**Testing VARIAMOS-WEB**

**Introduction**

The VariaMos Web application, given its size, requires a certain measure of stability in its codebase in order to ease the difficulty of collaborative work. Its complex network of components and dependencies is constantly growing; by using tests and their related libraries, one may ensure that one’s work does not fundamentally change the way the application behaves. This is particularly important in order to ensure that others can continue to integrate new features without the fear of having breaking changes made to the codebase. Writing tests, therefore, more than a good practice, is a core part of the development process. In fact, under most circumstances, tests should guide the implementation.

In this guide, we will explore the testing environment, its associated tools and what their use means for the development of VariaMos. It is quite important to remember that tests constitute not only a valuable source of information about the state and stability of the application but also documentation about the use of most, if not all, of the application’s components and the functions contained therein. We will now examine the tools that are required to create tests and how to best make use of them.

**The testing tools: Mocha, Vue-test-utils, Chai, Sinon**

**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 would 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 in isolation 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 or stubbed out. In order to accomplish this, it is necessary to make use of several tools that exist for testing Vue applications: ***Vue-test-utils,*** which enables one to perform all of the aforementioned interactions with vue components[[1]](#footnote-0); ***Chai***, which is an *assertion* library that allows one to formulate the expressions that evaluate the behaviour of that which is under test (i.e. the *assert/expect* expressions and their associated api); and finally ***Sinon***, which allows one to easily *stub* (i.e. to convert a function into a function that does nothing or behaves only as one desires) *or spy* (i.e. to wrap a function within a container that allows one to observe the behaviour of the underlying functions) on functions and make assertions about their behaviour.

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/>

It is highly recommended that one acquaint oneself with all of these testing libraries in order to ease the understanding of that which will follow. VariaMos has all of this already configured for use, and all that must be done to run all of the unit tests is the command ***npm run test:unit***.

**Other important libraries**

As a final remark regarding the testing environment, it is important to mention two other libraries that aid immensely in the creation of tests for VariaMos. The first of these is ***flush-promises*** (<https://github.com/kentor/flush-promises>). This library enables one to await the completion of any pending *promises* before continuing the execution; thus, it allows one to ensure that that which will run afterwards will be done after all asynchronous calls have returned. The second notable library that will be used in testing is ***babel-plugin-rewire*** (<https://github.com/speedskater/babel-plugin-rewire>). This library allows one to ‘rewire’ (i.e. replace with another function, variable, object, etc...) dependencies within an imported component or library. This is quite useful when components themselves have complex dependencies that cannot be easily mocked straight from the test.

**How tests are organized in VariaMos**

All tests are plain javascript files that exist under the ***<project-root>/tests/unit*** folder. The underlying folder structures aims to mirror that of the application itself. In addition, the following (standard) naming convention has been adopted: ***<component-name>.spec.js[[2]](#footnote-1)***. This ensures that it is easy to see the correspondence between tests and their respective components. In addition, within the ***<project-root>/tests/unit/util*** folder, one may find (and add) utility functions to ease the creation of tests. Currently, the only implemented utility function is ***stateFactory***, which, essentially, allows one to quickly create an application state for use with V*uex*.

**Example 1 Testing *Multi-models.vue’s* *clickActiveTab()* method**

In this example only the code relevant to this particular test (and the setup code) will be shown, however, the complete source is currently available in the VariaMos source as ***multimodels.spec.js***.

