### **1 Why Write Tests for your Rails Applications?**

Rails makes it super easy to write your tests. It starts by producing skeleton test code while you are creating your models and controllers.

By simply running your Rails tests you can ensure your code adheres to the desired functionality even after some major code refactoring.

Rails tests can also simulate browser requests and thus you can test your application's response without having to test it through your browser.

### **2 Introduction to Testing**

Testing support was woven into the Rails fabric from the beginning. It wasn't an "oh! let's bolt on support for running tests because they're new and cool" epiphany.

#### **2.1 Rails Sets up for Testing from the Word Go**

Rails creates a test directory for you as soon as you create a Rails project using rails new*application\_name*. If you list the contents of this directory then you shall see:

|  |
| --- |
| $ ls -F test  controllers/ helpers/ mailers/ test\_helper.rb  fixtures/ integration/ models/ |

The models directory is meant to hold tests for your models, the controllers directory is meant to hold tests for your controllers and the integration directory is meant to hold tests that involve any number of controllers interacting. There is also a directory for testing your mailers and one for testing view helpers.

Fixtures are a way of organizing test data; they reside in the fixtures directory.

The test\_helper.rb file holds the default configuration for your tests.

#### **2.2 The Test Environment**

By default, every Rails application has three environments: development, test, and production.

Each environment's configuration can be modified similarly. In this case, we can modify our test environment by changing the options found in config/environments/test.rb.

Your tests are run under RAILS\_ENV=test.

#### **2.3 Rails meets Minitest**

If you remember, we used the rails generate model command in the [Getting Started with Rails](http://guides.rubyonrails.org/getting_started.html)guide. We created our first model, and among other things it created test stubs in the test directory:

|  |
| --- |
| $ bin/rails generate model article title:string body:text  ...  create app/models/article.rb  create test/models/article\_test.rb  create test/fixtures/articles.yml  ... |

The default test stub in test/models/article\_test.rb looks like this:

|  |
| --- |
| require 'test\_helper'    class ArticleTest < ActiveSupport::TestCase  # test "the truth" do  # assert true  # end  end |

A line by line examination of this file will help get you oriented to Rails testing code and terminology.

|  |
| --- |
| require 'test\_helper' |

By requiring this file, test\_helper.rb the default configuration to run our tests is loaded. We will include this with all the tests we write, so any methods added to this file are available to all our tests.

|  |
| --- |
| class ArticleTest < ActiveSupport::TestCase |

The ArticleTest class defines a *test case* because it inherits from ActiveSupport::TestCase. ArticleTest thus has all the methods available from ActiveSupport::TestCase. Later in this guide, we'll see some of the methods it gives us.

Any method defined within a class inherited from Minitest::Test (which is the superclass of ActiveSupport::TestCase) that begins with test\_ (case sensitive) is simply called a test. So, methods defined as test\_password and test\_valid\_password are legal test names and are run automatically when the test case is run.

Rails also adds a test method that takes a test name and a block. It generates a normal Minitest::Unit test with method names prefixed with test\_. So you don't have to worry about naming the methods, and you can write something like:

|  |
| --- |
| test "the truth" do  assert true  end |

Which is approximately the same as writing this:

|  |
| --- |
| def test\_the\_truth  assert true  end |

However only the test macro allows a more readable test name. You can still use regular method definitions though.

The method name is generated by replacing spaces with underscores. The result does not need to be a valid Ruby identifier though, the name may contain punctuation characters etc. That's because in Ruby technically any string may be a method name. This may require use of define\_method and send calls to function properly, but formally there's little restriction on the name.

Next, let's look at our first assertion:

|  |
| --- |
| assert true |

An assertion is a line of code that evaluates an object (or expression) for expected results. For example, an assertion can check:

* does this value = that value?
* is this object nil?
* does this line of code throw an exception?
* is the user's password greater than 5 characters?

Every test may contain one or more assertions, with no restriction as to how many assertions are allowed. Only when all the assertions are successful will the test pass.

##### **2.3.1 Your first failing test**

To see how a test failure is reported, you can add a failing test to the article\_test.rb test case.

|  |
| --- |
| test "should not save article without title" do  article = Article.new  assert\_not article.save  end |

Let us run this newly added test (where 6 is the number of line where the test is defined).

|  |
| --- |
| $ bin/rails test test/models/article\_test.rb:6  Run options: --seed 44656    # Running:    F    Failure:  ArticleTest#test\_should\_not\_save\_article\_without\_title [/path/to/blog/test/models/article\_test.rb:6]:  Expected true to be nil or false      bin/rails test test/models/article\_test.rb:6        Finished in 0.023918s, 41.8090 runs/s, 41.8090 assertions/s.    1 runs, 1 assertions, 1 failures, 0 errors, 0 skips |

In the output, F denotes a failure. You can see the corresponding trace shown under Failure along with the name of the failing test. The next few lines contain the stack trace followed by a message that mentions the actual value and the expected value by the assertion. The default assertion messages provide just enough information to help pinpoint the error. To make the assertion failure message more readable, every assertion provides an optional message parameter, as shown here:

|  |
| --- |
| test "should not save article without title" do  article = Article.new  assert\_not article.save, "Saved the article without a title"  end |

Running this test shows the friendlier assertion message:

|  |
| --- |
| Failure:  ArticleTest#test\_should\_not\_save\_article\_without\_title [/path/to/blog/test/models/article\_test.rb:6]:  Saved the article without a title |

Now to get this test to pass we can add a model level validation for the *title* field.

|  |
| --- |
| class Article < ApplicationRecord  validates :title, presence: true  end |

Now the test should pass. Let us verify by running the test again:

|  |
| --- |
| $ bin/rails test test/models/article\_test.rb:6  Run options: --seed 31252    # Running:    .    Finished in 0.027476s, 36.3952 runs/s, 36.3952 assertions/s.    1 runs, 1 assertions, 0 failures, 0 errors, 0 skips |

Now, if you noticed, we first wrote a test which fails for a desired functionality, then we wrote some code which adds the functionality and finally we ensured that our test passes. This approach to software development is referred to as [*Test-Driven Development* (TDD)](http://c2.com/cgi/wiki?TestDrivenDevelopment).

