**Unit Testing Backbone JS and Marionette JS with Jasmine**

[What is Jasmine?](#_krgrqz8kzmlw)

[Quick Concepts :: Getting up to Speed](#_etsk78uw53t8)

[Suites](#_6klv088zo34b)

[Specs](#_5fcquf7mio4c)

[Expectations](#_nhwazvpqltn1)

[Matchers](#_r1uuojzg36sw)

[Manually Failing a Spec](#_apteqzvw2tir)

[Teardown](#_nvpno2nxltte)

[Setting up Jasmine with Rails](#_vn8oe93w8dgw)

[Using Jasmine to Test Backbone / Marionette JS](#_vigsc5ew53we)

[Unit Testing of Models](#_msrefzwbfxxw)

[Unit Testing of Views](#_5ueoirqljlu0)

[Unit Testing of Routers](#_504epa4rz7qt)

[Important Notes](#_kfbhg2h2i31u)

## 

## 

## What is Jasmine?

Jasmine is a behavior-driven development framework for testing JavaScript code. The key point is It does not depend on any other JavaScript frameworks. Also, it does not require a DOM. Additionally, it has a clean and obvious syntax so that tests can be easily written.

Throughout the document, we will be using the latest release (2.6.4) of Jasmine (as of 25 July 2017). The release can be downloaded by following [this](https://github.com/jasmine/jasmine/releases) link.

## **Quick Concepts :: Getting up to Speed**

### ***Suites***

* To begin with testing, it is a general practice to start with defining test suites.
* A test suite begins with a call to the global Jasmine function ***“describe”*** with two parameters: a string and a function.
* The string is a name or title for a spec suite - usually what is being tested.
* The function is a block of code that implements the suite.

*Quick Look:*

describe("A suite", **function**() {  
 // Specs Here  
});

### ***Specs***

* In Jasmine, we start writing test cases by specifying the specs inside the suite.
* Specs are defined by calling the global Jasmine function ***“it”***, which, like describe takes a string and a function. The string is the title of the spec and the function is the spec, or test.
* These specs contain one or more expectations to test the state of the code.

*Quick Look:*

describe("A suite", **function**() {  
 it("contains spec with an expectation", **function**() {  
 // Expectations  
 });  
});

### ***Expectations***

* Expectations are similar to ***assertions*** *(or in general a type of a boolean expression)*.
* An expectation in Jasmine is an assertion that is either true or false.
* A spec with all true expectations is a passing spec.
* A spec with one or more false expectations is a failing spec.
* Expectations are built with the function *expect* which takes a value, called the actual.
* It is chained with a ***Matcher*** function, which takes the expected value.
* Expectations are the key functions used to test our code state.

*Quick Look:*

describe("A suite", **function**() {  
 it("contains spec with an expectation", **function**() {  
 expect(true).toBe(true);  
 });  
});

As seen above, expectations needs to be chained with matcher functions to yield testing results.

### ***Matchers***

* Each matcher implements a boolean comparison between the actual value and the expected value.
* It is responsible for reporting to Jasmine if the expectation is true or false. Jasmine will then pass or fail the spec.
* Any matcher can evaluate to a negative assertion by chaining the call to expect with a not before calling the matcher.

*Jasmine provides a rich set of inbuilt matcher functions, all these functions are listed below:*

|  |  |
| --- | --- |
| **Matcher Name** | **Description** |
| toBe | compares with === |
| toEqual | works for simple literals and variables  should work for objects |
| toMatch | for regular expressions |
| toBeDefined | compares against `undefined` |
| toBeUndefined | compares against `undefined` |
| toBeNull | compares against null |
| toBeTruthy | for boolean casting testing |
| toBeFalsy | for boolean casting testing |
| toContain | works for finding an item in an Array  also works for finding a substring |
| toBeLessThan | for mathematical comparisons |
| toBeGreaterThan | for mathematical comparisons |
| toBeCloseTo | for precision math comparison |
| toThrow | for testing if a function throws an exception |
| toThrowError | for testing a specific thrown exception |

Sometimes, there may be a need to define custom matchers. This is also facilitated by Jasmine and can be learnt quickly by following [this](https://jasmine.github.io/2.6/custom_matcher.html) link.

### ***Manually Failing a Spec***

* Sometimes in a flow of writing test cases, there may be a need to manually fail the test case on occurrence of certain conditions.
* This can be easily achieved using the ***“fail”*** function which takes a message or an error object as a parameter.

