A Guide to Testing Rails Applications

January 13, 2015

This guide covers built-in mechanisms in Rails for testing your application.
After reading this guide, you will know:

- Rails testing terminology.
- How to write unit, functional, and integration tests for your application.
- Other popular testing approaches and plugins.

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. 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.

2.1 The Test Environment

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. Tests can mangle test data with confidence, that won't touch the data in the development or production databases.

2.2 Rails Sets up for Testing from the Word Go

Rails creates a `test` folder for you as soon as you create a Rails project using `rails new` *application_name*. If you list the contents of this folder 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.

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

The `test_helper.rb` file holds the default configuration for your tests.

2.3 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 fixture api documentation.

2.3.1 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 written in YAML. There is one file per model.

You'll find fixtures under your `test/fixtures` directory. When you run `rails generate model` to create a new model fixture stubs will be automatically created and placed in this directory.

2.3.2 YAML YAML-formatted fixtures are a very 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 space. You can place comments in a fixture file by using the # character in the first column. Keys which resemble YAML keywords such as 'yes' and 'no' are quoted so that the YAML Parser correctly interprets them.

If you are working with associations, 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
one:
  title: Welcome to Rails!
  body: Hello world!
  category: about
```

Note: For associations to reference one another by name, you cannot specify the `id:` attribute on the fixtures. Rails will auto assign a primary key to be consistent between runs. If you manually specify an `id:` attribute, this behavior will not work. For more information on this association behavior please read the fixture api documentation.

2.3.3 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 %>
```

2.3.4 Fixtures in Action Rails by default automatically loads all fixtures from the `test/fixtures` folder for your models and controllers test. Loading involves three steps:

- Remove any existing data from the table corresponding to the fixture
- Load the fixture data into the table
- Dump the fixture data into a variable in case you want to access it directly

2.3.5 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 setup as a local variable 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
email(david.girlfriend.email, david.location_tonight)
```

3 Unit Testing your Models

In Rails, models tests are what you write to test your models.

For this guide we will be using Rails *scaffolding*. It will create the model, a migration, controller and views for the new resource in a single operation. It will also create a full test suite following Rails best practices. We will be using examples from this generated code and will be supplementing it with additional examples where necessary.

For more information on Rails *scaffolding*, refer to Getting Started with Rails

When you use `rails generate scaffold`, for a resource among other things it creates a test stub in the `test/models` folder:

```
$ bin/rails generate scaffold 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'
```

As you know by now, `test_helper.rb` specifies the default configuration to run our tests. This is included with all the tests, so any methods added to this file are available to all your 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`. You'll see those methods a little later in this guide.

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, `test_password` and `test_valid_password` are legal test names and are run automatically when the test case is run.

Rails 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,

```
test "the truth" do
  assert true
end
```

acts as if you had written

```
def test_the_truth
  assert true
end
```

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. Odd ones need `define_method` and `send` calls, but formally there's no restriction.

```
assert true
```

This line of code is called an *assertion*. 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 contains one or more assertions. Only when all the assertions are successful will the test pass.

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. If so, 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.

3.2 Running Tests

Running a test is as simple as invoking the file containing the test cases through `rake test` command.

```
$ bin/rake test test/models/article_test.rb
.

Finished tests in 0.009262s, 107.9680 tests/s, 107.9680 assertions/s.

1 tests, 1 assertions, 0 failures, 0 errors, 0 skips
```

You can also run a particular test method from the test case by running the test and providing the `test method name`.

```
$ bin/rake test test/models/article_test.rb test_the_truth
.

Finished tests in 0.009064s, 110.3266 tests/s, 110.3266 assertions/s.

1 tests, 1 assertions, 0 failures, 0 errors, 0 skips
```

This will run all test methods from the test case. Note that `test_helper.rb` is in the `test` directory, hence this directory needs to be added to the load path using the `-I` switch.

The `.` (dot) above indicates a passing test. When a test fails you see an `F`; when a test throws an error you see an `E` in its place. The last line of the output is the summary.

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.

```
$ bin/rake test test/models/article_test.rb test_should_not_save_article_without_title
F

Finished tests in 0.044632s, 22.4054 tests/s, 22.4054 assertions/s.

  1) Failure:
test_should_not_save_article_without_title(ArticleTest) [test/models/article_test.rb:6]:
Failed assertion, no message given.

