After reading this guide, you'll know:

1. What Vue is, and why you should consider using it with Meteor
2. How to install Vue in your Meteor application, and how to use it correctly
3. How to integrate Vue with Meteor's realtime data layer
4. How to structure Meteor with Vue
5. How to Server-side Render (SSR) Vue with Meteor

Vue already has an excellent guide with many advanced topics already covered. Some of them are SSR (Server-side Rendering), Routing, Code Structure and Style Guide and State Management with Vuex.

This documentation is purely focused on integrating it with Meteor.

> *Meteor has a Vue skeleton which will prepare a basic Vue Meteor app for you. You can create one by running* `meteor create vue-meteor-app --vue` *There is also a Vue tutorial which covers the basics of this section.*

Introduction

Vue (pronounced /vju:/, like view) is a progressive framework for building user interfaces. Unlike other monolithic frameworks, Vue is designed from the ground up to be incrementally adoptable. The core library is focused on the view layer only, and is easy to pick up and integrate with other libraries or existing projects. On the other hand, Vue is also perfectly capable of powering sophisticated Single-Page Applications when used in combination with modern tooling and supporting libraries.

Vue has an excellent guide and documentation. This guide is about integrating it with Meteor.

Why use Vue with Meteor

Vue is a frontend library, like React, Blaze and Angular.

Some really nice frameworks are built around Vue. Nuxt.js for example, aims to create a framework flexible enough that you can use it as a main project base or in addition to your current project based on Node.js. Though Nuxt.js is full-stack and very pluggable, it lacks an API to communicate data from and to the server. Also unlike Meteor, Nuxt still relies on a configuration file.

Meteor's build tool and Pub/Sub API (or Apollo) provides Vue with this API that you would normally have to integrate yourself, greatly reducing the amount of boilerplate code you have to write.

Integrating Vue With Meteor

To start a new project:

```
meteor create vue-meteor-app
```

To install Vue in Meteor, you should add it as an npm dependency:

```
meteor npm install --save vue
```

To support Vue's Single File Components with the .vue file extensions, install the following Meteor package created by Vue Core developer Akryum (Guillaume Chau).

```
meteor add akryum:vue-component
```

You will end up with at least 3 files:

1. a `/client/App.vue` The root component of your app
2. a `/client/main.js` Initializing the Vue app in Meteor startup
3. a `/client/main.html` containing the body with the #app div

We need a base HTML document that has the `app` id. If you created a new project from `meteor create .`, put this in your `/client/main.html`.

```
<body>
  <div id="app"></div>
</body>
```

You can now start writing .vue files in your app with the following format. If you created a new project from `meteor create .`, put this in your `/client/App.vue`.

```
<template>
  <div>
    <p>This is a Vue component and below is the current date:<br />{{date}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      date: new Date(),
    };
  }
}
</script>

<style scoped>
  p {
    font-size: 2em;
    text-align: center;
  }
</style>
```

You can render the Vue component hierarchy to the DOM by using the below snippet in you client startup file. If you created a new project from `meteor create .`, put this in your `/client/main.js`.

```
import Vue from 'guide/site/content/vue';
import App from './App.vue';
import './main.html';

Meteor.startup(() => {
  new Vue({
    el: '#app',
    ...App,
```

```
  });
});
```

Run your new Vue+Meteor app with this command: `NO_HMR=1 meteor`

Using Meteor's data system

One of the biggest advantages of Meteor is definitely it's realtime data layer. It allows for so called full-stack reactivity and optimistic UI functionality. To accomplish full-stack reactivity, Meteor uses Tracker. In this section we will explain how to integrate Meteor Tracker with Vue to leverage the best of both tools.

1. Install the vue-meteor-tracker package from NPM:

```
meteor npm install --save vue-meteor-tracker
```

Next, the package needs to be plugged into Vue as a plugin. Add the following to your `/client/main.js`:

```
import Vue from 'vue';
import VueMeteorTracker from 'vue-meteor-tracker'; // import the integration
package!
import App from './App.vue';
import './main.html';

Vue.use(VueMeteorTracker);                          // Add the plugin to Vue!

Meteor.startup(() => {
  new Vue({
    el: '#app',
    ...App,
  });
});
```

Example app

If you've followed the integration guide, then your Vue application shows the time it was loaded.

Let's add some functionality that makes this part dynamic. To flex Meteor's plumbing, we'll create:

1. A Meteor Collection called `Time` with a `currentTime` doc.
2. A Meteor Publication called `Time` that sends all documents
3. A Meteor Method called `UpdateTime` to update the `currentTime` doc.
4. A Meteor Subscription to `Time`
5. Vue/Meteor Reactivity to update the Vue component

The first 3 steps are basic Meteor:

1. In `/imports/collections/Time.js`

```
Time = new Mongo.Collection("time");
```

2. In `/imports/publications/Time.js`

```
Meteor.publish('Time', function () {
  return Time.find({});
});
```

3. In `/imports/methods/UpdateTime.js`

```
Meteor.methods({
  UpdateTime() {
    Time.upsert('currentTime', { $set: { time: new Date() } });
  },
});
```