*Quick Look:*

describe("A spec using the fail function", **function**() {  
 **var** foo = **function**(x, callBack) {  
 if (x) {  
 callBack();  
 }  
 };  
  
 it("should not call the callBack", **function**() {  
 foo(false, **function**() {  
  ***fail("Callback has been called")***;  
 });  
 });  
});

### ***Teardown***

* After getting up to speed and creating some test cases, there is a possibility that some duplicated setup can be spotted in the test suite.
* To DRY (Do-not-Repeat-Yourself) Up your test-suite, Jasmine has the following list of global functions which can be used in your test suite:

|  |  |
| --- | --- |
| Function Name | Description |
| beforeEach | called once before each spec in the describe in which it is called |
| afterEach | called once after each spec |
| beforeAll | called only once before all the specs in describe are run |
| afterAll | called after all specs finish |

## **Setting up Jasmine with Rails**

Using Jasmine with rails makes it easier to integrate Jasmine with the rails workflow.

To setup, simply add the jasmine-rails to your gemfile.

*Quick Look:*

group :test, :development do

gem 'jasmine-rails'

end

Then run bundle install:

*Quick Look:*

$ bundle install

And finally, the rails generator:

*Quick Look:*

$ rails generate jasmine\_rails:install

The generator will create the necessary configuration files and mount a test runner to /specs so that you can get started with writing specs.

You can configure the jasmine.yml file for adding or tweaking configurations.

To add external JS libraries, (which we will need in this documentation ahead) please make corresponding changes in the jasmine.yml file.

To get all the additional JS libraries we will be using, please scroll to the end of the document.

For more details about this gem, please follow [this](https://github.com/searls/jasmine-rails) link.

To view the test results on the browser, simply run the following command:

***$ rails jasmine***

After this command is executed, a server will be started and the test results can be viewed on http://localhost:8888

## **Using Jasmine to Test Backbone / Marionette JS**

While developing an application using Backbone or Marionette JS, the key components of the front-end application are Models, Views and the Routers.

For effective testing, it is necessary to isolate the testing of Models, Views and Routers.

Therefore, while testing Backbone / Marionette Application, it is a good practice to write separate specs for Models, Views and Controllers.

### **Unit Testing of Models**

Testing of Model is pretty simple thing. It’s nothing more than testing some business logic, that might reside inside.

Let’s assume that we have a model named: Profile.js

The only one functionality that Profile.js model contains currently is “validation”, let’s begin with testing that.

|  |
| --- |
| describe('Profile.js spec', function () {  var model;  beforeEach(function () {  model = new Profile();  });  describe('when model is validating', function () {  var errors;  });  }); |

The above code spec, as you can see, does not do any asserts. This is written something to start with.

Now, let’s add some real test cases.

***First case is that, both the name and email attributes are absent.***

|  |
| --- |
| describe('when name and email fields are absent', function () {  beforeEach(function () {  errors = model.validate({});  });  it ('should have 2 errors', function () {  expect(errors.length).toBe(2);  });  it ('should have name field as invalid', function () {  expect(errors[0].name).toBe('name');  });  it ('should have email field as invalid', function () {  expect(errors[1].email).toBe('email');  });  }); |

The above test suite describes that if the name and email fields are empty, the validate function of the model will be invoked and it will return an array containing the validation with respect to the empty fields.

***Second Case: The user may enter name but forgot about email***

|  |
| --- |
| describe('when name is set, but email is absent', function () {  beforeEach(function () {  errors = model.validate({ name: 'abcd'});  });  it ('should have 1 error', function () {  expect(errors.length).toBe(1);  });  it ('should have email field as invalid', function () {  expect(errors[0].name).toBe('email');  });  it ('should have error message', function () {  expect(errors[0].message).toBeDefined();  });  }); |

The second test suite describes that if the user enters the name but forgets about email, then the model will validate and return the appropriate response stating that the email field is left invalid.

***Third Case: User might put email, but forgets about name***

|  |
| --- |
| describe('when email is set, but name is absent', function () {  beforeEach(function () {  errors = model.validate({ email: 'a@abc.com'});  });  it ('should have 1 error', function () {  expect(errors.length).toBe(1);  });  it ('should have email field as invalid', function () {  expect(errors[0].name).toBe('name');  });  it ('should have error message', function () {  expect(errors[0].message).toBeDefined();  });  }); |

The third test suite describes that if the user enters the email but forgets about name, then the model will validate and return the appropriate response stating that the name field is left invalid.

