Create a modular single-page app with Vue.js and Bluemix, Part 1: Develop and test the front end

Use Vue.js, webpack, Foundation, and NPM to build a RESTful app with a responsive UI and CRUD capability

Matt C. Tyson

February 03, 2016

In this two-part tutorial, build a simple but powerful single-page application with a responsive UI, and deploy your app in the cloud. In Part 1, develop the front end with the Vue.js JavaScript framework, using Node Package Manager (NPM) for dependency management, webpack as a build tool, and Foundation for the responsive UI. Learn Vue.js from basic principles to advanced usage, and use a modular design to support the application as it scales up. In Part 2, deploy the app to IBM® Bluemix®.

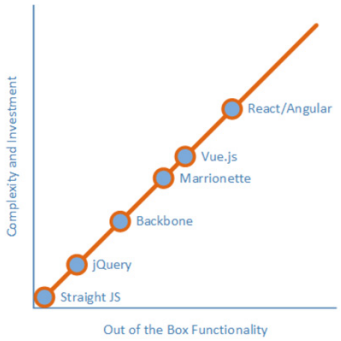
When you create an application that will be deployed into the browser environment, you must solve what I call the *application/view-state synchronization* problem: You need to control the end visual result (the shape of the DOM), and you need to manipulate the application data. The interaction between the DOM and the data grows in complexity as your application grows, so managing that interaction on your own becomes an error-prone proposition. Ideally, then, you offload that responsibility to a third-party framework. You want to solve the app/view-state synchronization problem with as little of your own effort as possible.

Among the several JavaScript frameworks that you can choose from, the new-generation Vue.js framework focuses tightly on solving the application/view-state synchronization problem, with a minimum of extraneous features. Vue.js is an object-oriented, data-driven DOM management system — or, put more succinctly, an object-to-DOM binding system. Vue.js deals with manifesting your application's data in the browser DOM.

This two-part tutorial series introduces you to the basics of Vue.js and then shows advanced Vue usage. In Part 1, you use Vue, NPM, and webpack to build a small but fully functional UI for a modular application, along with a production-grade build-and-dependency pipeline. In Part 2, you make the app live in the cloud via the IBM Bluemix Platform as a Service. When you're done, you'll be well prepared to put Vue to work in your own projects. See Downloadable resources to get the full sample code for Part 1.

A hands-on introduction to Vue.js

With a minimum of complexity, Vue.js supports complex and large-scale requirements. The framework gives you enough to build anything you need, without layering on a universe of other features. Compared to other frameworks, in my experience, Vue hits a sweet spot of simplicity and capability along the complexity-versus-features spectrum:



I start with a guided tour of Vue.js basics so that you understand how Vue works and how best to use it. Then, building on that foundation, you'll use more advanced Vue techniques and modern, production-grade supporting tools to build the app and pipeline.

Simple value binding

Create a directory called recruiteranking for a new project. Then download the latest Vue package, save it in a separate location, and create an index.html file in recruiteranking that includes the Vue package, as shown Listing 1.

Listing 1. Basic index.html file

```
<html>
    <head>

        <meta charset="utf-8">
        <script type="text/javascript" src="vue.js"></script>
    </head>
    <body>
  <div id="test">
  <p>User: {{ username }}</p>
  </div>
          <script>
test = new Vue({  // 1 – Instantiate a vue instance
  el: '#test',
  data: {
    username: "Luke Skywalker"
  }
});
        </script>
    </body>
</html>
```

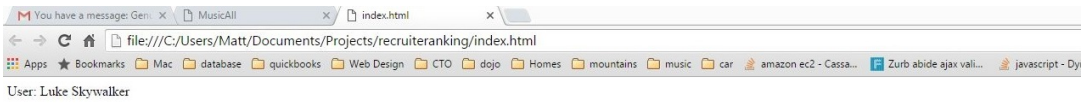
> *“ Most things in Vue happen through a Vue instance, which is why I describe Vue as object-oriented. ”*

In Listing 1, notice where you create a `Vue` instance within the `<script>` tag (flagged with comment 1). Most things in Vue happen through a `Vue` instance, which is why I describe Vue as object-

oriented. This `Vue` instance has only two fields, both of them key attributes of the `Vue` class. The `el` attribute tells Vue which DOM element it will bind to; this attribute can be any query resolving to a single element (or the actual element). If you're familiar with Backbone, think of this field as similar to the `el` property in that framework.

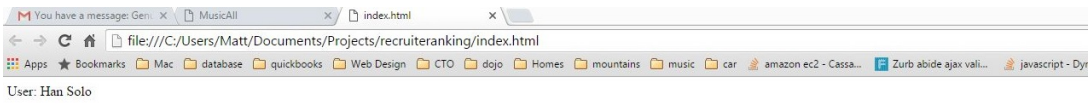
The second property is the `data` field. This property is the magical field on the `Vue` object that represents the instance state, and it can interact with the DOM automatically. The `data` property is also an object that contains a `username` field. The `username` field is one that I made up. The `data` property can contain any fields, of any type, including arrays and complex nested objects, with any names. Now look up into the body of the HTML, where the paragraph element inside the `<div>` with `id="test"` contains the `{{ username }}` token. This token is a template token that's exactly analogous to a mustache token (which is similar in intent to a JavaServer Pages Expression Language token of the form `${}`).

Vue associates the `test<div>` with the `test` instance and replaces the `username` token with the value of the `username` field on the instance. So if you load the page, it displays **User: Luke Skywalker**:



Now, open a JavaScript console in your browser. (If you're using Chrome, press F12 to open the developer console.) Because you gave your `Vue` instance a reference in the global scope, you can access the `test` instance in the console. You can change the user name that's displayed on the page by changing the `data` object on your `test Vue` instance. How do you get access to the `data` field? You don't. The `data` field is set directly on the instance: `test.username`.

