Table of Contents

Vue.js Introduction

Vue.js is a progressive framework for building user interfaces. Unlike other monolithic frameworks, Vue is designed from the ground up to be incrementally adoptable.

This book intends to help developers write Vue.js applications easy and fast. The book explains best practices and general development flows.

Example application is available on Github.

Project Initialization

Getting Started

We take advantage of the vue-cli to initialize the project.

```
$ npm install -g vue-cli
$ vue init xpepermint/vue-cli-template {my-project}
$ cd {my-project}
$ npm install
```

This will initialize a new Vue.js application with support for server-side rendering and hot module reload in development.

You can now start the HTTP server.

```
$ npm run start
```

Development

It is highly recommended to use Nodemon in development. It will restart you application when the server-side code changes (client-side changes are already handled by the application itself).

Install the required package.

```
$ npm install -g nodemon
```

Run the command below to start the server.

```
$ nodemon --exec npm start
```

Production

When you are ready to push your code in production, compile the project into the `./dist` directory and set at least the `env` environment variable (check the `./config/index.js`).

```
$ npm run build
$ npm config set {my-project}:env production
```

Serving static files through Node.js in production could slow down your application. It is recommended that you move the entire `./public` directory to CDN or you serve these files through Nginx or similar HTTP server. The `./dist/client` bundles are also served as public files thus don't forget to move them as well. In this case you'll need to configure the `publicPath` configuration variable.

```
$ npm config set {my-project}:publicPath https://{cdn-path}
```

Run the command below to start the server.

```
$ npm start
```

Using TypeScript

Although Vue.js officially provides TypeScript definition files, the framework it self is not written using this technology and thus not 100% ready to use it for you next big project.

If you search for a best-practice advice, then use TypeScript when writing Vue.js plugins (weh nusing only *.ts* files) and JavaScript for components and your actual projects (when using also *.vue* files). Currently, by using TypeScript in your Vue.js projects you accept a risk of having serious problems and lack of support.

If you wonder how to structure your TypeScript-based Vue.js plugin then take a look at the RawModel Vue.js Plugin.

Linting

Code style is important and eslint can significantly improve the way we write our code.

Installation

Start by installing the required packages.

```
$ npm install --save-dev eslint eslint-config-vue eslint-plugin-
vue
```

Create a new file `./.eslintrc` and add the configuration below.

```
{
  extends: ['vue'],
  plugins: ['vue'],
  rules: {}
}
```

Open `./package.json` and define a new command.

```
{
  "scripts": {
    "lint": "eslint ./src --ext .js --ext .vue"
  }
}
```

You can now run the `npm run lint` command to verify your code.

Configuring Text Editor

Atom

If you are using Atom then install the linter-eslint plugin. This will automatically install also the linter plugin.

After installation under `Preferences` open the `Packages` tab, find the `linter-eslint` plugin and click on the `Settings` button. Search for `Lint HTML Files` option and make sure it's marked as checked. This will enable linting in the `.vue` files.

Navigation Using Router

First install the official vue-router package.

```
$ npm install --save vue-router
```

Create a new file `./src/app/router/index.js` and define a router as shown below.

```
import Vue from 'vue'
import Router from 'vue-router'

Vue.use(Router)

export default new Router({
  mode: 'history',
  routes: [
    {path: '/', component: require('../components/home.vue')},
    {path: '/foo', component: require('../components/foo.vue')},
    {path: '/bar', component: require('../components/bar.vue')},
    {path: '*', redirect: '/'} // 404
  ],
  scrollBehavior (to, from, savedPosition) {
    return !savedPosition ? { x: 0, y: 0 } : savedPosition
  }
});
```

Create components which we've defined in the router file above.

```
<template>
  <div>./src/app/components/home.vue</div>
</template>
```

```
<template>
  <div>./src/app/components/foo.vue</div>
</template>
```

```
<template>
  <div>./src/app/components/bar.vue</div>
</template>
```

Open the `./src/app/components/app.vue` file and replace the existing template with the one below.

```
<template>
  <div id="app">
    <ul>
      <li><router-link to="/">Home</router-link></li>
      <li><router-link to="/foo">Foo</router-link></li>
      <li><router-link to="/bar">Bar</router-link></li>
    </ul>
    <router-view class="view"></router-view>
  </div>
</template>
```