***Test Reports:***

Now to view the test results, just run ***$ rails jasmine*** in the console (assuming you are in the project directory) and navigate to ***http://localhost:8888***

### **Unit Testing of Views**

Views are the core aspects of the framework. They are actually ones who do stuffs. In backbone or marionette the views could be treated as controllers in some way.

While testing views, one should focus on the following things:

1. Initialization: Testing whether the view is provided with all required inputs.

2. Rendering: Testing that the required html appeared in the view.

3. Events: Testing that the view is correctly handling the DOM events.

4. Model changes and persistence: Testing that the changes in the view are correctly propagated in model.

The best part of testing backbone or marionette views is that, we don’t need any special HTML on test page, since views hold its all DOM elements in ***this.el.*** This means, by using JQuery, its easy to change DOM values or trigger events.

Since view also holds reference to model, it’s easy to check model’s attributes changes or spying on particular models methods.

Let’s start testing the views with following examples:

**1. Testing *Initialization***

Test Case 1: Testing that ProfileView should exist, initially the test will fail, but as soon as the ProfileView function is defined, the test should be passed.

|  |
| --- |
| describe('ProfileView.js spec', function () {  var view, model;  beforeEach(function () {  view = new ProfileView();  });  describe('when view is constructing', function () {  it ('should exist', function () {  expect(view).toBeDefined();  });  });  }); |

Test Case 2: ProfileView could not exist without a model as well as default ‘name’ text, that should be used as initial value of textbox.

|  |
| --- |
| describe('when view is initialized', function () {  describe('without model', function () {  it('should throw exception', function () {  expect(function () {  new ProfileView();  }).toThrow(new Error('model is required'));  });  });  describe('without default name', function () {    it('should throw exception', function () {  expect(function () {  new ProfileView({model: new Backbone.Model() });  }).toThrow(new Error('name is required'));  });  });  }); |

The result of above test suite fails if the view is not assigned a model and a default value for ‘name’.

Once we add initialize function in the view and assign a model and a default value to ‘name’, then the test will pass.

Let’s add the initialize function in the view in this way:

|  |
| --- |
| initialize: function (options) {  if (!this.model) {  throw new Error('model is required');  }  if (!(options && options.name)) {  throw new Error('name is required');  }  this.name = options.name;  }, |

Now, once we have added the initialize function in the view, it is necessary to fix the previous Test Case in the following way:

|  |
| --- |
| describe('ProfileView.js spec', function () {  var view, model;  beforeEach(function () {  view = new ProfileView({model: new Profile(), name: 'abc' });  });  describe('when view is constructing', function () {  it ('should exist', function () {  expect(view).toBeDefined();  });  }); |

**2. Rendering**

Now, let’s test how the rendering works.

In this test suite, some of the matchers are from jasmine-jquery library. It adds a number of matchers useful for testing jQuery objects.

|  |
| --- |
| describe('when view is rendered', function () {  beforeEach(function () {  view.render();  });  it ('should email field be empty', function () {  expect(view.$el.find('input#email')).toHaveValue('');  });  it ('should website field be empty', function () {  expect(view.$el.find('input#website')).toHaveValue('');  });  it ('should name field with default name', function () {  expect(view.$el.find('textarea#name')).toHaveValue('abc');  });  }); |

We have tested initialization and rendering, now let’s test the last aspect, i.e. Events and Model changes and persistence.

**3. Events and Model Changes and Persistence**

Suppose a user inputs nothing, erases default name and presses submit button. Expected behavior is expressed with this test:

|  |
| --- |
| describe('when form is submitted', function () {  describe('no inputs are filled', function () {  beforeEach(function () {  view.$el.find('#email').val('').trigger('change');  view.$el.find('#name').val('').trigger('change');  });  beforeEach(function () {  view.$el.find('#submit').trigger('click');  });  it('email field should be invalidated', function () {  expect(view.$el.find('.control-group.email')).toHaveClass('error');  });  it('name field should be invalidated', function () {  expect(view.$el.find('.control-group.name')).toHaveClass('error');  });  it('website field should be valid', function () {  expect(view.$el.find('.control-group.website')).not.toHaveClass('error');  });  }); |