|  |
| --- |
| import {  shallowMount,  createLocalVue } from '@vue/test-utils'; import {  getters } from "../../../src/store/filetree"; import Vuex from 'vuex'; import MultiModels from '../../../src/views/Multi-models' import model from '../../../src/views/Models' import {  stateFactory } from '../util/StateManagement' 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)  describe.only('Multi-Models', () => {  /\*\*  \* The localgetters object permits us to spy  \* on the behaviour of the getters within the  \* Vuex Store. The assumption that they are  \* functionally correct would allow their use  \* as they are also tested themselves. This avoids  \* the need to reimplement them.  \*/  let localgetters   /\*\*  \* This function stubs the $router.push call within the  \* component.  \*/  let routPushStub   /\*\*  \* mockDispatch allows one to observe  \* the behaviour of the store.dispatch  \* function.  \*/  let mockDispatch;   /\*\*  \* WrapperFactory creates a Vue component wrapper  \* with the selected parameters.  \* @param {Boolean} empty Set whether the state is empty.  \* @param {string} projectName The project parameter for the route.  \* @param {string} folderName The folder parameter for the route.  \* @param {string} typeName The type parameter for the route.  \* @returns A wrapper for the component.  \*/  const wrapperFactory = (empty, projectName = '', folderName = '', typeName = '') => {  const filetree = {  state: stateFactory(empty),  getters: localgetters  }  const store = new Vuex.Store({  modules: {  filetree  }  })  mockDispatch = sinon.stub(store, 'dispatch')  routPushStub = sinon.stub();  /\*\*  \* We use shallowMount to avoid mounting the underlying  \* component structure so as to not overload the required  \* dependecies.  \*/  return shallowMount(MultiModels, {  localVue,  store,  mocks: {  $route: {  params: {  type: typeName,  project: projectName,  folder: folderName  }  },  $router: {  push: routPushStub  },  },  })  }   before( () => {  localgetters = {  getactivetab: sinon.spy(getters, 'getactivetab'),  getdata: sinon.spy(getters, 'getdata'),  getmodelcomponent: sinon.spy(getters, 'getmodelcomponent'),  getmodelcomponentindex: sinon.spy(getters, 'getmodelcomponentindex')  }  })   it('clickActiveTab() works when clicking on tab', () => {  const emptyState = false  const wrapper = wrapperFactory(emptyState, 'Model1', 'Domain-Model1', 'feature')  const link = wrapper.findAll('#atabs').at(1)  link.trigger('click')  expect(routPushStub).to.have.been.called  expect(mockDispatch).to.have.been.called  }) }) |

The code shown above can be split into three intermixed parts: first, the ***import*** and ***require*** statements and the ***localVue*** creation statement allows one to add every necessary dependency into the code; second, one has everything contained within the ***describe()*** function’s callback (except the ***it()*** function), which essentially controls the set-up and housekeeping features of the test; finally, the ***it()[[3]](#footnote-2)*** function’s callback is the test itself, making use of everything that has come before. We will now examine each of these parts in detail.

The first of these is the most straightforward of them all, as it simply details all that is necessary for the test to run. One may remark that two components are being imported, however, the second of these components is merely imported as a convenience as one may use it as the parameter to search for in the rendered component (e.g. in order to test whether it was included in the rendered component given certain conditions, particularly when conditional rendering is used), and, given the fact that it is a subcomponent of the component being tested, it therefore is quite useful. With that being said, it is not necessary for this example and should thus be disregarded. Other details of interest are the ***shallowMount()[[4]](#footnote-3)*** and ***createLocalVue()[[5]](#footnote-4)*** functions as these are quite fundamental to the test process. The first of these allows one to mount the component (i.e. render the component as if it were in the actual application) without any of its subcomponents so that the tests are completely self-contained and do not depend on anything other than the component itself[[6]](#footnote-5). The ***createLocalVue()*** function allows one to create a vue instance that exists only for the test and thus is unencumbered from the rest of the application’s configuration and likewise avoids polluting the base Vue instance of the application. In practice this means one may install all plugins, mixins and anything else that one might include in a Vue instance. Thus the use of the ***localVue.use(Vuex)*** statement makes itself evident.

Next, one finds the ***describe()*** function which exists to organize and describe the test suite that is contained within the callback function that is passed as the second parameter (the first is merely a string that will be its identifier in the output). It is within this callback that the tests and all the house-keeping code must live. These ***describe()*** functions may be nested at will in order to better organize the tests, though only one is required. In this case, the first thing that is done is set up variables of interest that will be queried during the tests. In particular, these are the variables that will hold methods that will be either stubbed or mocked so that one may finely control the behavior of the component. These are declared outside the scope of the test itself, as they are ‘set-up’ before the test itself runs and in order to avoid repeating their definition within every test. Afterwards, the utility function ***wrapperFactory()*** is defined to quickly construct a parametrized instance of the component. Though it may seem somewhat excessive, extracting and isolating logic akin to this for re-use is of the utmost importance in order to create small and comprehensible unit tests[[7]](#footnote-6). The construction of this particular component is done in two steps: first, a simplified version of the structure of the Vuex store is built and its ***dispatch()*** method is stubbed to disable the execution of actions and avoid unnecessary computations; then, by using ***shallowMount()***, a running instance of the component is returned. This function is then called from each test to create a new component, as it is imperative that there be no interdependence between tests for a particular order of execution cannot be counted on. The object that is passed as ***shallowMount()’s*** second parameter are the options that will determine the state, properties, etc. of the component[[8]](#footnote-7). In particular, in this example the router functionality is being mocked entirely[[9]](#footnote-8), as there is no need for routing functionality when testing the component, only the values contained are of interest and are thus able to be parametrized at will. Finally, one of mocha’s ***hooks[[10]](#footnote-9)*** is inserted: a ***before()*** function that will be called before any test is run and thus is useful for setting up the testing environment. Within the callback that is passed to this function the ***sinon.spy()*** method is used to obtain the functions that will fill the object that will contain the ***getters*** that will be used by the Vuex’s ***store*** once constructed in the ***wrapperFactory()*** function where the ***sinon.stub()*** method will be used to create an empty function that will mock ***vue-router’s push()*** function. It is also worthwhile to mention that though these are not used in this particular test, they may prove useful to understand how and when a component accesses the data contained in the ***store[[11]](#footnote-10)***.