So, if you enter `test.username = "Han Solo";` into the console, you see the change reflected in the UI, where **User: Han Solo** replaces **User: Luke Skywalker**:



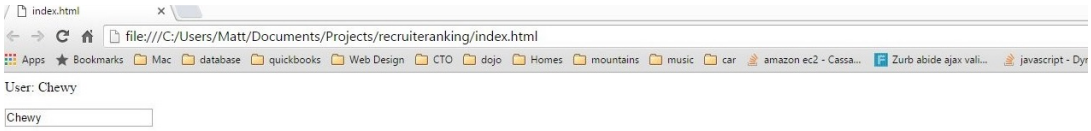
Methods

The change that you just made breaks encapsulation by reaching into the state of the `Vue` instance and altering it directly. Vue gives you a better alternative. Listing 2 adds a `methods` property to the `test` object.

Listing 2. `methods` property

```
test = new Vue({
  el: '#test',
  data: {
    username: "Luke Skywalker"
  },
  methods: {
    changeName: function(name){
      this.username=name;
    }
  }
});
```

The `test Vue` instance now has a `methods` field, with a `changeName` method on it. This simple method takes an argument that is then used to set the `username` data member that you saw previously. Now in the JavaScript console, you can access the `changeName` method **directly** as a method on the `test` instance. Open the JavaScript console and enter `test.changeName("Chewy");`.



Input bindings

Now, you take this simple example a step further and see how Vue supports binding to a user input field. Revisit the `test <div>` and add in an `input` field with the `v-model` attribute set to `username`:

```
<div id="test">
  <p>User: {{ username }}</p>
  <input v-model="username">
  </div>
```

Vue borrows the term *directive* from Angular, and the `v-model` attribute is considered a directive also. With this small addition, the UI now features a form input field that's bound to the Vue model. The binding is *two-way*: If the Vue data model changes, the form field changes, and vice versa.

The `v-model` directive makes it easy to create a two-way binding against the form input. You can test the two-way aspect by running `test.changeName("Yoda")` in your console. You'll see that both the text display and the form input values change to reflect the update.

Event handling

The last basic feature in this Vue intro is user event handling. You handle events via directive. In this case, the form is `v-on:events`, where `events` are the events that you want to listen on, such as `click`.

In the markup, add a button that uses event handling:

```
<div id="test">
  <p>User: {{ username }}</p>
  <input v-model="username">
  <button v-on:click="defaultUser">Default User</button>
  </div>
```

(Using Vue shorthand syntax, you can shorten the event directive to `@`. So you could rewrite the button handler as `<button @click="defaultUser">Default User</button>`.)

Listing 3 contains the corresponding change to the JavaScript: the addition of a `defaultUser` method that sets the `username` back to `Luke Skywalker`.

Listing 3. Adding a click handler to the Vue instance

```
test = new Vue({
  el: '#test',
  data: {
    username: "Luke Skywalker"
  },
  methods: {
    changeName: function(name){
      this.username=name;
    },
    defaultUser: function(){
      this.changeName("Luke Skywalker");
    }
  }
});
```

Verify that the `v-on:click` handler is working by clicking the button and observing that all of the elements that are bound to the `username` property are updated to the default value.

Now, you have the state, the `data` property, the behavior, and the `methods` property for Vue in your sights. Combined with the `el` property, event handling, and bindings, those are the core elements of the API. That simple foundation gives you great breadth of power, as you'll see as you read on.

Demo app and tech stack

The back end

The focus here is on the demo app's front end. A simple demo back end that I set up via the Spark framework provides simple endpoints to show off the front-end JavaScript. No database is involved; all data is in memory. You'll see the internals of the back end in Part 2, when you move into production deployment.

You're ready to start building a more sophisticated demo application: a RESTful single-page app. (You continue working in the recruiteranking project's files, so don't delete them.) Vue is the star of this show, but to put everything together, you use other technologies as the supporting cast. In particular, you use webpack as your build system. webpack is a powerful, highly configurable system; you use only a small range of its capabilities. You also use Foundation for responsive CSS layout, but Foundation remains fairly transparent. And you use NPM.

Using this tech stack, you build an application that helps you rank the recruiting companies that you've dealt with. The app sports a logo-and-menu bar at the top and a main grid that contains recruiting companies. The grid rows have a ranking from 1 to 5, and the rows are expandable to enable editing of the company details.

You need Node and NPM to install webpack and then include dependencies. Follow these instructions for installing both.

An aim of your build process is to install **no** global dependencies other than Node and NPM. This approach keeps your workspace clean. And all dependencies are explicitly listed in the project and can be deployed with a single command in any environment.

To verify that your system is ready, drop into the command line and run `npm -v`. If you get a version as a response, your NPM install is ready to go.

Managing dependencies with NPM

Create a directory for the project if you didn't do so earlier:

```
mkdir recruiteranking
```

Change to recruiteranking and get a default package.json file by running:

```
npm init
```

In a moment, you define your development dependencies in package.json — and through the power of webpack, you also specify deployment dependencies, such as jQuery. But before moving ahead, set up tracking in Git to have a safety net for changes:

```
git init
git add
git commit -m "a single step"
```

Or instead of `"a single step"`, you can make it `"initial commit"` or whatever else suits your style.

Now, open package.json in a front-end IDE. (Sublime is my favorite.) Add your dependencies to package.json, which then serves as your list of record for everything the client depends on (the same way that Apache Maven does for a Java™ app). Remember that you want everything else to go through NPM, with no hidden dependencies; then you can run a single NPM command to install all of the packages that you need. Listing 4 shows the updated dependency list.

Listing 4. package.json list of dependencies

```
{
  "name": "recruiteranking",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "jquery": "2.1.1",
    "foundation-sites": "5.5.3",
    "webpack": "~1.12.6",
    "webpack-dev-server": "~1.14.0",
    "style-loader": "~0.6",
    "script-loader": "~0.5",
    "css-loader": "~0.23.0",
    "node-sass-loader": "~0.1.7",
    "sass-loader": "~3.1.2",
    "vue": "1.0.10",
    "vue-loader": "7.2.0"
  }
}
```

Hereafter, I show you only the changes to make, not the entire list.