After these tests are passing, we can test that if only email is filled, but name is still empty, we are not able to submit the form:

|  |
| --- |
| describe('only email field filled', function () {  beforeEach(function () {  view.$el.find('#email').val('a@a.com').trigger('change');  view.$el.find('#name').val('').trigger('change');  });  beforeEach(function () {  view.$el.find('#submit').trigger('click');  });  it('email field should be valid', function () {  expect(view.$el.find('.control-group.email')).not.toHaveClass('error');  });  it('name field should be invalidated', function () {  expect(view.$el.find('.control-group.name')).toHaveClass('error');  });  it('website field should be valid', function () {  expect(view.$el.find('.control-group.website')).not.toHaveClass('error');  });  }); |

And lastly, if the view is filled correctly,

|  |
| --- |
| describe('email and name filled', function () {  beforeEach(function () {  spyOn(view.model, 'save').andCallThrough();  });  beforeEach(function () {  view.$el.find('#email').val('a@a.com').trigger('change');  view.$el.find('#name').val('some name').trigger('change');  });  beforeEach(function () {  view.$el.find('#submit').trigger('click');  });  it('should show no errors', function () {  expect(view.$el.find('.error').length).toBe(0);  });  it('should save model', function () {  expect(view.model.save).toHaveBeenCalled();  });  }); |

Here, we are testing two things. First that no validation errors appeared on form and second that ***save*** method of the model is called. Jasmine built-in spy framework is used here. You can setup on any object and then verify that function has (or has not) been called.

Following is the implementation of the view on which this testing is performed:

|  |
| --- |
| ProfileView.js |
| var ProfileView = Backbone.View.extend({  className: 'row',  template: '\  <form>\  <legend>Share the details</legend>\  <div class="control-group email">\  <label>Email</label>\  <input type="text" id="email" placeholder="Your email address...">\  <span class="help-inline"></span>\  </div>\  <div class="control-group website">\  <label>Website</label>\  <input type="text" id="website" placeholder="Your website...">\  <span class="help-inline"></span>\  </div>\  <div class="control-group name">\  <label>Name</label>\  <textarea id="name" class="input-xxlarge" placeholder="Name" rows="6"><%= name %></textarea>\  <span class="help-inline"></span>\  </div>\  <button type="submit" id="submit" class="btn">Submit</button>\  </form>\  ',  events: {  'click #submit': 'submitClicked'  },  initialize: function (options) {  if (!this.model) {  throw new Error('model is required');  }  if (!(options && options.name)) {  throw new Error('name is required');  }  this.name = options.name;  },  render: function () {  this.$el.html(\_.template(this.template, {name: this.name}));  return this;  },  submitClicked: function (e) {  e.preventDefault();  var me = this;  var options = {  success: function () {  me.hideErrors();  },  error: function (model, errors) {  me.showErrors(errors);  }  };  var profile = {  email: this.$('#email').val(),  website: this.$('#website').val(),  profile: this.$('#name').val()  };  this.model.save(name, options);  },  showErrors: function(errors) {  \_.each(errors, function (error) {  var controlGroup = this.$('.' + error.name);  controlGroup.addClass('error');  controlGroup.find('.help-inline').text(error.message);  }, this);  },  hideErrors: function () {  this.$('.control-group').removeClass('error');  this.$('.help-inline').text('');  }  }); |

### **Unit Testing of Routers**

In general, the task of routers in backbone or marionette js is as follows:

* The AppRouter responds to the home-page route (represented by an empty hash)
* The home route method instantiates a View and a collection or model.
* The collection or model is asked to fetch its contents from the server.
* When this response is received, the View renders the list.

Once the code for router is defined and ensured that routers are fired correctly, then we can test route methods by simply calling them.

Let’s assume we have an index method as a route method, and we need to ensure that it instantiates a ProfileView and Profiles collection in the correct way. There is one more addition over here, we will need to create fake objects for both.

Here, the routing procedures in Backbone and Marionette JS are different, therefore, for the examples shown below, the codes are written in Backbone JS perspective but their corresponding implementation in Marionette JS is written beside the applicable line of code, thereby it will be easy to map same examples with Marionette JS application along with Backbone JS.