##### **2.3.2 What an error looks like**

To see how an error gets reported, here's a test containing an error:

|  |
| --- |
| test "should report error" do  # some\_undefined\_variable is not defined elsewhere in the test case  some\_undefined\_variable  assert true  end |

Now you can see even more output in the console from running the tests:

|  |
| --- |
| $ bin/rails test test/models/article\_test.rb  Run options: --seed 1808    # Running:    .E    Error:  ArticleTest#test\_should\_report\_error:  NameError: undefined local variable or method `some\_undefined\_variable' for #<ArticleTest:0x007fee3aa71798>  test/models/article\_test.rb:11:in `block in <class:ArticleTest>'      bin/rails test test/models/article\_test.rb:9        Finished in 0.040609s, 49.2500 runs/s, 24.6250 assertions/s.    2 runs, 1 assertions, 0 failures, 1 errors, 0 skips |

Notice the 'E' in the output. It denotes a test with error.

The execution of each test method stops as soon as any error or an assertion failure is encountered, and the test suite continues with the next method. All test methods are executed in random order. The [config.active\_support.test\_order option](http://guides.rubyonrails.org/configuring.html#configuring-active-support) can be used to configure test order.

When a test fails you are presented with the corresponding backtrace. By default Rails filters that backtrace and will only print lines relevant to your application. This eliminates the framework noise and helps to focus on your code. However there are situations when you want to see the full backtrace. Simply set the -b (or --backtrace) argument to enable this behavior:

|  |
| --- |
| $ bin/rails test -b test/models/article\_test.rb |

If we want this test to pass we can modify it to use assert\_raises like so:

|  |
| --- |
| test "should report error" do  # some\_undefined\_variable is not defined elsewhere in the test case  assert\_raises(NameError) do  some\_undefined\_variable  end  end |

This test should now pass.

#### **2.4 Available Assertions**

By now you've caught a glimpse of some of the assertions that are available. Assertions are the worker bees of testing. They are the ones that actually perform the checks to ensure that things are going as planned.

Here's an extract of the assertions you can use with [Minitest](https://github.com/seattlerb/minitest), the default testing library used by Rails. The [msg] parameter is an optional string message you can specify to make your test failure messages clearer.

|  |  |
| --- | --- |
| **Assertion** | **Purpose** |
| assert( test, [msg] ) | Ensures that test is true. |
| assert\_not( test, [msg] ) | Ensures that test is false. |
| assert\_equal( expected, actual, [msg] ) | Ensures that expected == actual is true. |
| assert\_not\_equal( expected, actual, [msg] ) | Ensures that expected != actual is true. |
| assert\_same( expected, actual, [msg] ) | Ensures that expected.equal?(actual) is true. |
| assert\_not\_same( expected, actual, [msg] ) | Ensures that expected.equal?(actual) is false. |
| assert\_nil( obj, [msg] ) | Ensures that obj.nil? is true. |
| assert\_not\_nil( obj, [msg] ) | Ensures that obj.nil? is false. |
| assert\_empty( obj, [msg] ) | Ensures that obj is empty?. |
| assert\_not\_empty( obj, [msg] ) | Ensures that obj is not empty?. |
| assert\_match( regexp, string, [msg] ) | Ensures that a string matches the regular expression. |
| assert\_no\_match( regexp, string, [msg] ) | Ensures that a string doesn't match the regular expression. |
| assert\_includes( collection, obj, [msg] ) | Ensures that obj is in collection. |
| assert\_not\_includes( collection, obj, [msg] ) | Ensures that obj is not in collection. |
| assert\_in\_delta( expected, actual, [delta], [msg] ) | Ensures that the numbers expected and actual are within delta of each other. |
| assert\_not\_in\_delta( expected, actual, [delta], [msg] ) | Ensures that the numbers expected and actual are not within delta of each other. |
| assert\_throws( symbol, [msg] ) { block } | Ensures that the given block throws the symbol. |
| assert\_raises( exception1, exception2, ... ) { block } | Ensures that the given block raises one of the given exceptions. |
| assert\_nothing\_raised { block } | Ensures that the given block doesn't raise any exceptions. |
| assert\_instance\_of( class, obj, [msg] ) | Ensures that obj is an instance of class. |
| assert\_not\_instance\_of( class, obj, [msg] ) | Ensures that obj is not an instance of class. |
| assert\_kind\_of( class, obj, [msg] ) | Ensures that obj is an instance of class or is descending from it. |
| assert\_not\_kind\_of( class, obj, [msg] ) | Ensures that obj is not an instance of class and is not descending from it. |
| assert\_respond\_to( obj, symbol, [msg] ) | Ensures that obj responds to symbol. |
| assert\_not\_respond\_to( obj, symbol, [msg] ) | Ensures that obj does not respond to symbol. |
| assert\_operator( obj1, operator, [obj2], [msg] ) | Ensures that obj1.operator(obj2) is true. |
| assert\_not\_operator( obj1, operator, [obj2], [msg] ) | Ensures that obj1.operator(obj2) is false. |
| assert\_predicate ( obj, predicate, [msg] ) | Ensures that obj.predicate is true, e.g. assert\_predicate str, :empty? |
| assert\_not\_predicate ( obj, predicate, [msg] ) | Ensures that obj.predicate is false, e.g. assert\_not\_predicate str, :empty? |
| assert\_send( array, [msg] ) | Ensures that executing the method listed in array[1] on the object in array[0] with the parameters of array[2 and up] is true, e.g. assert\_send [@user, :full\_name, 'Sam Smith']. This one is weird eh? |
| flunk( [msg] ) | Ensures failure. This is useful to explicitly mark a test that isn't finished yet. |

The above are a subset of assertions that minitest supports. For an exhaustive & more up-to-date list, please check [Minitest API documentation](http://docs.seattlerb.org/minitest/), specifically [Minitest::Assertions](http://docs.seattlerb.org/minitest/Minitest/Assertions.html).