In Listing 4, the highlight is the added `dependencies` property, which includes several packages:

- jQuery v.2.1.1, the well-known, well-loved JavaScript library. NPM and webpack together make jQuery available to your JavaScript and to all the other libraries. (Note that in NPM 3 and higher, peer dependencies must be explicitly stated in this way.)
- Foundation, which is your responsive framework.
- Two webpack dependencies: webpack itself and its development server, which you see in action soon.
- The `*-loader` dependencies. These are all used by webpack. webpack can consume and combine a wide variety of file types and package them together efficiently for inclusion as a JavaScript file in your end project. So the loaders are something like dialects for webpack, and here, you're giving webpack the ability to speak JavaScript, CSS, and Vue.
- Vue.js.

Go to the command line at the root of your project, where the package.json file exists, and type `npm install`. This command adds the node_modules directory and downloads the specified packages to that directory. Wait (it might be a while) until everything downloads.

Configuring webpack

Now that everything that you need to rely on is ready for you, you can use webpack to put it all together.

The webpack.config.js file

In the project root directory, add a webpack.config.js file. This file configures webpack; it's true JavaScript rather than JSON, giving you more flexibility. Listing 5 shows the starter webpack config.

Listing 5. webpack.config.js

```
var path = require("path");

module.exports = {
    entry: {
      main: "./app/js/main.js"
    },
    output: {
        path: __dirname,
        filename: "bundle.js"
    },
    module: {
        loaders: [
            { test: /\.css$/, loader: "style!css" },
            { test: /\.scss$/, loaders: ["style", "css", "sass"] },
            { test: /\.vue$/, loader: 'vue' }
        ]
    },
     sassLoader: {
    includePaths: [path.resolve(__dirname, "./node_modules/foundation-sites/scss/")]
    }
};
```

I'll drive you through this config file quickly, stopping briefly at each attraction.

webpack.config.js: Including modules

First, you define an includes `path`, which is a module that provides functions that work with the file system.

Next is a call to `module.exports`, which is CommonJS-speak for "here's what my package contains." webpack relies on this module when it builds a project. If you're familiar with Maven, you can think of `module.exports` as the webpack equivalent of the Maven Project Object Model (POM).

webpack.config.js: `entry`

Multiple entry points, outputs, and chunks

webpack supports multiple entry points and breaking application output into multiple bundles. These features can come in handy as a project grows in size and complexity, and to support incremental loading of a project.

One common use case is to define a vendor bundle that is loaded separately from the application bundle. Then, the application can load the often-changing app code when necessary while leaving the unchanged vendor code in users' browser caches.

You don't need these features here, but you can find in-depth coverage of them here.

The `entry` field tells webpack where to look to begin the project, acting as a root for module inclusions. You can see in Listing 5 that /app/js/main.js defines all of your project's runtime dependencies.

webpack.config.js: `output`

`output` tells webpack where to put the final result. Your index.html refers to this file when it includes the build.

webpack.config.js: loaders

`module` configures the loaders for your webpack. Loaders are responsible for reading raw assets and transforming them into the final bundle. Loaders are pluggable and customizable. The details on `module` tell the loaders which file types they should handle (for example, .css for the style loader). The `test` field means "if the file passes this test, then apply this loader." After `module` comes `sassLoader.includePaths`, which directs the Sass loader to look in the foundation directory that was added by the `foundation-sites` dependency in NPM — necessary so that foundation SCSS includes from within your app resolve correctly.

Your final build asset is a JavaScript file. The CSS/SCSS loader creates a final product that is bundled and injected by the JavaScript. No final CSS file is output.

Now you have production-grade dependency management and a production-grade build setup. You're ready to use both to build a UI.

Modularized application

Add the entry file, main.js, in /app/js/ (relative to your project root) with the contents:

```
$ = jQuery = require('jquery');
foundation = require('foundation-sites');
Vue = require("vue");
require('../css/app.scss');
```

In this initial main.js file, you're requiring jQuery and the Foundation package, along with your own local app.scss file, which is the entry point for your styles. Note that you give jQuery a couple of global identifiers, `$` and `jQuery`, as a workaround for some loose references in the Foundation JavaScript. These identifiers are standard enough not to cause you any problems. You also make `Vue` global, which you address in a moment.

Add the app.scss file at /app/css/app.scss, with the following contents:

```
@import "settings";
@import "foundation";
```

This file is where you include the Foundation Sass and any global styles of your own.

Enabling local testing and deployment

The first thing to do with your fancy new JavaScript/Vue/Foundation rig is to switch how you deploy and test the app. webpack includes a Node/Express-based development server that hosts the app, making life easy during coding. The development server also supports proxying, which comes in handy soon for interacting with the back end. To run the dev server, you must make one change to your package.json file. Because all of your dependencies are locally managed NPM packages, you need to run webpack through NPM. So in package.json, enable the dev server by adding this entry to the `scripts` object:

```
"scripts": {
    "server": "webpack-dev-server"
  }
```

Now drop to the command line and run `npm run-script server` to get the app running on localhost, at 8080. `run-scripts` is an NPM facility that runs the designated script, with all of the local dependencies that are defined in package.json loaded. You now have a one-line way to load all of your dependencies, package them, and test them in the browser. Even better, `webpack-dev-server` hot-deploys your code changes in real time. The dev server bundles the build into memory and deploys from there; it doesn't place any output file in the file system, making for fast redeployment when the code changes.

Modifying index.html and main.js

Now, in the index.html file that you created in your root directory (see Listing 1), change the name of the JavaScript file that's included from `vue.js` to `bundle.js` :

```
<script type="text/javascript" src="bundle.js"></script>
```

In your browser, open http://localhost:8080/index.html. You can see that the app is loading and functioning as before, with the exception that the button and input are now styled — because Foundation has loaded successfully and its CSS has been injected into the page. Things already look much nicer. The page is also responsive; resize the browser window, and you see that the input field resizes accordingly.

Now it's time to eliminate that global `Vue` variable and move your JavaScript into main.js. Cut the code from the `<script>` tag in index.html and paste it into main.js. Also, move the `bundle.js` include to the end of index.html; it needs to be invoked after the markup is resolved, so that the code can reference the DOM after the browsers have rendered and the DOM is active. Now your index.html file looks like Listing 6.