1 tests, 1 assertions, 1 failures, 0 errors, 0 skips
```

In the output, `F` denotes a failure. You can see the corresponding trace shown under `1)` along with the name of the failing test. The next few lines contain the stack trace followed by a message which 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:

```
  1) Failure:
test_should_not_save_article_without_title(ArticleTest) [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 < ActiveRecord::Base
  validates :title, presence: true
end
```

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

```
$ bin/rake test test/models/article_test.rb test_should_not_save_article_without_title
.

Finished tests in 0.047721s, 20.9551 tests/s, 20.9551 assertions/s.

1 tests, 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).

Many Rails developers practice *Test-Driven Development* (TDD). This is an excellent way to build up a test suite that exercises every part of your application. TDD is beyond the scope of this guide, but one place to start is with 15 TDD steps to create a Rails application.

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/rake test test/models/article_test.rb test_should_report_error
E

Finished tests in 0.030974s, 32.2851 tests/s, 0.0000 assertions/s.

  1) Error:
test_should_report_error(ArticleTest):
NameError: undefined local variable or method ‘some_undefined_variable’ for #<
ArticleTest:0x007fe32e24afe0>
    test/models/article_test.rb:10:in ‘block in <class:ArticleTest>’

1 tests, 0 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 alphabetical 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 `BACKTRACE` environment variable to enable this behavior:

```
$ BACKTRACE=1 bin/rake test test/models/article_test.rb
```

3.3 What to Include in Your Unit Tests

Ideally, you would like to include a test for everything which could possibly break. It's a good practice to have at least one test for each of your validations and at least one test for every method in your model.

3.4 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.

There are a bunch of different types of assertions you can use. Here's an extract of the assertions you can use with `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. It's not required.

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`.

Assertion	Purpose
`assert_not_includes( collection, obj, [msg] )`	Ensures that `obj` is not in `collection`.
`assert_in_delta( expecting, actual, [delta], [msg] )`	Ensures that the numbers `expected` and `actual` are within `delta` of each other.
`assert_not_in_delta( expecting, 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( exception1, exception2, ... ) { block }`	Ensures that the given block doesn't raise one of the given 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 or descends from `class`.
`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?`

Assertion	Purpose
`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. 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 subset of assertions that minitest supports. For an exhaustive & more up-to-date list, please check Minitest API documentation, specifically `Minitest::Assertions`

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.

3.5 Rails Specific Assertions

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

Assertion	Purpose
`assert_difference(expressions, difference = 1, message = nil) {...}`	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, &amp;block)`	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)`	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)`	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.

Assertion	Purpose
`assert_response(type, message = nil)`	Asserts that the response comes with a specific status code. You can specify `:success` to indicate 200-299, `:redirect` to 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 and how their mapping works.
`assert_redirected _to(options = {}, message=nil)`	Assert 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`.
`assert_template(expected = nil, message=nil)`	Asserts that the request was rendered with the appropriate template file.

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

4 Functional Tests for Your Controllers

In Rails, testing the various actions of a single controller is called writing functional tests for that controller. Controllers handle the incoming web requests to your application and eventually respond with a rendered view.

4.1 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?

Now that we have used Rails scaffold generator for our `Article` resource, it has already created the controller code and tests. You can take look at the file `articles_controller_test.rb` in the `test/controllers` directory.

Let me take you through one such test, `test_should_get_index` from the file `articles_controller_test .rb`.

```
class ArticlesControllerTest < ActionController::TestCase
  test "should get index" do
    get :index
    assert_response :success
    assert_not_nil assigns(:articles)
  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 it assigns a valid `articles` instance variable.

The `get` method kicks off the web request and populates the results into the response. It accepts 4 arguments:

- The action of the controller you are requesting. This can be in the form of a string or a symbol.
- An optional hash of request parameters to pass into the action (eg. query string parameters or article variables).
- An optional hash of session variables to pass along with the request.
- An optional hash of flash values.

Example: Calling the `:show` action, passing an `id` of 12 as the `params` and setting a `user_id` of 5 in the session:

```
get(:show, {'id' => "12"}, {'user_id' => 5})
```

Another example: Calling the `:view` action, passing an `id` of 12 as the `params`, this time with no session, but with a flash message.

```
get(:view, {'id' => '12'}, nil, {'message' => 'booya!'})
```

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 :create, article: {title: 'Some title'}
  end

  assert_redirected_to article_path(assigns(:article))
end
```

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

4.2 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 are methods that you can use, however, you'll probably end up using the first two more often than the others.

Functional tests do not verify whether the specified request type should be accepted by the action. Request types in this context exist to make your tests more descriptive.

