com. You can also follow me there.

This means the content and user base already exists, we will be building a different client with some extra features.

Getting Started

We will bootstrap the app using `vue-cli`. If you haven't installed that, run `yarn global add vue-cli`. Now, create the application with `vue init webpack vue-steem`. I will be building a full app with routing, unit tests [jest] and e2e tests. If you want to follow along, say yes to all the prompts when running `vue init`.

(Optional) Install vue-component-scaffold

I wrote a small tool npm module to generate Vue component templates and the accompanying test called `vue-component-scaffold`. You can

read more it here if you are interested install with `npm install -g vue-component-scaffold` . I will be using during this series.

Install steem.js

Simply run `yarn install steem—save` . Steem is a blockchain based social media platform. We will use the API to serve profiles, posts, do upvotes, and claim rewards. The most common interface to access content is steemit.com.

Create the initial Profile route and components

Let's create some directories for the initial /profile route, and components. We will see what these are used for soon.

```
mkdir src/views
mkdir src/views/profile
vc src/views/profile/Index -t
vc src/views/profile/Profile -t
```

> `vc` *is the command for* `vue-component-scaffold` *. If you don't want to use it, just create empty* `.vue` *files with the same name and follow along.*

Inside of `views/profile/Index.vue` , add:

```
<template>
  <div>
    Profile!!!
    <ProfileContainer></ProfileContainer>
  </div>
</template>

<script>
  import ProfileContainer from './ProfileContainer'

  export default {
    name: 'ProfileIndex',

```

Initial views/profiles route.

Now edit `src/router/index.js` and use your newly created route.

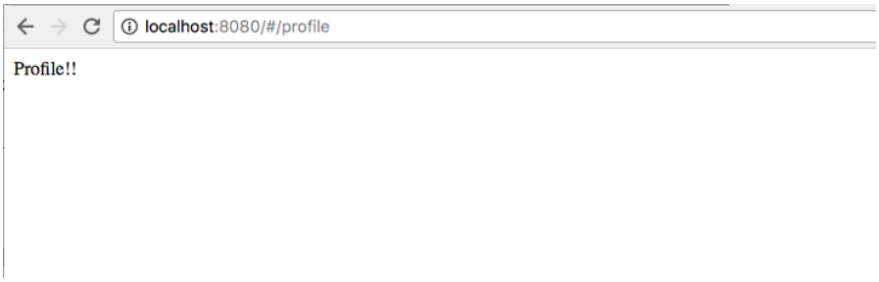
```
import Vue from 'vue'
import Router from 'vue-router'
import HelloWorld from '@/components/HelloWorld'
import ProfileView from '@/views/profile/Index'

Vue.use(Router)

export default new Router({
  routes: [
    {
      path: '/',
      name: 'HelloWorld',
      component: HelloWorld
```

Add the component to `vue-router`

Run `yarn dev` to start the dev server. Assuming you typed everything correctly and I didn't make any mistakes, `localhost:8080/#/profile` should show you a white page with Profile!!! like this:



Profile/Index.vue, added to routing

Looking good.

We installed `steem.js` earlier. Let's use that to grab some data, and then we will add storybook, and start to see container/component at work.

First, inside of `/views/profile/ProfileContainer.vue`, created a `mounted` method, in which we will make the API call and get some data about the user.. Also we will need to import the `steem` module.

The <script> section of ProfileContainer.vue, using steem.js to get some data for the user 'xenetics' (me).

If you refresh the page, you should see some data printed n the console with a ton of fields. We will be working with this data. The field we are interested in for now is `json_metadata`, which contains the account name, location, bio, and profile picture. Let's display it! But first, take a step back and and consider, what is the best way to do so?

We now have two components to work with: `Profile.vue` and `ProfileContainer.vue`. The latter currently makes an API call. Another way to look at it `ProfileContainer.vue` is that it is a smart component, or what is often called a *container* component. It fetches and manipulates data. `Profile.vue` will be a what is often known as a *presentational* component—it will show some data, but should not have knowledge of where the data comes from, or it's contents.

We will use storybook to work on our application UI—this will make the distinction between containers and components more clear.

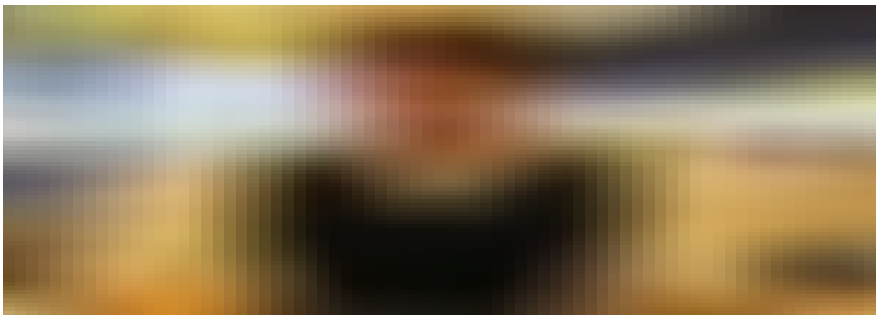
Install and Configure Storybook.js

Install storybook by running `yarn add @storybook/vue --save`. Next, inside of `package.json`, add:

```
{
  "scripts": {
    "storybook": "start-storybook -p 9001 -c .storybook"
  }
}
```

This is the storybook startup script. Now we need to create a `.storybook` directory, on the same level as `src`. Inside, add

`config.js` with the following:



.storybook/config.js.