Listing 6. Updated index.htm

```
<html>
    <head>
        <meta charset="utf-8">
    </head>
    <body>
  <div id="test">
  <p>User: {{ username }}</p>
  <input v-model="username">
  <button v-on:click="defaultUser">Default User</button>
  </div>
  <script src="http://localhost:8080/webpack-dev-server.js"></script>
  <script type="text/javascript" src="bundle.js"></script>
    </body>
</html>
```

And your main.js file now looks like Listing 7.

Listing 7. Updated main.js

```
$ = jQuery = require('jquery');
foundation = require('foundation-sites');

var Vue = require("vue");

require('../css/app.scss');

test = new Vue({
  el: '#test',
  data: {
  username: "Luke Skywalker"
  },
  methods: {
  changeName: function(name){
    this.username=name;
  },
  defaultUser: function(){
    this.changeName("Luke Skywalker");
  }
  }
});
```

Reload the page in the browser, and it works as before. Notice one more thing in Listing 7: You scope the `Vue` reference with `var`, so now you aren't leaking that variable into the global namespace.

Vue components

> *“ You'll see how to compose the app functionality into components. Components are an excellent way to promote reuse, deduplication, and modularity in application design. ”*

Your dependency management and build system are in place, you're pulling in the packages that you need, the packages are working, and the script code is broken out into an external JavaScript file. Now, you see how to compose the app functionality into components. Components are an excellent way to promote reuse, deduplication, and modularity in application design.

The Vue application has one master parent `Vue` instance that contains all the others and renders the children. (It's possible to have multiple root `Vue` instances, but you don't need that here.) You start with evolving the login feature to a component called `user`.

Here's the new body content for index.html for the `user` component:

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

The `<div>` called `app` is still a `Vue` instance, but you're making the custom `<user>` tag a Vue component. Now visit main.js. Leave the imports as is, and erase the remaining code. You add a component definition for the `user` component, and then the declaration for the `App Vue`. Listing 8 shows the component declaration.

Listing 8. Programmatically defining a `user` component

```
Vue.component('user', {
  template: '<div>User: {{ username }}</div>',
  data: function(){
    return {
      username: "Luke Skywalker"
    }
  }
});
```

Listing 8 makes use of the `Vue.component` method, which you use to register a component under a name (the first argument), followed by the component parameter object (the second argument).

Important

In a Vue component, use a function to return the data field, to avoid cross-instance data sharing.

Notice that the HTML markup has moved into the `template` attribute. (If you're familiar with Dojo, note the similarity to how Dojo's widgeting system's template attribute works.) Next is a `data` attribute — a function that **returns** the data object, instead of **being** the straight data object. The reason for this approach is that you're creating a component that can be reused in multiple places, like a type. You don't want all the type instances sharing the `data` field, which is what would happen if it were a straight object.

Next, add these lines — the instantiation of the `App` instance — to main.js:

```
var App = new Vue({
  el: "#app",
  data: {},
  methods: {}
});
```

The `App Vue` is simple — all it does is take the element.

If you reload the page, you see that things are working. The `App` instance has successfully injected the `user` component into its markup, and **User: Luke Skywalker** is rendered.

Now, you take componentizing one step further and move the `User` component into its own .vue file. From the root of the project, create an /app/vue folder, and add a user.vue file. Add the code in Listing 9 to the file.

Listing 9. User component defined in its own file, user.vue

```
<style>
</style>

<template>
  <p>User: {{ username }}</p>
</template>

<script>
module.exports = {
  data:function () {
  return {
    username: "Luke Skywalker"
  }
  }
}
</script>
```

The three tags in Listing 9—`<style>`, `<template>`, and `<script>`— represent the three elements of any component: CSS, markup, and code. (Tip for Sublime users: Click **View > syntax > HTML** to tell Sublime to show correct highlighting.)

So far, you're not making any use of custom CSS for the component, so the `<style>` tag is empty. The `<template>` element contains your familiar markup, and the `<script>` tag contains your familiar `data` attribute but wraps it differently. Because you're in a file that's being bundled as a CommonJS component, you use `module.exports` and define your component as you did in Listing 8). Notice that you don't declare a component name here.

Returning to main.js, how can you use this component? You must do two things: include the component, and reference it in your `App Vue`, as shown in Listing 10.

Listing 10. Using modular `User` component in the `App Vue`

```
var User = require("../vue/user.vue");

var App = new Vue({
  el: "#app",
  data: {},
  methods: {},
  components: {
  user: User
  }
});
```

In Listing 10, you require the user.vue file and give it a local variable reference. Next, in the `App` instance declaration, you add a `components` attribute. The `components` attribute does the job of mapping your `User` Vue type to the `<user>` tag. The component's keys are the tag names usable within the Vue markup, and the component values are the Vue components that these tags refer to.

Now if you reload, you again see **User: Luke Skywalker**, verifying that the app is still working, decomposed as it is into loadable components. All of the pieces are in place. You're ready to take the next step and start integrating those pieces into a real-world app.

First, add in a menu bar, courtesy of Foundation, and put your `user` component inside it. Listing 11 shows index.html updated with the menu-bar code.

Listing 11. Index.html with user name top bar

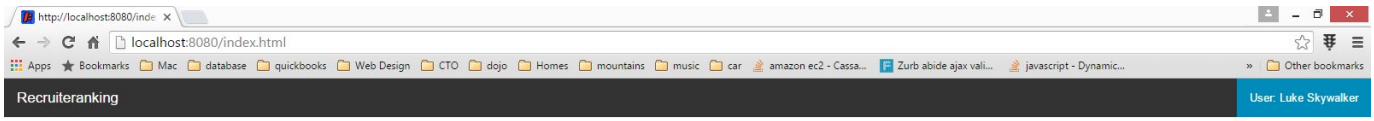
```
<html>

<head>
    <meta charset="utf-8">
</head>

<body>
    <div id="app">
        <div class="sticky">
            <nav class="top-bar" data-topbar role="navigation" id="top-bar">
                <ul class="title-area">
                    <li class="name">
                        <h1>
            <a href="#">Recruiteranking</a>
          </h1>
                    </li>
                </ul>
                <section class="top-bar-section">
                        <ul class="right">
                            <li class="active">
                              <user></user>
                            </li>
                        </ul>
                    </section>
            </nav>
        </div>
    </div>
    <script src="http://localhost:8080/webpack-dev-server.js"></script>
    <script type="text/javascript" src="bundle.js"></script>
</body>

</html>
```

Now, when reloaded, the app displays **Recruiteranking** in the left side of the top bar and **User: Luke Skywalker** on the right side:



Component hierarchy

Next up, you want to build a better data flow system. You begin by taking the `username` field out of the control of the `user` component and giving it to the `App`. Here, you begin to see the power of the Vue component hierarchy in action.