Now, let's add these to our server. First remove autopublish so our publications matter:

```
meteor remove autopublish
```

For fun, let's make a `settings.json` file:

```
{ "public": { "hello": "world" } }
```

Now, let's update our `/server/main.js` to use our new stuff:

```
import { Meteor } from 'meteor/meteor';

import '/imports/collections/Time';
import '/imports/publications/Time';
import '/imports/methods/UpdateTime';

Meteor.startup(() => {
  // Update the current time
  Meteor.call('UpdateTime');
  // Add a new doc on each start.
  Time.insert({ time: new Date() });
  // Print the current time from the database
  console.log(`The time is now ${Time.findOne().time}`);
});
```

Start your Meteor app, your should see a message pulling data from Mongo. We haven't made any changes to the client, so you should just see some startup messages.

```
meteor
```

4. and 5) Great, let's integrate this with Vue using Vue Meteor Tracker and update our `/client/App.vue` file:

```
<template>
  <div>
    <div v-if="!$subReady.Time">Loading...</div>
    <div v-else>
      <p>Hello {{hello}},
        <br>The time is now: {{currentTime}}
      </p>
      <button @click="updateTime">Update Time</button>
      <p>Startup times:</p>
      <ul>
        <li v-for="t in TimeCursor">
          {{t.time}}  -  {{t._id}}
        </li>
      </ul>
      <p>Meteor settings</p>
      <pre><code>
        {{settings}}
      </code></pre>
    </div>
  </div>
</template>

<script>
import '/imports/collections/Time';

export default {
  data() {
    console.log('Sending non-Meteor data to Vue component');
    return {
      hello: 'World',
      settings: Meteor.settings.public,   // not Meteor reactive
    }
  },
  // Vue Methods
  methods: {
    updateTime() {
      console.log('Calling Meteor Method UpdateTime');
      Meteor.call('UpdateTime');          // not Meteor reactive
    }
  },
  // Meteor reactivity
  meteor: {
    // Subscriptions - Errors not reported spelling and capitalization.
    $subscribe: {
      'Time': []
    },
    // A helper function to get the current time
    currentTime () {
      console.log('Calculating currentTime');
      var t = Time.findOne('currentTime') || {};
      return t.time;
```

```
    },
    // A Minimongo cursor on the Time collection is added to the Vue instance
    TimeCursor () {
      // Here you can use Meteor reactive sources like cursors or reactive vars
      // as you would in a Blaze template helper
      return Time.find({}, {
        sort: {time: -1}
      })
    },
  }
}
</script>

<style scoped>
  p {
    font-size: 2em;
  }
</style>
```

Restart your server to use the `settings.json` file.

```
meteor --settings=settings.json
```

Then refresh your browser to reload the client.

You should see:

- the current time
- a button to Update the current time
- startup times for the server (added to the Time collection on startup)
- The Meteor settings from your settings file

Excellent! That's a tour of some of Meteor's features, and how to integrate with Vue. Have a better approach? Please send a PR.

Style Guide and File Structure

Like code linting and style guides are tools for making code easier and more fun to work with.

These are practical means to practical ends.

1. Leverage existing tools
2. Leverage existing configurations

Meteor's style guide and Vue's style guide can be overlapped like this:

1. Configure your Editor
2. Configure eslint for Meteor
3. Review the Vue Style Guide
4. Open up the ESLint rules as needed.

Application Structure is documented here:

1. Meteor's Application Structure is the default start.

2. Vuex's Application Structure may be interesting.

SSR and Code Splitting

Vue has an excellent guide on how to render your Vue application on the server. It includes code splitting, async data fetching and many other practices that are used in most apps that require this.

Basic Example

Making Vue SSR to work with Meteor is not more complex then for example with Express. However instead of defining a wildcard route, Meteor uses its own server-render package that exposes an `onPageLoad` function. Every time a call is made to the server side, this function is triggered. This is where we should put our code like how its described on the VueJS SSR Guide.

To add the packages, run:

```
meteor add server-render
meteor npm install --save vue-server-renderer
```

then connect to Vue in `/server/main.js` :

```
import { Meteor } from 'meteor/meteor';
import Vue from 'vue';
import { onPageLoad } from 'meteor/server-render';
import { createRenderer } from 'vue-server-renderer';

const renderer = createRenderer();

onPageLoad(sink => {
  console.log('onPageLoad');

  const url = sink.request.url.path;

  const app = new Vue({
    data: {
      url
    },
    template: `<div>The visited URL is: {{ url }}</div>`
  });

  renderer.renderToString(app, (err, html) => {
    if (err) {
      res.status(500).end('Internal Server Error');
      return
    }
    console.log('html', html);

    sink.renderIntoElementById('app', html);
  })
})
```

Luckily Akryum has us covered and provided us with a Meteor package for this: akryum:vue-ssr allows us to write our server-side code like below:

```
import { VueSSR } from 'meteor/akryum:vue-ssr';
import createApp from './app';

VueSSR.createApp = function () {
  // Initialize the Vue app instance and return the app instance
  const { app } = createApp();
  return { app };
}
```