Open the `./src/app/index.js` file and install the router.

```
...
import router from './router';

export const app = new Vue({
  router,
  ...App
});
```

Open the `./src/app/server-entry.js` file and set router root path.

```
export default (ctx) => {
  app.$router.push(ctx.url);
  ...
};
```

Open the `./src/server/router/routes/app.js` file and set application context as show below.

```
exports.render = (req, res) => {
  let ctx = {url: req.originalUrl};
  let page = req.vue.renderToStream(ctx);
  ...
}
```

Needs update to 2.1: https://github.com/vuejs/vuex/releases

Application State

Vuex is used for managing application state and it's very similar to Flux. Vuex uses a concept of application state, getters, mutations and actions. The application `state` contains variables where application data is stored, `getters` are computed state variables, `mutations` are used for interacting with state variables and actions are used for loading data (e.g. from a server). **All these items are registered under the global namespace.**

Getting Started

Let's start by installing the dependencies.

```
$ npm install --save vuex
```

Create a new file `./src/app/store/index.js` and define the store.

```
import Vue from 'vue';
import Vuex from 'vuex';

Vue.use(Vuex);

import * as actions from './actions';
import * as getters from './getters';
import * as mutations from './mutations';
import * as state from './state';

export default new Vuex.Store({
  actions,
  getters,
  mutations,
  state
});
```

Open the `./src/app/index.js` file and attach the store.

```
...
import store from './store';

new Vue({
  ...
  store
});
```

Create a new file `./src/app/store/state.js` and define state variables.

```
export var isFetching = false;
export var users = [];
```

Create a new file `./src/app/store/actions.js` and add store actions.

```
export function fetchUsers ({commit}) {
  commit('startFetching');
  return new Promise((resolve) => {
    setTimeout(() => {
      commit('stopFetching');
      commit('addUser', {name: 'John Smith', enabled: true});
      commit('addUser', {name: 'Dona Black', enabled: false});
      resolve();
    }, 2000);
  });
}
```

Create a new file `./src/app/store/getters.js` and define store computed variables.

```
export function enabledUsers (state) {
  return state.users.filter(u => u.enabled);
}
```

Create a new file `./src/app/store/mutations.js` and define store mutations.

```
export function startFetching (state) {
  state.isFetching = true;
}

export function stopFetching (state) {
  state.isFetching = false;
}

export function addUser (state, data) {
  state.users.push(data);
}
```

We now have all the ingredients for retrieving and displaying a list of users. Add the code into one of your `.vue` file.

```
<template>
  <div>
    <div v-for="user in users">
      {{ user.name }}
    </div>
    <button v-on:click="loadMore">Load</button>
  </div>
</template>

<script>
export default {
  computed: {
    users () {
      return this.$store.state.users;
    }
  },
  methods: {
    loadMore () {
      this.$store.dispatch('fetchUsers');
    }
  }
}
</script>
```

State Hydration

Open the `./src/app/client-entry.js` file and add the line below.

```
app.$store.replaceState(window.STATE);
```

Open the `./src/app/server-entry.js` file and add a pre-fetch logic which will pre-populate the components with data.

```
...
export default (ctx) => {
  ...
  return Promise.all( // initialize components
    app.$router.getMatchedComponents().map(c => c.prefetch ? c.p
refetch.call(app) : null)
  ).then(() => {
    ctx.state = app.$store.state; // set initial state
  }).then(() => {
    return app;
  });
};
```

Open the `./src/server/router/routes/app.js` file and add the initial state source to the output.

```
page.on('end', () => {
  res.write(  `<script>window.STATE = JSON.parse('${JSON.stringi
fy(ctx.state)}')</script>`);
  ...
});
```

Open the component file and define the `prefetch` method which loads data on the server and the `beforeMount` method which loads data on the client.

```
<script>
export default {
  ...
  prefetch () {
    return this.$store.dispatch('fetchUsers');
  },
  beforeMount () {
    if (!this.$store.state.users.length) {
      return this.$store.dispatch('fetchUsers');
    }
  }
}
</script>
```

Forms

In this chapter we create a form which inserts a new user into the Vuex store. This chapter assumes that you've completed the previous chapter on how to manage application state.