Go to user.vue and make the change shown in Listing 12 — the addition of the `props` member.

Listing 12. Adding `props` member to user

```
<script>
module.exports = {
  props: ["username"],
  data:function () {
    return {
      username: "Luke Skywalker"
    }
  }
}
</script>
```

The `props` member tells the component which properties are accessible to its parent. So here, you're saying that `username` is accessible to the parent `Vue` instance.

Next, add a `data` field to your `App Vue` instance. This `data` field can be a straight object, not a function, because you won't reuse this instance. In the `data` field, put a `User` object that contains a `username` field. You're putting in a `User` object because down the line, you'll probably want to add other information about the logged-in user, such as ID. The new `data` field for `App` is:

```
data: {
    user: { username: "Luke Skywalker" }
  }
```

Note two things here. First, the parent `Vue` instance can interact with the child component. Second, Vue can deal seamlessly with the `data` field on the `App` instance being a complex object.

Now, to tie these pieces together, return to the index.html file and use the `v-bind` directive to tell the `user` component where to find its `username` data. Here's the markup for binding parent data to the `User` component:

```
<user v-bind:username="user.username"></user>
```

Without complaint, Vue navigates the dot operator on `user` to get to the `username` property.

`v-bind` shorthand syntax

You can use shorthand syntax for `v-bind`, similar to the shorthand for event handling. Vue interprets a colon as a `v-bind` directive. So the markup for binding parent data to the `User` component could be rewritten as `<user :username="user.username"></user>`.

Simply and elegantly, the `user` component's `username` field is now bound to the parent `user.username` field. By default, the data binding flows only *one way*, from parent down to child. Vue supports two-way communication, and even has an intracomponent eventing system. For now, you need only the basic parent-to-child communication handsaw, but it's good to know that you have the chainsaw if you need it.

Logging in

What do you need to log a user in? With everything that you have in place so far, you might be surprised at how quickly this step goes.

Begin by removing your default user, Luke Skywalker, by setting the `username` field in the user.vue file to `null`:

```
module.exports = {
  user: null
};
```

The `user` component now cares only about the display of the data, not the data itself.

At this point, if you reload and look at the console, you see an error: `[Vue warn]: Error when evaluating expression user.username. Turn on debug mode to see stack trace.` The error occurs because in the HTML in index.html, you reference `user.username`, and your `user` object is `null`. You want to show the user name only if the person is logged in. You can do that conditional rendering with the `v-if` directive:

```
<li class="active" v-if="user">
  <user v-bind:username="user.username"></user>
</li>
```

The `v-if` directive says, "Render this element only if the `v-if` condition is true." In this case, you render this list item only if the user data object is not `null`— exactly what you want. Now the user name only displays when a user is logged in.

To display a Log In button if the user is `null`, add an `else` block to your top bar:

```
<li class="active" v-if="user">
  <user v-bind:username="user.username"></user>
</li>
<li class="active" v-else>
  <a href="#" v-on:click="showLogin">Log In</a>
</li>
```

Notice that the login action has a `v-on:click` handler on it, pointing to `showLogin()`. Add that handler to your `App Vue`, as shown in Listing 13.

Listing 13. The `showLogin()` method on the `App Vue`

```
App = new Vue({
  el: "#app",
data: {
    user: null,
    loggingIn: false
  },
methods: {
    showLogin: function(){
      this.loggingIn=true;
    }
  }
//...
```

All Listing 13 does in the `logIn()` method is set the `loggingIn` flag in `App.data` to `true`. You can use that flag to reveal a login pane. (Vue also supports animated transitions.) Notice that you don't do any DOM manipulation directly; you set the flag and let the Vue directive in markup do the work.

In index.html, add a login form, as in Listing 14, immediately after the top-bar markup.

Listing 14. Adding a login form to index.html

```
<div id="app">
  <div class="sticky">
    <nav class="top-bar" data-topbar role="navigation" id="top-bar">
      ...
    </nav>
  </div>
  <div class="small-12 columns" id="sign-in-on" v-show="loggingIn">
    <div class="row">
        <div class="large-12 columns">
            <div class="signup-panel">
                <form id="signup-form" data-abide="ajax">
                    <div class="row collapse">
                        <div class="small-10  columns">
                            <input type="text" placeholder="Username" name="username" required>
                        </div>
                    </div>
                    <div class="row collapse">
                        <div class="small-10 columns ">
                            <input placeholder="Password" name="password" required type="password">
                        </div>
```

```
                    </div>
                    <button type="submit" v-on:click="login">Log In</button>
                </form>
            </div>
        </div>
    </div>
  </div>
</div>
```

The login page is displayed when `main.loggingIn` is set to `true`. Notice that you achieve this result with a `v-show` directive. The `v-show` differs from `v-if` in that `v-show` renders the content but hides it, whereas `v-if` does not render the content at all. In this case, you opt for hide-and-reveal instead of render-on-demand. Your Log In button is also attached to a method, `login`, again with `v-on:click`. That method performs the login action for you.

Integrating with a back end: Development

You've come to the point where you need to talk to a back end. For the purposes of this tutorial, I'm providing a back end that you can run locally (see Downloadable resources). If you're developing both the front end and back end (in other words, you are a "full-stack" developer) on your workstation, you can run your development back end on port 4567, and proxy requests to it as I am doing here with the demo app. Run the downloaded back-end server by executing `java -jar rr-backend-1.0.jar`. The server spins up and starts handling requests via embedded Jetty (via the Spark Framework).

You need a way to tell the client dev server to forward API requests to the back end. webpack provides that capability in the dev server proxy. Add the code in Listing 15 to the root-level object of your webpack.config.js file.