First, Let’s start with initializing a test suite for testing router methods.

|  |
| --- |
| describe("AppRouter", function() {  beforeEach(function() {  this.router = new AppRouter();  */\** *Marionette: this.router = new AppRouter({controller: controller\_name);* *\*/*  this.collection = new Backbone.Collection();  */\* Marionette: Backbone.Marionette.Collection(); \*/*  this.profileViewStub = sinon.stub(window, "ProfileView")  .returns(new Backbone.View()); */\* Marionette: Marionette.View() \*/*  this.profilesCollectionStub = sinon.stub(window, "Profiles")  .returns(this.collection);  });  afterEach(function() {  window.ProfileView.restore();  window.Profiles.restore();  });  }); |

Here, first we create our router instance for testing. We then create a bare Backbone.js Collection object to act as the Profiles collection that will be returned when we stub out its constructor function. Finally, we create Sinon.JS stubs for both the ProfileView constructor and the Profiles collection constructor, returning a new Backbone.js View and our bare collection respectively.

Now, let’s start with writing specs:

|  |
| --- |
| describe("Index handler", function() {  describe("when no Profile exists", function() {  beforeEach(function() {  this.router.index(); */\* Marionette: this.router.controller.index(); \*/*  });  it("creates a Profile collection", function() {  expect(this.profilesCollectionStub)  .toHaveBeenCalledOnce();  expect(this.profilesCollectionStub)  .toHaveBeenCalledWithExactly();  });  it("creates a Profile view", function() {  expect(this.profileListViewStub)  .toHaveBeenCalledOnce();  expect(this.profileListViewStub)  .toHaveBeenCalledWith({  collection: this.collection  });  });  });  }); |

Before each spec, we call our index method for testing.

In the first spec we check that the Profiles collection constructor has been called exactly once, and that it was called with no arguments.

In the second spec, we check that the ProfileView constructor was also called once, and that it was called with a hash object containing our stubbed collection instance. In this way we are testing that the application is linking the ProfileView with its data source, the Profiles collection.

When these specs run, we may get four failures:

|  |
| --- |
| creates a Profile collection  Expected Function to have been called once.  Expected Function to have been called with exactly.  creates a Profile view  Expected Function to have been called once  Expected Function to have been called with ... |

To fix these, additional code is to be written to the AppRouter.js file:

|  |
| --- |
| var AppRouter = Backbone.Router.extend({ *// Marionette: Marionette.AppRouter.extend*  ...  index: function() {  this.profiles = new Profiles();  this.profilesView = new ProfileView({  collection: this.profiles  });  }  }); |

This is part is done.

We now need to test that collection’s data is fetched when the index route is run. This is done by simply calling the Profiles collection’s fetch method. Let’s write another spec.

First, we need to stub the collection’s fetch method so that it performs no action, but allows us to spy on it. We add the following line to our beforeEach method just after creating this.collection:

|  |
| --- |
| describe("AppRouter", function() {  beforeEach(function() {  ...  this.collection = new Backbone.Collection();  */\* Marionette: Backbone.Marionette.Collection(); \*/*  this.fetchStub = sinon.stub(this.collection, "fetch")  .returns(null);  ...  });  ...  }); |

Then, we can add our new spec after the previous two:

|  |
| --- |
| it("fetches the Profiles from the server", function() {  expect(this.fetchStub).toHaveBeenCalledOnce();  expect(this.fetchStub).toHaveBeenCalledWith();  }); |

Again, as expected, the test fails returning the following:

|  |
| --- |
| fetches the Profiles from the server  Expected Function to have been called once.  Expected Function to have been called with. |

To fix this, just add one more line in the AppRouter.js file:

|  |
| --- |
| var AppRouter = Backbone.Router.extend({ *// Marionette: Marionette.AppRouter.extend*  ...  index: function() {  this.profiles = new Profiles();  this.profilesView = new ProfileView({  collection: this.profiles  });  this.profiles.fetch();  }  }); |

If you run the test again, the tests should pass.

In this way, we have successfully tested the router methods in our Backbone JS or Marionette JS application.

**Key Points to Note while testing routers:**

1. There is no need to test how the routing mechanism is working as these are very well tested by the creators and the corresponding test suites can be found on their official links.
2. To get started with very basic route testing, you can use the following code as boilerplate code, this code tests the very basic functionality of the routing:

|  |
| --- |
| describe("Routing::", function(){  var router;  beforeEach(function(){  router = new AppRouter({controller: myController});  });  it("Default route should load 'defaultView' view", function(){  expect(router.appRoutes[""]).toEqual('defaultView');  });  }); |

## Important Notes

***Gems used in the entire documentation:***

gem 'jasmine'

***JQuery Libraries:***

Jasmine-jquery.js

Jasmine-sinon.js