Getting Started

Create a new component file or open an existing one (e.g. `./src/app/components/app.vue`), then set the code as shown below.

```
<template>
  <div>
    <!-- Form -->
    <form novalidate v-on:submit.prevent="adduser">
      <input type="text" v-model="user.name" placeholder="User n
ame"/>
      <input type="checkbox"v-model="user.enabled"/> Enabled
      <button>+ Add User</button>
    </form>
    <!-- List -->
    <pre v-for="user in users">{{user}}</pre>
  </div>
</template>

<script>
























</script>
```

Dynamic Inputs

Here we add a `tags` field to the user, to show how easy we can create dynamic form fields.

Let's make some adjustments in the code example above.

```
<template>
  <div>
    <!-- Form -->
    <form ...>
      ...
      <template v-for="tag in user.tags">
        <input type="text" v-model="tag.name" placeholder="User
tag"/>
      </template>
      <button v-on:click.prevent="newTag">+ New Tag</button>
      ...
    </form>
  </div>
</template>

<script>
export default {
  data () {
    return {
      user: {
        ...
        tags: [{name: ''}]
      }
    };
  },
  methods: {
    ...
    newTag () {
      this.user.tags.push({name: ''});
    }
  }
};
</script>
```

Validation

The simplest and the most convenient way for validating a form is to use the RawModel.js framework. Use vue-rawmodel plugin to implement this into your Vue application.

Internationalization (i18n) & Localization (l10n)

It seams that Strawman's proposal is dictating the future of internationalization and localization in Javascript. It's using ECMAScript Internationalization API which is supported by all modern browsers and `Node.js`.

This chapter explains how to add `i18n` and `l10n` support into your Vue.js application using vue-translated.

Getting Started

Install the dependencies.

```
$ npm install --save vue-translated translated
```

Create a new file `./src/app/i18n/index.js` and configure the plugin.

```
import Vue from 'vue';
import {I18n} from 'translated';
import VueTranslated from 'vue-translated';

Vue.use(VueTranslated);

const locale = 'en-US'; // replace this with `process.env.LOCALE
` or similar
const messages = require(`./messages/${locale}`).default;
const formats = require(`./formats/${locale}`).default;

export default new I18n({locale, messages, formats});
```

Create a new file `./src/app/i18n/messages/en-US.js` and define translated messages using the ICU Message Format.

```
export default {
  'home.text': 'Hello {name}!'
}
```

Create a new file `./src/app/i18n/formats/en-US.js` and define custom formats.

```
export default {
  number: {},
  date: {},
  time: {}
}
```

Open the application main file `./src/app/index.js` and attach the `i18n` instance.

```
import i18n from './i18n';

export const app = new Vue(
  Vue.util.extend({
    i18n,
    ...
  }, App)
);
```

You can now use the nee API.

```
<template>
  <div>
    {{ $formatMessage('home.text' {name: 'John'}) }}
  </div>
</template>
```

Intl Support

Browsers

If you need to support old browsers with no support for `Intl` then you need to include a polyfill. The easiest way is to use FT Polyfill Service. If you've follow this book and you are using server-side rendering then open the `./src/server/middlewares/app.js` and load the `polyfill.io` service.

```
exports.appServer = function () {
  return (req, res) => {
    ...
    page.on('end', () => {
      res.write(`<script src="https://cdn.polyfill.io/v2/polyfil
l.min.js?features=Intl.~locale.en"></script>`);
      ...
    });
    ...
  };
};
```

Node.js

Node.js already supports `Intl` out of the box but it comes with English support only. To support other languages we need to download the latest ICU data file and then run our scripts with the `--icu-data-dir` option.

To enable other languages in your Vue.js application on the server-side, first download the ICU data.

```
$ npm install --save icu4c-data@0.58.2
```

Open the `./package.json` file and update your `node` scripts by adding the `--icu-data-dir` option.

```
{
  "scripts": {
    ...
    "start": "node --harmony --icu-data-dir=node_modules/icu4c-d
ata ./scripts/start"
  },
}
```

Testing

This tutorial explains how to implement the webdriver testing utility to your `Vue.js` application.

Getting Started

If you don't have a `Vue.js` , then clone the vue-example.