Listing 15. Adding dev server proxy to webpack.config.js

```
module.exports = {
   ...
    devServer: {
        proxy: {
            '/api/*': {
                target: 'http://localhost:4567',
                secure: false
            }
        }
    }
}
```

Now, with the back-end application running, any requests to the `/api/*` path go to the server, listening on port 4567.

With the back-end server running, go to http://localhost:8080/api/test in your browser to verify that the server is pushing its response back through the dev server. If the response is `TEST OK`, all is well.

To continue with your modular design, you evolve your login form into a component. Listing 16 shows this component.

Listing 16. The `login` component

```
<style>
</style>
<template>
  <form action="{{ action }}" method="{{ method }}" v-on:submit.stop.prevent="login">
    <div class="row collapse">
        <div class="small-10  columns">
      <input type="text" placeholder="Username" name="username" v-model="payload.username" required>
        </div>
    </div>
    <div class="row collapse">
        <div class="small-10 columns ">
      <input placeholder="Password" name="password" v-model="payload.password" required type="password">
        </div>
    </div>
    <button type="submit">Log In</button>
  </form>
</template>
<script>
module.exports = {
  props: {
    'action': {
        type: String,
        required: true
    },
    'method': {
        type: String,
        default: "post"
    }
  },
  data: function() {
    return {
    payload: {
      username: null,
      password: null
    }
    }
  },
  methods: {
    login: function() {
        this.$http.post(this.action, this.payload, function (data, status, request) {

        this.$dispatch('onLoginSuccess', data);
        }).error(function (data, status, request) {
      console.error(status);
        });
    }
  }
}
</script>
```

The `login` component should be mostly familiar to you, with a couple of notable new items. Notice that you have a more elaborate `props` field, and you use it to bring in the `method` and `action` from the form. The `props` field shows of some of Vue's support for `required`, `type`, and `default` parameters. The `props` field even includes a validator, in case you need one.

You listen on the template form's submit, with `v-on:submit.stop.prevent="login"`, and here you see a couple of modifiers on the `v-on` directive: `stop` and `prevent`. These modifiers perform the service of preventing the default browser action and stopping the event propagation. The modifiers can be used together as here, or individually, depending on your need.

When the `login` method is called, a `post` is sent with the `payload` field from the `data` object, which is bound with the `v-bind` to the input fields. The `$http` object is from the `vue-resource` plugin, which you install in a moment. But first, notice that you execute `#dispatch` with a string and your return data from the `post`. Using `$dispatch` — part of Vue's component eventing system — child components can send an event up the component tree to the parents. You handle this event in your `App Vue`.

Go to main.js and add the `events` field:

```
events: {
      onLoginSuccess: function(data) {
        this.user = data;
        this.loggingIn = false;
      }
    }
```

`events.onLoginSuccess` catches the event. The back-end test service is a stub that simply returns the user object with a UUID added. You set the `user` object on `App.user`, and you also set `loggingIn` to `false`, which hides the login form.

Before these changes will work, you need to include the `vue-resource` plugin. You could use straight XHR or jQuery, or whatever Ajax support you want, but the `vue-resource` plugin makes things clean and simple, as you saw with the `this.$http` call in the `login` method in Listing 16. To add `vue-resource` support, go to package.json and add `"vue-resource": "0.5.1"` to the dependencies. Next, go into main.js and add this line to the imports:

```
Vue.use(require('vue-resource'));
```

Notice that you call `Vue.use()` to add the module as a plugin into the Vue runtime.

You have a couple of options for adding validation to your form — both Foundation and Vue include validator support — but you skip validation and move on to the recruiter-related functionality. You want a grid that users can click for details, and you want logged-in users to be able to edit, delete, and create recruiter entries. The login requirement is the authorization part of the app. (You could easily extrapolate to make user roles or access-control lists the basis for authorization, but for brevity's sake, you use only the login check here.)

You map your CRUD operations to the RESTful semantic like this: `PUT` for create, `GET` for read, `POST` for update, and `DELETE` for delete. So your server's API for `recruiter` looks like this:

```
GET /api/recruiter
POST /api/recruiter
PUT /api/recruiter
DELETE /api/recruiter
```

The only slightly controversial aspect here is the meaning of the `POST` and `PUT` verbs, but this is a slight issue that won't keep anyone up at night.

Warning

The test server is all in-memory and does no validation or authorization checks, so if you restart the server, you lose your changes.

Start with your grid component, which displays the list of recruiters, using the grid example from the Vue docs as a starting point. This component brings together much of what you've learned here and also introduces a few new things.

Create the `grid` component by putting code in Listing 17 into an /app/js/vue/grid.vue file.

Listing 17. The `grid` component: /app/js/vue/grid.vue

```
<style>
/* CSS truncated for brevity */
</style>
<template>
    <table>
        <thead>
            <tr>
                <th v-for="key in columns" @click="sortBy(key)" :class="{active: sortKey == key}">
                    {{key | capitalize}}
                    <span class="arrow" :class="sortOrders[key] > 0 ? 'asc' : 'dsc'">
          </span>
                </th>
            </tr>
        </thead>
        <tbody>
            <tr v-for="entry in data | orderBy sortKey sortOrders[sortKey]">
                <td v-for="key in columns" v-on:click="rowClick(entry)">
                    {{entry[key]}}
                </td>
            </tr>
        </tbody>
    </table>
</template>
<script>
module.exports = {
    props: {
        data: Array,
        columns: Array
    },
    data: function() {
        var sortOrders = {}
        this.columns.forEach(function(key) {
            sortOrders[key] = 1
        })
        return {
            sortKey: '',
            sortOrders: sortOrders
        }
    },
    methods: {
        sortBy: function(key) {
            this.sortKey = key
            this.sortOrders[key] = this.sortOrders[key] * -1
        },
        rowClick: function(row){
          this.$dispatch('onRecruiterDetail', row);
        }
    }
}
</script>
```

Notice that this component includes styling that I'm omitting from the listing for the sake of brevity. You can see the styling — simple CSS to give the grid a look and feel — in the downloadable app.