Because of the modular nature of the testing framework, it is possible to create your own assertions. In fact, that's exactly what Rails does. It includes some specialized assertions to make your life easier.

Creating your own assertions is an advanced topic that we won't cover in this tutorial.

#### **2.5 Rails Specific Assertions**

Rails adds some custom assertions of its own to the minitest framework:

|  |  |
| --- | --- |
| **Assertion** | **Purpose** |
| [assert\_difference(expressions, difference = 1, message = nil) {...}](http://api.rubyonrails.org/classes/ActiveSupport/Testing/Assertions.html#method-i-assert_difference) | Test numeric difference between the return value of an expression as a result of what is evaluated in the yielded block. |
| [assert\_no\_difference(expressions, message = nil, &block)](http://api.rubyonrails.org/classes/ActiveSupport/Testing/Assertions.html#method-i-assert_no_difference) | Asserts that the numeric result of evaluating an expression is not changed before and after invoking the passed in block. |
| [assert\_recognizes(expected\_options, path, extras={}, message=nil)](http://api.rubyonrails.org/classes/ActionDispatch/Assertions/RoutingAssertions.html#method-i-assert_recognizes) | Asserts that the routing of the given path was handled correctly and that the parsed options (given in the expected\_options hash) match path. Basically, it asserts that Rails recognizes the route given by expected\_options. |
| [assert\_generates(expected\_path, options, defaults={}, extras = {}, message=nil)](http://api.rubyonrails.org/classes/ActionDispatch/Assertions/RoutingAssertions.html#method-i-assert_generates) | Asserts that the provided options can be used to generate the provided path. This is the inverse of assert\_recognizes. The extras parameter is used to tell the request the names and values of additional request parameters that would be in a query string. The message parameter allows you to specify a custom error message for assertion failures. |
| [assert\_response(type, message = nil)](http://api.rubyonrails.org/classes/ActionDispatch/Assertions/ResponseAssertions.html#method-i-assert_response) | Asserts that the response comes with a specific status code. You can specify :success to indicate 200-299, :redirectto indicate 300-399, :missing to indicate 404, or :error to match the 500-599 range. You can also pass an explicit status number or its symbolic equivalent. For more information, see [full list of status codes](http://rubydoc.info/github/rack/rack/master/Rack/Utils#HTTP_STATUS_CODES-constant) and how their [mapping](http://rubydoc.info/github/rack/rack/master/Rack/Utils#SYMBOL_TO_STATUS_CODE-constant) works. |
| [assert\_redirected\_to(options = {}, message=nil)](http://api.rubyonrails.org/classes/ActionDispatch/Assertions/ResponseAssertions.html#method-i-assert_redirected_to) | Asserts that the redirection options passed in match those of the redirect called in the latest action. This match can be partial, such that assert\_redirected\_to(controller: "weblog") will also match the redirection of redirect\_to(controller: "weblog", action: "show") and so on. You can also pass named routes such as assert\_redirected\_to root\_path and Active Record objects such as assert\_redirected\_to @article. |

You'll see the usage of some of these assertions in the next chapter.

#### **2.6 A Brief Note About Test Cases**

All the basic assertions such as assert\_equal defined in Minitest::Assertions are also available in the classes we use in our own test cases. In fact, Rails provides the following classes for you to inherit from:

* [ActiveSupport::TestCase](http://api.rubyonrails.org/classes/ActiveSupport/TestCase.html)
* [ActionMailer::TestCase](http://api.rubyonrails.org/classes/ActionMailer/TestCase.html)
* [ActionView::TestCase](http://api.rubyonrails.org/classes/ActionView/TestCase.html)
* [ActionDispatch::IntegrationTest](http://api.rubyonrails.org/classes/ActionDispatch/IntegrationTest.html)
* [ActiveJob::TestCase](http://api.rubyonrails.org/classes/ActiveJob/TestCase.html)

Each of these classes include Minitest::Assertions, allowing us to use all of the basic assertions in our tests.

For more information on Minitest, refer to [its documentation](http://docs.seattlerb.org/minitest).

#### **2.7 The Rails Test Runner**

We can run all of our tests at once by using the bin/rails test command.

Or we can run a single test by passing the bin/rails test command the filename containing the test cases.

|  |
| --- |
| $ bin/rails test test/models/article\_test.rb  Run options: --seed 1559    # Running:    ..    Finished in 0.027034s, 73.9810 runs/s, 110.9715 assertions/s.    2 runs, 3 assertions, 0 failures, 0 errors, 0 skips |

This will run all test methods from the test case.

You can also run a particular test method from the test case by providing the -n or --name flag and the test's method name.

|  |
| --- |
| $ bin/rails test test/models/article\_test.rb -n test\_the\_truth  Run options: -n test\_the\_truth --seed 43583    # Running:    .    Finished tests in 0.009064s, 110.3266 tests/s, 110.3266 assertions/s.    1 tests, 1 assertions, 0 failures, 0 errors, 0 skips |

You can also run a test at a specific line by providing the line number.

|  |
| --- |
| $ bin/rails test test/models/article\_test.rb:6 # run specific test and line |

You can also run an entire directory of tests by providing the path to the directory.

|  |
| --- |
| $ bin/rails test test/controllers # run all tests from specific directory |

The test runner provides lot of other features too like failing fast, deferring test output at the end of test run and so on. Check the documentation of the test runner as follows:

|  |
| --- |
| $ bin/rails test -h  minitest options:  -h, --help Display this help.  -s, --seed SEED Sets random seed. Also via env. Eg: SEED=n rake  -v, --verbose Verbose. Show progress processing files.  -n, --name PATTERN Filter run on /regexp/ or string.  --exclude PATTERN Exclude /regexp/ or string from run.    Known extensions: rails, pride    Usage: bin/rails test [options] [files or directories]  You can run a single test by appending a line number to a filename:    bin/rails test test/models/user\_test.rb:27    You can run multiple files and directories at the same time:    bin/rails test test/controllers test/integration/login\_test.rb    By default test failures and errors are reported inline during a run.    Rails options:  -e, --environment ENV Run tests in the ENV environment  -b, --backtrace Show the complete backtrace  -d, --defer-output Output test failures and errors after the test run  -f, --fail-fast Abort test run on first failure or error  -c, --[no-]color Enable color in the output |

### **3 The Test Database**

Just about every Rails application interacts heavily with a database and, as a result, your tests will need a database to interact with as well. To write efficient tests, you'll need to understand how to set up this database and populate it with sample data.