Next, inside of `.storybook` again, create `webpack.config.js`. We need to extend the default webpack config, by just a little (this setup can take a bit of work, but it's worth it)!

.storybook/webpack.config.js. Extended config to load .vue files.

In the above `config.js` (not `webpack.config.js`, the one above), we did `require` `..src/stories` which doesn't exist. Go ahead and makea `src/stories` directory, and inside create `index.js` and `profile.js`. `index.js` will simply import the story for `Profile.vue` which we are about to create!

`index.js` simply contains:

```
import profile from './profile'
```

`profile.js` contains the story:

stories/profile.js, our first story.

If you type/copypasted all that correctly, `run yarn storybook -c` and visit `localhost:9001`, and you should see:

Our newly created storybook with an empty story for Profile.vue.

Now we can get to work. Notice in `profile.js` in the `template` property, we just render `<Profile />`. The `Profile.vue` component will not be able to make API calls, or dispatch actions, or anything fancy. Image storybook in an isolated environment, without an internet connection. The UI should have nothing to do with external services, APIs, or business logic. Just render data from `props`. This way, we can use storybook to see if the UI is correct, and unit tests to make sure the data and props are correct.

We want to be able to develop our presentation components, like `Profile.vue` completely decoupled from the application logic. Let's start there. Inside of `Profile.vue`, add the following:

Passing a prop to Profile.vue to display.

Now `Profile.vue` knows it will be receiving a `profileImage`, by the `props`, which will be a link to wherever the user's image is stored. Inside of `stories/profile.js`, update the story to include an image like so:

Updated story to render some an image

I know I said storybook should operate without relying on external services! You probably are thinking "hey, Vue.js is an external service…!", and of course you are correct. Later on, we will set up some static images saved locally. This is fine for now, to make sure storybook and our first story are set up and working correctly.

Assuming you still have the storybook server running, visit `localhost:9001`. You should be greeted with:

The Profile story, rending a profile image.

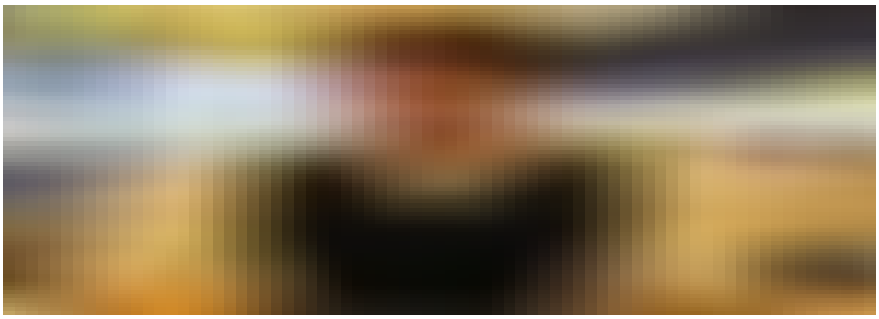
Great! Fantastic work. You just wrote your first story for a completely decoupled UI components. Let's go ahead and write some logic in `ProfileContainer.vue`, so we can use the image returned from the Steem API.

First we will update `ProfileContainer.vue`, `data` and `mounted`.

Updating ProfileContainer.vue mounted and data to save the response.

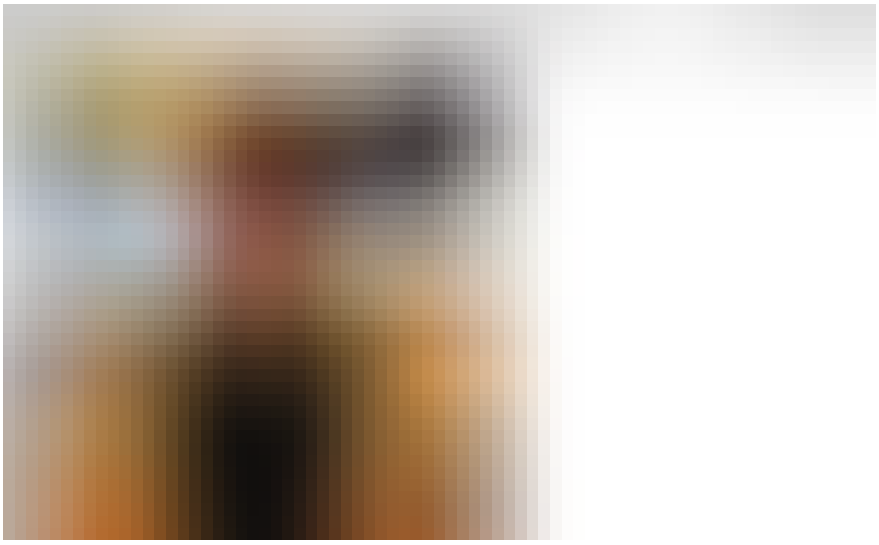
We added `result`, to save the response, and `loaded`. We don't want to render `<Profile>` until the API call is complete. We write the result of the API call to `result` and set `loaded` to be `true` when the API call (hopefully) succeeds.