4.3 The Four Hashes of the Apocalypse

After a request has been made using one of the 6 methods (`get`, `post`, etc.) and processed, you will have 4 Hash objects ready for use:

- `assigns` - Any objects that are stored as instance variables in actions for use in views.
- `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, except for `assigns`. For example:

```
flash["gordon"]               flash[:gordon]
session["shmession"]          session[:shmession]
cookies["are_good_for_u"]     cookies[:are_good_for_u]

# Because you can't use assigns[:something] for historical reasons:
assigns["something"]          assigns(:something)
```

4.4 Instance Variables Available

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

- `@controller` - The controller processing the request
- `@request` - The request
- `@response` - The response

4.5 Setting Headers and CGI variables

HTTP headers and CGI variables can be set directly on the `@request` instance variable:

```
# setting a HTTP Header
@request.headers["Accept"] = "text/plain, text/html"
get :index # simulate the request with custom header

# setting a CGI variable
@request.headers["HTTP_REFERER"] = "http://example.com/home"
post :create # simulate the request with custom env variable
```

4.6 Testing Templates and Layouts

If you want to make sure that the response rendered the correct template and layout, you can use the `assert_template` method:

```
test "index should render correct template and layout" do
  get :index
  assert_template :index
  assert_template layout: "layouts/application"
end
```

Note that you cannot test for template and layout at the same time, with one call to `assert_template` method. Also, for the `layout` test, you can give a regular expression instead of a string, but using the string, makes things clearer. On the other hand, you have to include the “layouts” directory name even if you save your layout file in this standard layout directory. Hence,

```
assert_template layout: "application"
```

will not work.

If your view renders any partial, when asserting for the layout, you have to assert for the partial at the same time. Otherwise, assertion will fail.

Hence:

```
test "new should render correct layout" do
  get :new
  assert_template layout: "layouts/application", partial: "_form"
end
```

is the correct way to assert for the layout when the view renders a partial with name `_form`. Omitting the `:partial` key in your `assert_template` call will complain.

4.7 A Fuller Functional Test Example

Here’s another example that uses `flash`, `assert_redirected_to`, and `assert_difference`:

```
test "should create article" do
  assert_difference('Article.count') do
    post :create, article: {title: 'Hi', body: 'This is my first article.'}
  end
  assert_redirected_to article_path(assigns(:article))
  assert_equal 'Article was successfully created.', flash[:notice]
end
```

4.8 Testing Views

Testing the response to your request by asserting the presence of key HTML elements and their content is a useful way to test the views of your application. The `assert_select` assertion allows you to do this by using a simple yet powerful syntax.

You may find references to `assert_tag` in other documentation. This has been removed in 4.2. Use `assert_select` instead.

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. In this case the inner `assert_select` runs the assertion on the complete collection of elements selected by the outer `assert_select` block:

```
assert_select 'ul.navigation' do
  assert_select 'li.menu_item'
end
```

Alternatively the collection of elements selected by the outer `assert_select` may be iterated through so that `assert_select` may be called separately for each element. Suppose for example that the response contains two ordered lists, each with four 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
```

The `assert_select` assertion is quite powerful. For more advanced usage, refer to its documentation.

4.8.1 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
```

5 Integration Testing

Integration tests are used to test the interaction among any number of controllers. They are generally used to test important work flows within your application.

Unlike Unit and Functional tests, integration tests have to be explicitly created under the 'test/integration' folder within your application. Rails provides a generator to create an integration test skeleton for you.

```
$ 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
```

Integration tests inherit from `ActionDispatch::IntegrationTest`. This makes available some additional helpers to use in your integration tests. Also you need to explicitly include the fixtures to be made available to the test.

5.1 Helpers Available for Integration Tests

In addition to the standard testing helpers, there are some additional helpers available to integration tests:

Helper	Purpose
`https?`	Returns `true` if the session is mimicking a secure HTTPS request.
`https!`	Allows you to mimic a secure HTTPS request.
`host!`	Allows you to set the host name to use in the next request.
`redirect?`	Returns `true` if the last request was a redirect.
`follow_redirect!`	Follows a single redirect response.
`request_via_redirect(http_method, path, [parameters], [headers])`	Allows you to make an HTTP request and follow any subsequent redirects.
`post_via_redirect(path, [parameters], [headers])`	Allows you to make an HTTP POST request and follow any subsequent redirects.
`get_via_redirect(path, [parameters], [headers])`	Allows you to make an HTTP GET request and follow any subsequent redirects.
`patch_via_redirect(path, [parameters], [headers])`	Allows you to make an HTTP PATCH request and follow any subsequent redirects.
`put_via_redirect(path, [parameters], [headers])`	Allows you to make an HTTP PUT request and follow any subsequent redirects.
`delete_via_redirect(path, [parameters], [headers])`	Allows you to make an HTTP DELETE request and follow any subsequent redirects.
`open_session`	Opens a new session instance.