Server-side Routing

Sweet, but most apps have some sort of routing functionality. We can use the VueSSR context parameter for this. It simply passes the Meteor server-render request url which we need to push into our router instance:

```
import { VueSSR } from 'meteor/akryum:vue-ssr';
import createApp from './app';

VueSSR.createApp = function (context) {
  // Initialize the Vue app instance and return the app + router instance
  const { app, router } = createApp();

  // Set router's location from the context
  router.push(context.url);

  return { app };
}
```

Async data and Hydration

Hydration is the the word for loading state into components on the serverside and then reusing that data on the clientside. This allows components to fully render their markup on the server and prevents a 're-render' on the clientside when the bundle is loaded.

Nuxt solves this gracefully with a feature called asyncData.

Meteor's pub/sub system is at this moment not suitable for SSR which means that if we want the same functionality, we will have to implement it ourselves. How that can be done is exactly what we are going to explain here!

> *Important reminder here is the fact that Server Rendering on its own is already worth a guide - which is exactly what the guys from Vue did. Most of the code is needed in any platform except Nuxt (Vue based) and Next (React based). We simply describe the best way to do this for Meteor. To really understand what is happening read that SSR guide from Vue.*

SSR follows a couple of steps that are almost always the same for any frontend library (React, Vue or Angular).

1. Resolve the url with the router
2. Fetch any matching components from the router
3. Filter out components that have no asyncData
4. Map the components into a list of promises by return the asyncData method's result

5. Resolve all promises
6. Store the resulting data in the HTML for later hydration of the client bundle
7. Hydrate the clientside

Its better documented in code:

```
VueSSR.createApp = function (context) {

  // Wait with sending the app to the client until the promise resolves (thanks
Akryum)
  return new Promise((resolve, reject) => {
    const { app, router, store } = createApp({
      ssr: true,
    });

    // 1. Resolve the URL with the router
    router.push(context.url);

    router.onReady(async () => {
      // 2, Fetch any matching components from the router
      const matchedComponents = router.getMatchedComponents();

      const route = router.currentRoute;

      // No matched routes
      if (!matchedComponents.length) {
        reject(new Error('not-found'));
      }

      // 3. Filter out components that have no asyncData
      const componentsWithAsyncData = matchedComponents.filter(component =>
component.asyncData);

      // 4. Map the components into a list of promises
      // by returning the asyncData method's result
      const asyncDataPromises = componentsWithAsyncData.map(component => (
        component.asyncData({ store, route })
      ));

      // You can have the asyncData methods resolve promises with data.
      // However to avoid complexity its recommended to leverage Vuex
      // In our case we're simply calling Vuex actions in our methods
      // that do the fetching and storing of the data. This makes the below
      // step really simple

      // 5. Resolve all promises. (that's it)
      await Promise.all(asyncDataPromises);

      // From this point on we can assume that all the needed data is stored
      // in the Vuex store. Now we simply need to grap it and push it into
      // the HTML as a "javascript string"
```

```
      // 6. Store the data in the HTML for later hydration of the client bundle
      const js = `window.__INITIAL_STATE__=${JSON.stringify(store.state)};`;

      // Resolve the promise with the same object as the simple version
      // Push our javascript string into the resolver.
      // The VueSSR package takes care of the rest
      resolve({
        app,
        js,
      });
    });
  });
};
```

Awesome. When we load our app in the browser you should see a weird effect. The app seems to load correctly. That's the server-side rendering doing its job well. However, after a split second the app suddenly is empty again.

That's because when the client-side bundle takes over, it doesn't have its data yet. It will override the HTML with an empty app! We need to hydrate the bundle with the JSON data in the HTML.

If you inspect the HTML via the source code view, you will see the HTML source of your app accompanied by the `__INITIAL_STATE=""` filled with the JSON string. We need to use this to hydrate the clientside. Luckily this is fairly easy, because we have only one place that needs hydration: the Vuex store!

```
import { Meteor } from 'meteor/meteor';
import createApp from './app';

Meteor.startup(() => {
  const { store, router } = createApp({ // Same function as the server
    ssr: false,
  });

  // Hydrate the Vuex store with the JSON string
  if (window.__INITIAL_STATE__) {
    store.replaceState(window.__INITIAL_STATE__);
  }
});
```

Now when we load our bundle, the components should have data from the store. All fine. However there is one more thing to do. If we navigate, our newly rendered clientside components will again not have any data. This is because the `asyncData` method is not yet being called on the client side. We can fix this using a mixin like below as documented in the Vue SSR Guide.

```
Vue.mixin({
  beforeMount () {
    const { asyncData } = this.$options
    if (asyncData) {
      // assign the fetch operation to a promise
      // so that in components we can do `this.dataPromise.then(...)` to
      // perform other tasks after data is ready
      this.dataPromise = asyncData({
```

```
        store: this.$store,
        route: this.$route
      })
    }
  }
})
```

We now have a fully functioning and server-rendered Vue app in Meteor!