Notice in the `<template>` in Listing 17 that you have a basic table layout. The header tag, `<th>`, contains some new items:

- The `v-for` directive, which is Vue's iterator support; with it, you can emit the specified markup based on an array. In this case, you do a `v-for="key in columns"`, which is analogous to a normal JavaScript `for` loop, and means, "For each item in the `columns` collection, render the markup, exposing the `key` variable as a reference to the current element."
- A click handler on the header, which calls a `sortBy` method, passing in the current key with `@click="sortBy(key)"`.
- A *class* directive: `:class="{active: sortKey == key}"`. Remember that the colon alone is a shorthand for the `v-bind` syntax, so you're looking at a binding to the `Vue` instance's `data` member — but in this case, for the `class` attribute. Vue offers some special handling for the `class` and `style` attributes. Each of those attributes takes a comma-separated list of names, pointing to an expression that resolves to a Boolean. In this case, if it's true that `sortKey == key`, this `<th>` element receives the `active` class: `<th class="active">`. This expression implies that the Vue component instance has a `sortKey` data field that is being compared to the `key` variable.
- A basic token as the body of the `<th>` that includes a feature that's new to you — a *filter*: `{{key | capitalize}}`. The filter is called `capitalize`, and it is added to the base value of `key`, by way of the pipe character. `capitalize` is a built-in filter that does exactly what it sounds like. (With Vue, you can also define your own custom filters, but you won't do that here.)

The next item of interest in Listing 17 is the `v-for` directive on the table body's `<td>` element: `v-for="entry in data | orderBy sortKey sortOrders[sortKey]"`. This loop executes over the `data` property and modifies the result with the `orderBy` function, passing in two arguments: `sortKey` and `sortOrders[sortKey]`. `orderBy` uses the arguments to determine the ordering algorithm. The first argument defines which field on the sorted object to use, and the second tells the algorithm to sort ascending (`true`) or descending (`false`).

The JavaScript part of the component is clearly concerned with supporting various parts of the markup. The `props` field provides the `data` (the array that the table body iterates over) and the `columns` (the array that the header iterates over). The component parent sets the data and columns arrays; the parent is concerned with supplying the application data to the child for rendering.

The `data` field demonstrates that you can execute arbitrary JavaScript before returning the `data` object. In Listing 17, the `data` code prepares the `sortKey` and `sortOrders` fields. `sortOrders` is set to `1` for every key in the columns array, which resolves to `true` (ascending) in the `orderBy` operation on the table body.

The component has two methods: `sortBy` and `rowClick`. The `sortBy` method sets the active sort key and then reverses the `sortOrders` value for that key by multiplying by -1:

`this.sortOrders[key] = this.sortOrders[key] * -1`. Finally, the `rowClick` method takes the item passed in (the object item currently being iterated over) and raises a `onRowClicked` event to be handled by a parent `Vue` (it bubbles up the parent chain and continues hitting any `onRowClicked` handlers until `false` is returned or the chain ends).

All you need to do to get the grid going is to include it in main.js and provide the `columns` and `data` properties. In main.js, include the component by adding `var grid = require("../vue/grid.vue");`. Remember to register the grid component in the `components` property:

```
components: {
        user: user,
        login: login,
        grid: grid
    }
```

Also, add in the `gridData` and `columns` fields to the `data` property, and notice that you set the grid's column titles here:

```
gridColumns: ['name', 'rank'],
      gridData: []
```

Now you need to populate the `gridData` via the `ready` method. You haven't seen this method until now. `ready` is a lifecycle callback that executes after the `Vue` instance has rendered and is completely ready. Listing 18 shows the `App.ready` method.

Listing 18. Using the `ready()` callback to load data in the `App Vue`

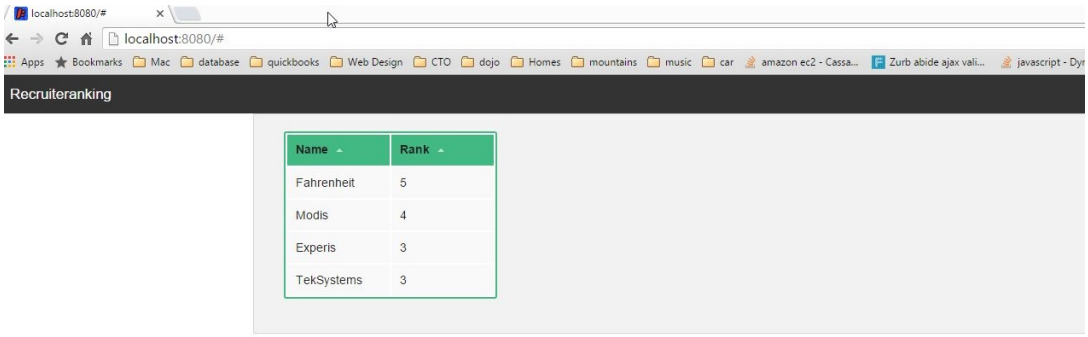
```
ready: function() {
        this.$http.get("/api/recruiter").then(function(response) {
            this.gridData = response.data;
        }, function(response) {
            console.error(response.status);
        });
    }
```

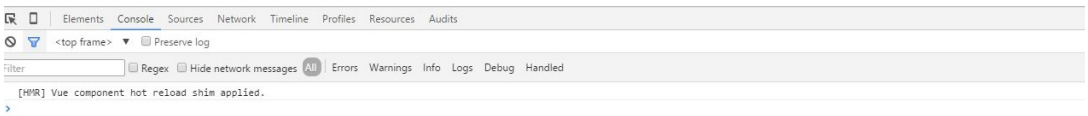
In the `ready` method, you again make use of the `vue-resource $http()` method, as you did for `login`. This time, you send a `GET` request to the `/api/recruiter` URL (remembering that your proxy forwards this request to the server based on the `/api/` route). You use `.then` to provide a handler function. The response argument contains everything you need to handle the response. In this case, you set the `data` field as your `gridData`. When the `main Vue` is ready, your grid is populated from the server.