5.2 Integration Testing Examples

A simple integration test that exercises multiple controllers:

```
require 'test_helper'

class UserFlowsTest < ActionDispatch::IntegrationTest
  test "login and browse site" do
    # login via https
    https!
    get "/login"
    assert_response :success

    post_via_redirect "/login", username: users(:david).username, password: users(:
david).password
    assert_equal '/welcome', path
    assert_equal 'Welcome david!', flash[:notice]

    https!(false)
    get "/articles/all"
    assert_response :success
    assert assigns(:articles)
  end
end
```

As you can see the integration test involves multiple controllers and exercises the entire stack from database to dispatcher. In addition you can have multiple session instances open simultaneously in a test and extend those instances with assertion methods to create a very powerful testing DSL (domain-specific language) just for your application.

Here's an example of multiple sessions and custom DSL in an integration test

```
require 'test_helper'

class UserFlowsTest < ActionDispatch::IntegrationTest
  test "login and browse site" do
    # User david logs in
    david = login(:david)
    # User guest logs in
    guest = login(:guest)

    # Both are now available in different sessions
    assert_equal 'Welcome david!', david.flash[:notice]
    assert_equal 'Welcome guest!', guest.flash[:notice]

    # User david can browse site
    david.browses_site
    # User guest can browse site as well
    guest.browses_site

    # Continue with other assertions
  end

  private

    module CustomDsl
      def browses_site
        get "/products/all"
        assert_response :success
        assert assigns(:products)
      end
    end

    def login(user)
      open_session do |sess|
        sess.extend(CustomDsl)
        u = users(user)
        sess.https!
        sess.post "/login", username: u.username, password: u.password
        assert_equal '/welcome', sess.path
        sess.https!(false)
      end
    end
end
```

6 Rake Tasks for Running your Tests

Rails comes with a number of built-in rake tasks to help with testing. The table below lists the commands included in the default Rakefile when a Rails project is created.

Tasks	Description
`rake test`	Runs all tests in the `test` folder. You can also simply run `rake` as Rails will run all the tests by default
`rake test:controllers`	Runs all the controller tests from `test/controllers`
`rake test:functionals`	Runs all the functional tests from `test/controllers`, `test/mailers`, and `test/functional`
`rake test:helpers`	Runs all the helper tests from `test/helpers`
`rake test:integration`	Runs all the integration tests from `test/integration`
`rake test:jobs`	Runs all the job tests from `test/jobs`
`rake test:mailers`	Runs all the mailer tests from `test/mailers`
`rake test:models`	Runs all the model tests from `test/models`
`rake test:units`	Runs all the unit tests from `test/models`, `test/helpers`, and `test/unit`
`rake test:db`	Runs all tests in the `test` folder and resets the db

7 A Brief Note About Minitest

Ruby ships with a vast Standard Library for all common use-cases including testing. Since version 1.9, Ruby provides `Minitest`, a framework for testing. All the basic assertions such as `assert_equal` discussed above are actually defined in `Minitest::Assertions`. The classes `ActiveSupport::TestCase`, `ActionController::TestCase`, `ActionMailer::TestCase`, `ActionView::TestCase` and `ActionDispatch::IntegrationTest` - which we have been inheriting in our test classes - include `Minitest::Assertions`, allowing us to use all of the basic assertions in our tests.

For more information on `Minitest`, refer to Minitest

8 Setup and Teardown

If you would like to run a block of code before the start of each test and another block of code after the end of each test you have two special callbacks for your rescue. Let's take note of this by looking at an example for our functional test in `Articles` controller:

```
require 'test_helper'

class ArticlesControllerTest < ActionController::TestCase

  # called before every single test
  def setup
    @article = articles(:one)
  end

  # called after every single test
  def teardown
    # as we are re-initializing @article before every test
    # setting it to nil here is not essential but I hope
    # you understand how you can use the teardown method
    @article = nil
  end

  test "should show article" do
    get :show, id: @article.id
    assert_response :success
  end

  test "should destroy article" do
    assert_difference('Article.count', -1) do
      delete :destroy, id: @article.id
    end

    assert_redirected_to articles_path
  end
```

```
end
```