By default, every Rails application has three environments: development, test, and production. The database for each one of them is configured in config/database.yml.

A dedicated test database allows you to set up and interact with test data in isolation. This way your tests can mangle test data with confidence, without worrying about the data in the development or production databases.

#### **3.1 Maintaining the test database schema**

In order to run your tests, your test database will need to have the current structure. The test helper checks whether your test database has any pending migrations. It will try to load your db/schema.rb or db/structure.sql into the test database. If migrations are still pending, an error will be raised. Usually this indicates that your schema is not fully migrated. Running the migrations against the development database (bin/rails db:migrate) will bring the schema up to date.

If there were modifications to existing migrations, the test database needs to be rebuilt. This can be done by executing bin/rails db:test:prepare.

#### **3.2 The Low-Down on Fixtures**

For good tests, you'll need to give some thought to setting up test data. In Rails, you can handle this by defining and customizing fixtures. You can find comprehensive documentation in the [Fixtures API documentation](http://api.rubyonrails.org/classes/ActiveRecord/FixtureSet.html).

##### **3.2.1 What Are Fixtures?**

*Fixtures* is a fancy word for sample data. Fixtures allow you to populate your testing database with predefined data before your tests run. Fixtures are database independent and written in YAML. There is one file per model.

Fixtures are not designed to create every object that your tests need, and are best managed when only used for default data that can be applied to the common case.

You'll find fixtures under your test/fixtures directory. When you run rails generate modelto create a new model, Rails automatically creates fixture stubs in this directory.

##### **3.2.2 YAML**

YAML-formatted fixtures are a human-friendly way to describe your sample data. These types of fixtures have the **.yml** file extension (as in users.yml).

Here's a sample YAML fixture file:

|  |
| --- |
| # lo & behold! I am a YAML comment!  david:  name: David Heinemeier Hansson  birthday: 1979-10-15  profession: Systems development    steve:  name: Steve Ross Kellock  birthday: 1974-09-27  profession: guy with keyboard |

Each fixture is given a name followed by an indented list of colon-separated key/value pairs. Records are typically separated by a blank line. You can place comments in a fixture file by using the # character in the first column.

If you are working with [associations](http://guides.rubyonrails.org/association_basics.html), you can simply define a reference node between two different fixtures. Here's an example with a belongs\_to/has\_many association:

|  |
| --- |
| # In fixtures/categories.yml  about:  name: About    # In fixtures/articles.yml  first:  title: Welcome to Rails!  body: Hello world!  category: about |

Notice the category key of the first article found in fixtures/articles.yml has a value of about. This tells Rails to load the category about found in fixtures/categories.yml.

For associations to reference one another by name, you can use the fixture name instead of specifying the id: attribute on the associated fixtures. Rails will auto assign a primary key to be consistent between runs. For more information on this association behavior please read the [Fixtures API documentation](http://api.rubyonrails.org/classes/ActiveRecord/FixtureSet.html).

##### **3.2.3 ERB'in It Up**

ERB allows you to embed Ruby code within templates. The YAML fixture format is pre-processed with ERB when Rails loads fixtures. This allows you to use Ruby to help you generate some sample data. For example, the following code generates a thousand users:

|  |
| --- |
| <% 1000.times do |n| %>  user\_<%= n %>:  username: <%= "user#{n}" %>  email: <%= "user#{n}@example.com" %>  <% end %> |

##### **3.2.4 Fixtures in Action**

Rails automatically loads all fixtures from the test/fixtures directory by default. Loading involves three steps:

1. Remove any existing data from the table corresponding to the fixture
2. Load the fixture data into the table
3. Dump the fixture data into a method in case you want to access it directly

In order to remove existing data from the database, Rails tries to disable referential integrity triggers (like foreign keys and check constraints). If you are getting annoying permission errors on running tests, make sure the database user has privilege to disable these triggers in testing environment. (In PostgreSQL, only superusers can disable all triggers. Read more about PostgreSQL permissions [here](http://blog.endpoint.com/2012/10/postgres-system-triggers-error.html)).

##### **3.2.5 Fixtures are Active Record objects**

Fixtures are instances of Active Record. As mentioned in point #3 above, you can access the object directly because it is automatically available as a method whose scope is local of the test case. For example:

|  |
| --- |
| # this will return the User object for the fixture named david  users(:david)    # this will return the property for david called id  users(:david).id    # one can also access methods available on the User class  david = users(:david)  david.call(david.partner) |

To get multiple fixtures at once, you can pass in a list of fixture names. For example:

|  |
| --- |
| # this will return an array containing the fixtures david and steve  users(:david, :steve) |

### **4 Model Testing**

Model tests are used to test the various models of your application.

Rails model tests are stored under the test/models directory. Rails provides a generator to create a model test skeleton for you.

|  |
| --- |
| $ bin/rails generate test\_unit:model article title:string body:text  create test/models/article\_test.rb  create test/fixtures/articles.yml |

Model tests don't have their own superclass like ActionMailer::TestCase instead they inherit from [ActiveSupport::TestCase](http://api.rubyonrails.org/classes/ActiveSupport/TestCase.html).