The last step is to introduce the `grid` component into the layout, which is easy. In index.html, add the following markup to the HTML body, directly below the code for the top bar:

```
<div class="row panel" id="main">
              <div class="large-6 columns">
                  <grid :data="gridData" :columns="gridColumns"></grid>
                </div>
```

In the preceding code, you bind your required fields to the data that you provided: `data` to `gridData` and `columns` to `gridColumns`. Now, if you reload the page, you see the recruiter grid, seeded with a few starter rows:





Now you want to be able to click a row and see the details. The full sample application (see Download) includes a `recruiter-detail` component in the /js/app/vue/recruiter-detail.vue file. That component doesn't contain anything new and different, so I won't go into it in depth. Add `recruiter-detail` to main.js as a component, and then include it in the markup, as shown in Listing 19.

Listing 19. Adding the `recruiter-detail` component to the index.html markup

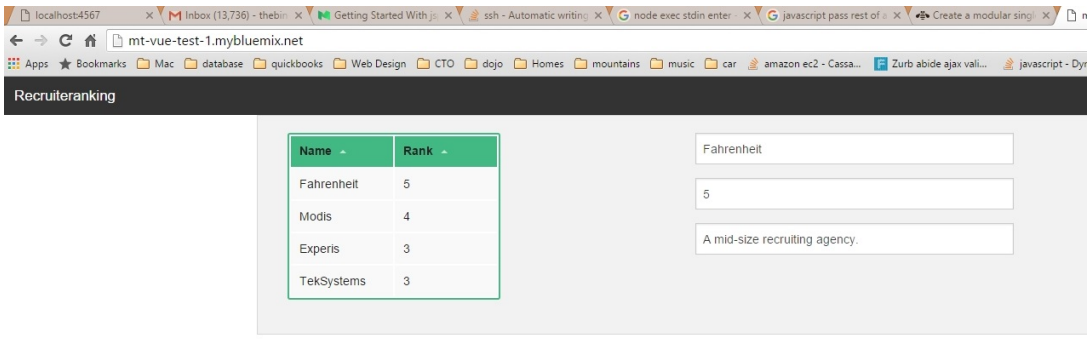
```
<div class="row panel" id="main">
              <div class="large-6 columns">
                  <grid :data="gridData" :columns="gridColumns"></grid>
                </div>
                <div class="large-6 columns" v-show="detail">
                  <detail :payload="detail" :editable="user" action="/api/recruiter">
                  </detail>
                </div>
        </div>
```

The noteworthy piece here is the `:payload` binding, which resolves to `detail` on the `main Vue`. Recall that the `grid` component issues an event when a row is clicked, and that event is handled by the `onRecuiterDetail` event handler on `main`:

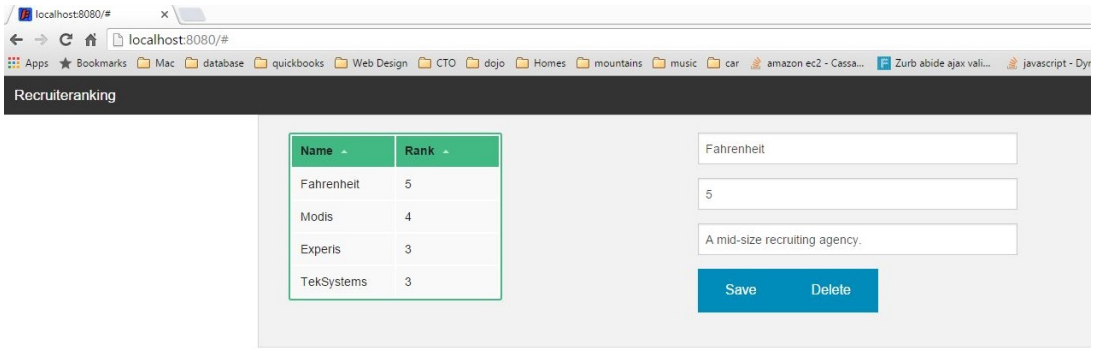
```
onRecruiterDetail: function(recruiter) {
            this.detail = recruiter;
        }
```

The `onRecuiterDetail` method sets the content of the form in `recruiter-detail`, and it also shows the view (by way of the `v-show` directive on the `<div>`).

Now if you click a row, you get the recruiter detail:



Notice that when you include the detail component in the HTML, you bind the `editable` field to the `user` object on the `main Vue`. As a result, the editing functionality is available only if the user is logged in. So, if you log in, you see buttons on the detail. Notice that everything works the same here whether you log in and then open a detail, or if the detail is already visible and then you log in:



The Save and Delete buttons both emit events that are handled by the `main Vue`— none of the business logic happens in the detail component. The events are `onRecruiterSave` and `onRecruiterDelete`. They both use the `vue-resource` support to issue the appropriate REST calls to the API, using the provided `recruiter` instance to populate the data. Because the `recruiter` instance in the detail is pulled from the grid array, the grid changes immediately to reflect any updates that the user makes to the values in the detail form.

The last piece that you need is a button on the toolbar to create a recruiter:

```
<li class="active" v-if="user">
                            <a href="#" v-on:click="createRecruiter">New Recruiter</a>
                        </li>
```

Here again, you enable this feature only if the user is logged in. The `createRecruiter` function is simple:

```
createRecruiter: function() {
            this.detail = {};
        }
```

By setting a blank object on the `detail` field, `createRecruiter` reveals the recruiter detail pane. If the `id` field is present, the form is handled as an update, and otherwise as a create.

You now have complete CRUD support, including a sortable grid and basic authentication and authorization.

Conclusion to Part 1

You can use core features of Vue.js to build anything you can imagine while keeping keep your project clean and manageable. Vue's extensive capabilities come with minimal overhead. These qualities account for rapid gains in Vue's popularity, and more community-created functionality is being added all the time. And you can take advantage of tools that work well with Vue to maximize your development experience. In Part 2 in this series, see a couple of ways to deploy your recruiter-ranking application in the cloud with IBM Bluemix.

Downloadable resources

Description	Name	Size
Recruiteranking front-end code	recruiteranking.zip	75KB
Stand-alone back-end WAR	rr-backend-war.zip	321KB

Related topics

- Vue.js: Visit the Vue.js project website for documentation, examples, and community discussion.
- webpack: Learn more about the webpack module bundler.
- Foundation: Dig into the Foundation responsive-UI framework.
- NPM: Explore the NPM package manager.
- Vue.js: Download Vue.
- Node.js and NPM: Find out how to install Node and make sure that you have the latest NPM version.