Above, the `setup` method is called before each test and so `@article` is available for each of the tests. Rails implements `setup` and `teardown` as `ActiveSupport::Callbacks`. Which essentially means you need not only use `setup` and `teardown` as methods in your tests. You could specify them by using:

- a block
- a method (like in the earlier example)
- a method name as a symbol
- a lambda

Let's see the earlier example by specifying `setup` callback by specifying a method name as a symbol:

```
require 'test_helper'

class ArticlesControllerTest < ActionController::TestCase

  # called before every single test
  setup :initialize_article

  # called after every single test
  def teardown
    @article = nil
  end

  test "should show article" do
    get :show, id: @article.id
    assert_response :success
  end

  test "should update article" do
    patch :update, id: @article.id, article: {}
    assert_redirected_to article_path(assigns(:article))
  end

  test "should destroy article" do
    assert_difference('Article.count', -1) do
      delete :destroy, id: @article.id
    end

    assert_redirected_to articles_path
  end

  private

    def initialize_article
```

```
      @article = articles(:one)
    end
end
```

9 Testing Routes

Like everything else in your Rails application, it is recommended that you test your routes. Below are example tests for the routes of default `show` and `create` action of `Articles` controller above and it should look like:

```
class ArticleRoutesTest < ActionController::TestCase
  test "should route to article" do
    assert_routing '/articles/1', { controller: "articles", action: "show", id: "1" }
  end

  test "should route to create article" do
    assert_routing({ method: 'post', path: '/articles' }, { controller: "articles", action:
 "create" })
  end
end
```

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 it is 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 make 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 `invite` action.

```
require 'test_helper'

class UserMailerTest < ActionMailer::TestCase
  test "invite" do
    # Send the email, then test that it got queued
    email = UserMailer.create_invite('me@example.com',
                                     'friend@example.com', Time.now).deliver_now
    assert_not ActionMailer::Base.deliveries.empty?

    # 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` tests. If you want to have a clean slate outside Action Mailer tests, 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 < ActionController::TestCase
  test "invite friend" do
    assert_difference 'ActionMailer::Base.deliveries.size', +1 do
      post :invite_friend, 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 helpers

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.

A helper test looks like so:

```
require 'test_helper'

class UserHelperTest < ActionView::TestCase
end
```

A helper is just a simple module where you can define methods which are available into your views. To test the output of the helper's methods, you just have to use a mixin like this:

```
class UserHelperTest < ActionView::TestCase
  include UserHelper

  test "should return the user name" do
    # ...
  end
end
```

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

12 Testing Jobs

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

12.1 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 `:test` so that your jobs are performed inline. 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.

12.2 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:

Assertion	Purpose
`assert_enqueued_jobs(number)`	Asserts that the number of enqueued jobs matches the given number.
`assert_performed_jobs(number)`	Asserts that the number of performed jobs matches the given number.
`assert_no_enqueued_jobs { ... }`	Asserts that no jobs have been enqueued.
`assert_no_performed_jobs { ... }`	Asserts that no jobs have been performed.
`assert_enqueued_with([args]) { ... }`	Asserts that the job passed in the block has been enqueued with the given arguments.
`assert_performed_with([args]) { ... }`	Asserts that the job passed in the block has been performed with the given arguments.

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 < ActiveSupport::TestCase
  test 'billing job scheduling' do
    assert_enqueued_with(job: BillingJob) do
      product.charge(account)
    end
  end
end
```

13 Other Testing Approaches

The built-in `minitest` based testing is not the only way to test Rails applications. Rails developers have come up with a wide variety of other approaches and aids for testing, including:

- NullDB, a way to speed up testing by avoiding database use.
- Factory Girl, a replacement for fixtures.
- Fixture Builder, a tool that compiles Ruby factories into fixtures before a test run.
- MiniTest::Spec Rails, use the MiniTest::Spec DSL within your rails tests.
- Shoulda, an extension to `test/unit` with additional helpers, macros, and assertions.
- RSpec, a behavior-driven development framework
- Capybara, Acceptance test framework for web applications

14 Feedback

You're encouraged to help improve the quality of this guide.

Please contribute if you see any typos or factual errors. To get started, you can read our documentation contributions section.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Make sure to check Edge Guides first to verify if the issues are already fixed or not on the master branch. Check the Ruby on Rails Guides Guidelines for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please open an issue.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the rubyonrails-docs mailing list.