### **5 Integration Testing**

Integration tests are used to test how various parts of your application interact. They are generally used to test important workflows within our application.

For creating Rails integration tests, we use the test/integration directory for our application. Rails provides a generator to create an integration test skeleton for us.

|  |
| --- |
| $ bin/rails generate integration\_test user\_flows  exists test/integration/  create test/integration/user\_flows\_test.rb |

Here's what a freshly-generated integration test looks like:

|  |
| --- |
| require 'test\_helper'    class UserFlowsTest < ActionDispatch::IntegrationTest  # test "the truth" do  # assert true  # end  end |

Here the test is inheriting from ActionDispatch::IntegrationTest. This makes some additional helpers available for us to use in our integration tests.

#### **5.1 Helpers Available for Integration Tests**

In addition to the standard testing helpers, inheriting from ActionDispatch::IntegrationTestcomes with some additional helpers available when writing integration tests. Let's get briefly introduced to the three categories of helpers we get to choose from.

For dealing with the integration test runner, see [ActionDispatch::Integration::Runner](http://api.rubyonrails.org/classes/ActionDispatch/Integration/Runner.html).

When performing requests, we will have [ActionDispatch::Integration::RequestHelpers](http://api.rubyonrails.org/classes/ActionDispatch/Integration/RequestHelpers.html)available for our use.

If we need to modify the session, or state of our integration test, take a look at [ActionDispatch::Integration::Session](http://api.rubyonrails.org/classes/ActionDispatch/Integration/Session.html) to help.

#### **5.2 Implementing an integration test**

Let's add an integration test to our blog application. We'll start with a basic workflow of creating a new blog article, to verify that everything is working properly.

We'll start by generating our integration test skeleton:

|  |
| --- |
| $ bin/rails generate integration\_test blog\_flow |

It should have created a test file placeholder for us. With the output of the previous command we should see:

|  |
| --- |
| invoke test\_unit  create test/integration/blog\_flow\_test.rb |

Now let's open that file and write our first assertion:

|  |
| --- |
| require 'test\_helper'    class BlogFlowTest < ActionDispatch::IntegrationTest  test "can see the welcome page" do  get "/"  assert\_select "h1", "Welcome#index"  end  end |

We will take a look at assert\_select to query the resulting HTML of a request in the "Testing Views" section below. It is used for testing the response of our request by asserting the presence of key HTML elements and their content.

When we visit our root path, we should see welcome/index.html.erb rendered for the view. So this assertion should pass.

##### **5.2.1 Creating articles integration**

How about testing our ability to create a new article in our blog and see the resulting article.

|  |
| --- |
| test "can create an article" do  get "/articles/new"  assert\_response :success    post "/articles",  params: { article: { title: "can create", body: "article successfully." } }  assert\_response :redirect  follow\_redirect!  assert\_response :success  assert\_select "p", "Title:\n can create"  end |

Let's break this test down so we can understand it.

We start by calling the :new action on our Articles controller. This response should be successful.

After this we make a post request to the :create action of our Articles controller:

|  |
| --- |
| post "/articles",  params: { article: { title: "can create", body: "article successfully." } }  assert\_response :redirect  follow\_redirect! |

The two lines following the request are to handle the redirect we setup when creating a new article.

Don't forget to call follow\_redirect! if you plan to make subsequent requests after a redirect is made.

Finally we can assert that our response was successful and our new article is readable on the page.

##### **5.2.2 Taking it further**

We were able to successfully test a very small workflow for visiting our blog and creating a new article. If we wanted to take this further we could add tests for commenting, removing articles, or editing comments. Integration tests are a great place to experiment with all kinds of use-cases for our applications.

### **6 Functional Tests for Your Controllers**

In Rails, testing the various actions of a controller is a form of writing functional tests. Remember your controllers handle the incoming web requests to your application and eventually respond with a rendered view. When writing functional tests, you are testing how your actions handle the requests and the expected result or response, in some cases an HTML view.

#### **6.1 What to include in your Functional Tests**

You should test for things such as:

* was the web request successful?
* was the user redirected to the right page?
* was the user successfully authenticated?
* was the correct object stored in the response template?
* was the appropriate message displayed to the user in the view?

The easiest way to see functional tests in action is to generate a controller using the scaffold generator:

|  |
| --- |
| $ bin/rails generate scaffold\_controller article title:string body:text  ...  create app/controllers/articles\_controller.rb  ...  invoke test\_unit  create test/controllers/articles\_controller\_test.rb  ... |

This will generate the controller code and tests for an Article resource. You can take a look at the file articles\_controller\_test.rb in the test/controllers directory.

If you already have a controller and just want to generate the test scaffold code for each of the seven default actions, you can use the following command:

|  |
| --- |
| $ bin/rails generate test\_unit:scaffold article  ...  invoke test\_unit  create test/controllers/articles\_controller\_test.rb  ... |

Let's take a look at one such test, test\_should\_get\_index from the file articles\_controller\_test.rb.

|  |
| --- |
| # articles\_controller\_test.rb  class ArticlesControllerTest < ActionDispatch::IntegrationTest  test "should get index" do  get articles\_url  assert\_response :success  end  end |

In the test\_should\_get\_index test, Rails simulates a request on the action called index, making sure the request was successful and also ensuring that the right response body has been generated.

The get method kicks off the web request and populates the results into the @response. It can accept up to 6 arguments:

* The action of the controller you are requesting. This can be in the form of a string or a route (i.e. articles\_url).
* params: option with a hash of request parameters to pass into the action (e.g. query string parameters or article variables).
* headers: for setting the headers that will be passed with the request.
* env: for customizing the request environment as needed.
* xhr: whether the request is Ajax request or not. Can be set to true for marking the request as Ajax.
* as: for encoding the request with different content type. Supports :json by default.

All of these keyword arguments are optional.

