**General remarks on the utilisation and scope of Mocha**

**Mocha**, as a *test-runner* exists to ease the creation and execution of unit tests on different components of the Vue application. In addition, it provides a variety of *hooks* as part of its API that allows one to easily control how the tests are carried out. As such, its usage should concern itself principally with what one could call the component’s *contract*, that is to say, its outputs based on particular inputs, but not necessarily in terms of UI changes. In particular, one would *mount* a component whilst also *mocking* most of its properties. In practice, this means that the component exists isolated from the rest of the application and perhaps even its subcomponents if it is *shallow mounted*. The generated html may be examined, interacted with, and even compared, but little or no access will be had to other parts of the application. All external components, APIs and the like must be simulated. In order to accomplish this, it is necessary to make use of several tools that exist around mocha: ***Vue-test-utils,*** which exists to allow one to perform all the aforementioned interaction with vue components; ***Chai***, which is an *assertion* library that allows one to formulate the expressions that evaluate the output of the unit tests (i.e. the *assert/expect* and associated api); and finally ***Sinon***, which allows one to easily *stub or spy* on functions and make assertions about their behaviour. To install the mocha plugin one must run **vue add @vue/unit-mocha** if it has not been added at the time of project creation. It is important to remark that the manually configured Vue installation with tests will add *mocha* and *chai*, but one must run: **npm i --save-dev sinon sinon-chai** to access *sinon’s* functionality, and to add sinon tests to chai’s syntax.

Their respective APIs can be found in the following links:

* <https://mochajs.org/>
* <https://www.chaijs.com/>
* <https://sinonjs.org/>
* <https://github.com/domenic/sinon-chai>
* <https://vue-test-utils.vuejs.org/>

In order to better understand the interaction between all these components it is useful to examine a simple but complete example. To that end, we will examine a component from a simple Vuejs app. This component has been designed to make use of Vuex in place of event emission, and as such it seeks to be as clean and succinct as possible. Nonetheless, in order to avoir overencumbering the implementation with boilerplate code, some local variables remain. These however, have no bearing on the global state of the application. With that being said, we can now proceed to examine the component’s code.

|  |
| --- |
| <template>  <div class="row align-items-center justify-content-center mb-3">  <b-form inline @submit.prevent>  <label class="sr-only" for="iFormYearSelect">Year</label>  <b-form-select  class="mb-2 mr-sm-2 mb-sm-0"  id="iFormYearSelect"  :options="yearOptions"  v-model="yearSelected"  @change="changeYear"  data-test="yearSelector"  ></b-form-select>   <label class="sr-only" for="iFormMonthSelect">Month</label>  <b-form-select  class="mb-2 mr-sm-2 mb-sm-0"  id="iFormMonthSelect"  :options="monthOptions"  v-model="monthSelected"  @change="changeMonth"  data-test="monthSelector"  ></b-form-select>   <label class="mr-sm-2" for="iForm2Max">MaxBudget</label>  <b-input-group prepend="€" class="mb-2 mr-sm-2 mb-sm-0">  <b-input  id="mb-2 mr-sm-2 mb-sm-0"  placeholder="0.00"  v-model="currentMax"  type="number"  data-test="maxInput"  />  </b-input-group>   <b-button  variant="outline-primary"  data-test="maxSet"  @click="changeMax"  :disabled="!dateSet"  >Set Max</b-button>  </b-form>  </div> </template> |

As it can be seen from the html template above, what interests us is the fact that there are three main interactive elements, two *selects* and an *input-group* (both from the *bootstrap-vue* library: <https://bootstrap-vue.js.org/docs/>, this will be important later when the tests are being run.) Now, each of these components has been marked with ***data-test=”X”***attributesaccording to its functionality in order to ease their retrieval during tests. In addition the component contains the following simple methods that respond to the ***change***events for the selects, and ***click***events for the button to submit the text in the input field. They are relatively self explanatory. In essence they aim to set within the application’s state the current month and year, so that the relevant entry is shown, and, when a **max** amount is set, that it is then inserted into the state’s entry for that month. In addition the **dateSet** property that determines the buttons enabled or disabled state is simply a boolean that determines whether the month and year have both been set (this is due to the way the information for each year + month pair is stored.) The following code shows the actual functionality of the component.