Finally, the ***it()*** function contains the test itself, with a syntax very much akin to that of ***describe()***. The callback function follows the ***AAA (Arrange, Act, Assert) pattern*** as follows: the first two lines obtain the component instance (the ***wrapper***) that will be used for testing; the next two lines make use of ***vue-test-utils*** capabilities to select a component (using a CSS selector), and then interact with it by triggering an event on that element; finally the final two lines contain assertions about the state of the two functions that should be triggered by this particular configuration of the component when the click event is fired and the ***clickActiveTab()*** method is invoked through vue’s ***@click*** attribute[[12]](#footnote-11).

This, then, is how one would proceed to test a single file component within a Vue application. The next example will examine a test for another component where some additional work is needed to make the application behave as expected, and it will serve to better illustrate the capabilities and limitations of the test environment.

**Example 2 Testing *Verification.vue’s* *test()* method**

As before, the complete source is currently available in the VariaMos source as ***Verification.spec.js***.

|  |
| --- |
| import {  shallowMount,  createLocalVue } from '@vue/test-utils'; import Vuex from 'vuex'; import VueI18n from 'vue-i18n' import i18n from '../../../../src/i18n' import Verification from '../../../../src/components/model\_actions/Verification' import axios from 'axios' import flushPromises from 'flush-promises' 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(VueI18n)  describe.only('Verification', () => {  /\*\*  \* This function stubs the $router.push call within the  \* component.  \*/  let routPushStub;   /\*\*  \* mockGraph exists to simulate the behaviour of the  \* mxGraph model.  \*/  let mockGraph;   /\*\*  \* These variables will contain the stubs that will replace  \* all of the modal functionality.  \*/  let setupModalStub;  let modalH3Stub;  let modalSimpleTextStub;  let modalButtonStub;   /\*\*  \* This class allows one to mock localStorage without the  \* need for a browser. Unfortunately the setItem method  \* behaves somewhat stragely but it has been necessary to  \* implement it in this way to solve the issue of the different  \* ways one has to interact with localStorage and that its  \* unique behaviour as part of the browser.  \*/  class LocalStorageMock {  constructor() {  this.store = {};  }  clear() {  this.store = {};  }  getItem(key) {  return this.store[key] || null;  }  setItem(key, value) {  this.store[key] = value.toString();  this[key] = value.toString();  }  removeItem(key) {  delete this.store[key];  }  }   /\*\*  \* WrapperFactory creates a Vue component wrapper  \* with the selected parameters.  \* @param {Boolean} empty Set whether the state is empty.  \* @param {string} projectName The project parameter for the route.  \* @param {string} folderName The folder parameter for the route.  \* @param {string} typeName The type parameter for the route.  \* @returns A wrapper for the component.  \*/  const wrapperFactory = (empty, projectName = '', folderName = '', typeName = '') => {  mockGraph = {  getModel: sinon.stub(),  removeCellOverlay: sinon.stub(),  }  /\*\*  \* We use shallowMount to avoid mounting the underlying  \* component structure so as to not overload the required  \* dependecies.  \*/  return shallowMount(Verification, {  localVue,  mocks: {  $route: {  params: {  type: typeName,  project: projectName,  folder: folderName  }  },  $router: {  push: routPushStub  },  },  propsData: {  current\_graph: mockGraph  },  i18n  })  }   beforeEach( () => {   global.localStorage = new LocalStorageMock;  routPushStub = sinon.stub();  setupModalStub = sinon.stub();  modalH3Stub = sinon.stub();  modalSimpleTextStub = sinon.stub();  modalButtonStub = sinon.stub();  })   it.only('test()', async () => {  Verification.\_\_Rewire\_\_('setupModal', setupModalStub)  Verification.\_\_Rewire\_\_('modalH3', modalH3Stub)  Verification.\_\_Rewire\_\_('modalSimpleText', modalSimpleTextStub)  Verification.\_\_Rewire\_\_('modalButton', modalButtonStub)  sinon.stub(axios, 'post').resolves({ data: "Success" });  global.mxCodec = function() {  this.encode = sinon.stub()  }  global.mxUtils = {  getPrettyXml: sinon.stub()  }  global.localStorage.setItem("domain\_implementation\_main\_path", 'XXXX')  const emptyState = false;  const wrapper = wrapperFactory(emptyState, '', '', 'feature');  wrapper.vm.test()  await flushPromises()  expect(setupModalStub).to.have.been.called  expect(modalH3Stub).to.have.been.called  expect(modalSimpleTextStub).to.have.been.called  \_\_rewire\_reset\_all\_\_();  }) }) |