Example: Calling the :show action, passing an id of 12 as the params and setting HTTP\_REFERERheader:

|  |
| --- |
| get :show, params: { id: 12 }, headers: { "HTTP\_REFERER" => "http://example.com/home" } |

Another example: Calling the :update action, passing an id of 12 as the params as an Ajax request.

|  |
| --- |
| patch update\_url, params: { id: 12 }, xhr: true |

If you try running test\_should\_create\_article test from articles\_controller\_test.rb it will fail on account of the newly added model level validation and rightly so.

Let us modify test\_should\_create\_article test in articles\_controller\_test.rb so that all our test pass:

|  |
| --- |
| test "should create article" do  assert\_difference('Article.count') do  post articles\_url, params: { article: { body: 'Rails is awesome!', title: 'Hello Rails' } }  end    assert\_redirected\_to article\_path(Article.last)  end |

Now you can try running all the tests and they should pass.

#### **6.2 Available Request Types for Functional Tests**

If you're familiar with the HTTP protocol, you'll know that get is a type of request. There are 6 request types supported in Rails functional tests:

* get
* post
* patch
* put
* head
* delete

All of request types have equivalent methods that you can use. In a typical C.R.U.D. application you'll be using get, post, put and delete more often.

Functional tests do not verify whether the specified request type is accepted by the action, we're more concerned with the result. Request tests exist for this use case to make your tests more purposeful.

#### **6.3 Testing XHR (AJAX) requests**

To test AJAX requests, you can specify the xhr: true option to get, post, patch, put, and delete methods. For example:

|  |
| --- |
| test "ajax request" do  article = articles(:one)  get article\_url(article), xhr: true    assert\_equal 'hello world', @response.body  assert\_equal "text/javascript", @response.content\_type  end |

#### **6.4 The Three Hashes of the Apocalypse**

After a request has been made and processed, you will have 3 Hash objects ready for use:

* cookies - Any cookies that are set
* flash - Any objects living in the flash
* session - Any object living in session variables

As is the case with normal Hash objects, you can access the values by referencing the keys by string. You can also reference them by symbol name. For example:

|  |
| --- |
| flash["gordon"] flash[:gordon]  session["shmession"] session[:shmession]  cookies["are\_good\_for\_u"] cookies[:are\_good\_for\_u] |

#### **6.5 Instance Variables Available**

You also have access to three instance variables in your functional tests:

* @controller - The controller processing the request
* @request - The request object
* @response - The response object

#### **6.6 Setting Headers and CGI variables**

[HTTP headers](http://tools.ietf.org/search/rfc2616#section-5.3) and [CGI variables](http://tools.ietf.org/search/rfc3875#section-4.1) can be passed as headers:

|  |
| --- |
| # setting an HTTP Header  get articles\_url, headers: "Content-Type" => "text/plain" # simulate the request with custom header    # setting a CGI variable  get articles\_url, headers: "HTTP\_REFERER" => "http://example.com/home" # simulate the request with custom env variable |

#### **6.7 Testing flash notices**

If you remember from earlier, one of the Three Hashes of the Apocalypse was flash.

We want to add a flash message to our blog application whenever someone successfully creates a new Article.

Let's start by adding this assertion to our test\_should\_create\_article test:

|  |
| --- |
| test "should create article" do  assert\_difference('Article.count') do  post article\_url, params: { article: { title: 'Some title' } }  end    assert\_redirected\_to article\_path(Article.last)  assert\_equal 'Article was successfully created.', flash[:notice]  end |

If we run our test now, we should see a failure:

|  |
| --- |
| $ bin/rails test test/controllers/articles\_controller\_test.rb -n test\_should\_create\_article  Run options: -n test\_should\_create\_article --seed 32266    # Running:    F    Finished in 0.114870s, 8.7055 runs/s, 34.8220 assertions/s.    1) Failure:  ArticlesControllerTest#test\_should\_create\_article [/test/controllers/articles\_controller\_test.rb:16]:  --- expected  +++ actual  @@ -1 +1 @@  -"Article was successfully created."  +nil    1 runs, 4 assertions, 1 failures, 0 errors, 0 skips |

Let's implement the flash message now in our controller. Our :create action should now look like this:

|  |
| --- |
| def create  @article = Article.new(article\_params)    if @article.save  flash[:notice] = 'Article was successfully created.'  redirect\_to @article  else  render 'new'  end  end |

Now if we run our tests, we should see it pass:

|  |
| --- |
| $ bin/rails test test/controllers/articles\_controller\_test.rb -n test\_should\_create\_article  Run options: -n test\_should\_create\_article --seed 18981    # Running:    .    Finished in 0.081972s, 12.1993 runs/s, 48.7972 assertions/s.    1 runs, 4 assertions, 0 failures, 0 errors, 0 skips |

#### **6.8 Putting it together**

At this point our Articles controller tests the :index as well as :new and :create actions. What about dealing with existing data?

Let's write a test for the :show action:

|  |
| --- |
| test "should show article" do  article = articles(:one)  get article\_url(article)  assert\_response :success  end |

Remember from our discussion earlier on fixtures, the articles() method will give us access to our Articles fixtures.

How about deleting an existing Article?

|  |
| --- |
| test "should destroy article" do  article = articles(:one)  assert\_difference('Article.count', -1) do  delete article\_url(article)  end    assert\_redirected\_to articles\_path  end |

We can also add a test for updating an existing Article.

|  |
| --- |
| test "should update article" do  article = articles(:one)    patch article\_url(article), params: { article: { title: "updated" } }    assert\_redirected\_to article\_path(article)  # Reload association to fetch updated data and assert that title is updated.  article.reload  assert\_equal "updated", article.title  end |

Notice we're starting to see some duplication in these three tests, they both access the same Article fixture data. We can D.R.Y. this up by using the setup and teardown methods provided by ActiveSupport::Callbacks.