|  |
| --- |
| <script> import { mapActions, mapGetters } from "vuex";  export default {  name: "MonthSettingsForm",  data() {  return {  yearSelected: null,  yearOptions: [  …   ],  monthSelected: null,  monthOptions: [  …   ],  currentMax: null  };  },  computed: {  ...mapGetters("spending", ['dateSet'])  },  methods: {  changeYear() {  this.setYear(this.yearSelected);  },  changeMonth() {  this.setMonth(this.monthSelected);  },  changeMax() {  this.setMax(this.currentMax)  this.currentMax = null  },  ...mapActions("spending",  ["setYear",  "setMonth",  "setMax"]  )  } }; </script> |

Here one may observe that the component deals with no props or events, but rather, obtains and updates the state of the application directly from the vuex store. (It does however retain a local state for some of its variables. Though this is either to supply the props of the bootstrap-vue elements or to have a variable to bind to the value of these components. To insert and sync this directly to the store would have been overly verbose. (Although it is possible, see: <https://vuex.vuejs.org/guide/forms.html#two-way-computed-property>**)**

What is presented here below is the unit test suite for this component.

|  |
| --- |
| import { mount, createLocalVue } from '@vue/test-utils'; import Vuex from 'vuex'; import BootstrapVue from 'bootstrap-vue'; import flushPromises from 'flush-promises'; import MonthSettingsForm from '../../src/components/MonthSettingsForm'; const chai = require("chai"); const expect = chai.expect; const sinon = require("sinon"); const sinonChai = require("sinon-chai");  chai.use(sinonChai)  const localVue = createLocalVue()  localVue.use(Vuex) localVue.use(BootstrapVue)  //The describe block represents the test 'suite' //It represents a logical grouping of tests. //It can be nested as much as is desired. //Caution: Hooks from outer 'describes' will run within inner 'describes' too. describe('MonthSettingsForm', () => {  //Only these two are of interest for consultation within the tests  //As such they are left within the scope of the suite.  //As they are redefined in every execution, they cannot pollute  //independent tests.  let wrapper  const actions = {  setYear: sinon.stub(),  setMonth: sinon.stub(),  setMax: sinon.stub()  }   //The wrapper factory pattern is suggested in the Vue.js documentation,  //and allows easy control of wrapper properties while also avoiding the  //repetition of code.  const wrapperFactory = (isDateSet) => {  const getters = {  dateSet: sinon.stub().returns(isDateSet)  }  const spending = {  namespaced: true,  state: {},  actions,  getters  }  const store = new Vuex.Store({  modules: {  spending  }  })  //shallowMount cannot be used, as the interactives bootstrap-vue  //elements won't work without being instanced with the component.  return mount(MonthSettingsForm, {  localVue,  store: store,  })  }  //An it block can be considered as the unit test itself  //Within this particular test, we are testing both that  //a function that is local to the component is called and  //that a function that is within the store is called; thus,  //the use of two assertions is justified.  it('Calls setYear on the Store when setting the year', async () => {  wrapper = wrapperFactory(false)  let spy = sinon.spy(wrapper.vm, 'changeYear')  wrapper.find('[data-test="yearSelector"]').setValue('1');  //Given that calls to Vuex, and even the components methods  //are asynchronous, it is necessary to "flush" all promises  //in order to insure execution has been completed.  //The async/await pattern allows execution to halt in order  //to 'wait' for asynchronous code to complete.  await flushPromises()  expect(spy).to.have.been.called  expect(actions.setYear).to.have.been.called  })  …  …  …  }) |