Installation

Install the required packages.

```
$ npm install --save-dev webdriverio
```

Generate a configuration file.

```
$ ./node_modules/.bin/wdio config
* Where do you want to execute your tests?
> On my local machine
* Which framework do you want to use?
> mocha
* Shall I install the framework adapter for you?
> Yes
* Where are your test specs located?
> ./tests/**/*.js
* Which reporter do you want to use spec?
> spec
* Shall I install the reporter library for you?
> Yes
* Do you want to add a service to your test setup sauce?
> selenium-standalone
* Shall I install the services for you?
> Yes
* Level of logging verbosity?
> silent
* In which directory should screenshots gets saved if a command
fails?
> ./wdio/shots/
* What is the base url?
> http://localhost:3000
```

Add the `wdio` and `hs_err_pid*` keys to your `.gitignore`. If you are using `nodemon.json` then also add these keys under `ignore`.

Open the `./package.json` file and add the `npm test` command.

```
{
  "scripts": {
    "test": "wdio wdio.conf.js"
  },
}
```

Create a new file `./tests/index.js` and write your first test.

```
const assert = require('assert');

describe('foo', () => {
  it('should be true', () => {
    assert.equal(true, true);
  });
});
```

You can now run your test by executing the `npm test` command.

Linting Support

If you are using eslint, you'll need to register a plugin tu support mocha which we selected while generating the configuration file.

First install the plugin.

```
$ npm install --save-dev eslint-plugin-mocha
```

Then open the `.eslintrc` and reconfigure the file as shown below.

```
module.exports = {
  plugins: ['mocha'],
  env: {
    mocha: true
  },
  globals: {
    browser: false
  },
  rules: {
    'mocha/no-exclusive-tests': 'error'
  }
}
```

Testing Application

Our first test doesn't do much so let's replace it with something useful. The code below navigates to the home page and checks for a valid page title.

```
const assert = require('assert');

describe('/', () => {
  it('should have title', () => {
    browser.url('/');
    assert.equal(browser.getTitle(), 'Example');
  });
});
```

If we execute the `npm test` command, the test will fail because the application is not started. We can start the application manually, we can also use concurrently for starting the npm command but even better solution is to create a custom script for that.

Create a new file `./scripts/test.js` and write a script which starts the application first and then runs the tests.

```
process.env.NODE_ENV = 'production';

const {spawn} = require('child_process');

(async () => {
  // starting your application
  ...
  // running tests
  spawn('./node_modules/.bin/wdio', ['./wdio.conf.js'], {stdio:
'inherit'})
  .on('close', (code) => app.close())
  .on('close', (code) => process.exit(code));
})()
.catch((error) => {
  throw error;
});
```

Open the `./package.json` and change the `npm test` script.

```
{
  "scripts": {
    "test": "node --harmony ./scripts/test.js"
  },
}
```

Using Sauce Service

Install the dependencies.

```
$ npm install --save-dev wdio-sauce-service
```

Open the `./wdio.conf.js` file and the code below at the bottom of the configuration file.

```
const {version} = require('./package');

exports.config = Object.assign(exports.config, {
  user: '{saucelabs-username}',
  key: '{saucelabs-access-key}',
  capabilities: [
    {browserName: 'firefox', build: version},
    {browserName: 'chrome', build: version},
  ],
  services: ['sauce'],
  sauceConnect: true,
  protocol: 'http',
  port: 80
});
```

The `npm test` command will now run tests using `sauce` service. Use the Platform Configurator for generating additional `capabilities`. Don't forget to adding the `build` option to each capability to group tests and display them under the `Automated Builds` tab in your Sauselabs account.

Continuous Deployment

If the test process works on your local machine, then it will work in cloud.

CircleCI is one of the popular continuous integration platforms which we will use here.

First, create a new file `./circle.yml`.

```
machine:
  node:
    version: 6 # until https://github.com/laverdet/node-fibers/issues/321 is resolved
dependencies:
  override:
    - NODE_ENV=development npm install
```

Push your code to Github or Bitbucket repository then navigate to circleci.com and connect your repository. Tests will now be automatically run every time you push an update to that repository.

Note that you can use the `process.env.CI` variable to check if tests are run in cloud.