This test is structured much like the other, with some small differences[[13]](#footnote-12), which will be the focus of this example. The first of these differences is the creation of a ***LocalStorageMock*** class[[14]](#footnote-13), which copies the behaviour of the browser’s localstorage. This is necessary as the component will not run in an actual browser instance and will therefore be unable to access the browser’s functionalities. An instance of this class can now be set, in this case in the ***beforeEach()*** function, as part of the global namespace so that the component can access it just as it would the normal localStorage. The choice of doing so in the ***beforeEach()*** hook rests on the fact that if it were set only once, it could potentially pollute other tests; the same rationale applies to the stubs, as they may be queried and having them already contain extraneous information might damage the test results. This exemplifies one of the most glaring limitations of the testing environment: there is no access to anything that would be provided by the browser; should a component make use of anything that might be provided by the browser, there would be a need to simulate it.

Further on, in the ***it()*** function, another of these differences is made manifest. There, the use of ***babel-plugin-rewire’s*** ***\_\_Rewire\_\_()*** method is used to change what a certain dependency points to in a given component[[15]](#footnote-14). This plugin basically steps in whilst babel builds the source and ‘rewires’ what a given dependency resolves to. Also, just as in the ***before()*** hook, the global namespace is written to in order to provide items that are only included as part of ***index.html’s*** code. To avoid any errors, ***localStorage*** is set prior to the creation of the component instance by calling its ***setItem()*** method. In contrast with the other example, the method in question is called directly[[16]](#footnote-15) by first accessing the ***vm*** property of the ***wrapper*** as this is what contains the actual Vue instance.

The method in question contains a call to code that runs asynchronously, which necessitates the use of two important constructs: first, the callback passed to ***it()*** must be declared as ***async*** so that one may use the ***await*** keyword within, which enables one to easily handle[[17]](#footnote-16) the execution of asynchronous calls within the test; finally, using the previous construct, a call is made to ***flushPromises()*** which essentially awaits the end of all pending promises. Afterwards all that remains is to verify that the function made the correct calls and to call ***\_\_rewire\_reset\_all\_\_()*** which undoes the rewiring of dependencies so that they may be used in other tests if need be.

**Example 3 Testing the Vuex Store’s setitemselect() mutation**

The code that follows illustrates how one would go about testing the Vuex store, which, given the fact that it is plain Javascript, is considerably simpler than testing the components themselves. The benefit of testing the store and its functions rests on the fact that once one ensures that the store behaves correctly and that the application’s state behaves in a consistent and predictable manner, one can now assume its correctness when testing the components themselves. In addition, this example shows how one may arbitrarily nest ***describe()*** hooks, as it aids in organizing the different types of functions contained in the store.

|  |
| --- |
| import {  getters,  actions,  mutations } from '../../../src/store/filetree'; import {  stateFactory } from '../util/StateManagement' const chai = require("chai"); const expect = chai.expect; const sinon = require("sinon"); const sinonChai = require("sinon-chai");  chai.use(sinonChai)  describe('Filetree Store Unit tests', () => {  describe('Mutations', () => {  it('setitemselect', () => {  const mutationIndex = 3  const state = stateFactory(false)  mutations.setitemselect(state, mutationIndex)  //Check that only one item is selected.  const correctSelection = state.data.every((element, index) => {  if (index !== mutationIndex) {  return element.data.isSelected === false  } else {  return element.data.isSelected === true  }  })  expect(correctSelection).to.deep.equal(true)  })  }) }) |

This example follows a very similar pattern to the two previous examples with the notable exceptions of not making use of ***describe()*** hooks and not making use of anything related to Vue itself. The reason for this stems from the fact that the store exists as plain Javascript and thus the tests merely query whether the functions are correctly implemented. Here it suffices to import the store’s functions and to call them directly.