Several remarks must be made regarding the imports: the use of ***const x = require(...)*** is done simply because it is necessary in order to be able to use chai and sinon together. Otherwise it would suffice to write ***import {expect} from ‘chai’***. Next we create the *wrapperFactory* that which will allow us to control how and with what parameters a component and its local vue instance is constructed. Afterwards, a single test is shown and commented in detail. As this example shows, Vue’s idiosyncrasies occasionally force one to change the approach one uses to examine the data. For instance, given the fact that the control calls a function, which itself calls another function, it is probable that this function might not have finished by the time the test continues, and thus, it is necessary to *wait* in order to be able to test this asynchronous behaviour. For more information the following links are recommended:

* <https://vue-test-utils.vuejs.org/guides/testing-async-components.html>
* <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function>
* <https://vuejs.org/v2/guide/reactivity.html#Async-Update-Queue>

In addition the mocha plugin for vue allows us to test any other part of the application that we desire, for instance, the Vuex store. Here is what the tests for the Vuex store of the application look like.

|  |
| --- |
| import { mutations, getters, actions } from '../../src/store/modules/spending'; const chai = require("chai"); const expect = chai.expect; const sinon = require("sinon"); const sinonChai = require("sinon-chai");  chai.use(sinonChai)  describe('Store\'s Spending Module tests', () => {  describe('actions', () => {  it('setYear with no month', () => {  const commit = sinon.spy()  const year = '1'  const currentYear = null  const currentMonth = null  const context = {  state: {  dataObject: {},  currentYear: currentYear,  currentMonth: currentMonth  },  commit: commit  }  actions.setYear(context, year)  expect(commit.args).to.deep.equal([['SET\_YEAR', '1']])  })  …   })   describe('getters', () => {  it('getCurrentMonthEntry', () => {  const year = '1'  const month = '1'  const state = {  dataObject: {  '1': [  {'month': '1', 'max': 0, 'items': []}  ]  },  currentYear: year,  currentMonth: month  }  const result = getters.getCurrentMonthEntry(state)  expect(result).to.deep.equal({'month': '1', 'max': 0, 'items': []})  })  …   })   describe('mutations', () => {  it('CREATE\_ENTRY empty data object', () => {  const year = '1'  const month = '1'  const state = {  dataObject: {}  }  mutations.CREATE\_ENTRY(state, {year: year, month: month})  expect(state.dataObject).to.deep.equal(  {  '1': [  {'month': '1', 'max': 0, 'items': []}  ]  }  )  })  …   }); |

What can be seen up above are example tests for each of the three types of actions that can be done on a Vuex Store. Remarkably, it is not necessary to instantiate a vue or component instance, as the stores act like native javascript and can be tested as such. What must be stated is that generally the functions need to have everything they might need mocked, as there will be no running store to handle requests. This is advantageous in the sense that it allows one to truly reason about what the functions do and how they should behave. In order to truly get acquainted with these types of tests it is well worth going through this page:

* <https://vuex.vuejs.org/guide/testing.html>
* <https://vue-test-utils.vuejs.org/guides/#testing-getters-mutations-and-actions-separately>

The project for these examples can be found here: <https://github.com/kaiser185/vuejsDemo>

**General remarks on the utilisation and scope of Nightwatch**

**Nightwatch** is an automated testing framework, designed to allow its users to automate user interaction tests. In essence, it allows the creation and automation of user *workflows*, to ensure the correct behaviour of an application from the perspective of the user. These are End to End tests (E2E). E2E tests have the particularity that they have no interaction with the internal functionality of the application and can only interact with the rendered HTML. **Nightwatch** makes use of the WebDriver technology, and given its standardized status and stability, it allows one to have confidence in both the capabilites and limitations of the framework. In practice this means that one may write tests in javascript using nightwatch’s API and its useful abstractions of WebDriver commands.