Our test should now look something as what follows. Disregard the other tests for now, we're leaving them out for brevity.

|  |
| --- |
| require 'test\_helper'    class ArticlesControllerTest < ActionDispatch::IntegrationTest  # called before every single test  setup do  @article = articles(:one)  end    # called after every single test  teardown do  # when controller is using cache it may be a good idea to reset it afterwards  Rails.cache.clear  end    test "should show article" do  # Reuse the @article instance variable from setup  get article\_url(@article)  assert\_response :success  end    test "should destroy article" do  assert\_difference('Article.count', -1) do  delete article\_url(@article)  end    assert\_redirected\_to articles\_path  end    test "should update article" do  patch article\_url(@article), params: { article: { title: "updated" } }    assert\_redirected\_to article\_path(@article)  # Reload association to fetch updated data and assert that title is updated.  @article.reload  assert\_equal "updated", @article.title  end  end |

Similar to other callbacks in Rails, the setup and teardown methods can also be used by passing a block, lambda, or method name as a symbol to call.

#### **6.9 Test helpers**

To avoid code duplication, you can add your own test helpers. Sign in helper can be a good example:

|  |
| --- |
| #test/test\_helper.rb    module SignInHelper  def sign\_in\_as(user)  post sign\_in\_url(email: user.email, password: user.password)  end  end    class ActionDispatch::IntegrationTest  include SignInHelper  end |

|  |
| --- |
| require 'test\_helper'    class ProfileControllerTest < ActionDispatch::IntegrationTest    test "should show profile" do  # helper is now reusable from any controller test case  sign\_in\_as users(:david)    get profile\_url  assert\_response :success  end  end |

### **7 Testing Routes**

Like everything else in your Rails application, you can test your routes.

If your application has complex routes, Rails provides a number of useful helpers to test them.

For more information on routing assertions available in Rails, see the API documentation for [ActionDispatch::Assertions::RoutingAssertions](http://api.rubyonrails.org/classes/ActionDispatch/Assertions/RoutingAssertions.html).

### **8 Testing Views**

Testing the response to your request by asserting the presence of key HTML elements and their content is a common way to test the views of your application. Like route tests, view tests reside in test/controllers/ or are part of controller tests. The assert\_select method allows you to query HTML elements of the response by using a simple yet powerful syntax.

There are two forms of assert\_select:

assert\_select(selector, [equality], [message]) ensures that the equality condition is met on the selected elements through the selector. The selector may be a CSS selector expression (String) or an expression with substitution values.

assert\_select(element, selector, [equality], [message]) ensures that the equality condition is met on all the selected elements through the selector starting from the *element* (instance of Nokogiri::XML::Node or Nokogiri::XML::NodeSet) and its descendants.

For example, you could verify the contents on the title element in your response with:

|  |
| --- |
| assert\_select 'title', "Welcome to Rails Testing Guide" |

You can also use nested assert\_select blocks for deeper investigation.

In the following example, the inner assert\_select for li.menu\_item runs within the collection of elements selected by the outer block:

|  |
| --- |
| assert\_select 'ul.navigation' do  assert\_select 'li.menu\_item'  end |

A collection of selected elements may be iterated through so that assert\_select may be called separately for each element.

For example if the response contains two ordered lists, each with four nested list elements then the following tests will both pass.

|  |
| --- |
| assert\_select "ol" do |elements|  elements.each do |element|  assert\_select element, "li", 4  end  end    assert\_select "ol" do  assert\_select "li", 8  end |

This assertion is quite powerful. For more advanced usage, refer to its [documentation](https://github.com/rails/rails-dom-testing/blob/master/lib/rails/dom/testing/assertions/selector_assertions.rb).

##### **8.1 Additional View-Based Assertions**

There are more assertions that are primarily used in testing views:

|  |  |
| --- | --- |
| **Assertion** | **Purpose** |
| assert\_select\_email | Allows you to make assertions on the body of an e-mail. |
| assert\_select\_encoded | Allows you to make assertions on encoded HTML. It does this by un-encoding the contents of each element and then calling the block with all the un-encoded elements. |
| css\_select(selector)or css\_select(element, selector) | Returns an array of all the elements selected by the *selector*. In the second variant it first matches the base *element* and tries to match the *selector* expression on any of its children. If there are no matches both variants return an empty array. |

Here's an example of using assert\_select\_email:

|  |
| --- |
| assert\_select\_email do  assert\_select 'small', 'Please click the "Unsubscribe" link if you want to opt-out.'  end |

### **9 Testing Helpers**

A helper is just a simple module where you can define methods which are available into your views.

In order to test helpers, all you need to do is check that the output of the helper method matches what you'd expect. Tests related to the helpers are located under the test/helpers directory.

Given we have the following helper:

|  |
| --- |
| module UserHelper  def link\_to\_user(user)  link\_to "#{user.first\_name} #{user.last\_name}", user  end  end |

We can test the output of this method like this:

|  |
| --- |
| class UserHelperTest < ActionView::TestCase  test "should return the user's full name" do  user = users(:david)    assert\_dom\_equal %{<a href="/user/#{user.id}">David Heinemeier Hansson</a>}, link\_to\_user(user)  end  end |

Moreover, since the test class extends from ActionView::TestCase, you have access to Rails' helper methods such as link\_to or pluralize.

### **10 Testing Your Mailers**

Testing mailer classes requires some specific tools to do a thorough job.

#### **10.1 Keeping the Postman in Check**

Your mailer classes - like every other part of your Rails application - should be tested to ensure that they are working as expected.

The goals of testing your mailer classes are to ensure that:

* emails are being processed (created and sent)
* the email content is correct (subject, sender, body, etc)
* the right emails are being sent at the right times

##### **10.1.1 From All Sides**

There are two aspects of testing your mailer, the unit tests and the functional tests. In the unit tests, you run the mailer in isolation with tightly controlled inputs and compare the output to a known value (a fixture.) In the functional tests you don't so much test the minute details produced by the mailer; instead, we test that our controllers and models are using the mailer in the right way. You test to prove that the right email was sent at the right time.

