A Guide to Testing Rails Applications

This guide covers built-in mechanisms offered by Rails to test your application. By referring to this guide, you will be able to:

Understand Rails testing terminology

Write unit, functional and integration tests for your application

Identify other popular testing approaches and plugins

This guide won't teach you to write a Rails application; it assumes basic familiarity with the Rails way of doing things.

Chapters

1 Why Write Tests for your Rails Applications?

Rails makes it super easy to write your tests. It starts by producing skeleton test code in the background 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 Three Environments

Every Rails application you build has 3 sides: a side for production, a side for development, and a side for testing.

One place you'll find this distinction is in the `config/database.yml` file. This YAML configuration file has 3 different sections defining 3 unique database setups:

production
development
test

This allows you to set up and interact with test data without any danger of your tests altering data from your production environment.

For example, suppose you need to test your new `delete_this_user_and_every_everything_associated_with_it` function. Wouldn't you want to run this in an environment where it makes no difference if you destroy data or not?

When you do end up destroying your testing database (and it will happen, trust me), you can rebuild it from scratch according to the specs defined in the development database. You can do this by running `rake db:test:prepare`.

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/

fixtures/       functional/     integration/    test_helper.rb  unit/
```

The `unit` folder is meant to hold tests for your models, the `functional` folder is meant to hold tests for your controllers, and the `integration` folder 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.

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 and assume a single format: **YAML**.

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 separated by a blank space. You can place comments in a fixture file by using the # character in the first column.

2.3.3 ERB'in It Up

ERB allows you to embed ruby code within templates. YAML fixture format is pre-processed with ERB when you load fixtures. This allows you to use Ruby to help you generate some sample data.

```
<%

earth_size =
20

%>
mercury:

size:
<%=

earth_size /
50

%>

brightest_on:
<%=

113
.days.ago.to_s(
:db
)
%>

venus:

size:
<%=

earth_size /
2

%>

brightest_on:
<%=

67
.days.ago.to_s(
:db
)
%>

mars:

size:
<%=

earth_size -
69

%>

brightest_on:
<%=

13
.days.from_now.to_s(
:db
)
%>
```

Anything encased within the

```
<%

%>
```

tag is considered Ruby code. When this fixture is loaded, the `size` attribute of the three records will be set to 20/50, 20/2, and 20-69 respectively. The `brightest_on` attribute will also be evaluated and formatted by Rails to be compatible with the database.

2.3.4 Fixtures in Action

Rails by default automatically loads all fixtures from the `test/fixtures` folder for your unit and functional 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 Hashes with Special Powers

Fixtures are basically Hash objects. As mentioned in point #3 above, you can access the hash object directly because it is automatically setup as a local variable of the test case. For example:

```
# this will return the Hash for the fixture named david
users(
:david
)

# this will return the property for david called id
users(
:david
).id
```

Fixtures can also transform themselves into the form of the original class. Thus, you can get at the methods only available to that class.

```
# using the find method, we grab the "real" david as a User
david = users(
:david
).find

# and now we have access to methods only available to a User class
email(david.girlfriend.email, david.location_tonight)
```

3 Unit Testing your Models

In Rails, unit 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. I 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/unit` folder:

```
$ rails generate scaffold post title:string body:text
...
create  app/models/post.rb
create  test/unit/post_test.rb
create  test/fixtures/posts.yml
...
```

The default test stub in `test/unit/post_test.rb` looks like this:

```
require
'test_helper'

class

PostTest < ActiveSupport::TestCase

# Replace this with your real tests.

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

PostTest < ActiveSupport::TestCase
```

The `PostTest` class defines a *test case* because it inherits from `ActiveSupport::TestCase`. `PostTest` 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 `Test::Unit` test case that begins with `test` (case sensitive) is simply called a test. So, `test_password`, `test_valid_password` and `testValidPassword` all 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 `Test::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 Preparing your Application for Testing

Before you can run your tests, you need to ensure that the test database structure is current. For this you can use the following rake commands:

```
$ rake db:migrate
...
$ rake db:test:load
```

The `rake db:migrate` above runs any pending migrations on the *development* environment and updates `db/schema.rb`. The `rake db:test:load` recreates the test database from the current `db/schema.rb`. On subsequent attempts, it is a good idea to first run `db:test:prepare`, as it first checks for pending migrations and warns you appropriately.

`db:test:prepare` will fail with an error if `db/schema.rb` doesn't exist.

3.1.1 Rake Tasks for Preparing your Application for Testing

Tasks	Description
`rake db:test:clone`	Recreate the test database from the current environment's database schema
`rake db:test:clone_structure`	Recreate the test database from the development structure
`rake db:test:load`	Recreate the test database from the current `schema.rb`
`rake db:test:prepare`	Check for pending migrations and load the test schema
`rake db:test:purge`	Empty the test database.

You can see all these rake tasks and their descriptions by running `rake --tasks --describe`

3.2 Running Tests

Running a test is as simple as invoking the file containing the test cases through Ruby:

```
$ ruby -Itest test/unit/post_test.rb

Loaded suite unit/post_test
Started
.
Finished in 0.023513 seconds.

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

This will run all the 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.

You can also run a particular test method from the test case by using the `-n` switch with the `test method name`.

```
$ ruby -Itest test/unit/post_test.rb -n test_the_truth

Loaded suite unit/post_test
Started
.
Finished in 0.023513 seconds.

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

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 `post_test.rb` test case.

```
test
"should not save post without title"

do

post = Post.
new

assert !post.save
end
```

Let us run this newly added test.

```
$ ruby unit/post_test.rb -n test_should_not_save_post_without_title
Loaded suite -e
Started
F
Finished in 0.102072 seconds.


1) Failure:
test_should_not_save_post_without_title(PostTest) [/test/unit/post_test.rb:6]:
<false> is not true.

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

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 post without title"

do

post = Post.
new

assert !post.save,
"Saved the post without a title"
end
```

Running this test shows the friendlier assertion message:

```
1) Failure:
test_should_not_save_post_without_title(PostTest) [/test/unit/post_test.rb:6]:
Saved the post without a title.
<false> is not true.
```

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

```
class

Post < ActiveRecord::Base

validates
:title
,
:presence

=>
true
end
```

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

```
$ ruby unit/post_test.rb -n test_should_not_save_post_without_title
Loaded suite unit/post_test
```

```
Started
.
Finished in 0.193608 seconds.

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

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:

```
$ ruby unit/post_test.rb -n test_should_report_error
Loaded suite -e
Started
E
Finished in 0.082603 seconds.


1) Error:
test_should_report_error(PostTest):
NameError: undefined local variable or method `some_undefined_variable' for #<PostTest:0x249d354>

/test/unit/post_test.rb:6:in `test_should_report_error'

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

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.

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 Assertions Available

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 the complete list of assertions that ship with `test/unit`, 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( boolean, [msg] )`	Ensures that the object/expression is true.
`assert_equal( obj1, obj2, [msg] )`	Ensures that `obj1 == obj2` is true.
`assert_not_equal( obj1, obj2, [msg] )`	Ensures that `obj1 == obj2` is false.
`assert_same( obj1, obj2, [msg] )`	Ensures that `obj1.equal?(obj2)` is true.

`assert_not_same( obj1, obj2, [msg] )`	Ensures that `obj1.equal?(obj2)` is false.
`assert_nil( obj, [msg] )`	Ensures that `obj.nil?` is true.
`assert_not_nil( obj, [msg] )`	Ensures that `obj.nil?` is false.
`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_in_delta( expecting, actual, delta, [msg] )`	Ensures that the numbers `expecting` and `actual` are within `delta` of each other.
`assert_throws( symbol, [msg] ) { block }`	Ensures that the given block throws the symbol.
`assert_raise( 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 of the `class` type.
`assert_kind_of( class, obj, [msg] )`	Ensures that `obj` is or descends from `class`.
`assert_respond_to( obj, symbol, [msg] )`	Ensures that `obj` has a method called `symbol`.
`assert_operator( obj1, operator, obj2, [msg] )`	Ensures that `obj1.operator(obj2)` is true.
`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.

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 `test/unit` framework:

`assert_valid(record)` has been deprecated. Please use `assert(record.valid?)` instead.

Assertion	**Purpose**
`assert_valid(record)`	Ensures that the passed record is valid by Active Record standards and returns any error messages if it is not.
`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, &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.
`assert_response(type, message = nil)`	Asserts that the response comes with a specific status code. You can specify `:success` to indicate 200, `:redirect` to indicate 300-399, `:missing` to indicate 404, or `:error` to match the 500-599 range
`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.
`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 `Post` resource, it has already created the controller code and functional tests. You can take look at the file `posts_controller_test.rb` in the `test/functional` directory.

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