Hopefully this guide has made clear how to test within the context of VariaMos, and how one might go about implementing different types of tests. Any comments on either the text or code are very much welcome.

1. Given that one may test any valid javascript, and that the application contains elements other than vue components, this isn’t always required. [↑](#footnote-ref-0)
2. Should the files being tested not be components, one should nonetheless respect this naming rule. [↑](#footnote-ref-1)
3. The actual test suite would include several of these ***it()*** functions. [↑](#footnote-ref-2)
4. <https://vue-test-utils.vuejs.org/api/#shallowmount> [↑](#footnote-ref-3)
5. <https://vue-test-utils.vuejs.org/api/#createlocalvue> [↑](#footnote-ref-4)
6. Evidently, if one were to not do such a thing one would move into the territory of integration tests, which are resolutely beyond the scope of this document. [↑](#footnote-ref-5)
7. All of the other vue component test suites have similar ***wrapperFactory()*** functions with slight variations in content and features; thus, it stands to reason that one might extract this function into a utility file to deal with all components in a generic manner, with the risk of losing the granularity of control. [↑](#footnote-ref-6)
8. These are discussed at length here <https://vue-test-utils.vuejs.org/api/options.html#context>, which contains several quite good examples. [↑](#footnote-ref-7)
9. The guide at <https://lmiller1990.github.io/vue-testing-handbook/vue-router.html#using-a-mock-router> explains this quite well, but in essence all one is doing is replacing the objects on the component’s ***this***. [↑](#footnote-ref-8)
10. <https://mochajs.org/#hooks> <https://mochajs.org/#run-cycle-overview> It is very much advisable that one read at least the information contained in these two links to grasp the basics of mocha and how its hooks interact with the test suite. [↑](#footnote-ref-9)
11. In this case they are mainly included for illustrative purposes, as this mechanism can also be used to modify their behaviour and thus increase the control over the ***store*** that is passed to the component by, for example, stubbing the functions and returning a set content. They may also prove useful for debugging purposes, as querying them with ***sinon’s*** api is very straightforward. [↑](#footnote-ref-10)
12. It is unnecessary to make assertions about the framework itself and thus this interaction is comparable to a direct invocation of the method in question, but with the added benefit of receiving the same parameters that the method would be called with during the actual usage of the application. In *general* these two approaches are comparable and the choice is to be made in terms of the complexity of the test. [↑](#footnote-ref-11)
13. For the sake of brevity the differences in ***shallowMounts()’s*** callwill be but a footnote, as all that has changed is that the component is being fed some specific data for its ***props*** object and it is also passed the project’s ***i18n*** to allow access to ***$t()*** in the component. [↑](#footnote-ref-12)
14. This has been adapted from the answers on this StackOverflow question: <https://stackoverflow.com/questions/32911630/how-do-i-deal-with-localstorage-in-jest-tests> [↑](#footnote-ref-13)
15. One might remark that ***sinon’s stub()*** and ***spy()*** could theoretically be used to do this. In practice, however, per <https://sinonjs.org/releases/latest/stubs/>, they recommend the use of helper libraries to replace complex dependencies. Further, the use of ***babel-plugin-rewire*** is quite straightforward, and implies no changes to the behaviour of the application itself during compilation as it has been configured to only fire in test environments. This was done by modifying the application’s ***babel.config.js*** after the plugin was installed. [↑](#footnote-ref-14)
16. It would be nearly identical if the other method had been employed, as the function is merely invoked when the link is clicked with no additional information being passed. [↑](#footnote-ref-15)
17. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function> This link explains this quite succinctly, though, if one thing must be retained it is the idea that once one declares a function to be ***async*** one may now use the ***await*** keyword before function calls, which signals that the execution must stop and await the completion of that which is marked with ***await***. It does however introduce some interesting side effects such as converting the outer function into an implicit promise, though for our purposes it does not affect how tests behave in any noticeable way. Interestingly, one may also force the vue instance to run through its event loop and move onto the next ***tick*** by using ***wrapper.vm.$nextTick()***, and, given that it returns a promise one can either pass it a callback for the test, ***await*** it, or chain a ***then()*** function call. Making use of ***flush-promises*** is still perhaps the most elegant approach, and necessitates the least amount of interaction with the internals of the vue instance. [↑](#footnote-ref-16)