In order to write tests with **nightwatch**, one must have a relatively solid understanding of the html that the page generates and how it behaves, particularly when it has dynamic components. As such it is occasionally useful to add test attributes to html elements in order to make their use and identification easier. The following code demonstrates a simple user workflow with the VariaMos application (development branch, as of April 5 2019), wherein a user creates a project, navigates to his first feature model, creates a very simple model with two features and a relationship and then saves the model. Finally localstorage is queried for the presence of the model, indicating a correct insertion of data.

|  |
| --- |
| module.exports = {  'create a model structure': browser => {  browser  .maximizeWindow()  .url(process.env.VUE\_DEV\_SERVER\_URL)  .waitForElementVisible('#app', 5000)  .assert.elementPresent('[id="app"]')  .click('[data-test="newprojectbutton"]')  .waitForElementVisible('[data-test="newprojectmodal"]', 5000)  .setValue('[data-test="newprojectmodalinput"]', 'nightwatch')  .click('[data-test="newprojectmodal"] button:last-child')  .waitForElementVisible('.naza-tree-row', 5000)  .pause(300)  .moveToElement('.naza-tree-row a i', 0, 0)  .mouseButtonClick(0)  .pause(300)  .moveToElement('[data-test="modelFolder"]', 0, 0)  .pause(100)  .mouseButtonClick(0)  .pause(300)  .moveToElement('[title="Root Feature"]', 0, 0)  .mouseButtonDown(0)  .moveToElement('#graphContainer svg', 300, 200)  .mouseButtonUp(0)  .moveToElement('[title="Leaf Feature"]', 0, 0)  .mouseButtonDown(0)  .moveToElement('#graphContainer svg', 500, 200)  .mouseButtonUp(0)  .pause(50)  .useXpath()  .moveToElement('//\*/text()[normalize-space(.)="leaf"]/parent::\*', 5, 5)  .mouseButtonDown(0)  .moveToElement('//\*/text()[normalize-space(.)="root"]/parent::\*', 5, 5)  .mouseButtonUp(0)  .pause(1000)  .useCss()  .click('#buttonSAVE')  .pause(100)  .click('.modal-container button')  .pause(100)  .useXpath()  .click('//div[@class="tabs"]//\*/text()[normalize-space(.)="binding\_feature\_component"]/parent::\*')  .execute(function() {  return window.localStorage.getItem('Domain - nightwatch');  }, [], function(result) {  this.assert.equal(result.status, 0)  this.assert.notStrictEqual(result.value, '')  })  .pause(9000)  .end()  } } |

As can be seen from the aforementioned code, nightwatch tests have a very simple structure, they essentially receive an object and pass in a browser object which will enable all interactions. Afterwards, essentially all interaction is chained off the browser object. It is worth noting that in order to select elements one must use either CSS or XPATH selectors. Most commands are relatively self-explanatory. In order to get a better feel for the inner workings of nightwatch it is highly recommended to read through their documentation: <http://v09.nightwatchjs.org/api#commands>

**Travis CI**

**Travis** is a continuous integration platform, in other words, it checks the integrity of every build that is pushed to a github repository. In essence, it boots up a virtual machine, clones one’s repository and subsequently runs whatever verification script it is given. If any of the processes exit with a non-zero code, it will mark the build as failed and report said result. Its use is quite simple, one needs only to connect the repository and make a commit. It is however important to remember that it is not a replacement for in-place testing, as it is much better to make sure that no errors occur before commiting the result. That being said, it is also worthwhile having it, as it will catch errors if tests are forgotten, and will also show errors that may present themselves only on linux systems. (As it typically mounts a debian *vm*). The code that follows shows the a configuration file for running nightwatch tests on travis.

|  |
| --- |
| sudo: required dist: trusty addons:  chrome: stable language: node\_js node\_js:  - 10 jdk:  - oraclejdk8 cache:  npm: true # this caches ~/.npm folder on Travis  directories:  - ~/.cache before\_install:  - export CHROME\_BIN=chromium-browser  - export DISPLAY=:99.0  - sh -e /etc/init.d/xvfb start install:  - npm ci script:  # In order to run unit tests as well use:  #- npm run test:unit  - npm run test:e2e |