```
test
"should get index"

do

get
:index

assert_response
:success

assert_not_nil assigns(
:posts
)
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 `posts` 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 post 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_post` test from `posts_controller_test.rb` it will fail on account of the newly added model level validation and rightly so.

Let us modify `test_should_create_post` test in `posts_controller_test.rb` so that all our test pass:

```
test
"should create post"

do

assert_difference(
'Post.count'
)
do

post
:create
,
:post

=> {
:title

=>
'Some title'
}

end


assert_redirected_to post_path(assigns(
:post
))
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 5 request types supported in Rails functional tests:

```
get
post
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 by using one of the 5 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 A Fuller Functional Test Example

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

```
test
"should create post"

do

assert_difference(
'Post.count'
)
do

post
:create
,
:post

=> {
:title

=>
'Hi'
,
:body

=>
'This is my first post.'
}

end

assert_redirected_to post_path(assigns(
:post
))

assert_equal
'Post was successfully created.'
, flash[
:notice
]
end
```

4.6 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, but this is now deprecated in favor of `assert_select`.

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), an expression with substitution values, or an `HTML::Selector` object.

`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 `HTML::Node`) 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**.

4.6.1 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.

```
$ 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

fixtures
:all


# Replace this with your real tests.

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.
`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

fixtures
:users


test
"login and browse site"

do

# login via https

https!

get
"/login"

assert_response
:success


post_via_redirect
"/login"
,
:username

=> users(
:avs
).username,
:password

=> users(
:avs
).password

assert_equal
'/welcome'
, path

assert_equal
'Welcome avs!'
, flash[
:notice
]
```

```
https!(
false
)

get
"/posts/all"

assert_response
:success

assert assigns(
:products
)

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

fixtures
:users


test
"login and browse site"

do


# User avs logs in

avs = login(
:avs
)

# User guest logs in

guest = login(
:guest
)


# Both are now available in different sessions

assert_equal
'Welcome avs!'
, avs.flash[
:notice
]

assert_equal
'Welcome guest!'
, guest.flash[
:notice
]


# User avs can browse site

avs.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'
, path
sess.https!(
false
)
end
end
end
```

6 Rake Tasks for Running your Tests

You don't need to set up and run your tests by hand on a test-by-test basis. Rails comes with a number of rake tasks to help in testing. The table below lists all rake tasks that come along in the default Rakefile when you initiate a Rails project.