#### **10.2 Unit Testing**

In order to test that your mailer is working as expected, you can use unit tests to compare the actual results of the mailer with pre-written examples of what should be produced.

##### **10.2.1 Revenge of the Fixtures**

For the purposes of unit testing a mailer, fixtures are used to provide an example of how the output *should* look. Because these are example emails, and not Active Record data like the other fixtures, they are kept in their own subdirectory apart from the other fixtures. The name of the directory within test/fixtures directly corresponds to the name of the mailer. So, for a mailer named UserMailer, the fixtures should reside in test/fixtures/user\_mailer directory.

When you generated your mailer, the generator creates stub fixtures for each of the mailers actions. If you didn't use the generator, you'll have to create those files yourself.

##### **10.2.2 The Basic Test Case**

Here's a unit test to test a mailer named UserMailer whose action invite is used to send an invitation to a friend. It is an adapted version of the base test created by the generator for an inviteaction.

|  |
| --- |
| require 'test\_helper'    class UserMailerTest < ActionMailer::TestCase  test "invite" do  # Create the email and store it for further assertions  email = UserMailer.create\_invite('me@example.com',  'friend@example.com', Time.now)    # Send the email, then test that it got queued  assert\_emails 1 do  email.deliver\_now  end    # Test the body of the sent email contains what we expect it to  assert\_equal ['me@example.com'], email.from  assert\_equal ['friend@example.com'], email.to  assert\_equal 'You have been invited by me@example.com', email.subject  assert\_equal read\_fixture('invite').join, email.body.to\_s  end  end |

In the test we send the email and store the returned object in the email variable. We then ensure that it was sent (the first assert), then, in the second batch of assertions, we ensure that the email does indeed contain what we expect. The helper read\_fixture is used to read in the content from this file.

Here's the content of the invite fixture:

|  |
| --- |
| Hi friend@example.com,    You have been invited.    Cheers! |

This is the right time to understand a little more about writing tests for your mailers. The line ActionMailer::Base.delivery\_method = :test in config/environments/test.rb sets the delivery method to test mode so that email will not actually be delivered (useful to avoid spamming your users while testing) but instead it will be appended to an array (ActionMailer::Base.deliveries).

The ActionMailer::Base.deliveries array is only reset automatically in ActionMailer::TestCase and ActionDispatch::IntegrationTest tests. If you want to have a clean slate outside these test cases, you can reset it manually with: ActionMailer::Base.deliveries.clear

#### **10.3 Functional Testing**

Functional testing for mailers involves more than just checking that the email body, recipients and so forth are correct. In functional mail tests you call the mail deliver methods and check that the appropriate emails have been appended to the delivery list. It is fairly safe to assume that the deliver methods themselves do their job. You are probably more interested in whether your own business logic is sending emails when you expect them to go out. For example, you can check that the invite friend operation is sending an email appropriately:

|  |
| --- |
| require 'test\_helper'    class UserControllerTest < ActionDispatch::IntegrationTest  test "invite friend" do  assert\_difference 'ActionMailer::Base.deliveries.size', +1 do  post invite\_friend\_url, params: { email: 'friend@example.com' }  end  invite\_email = ActionMailer::Base.deliveries.last    assert\_equal "You have been invited by me@example.com", invite\_email.subject  assert\_equal 'friend@example.com', invite\_email.to[0]  assert\_match(/Hi friend@example.com/, invite\_email.body.to\_s)  end  end |

### **11 Testing Jobs**

Since your custom jobs can be queued at different levels inside your application, you'll need to test both, the jobs themselves (their behavior when they get enqueued) and that other entities correctly enqueue them.

#### **11.1 A Basic Test Case**

By default, when you generate a job, an associated test will be generated as well under the test/jobs directory. Here's an example test with a billing job:

|  |
| --- |
| require 'test\_helper'    class BillingJobTest < ActiveJob::TestCase  test 'that account is charged' do  BillingJob.perform\_now(account, product)  assert account.reload.charged\_for?(product)  end  end |

This test is pretty simple and only asserts that the job get the work done as expected.

By default, ActiveJob::TestCase will set the queue adapter to :async so that your jobs are performed in an async fashion. It will also ensure that all previously performed and enqueued jobs are cleared before any test run so you can safely assume that no jobs have already been executed in the scope of each test.

#### **11.2 Custom Assertions And Testing Jobs Inside Other Components**

Active Job ships with a bunch of custom assertions that can be used to lessen the verbosity of tests. For a full list of available assertions, see the API documentation for [ActiveJob::TestHelper](http://api.rubyonrails.org/classes/ActiveJob/TestHelper.html).

It's a good practice to ensure that your jobs correctly get enqueued or performed wherever you invoke them (e.g. inside your controllers). This is precisely where the custom assertions provided by Active Job are pretty useful. For instance, within a model:

|  |
| --- |
| require 'test\_helper'    class ProductTest < ActiveJob::TestCase  test 'billing job scheduling' do  assert\_enqueued\_with(job: BillingJob) do  product.charge(account)  end  end  end |

### **12 Additional Testing Resources**

#### **12.1 Testing Time-Dependent Code**

Rails provides built-in helper methods that enable you to assert that your time-sensitive code works as expected.

Here is an example using the [travel\_to](http://api.rubyonrails.org/classes/ActiveSupport/Testing/TimeHelpers.html#method-i-travel_to) helper:

|  |
| --- |
| # Lets say that a user is eligible for gifting a month after they register.  user = User.create(name: 'Gaurish', activation\_date: Date.new(2004, 10, 24))  assert\_not user.applicable\_for\_gifting?  travel\_to Date.new(2004, 11, 24) do  assert\_equal Date.new(2004, 10, 24), user.activation\_date # inside the travel\_to block `Date.current` is mocked  assert user.applicable\_for\_gifting?  end  assert\_equal Date.new(2004, 10, 24), user.activation\_date # The change was visible only inside the `travel\_to` block. |