Next, the markup:



Updating ProfiileContainer.vue markup to pass the prop.

Nothing too exciting—just pass the `profile_image` property of `result.profile`. Now. visiting `localhost:8080/#/profile` …



Showing an image passed by props to Profile.vue.

Great job! We knew this would work since we saw it in `stories/profile.js`, but it's good to see it live as well.

Let's add in the rest of the profile data. This time, I'm just going to pass the entire `result.profile` as a `prop`. In the future, I'd like to extract the avatar into a separate component, which I might use in other places on the website, such as a list of followers or next my posts, so I want to key it separate from the other data.

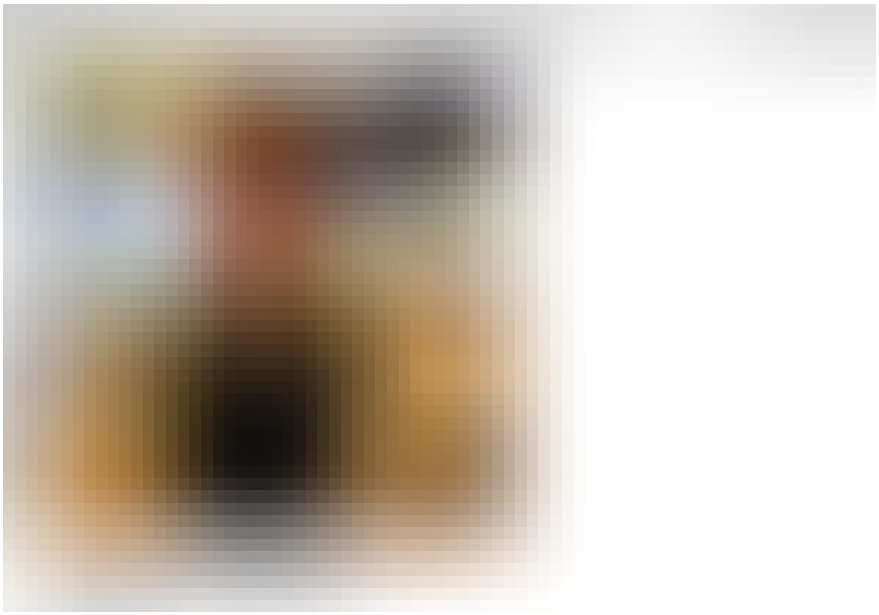
We only need to update the `<template>` section of `ProfileContainer.vue`:

Passing profile-data to Profile.vue.

And then add the new prop and markup in `Profile.vue` :

Adding the profile data in Profile.vue's markup.

`localhost:8080/#/profile` should be looking like this:



Profile.vue with some additional details.

Don't forget to update `stories/profile.js`.

Updated stories/profile.js for Profile.vue.

Beautiful!

Now we are set up for a large app—our data layer in `ProfileContainer.vue` and the presentation layer in `Profile` are decoupled. We also set up storybook. The app looks pretty ugly now—but that's fine. The developers can go ahead and add more functionality, and we can hand the repository over to the designers, who can work on the UI without even needing to make an API calls—armed with storybook, all the designers need to do is work in `Profile.vue`, while the developers focus on build business logic in the form of containers.

Better yet, the designers could even go ahead and make components using storybook, assuming they know what kind of data they can expect to get. Perhaps a designer could start building a `Post.vue` component, which formats a blog post nicely. The props are obvious enough, and when the developers can, they can hook it up to a `PostContainer.vue`, and pass the props to the `Post.vue` presentational component.

The last thing to do is write some unit tests. We want to make sure the API call is made and the result assigned correctly. At that time, `loaded` should be set to `true` and reveal the `<Profile>` component. I won't do that, because in the next part of this series we will move the API call to a service, which is in turn called by dispatching a Vuex action.

Next article we will improve the UI a bit, and show the followers, as well as start setting up authentication.

Thanks for reading. Any comments or questions are welcome, and again, the source for this part is available here.