Tasks	Description
`rake test`	Runs all unit, functional and integration tests. You can also simply run `rake` as the *test* target is the default.
`rake test:benchmark`	Benchmark the performance tests
`rake test:functionals`	Runs all the functional tests from `test/functional`
`rake test:integration`	Runs all the integration tests from `test/integration`
`rake test:plugins`	Run all the plugin tests from `vendor/plugins/*/**/test` (or specify with `PLUGIN=_name_`)
`rake test:profile`	Profile the performance tests
`rake test:recent`	Tests recent changes

`rake test:uncommitted`	Runs all the tests which are uncommitted. Supports Subversion and Git
`rake test:units`	Runs all the unit tests from `test/unit`

7 Brief Note About `Test::Unit`

Ruby ships with a boat load of libraries. One little gem of a library is `Test::Unit`, a framework for unit testing in Ruby. All the basic assertions discussed above are actually defined in `Test::Unit::Assertions`. The class `ActiveSupport::TestCase` which we have been using in our unit and functional tests extends `Test::Unit::TestCase`, allowing us to use all of the basic assertions in our tests.

For more information on `Test::Unit`, refer to **test/unit Documentation**

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 `Posts` controller:

```
require
'test_helper'

class

PostsControllerTest < ActionController::TestCase


# called before every single test

def

setup

@post

= posts(
:one
)

end


# called after every single test

def

teardown

# as we are re-initializing @post before every test

# setting it to nil here is not essential but I hope

# you understand how you can use the teardown method

@post

=
nil

end


test
"should show post"

do

get
:show
,
:id

=>
@post
.id

assert_response
:success

end


test
```

```
"should destroy post"

do

assert_difference(
'Post.count'
, -
1
)
do

delete
:destroy
,
:id

=>
@post
.id

end


assert_redirected_to posts_path

end

end
```

Above, the `setup` method is called before each test and so `@post` 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

PostsControllerTest < ActionController::TestCase


# called before every single test

setup
:initialize_post


# called after every single test

def

teardown

@post

=
nil

end


test
"should show post"

do

get
:show
,
:id

=>
@post
.id
```

```
assert_response
:success

end


test
"should update post"

do

put
:update
,
:id

=>
@post
.id,
:post

=> { }

assert_redirected_to post_path(assigns(
:post
))

end


test
"should destroy post"

do

assert_difference(
'Post.count'
, -
1
)
do

delete
:destroy
,
:id

=>
@post
.id

end


assert_redirected_to posts_path

end


private


def

initialize_post

@post

= posts(
:one
)

end

end
```

9 Testing Routes

Like everything else in your Rails application, it is recommended that you test your routes. An example test for a route in the default `show` action of `Posts` controller above should look like:

```
test
"should route to post"

do

assert_routing
'/posts/1'
, {
:controller

=>
"posts"
,
:action

=>
"show"
,
:id

=>
"1"

}
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

tests UserMailer
```

```
test
"invite"

do

@expected
.from    =
'me@example.com'

@expected
.to      =
'friend@example.com'

@expected
.subject =
"You have been invited by #{@expected.from}"

@expected
.body    = read_fixture(
'invite'
)

@expected
.date    =
Time
.now


assert_equal
@expected
.encoded, UserMailer.create_invite(
'me@example.com'
,
'friend@example.com'
,
@expected
.date).encoded

end

end
```

In this test, `@expected` is an instance of `TMail::Mail` that you can use in your tests. It is defined in `ActionMailer::TestCase`. The test above uses `@expected` to construct an email, which it then asserts with email created by the custom mailer. The `invite` fixture is the body of the email and is used as the sample content to assert against. 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`).

However often in unit tests, mails will not actually be sent, simply constructed, as in the example above, where the precise content of the email is checked against what it should be.

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)

end
end
```

11 Other Testing Approaches

The built-in `test/unit` 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.
Machinist, another replacement for fixtures.
Shoulda, an extension to `test/unit` with additional helpers, macros, and assertions.
RSpec, a behavior-driven development framework

Feedback

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

If you see any typos or factual errors you are confident to patch, please clone **docrails** and push the change yourself. That branch of Rails has public write access. Commits are still reviewed, but that happens after you've submitted your contribution. **docrails** is cross-merged with master periodically.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. 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**.