DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.

Testing Rails Applications

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, integration, and system tests for your application.
- Other popular testing approaches and plugins.

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 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.

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.

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
application_system_test_case.rb  controllers/                    helpers/
channels/                        fixtures/                       integration/
```

The `helpers`, `mailers`, and `models` directories are meant to hold tests for view helpers, mailers, and models, respectively. The `channels` directory is meant to hold tests for Action Cable connection and channels. The `controllers` directory is meant to hold tests for controllers, routes, and views. The `integration` directory is meant to hold tests for interactions between controllers.

The system test directory holds system tests, which are used for full browser testing of your application. System tests allow you to test your application the way your users experience it and help you test your JavaScript as well. System tests inherit from Capybara and perform in browser tests for your application.

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

A `jobs` directory will also be created when an associated test is first generated.

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

The `application_system_test_case.rb` holds the default configuration for your system tests.

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`.

NOTE: Your tests are run under `RAILS_ENV=test`.

Rails meets Minitest

If you remember, we used the `bin/rails generate model` command in the Getting Started with Rails 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_` 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
```

Although you can still use regular method definitions, using the `test` macro allows for a more readable test name.

NOTE: 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.

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_te
Expected true to be nil or false


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_te
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).

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:0x
    test/models/article_test.rb:11:in 'block in <class:ArticleTest>'


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.

NOTE: 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 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. 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.

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`, 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?`.

Assertion	Purpose
`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`.

Assertion	Purpose
`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_in_epsilon ( expected, actual, [epsilon], [msg] )`	Ensures that the numbers `expected` and `actual` have a relative error less than `epsilon`.

Assertion	Purpose
`assert_not_in_epsilon ( expected, actual, [epsilon], [msg] )`	Ensures that the numbers `expected` and `actual` have a relative error not less than `epsilon`.
`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_instance_of( class, obj, [msg] )`	Ensures that `obj` is an instance of `class`.

Assertion	Purpose
`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`.

Assertion	Purpose
`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?`
`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, 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.

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

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.

Assertion	Purpose
`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_changes(expressions, message = nil, from:, to:, &block)`	Test that the result of evaluating an expression is changed after invoking the passed in block.

Assertion	Purpose
`assert_no_changes(expressions, message = nil, &block)`	Test the result of evaluating an expression is not changed after invoking the passed in block.
`assert_nothing_raised { block }`	Ensures that the given block doesn't raise any exceptions.

Assertion	Purpose
`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.

Assertion	Purpose
`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 pa-

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 informa-

Assertion	Purpose
`assert_redirected_to(options = {}, message=nil)`	Asserts that the response is a redirect to a URL matching the given options. 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.

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`
- `ActionMailer::TestCase`
- `ActionView::TestCase`
- `ActiveJob::TestCase`
- `ActionDispatch::IntegrationTest`
- `ActionDispatch::SystemTestCase`
- `Rails::Generators::TestCase`

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

NOTE: For more information on `Minitest`, refer to its documentation.

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

```
$ bin/rails test -h
Usage: 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.

minitest options:
    -h, --help                       Display this help.
        --no-plugins                 Bypass minitest plugin auto-loading (or set $MT_NO_PLUG
    -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
    -w, --warnings                   Run with Ruby warnings enabled
    -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
    -p, --pride                      Pride. Show your testing pride!
```

Parallel Testing

Parallel testing allows you to parallelize your test suite. While forking processes is the default method, threading is supported as well. Running tests in parallel reduces the time it takes your entire test suite to run.

Parallel Testing with Processes

The default parallelization method is to fork processes using Ruby's DRb system. The processes are forked based on the number of workers provided. The default number is the actual core count on the machine you are on, but can be changed by the number passed to the parallelize method.

To enable parallelization add the following to your `test_helper.rb`:

```
class ActiveSupport::TestCase
  parallelize(workers: 2)
end
```

The number of workers passed is the number of times the process will be forked. You may want to parallelize your local test suite differently from your CI, so an environment variable is provided to be able to easily change the number of workers a test run should use:

```
$ PARALLEL_WORKERS=15 bin/rails test
```

When parallelizing tests, Active Record automatically handles creating a database and loading the schema into the database for each process. The databases will be suffixed with the number corresponding to the worker. For example, if you have 2 workers the tests will create `test-database-0` and `test-database-1` respectively.

If the number of workers passed is 1 or fewer the processes will not be forked and the tests will not be parallelized and the tests will use the original `test-database` database.

Two hooks are provided, one runs when the process is forked, and one runs before the forked process is closed. These can be useful if your app uses multiple databases or performs other tasks that depend on the number of workers.

The `parallelize_setup` method is called right after the processes are forked. The `parallelize_teardown` method is called right before the processes are closed.

```
class ActiveSupport::TestCase
  parallelize_setup do |worker|
    # setup databases
  end

  parallelize_teardown do |worker|
    # cleanup databases
  end

  parallelize(workers: :number_of_processors)
end
```

These methods are not needed or available when using parallel testing with threads.

Parallel Testing with Threads

If you prefer using threads or are using JRuby, a threaded parallelization option is provided. The threaded parallelizer is backed by Minitest's `Parallel::Executor`.

To change the parallelization method to use threads over forks put the following in your `test_helper.rb`

```
class ActiveSupport::TestCase
  parallelize(workers: :number_of_processors, with: :threads)
end
```

Rails applications generated from JRuby or TruffleRuby will automatically include the `with: :threads` option.

The number of workers passed to `parallelize` determines the number of threads the tests will use. You may want to parallelize your local test suite differently from your CI, so an environment variable is provided to be able to easily change the number of workers a test run should use:

```
$ PARALLEL_WORKERS=15 bin/rails test
```

Testing Parallel Transactions

Rails automatically wraps any test case in a database transaction that is rolled back after the test completes. This makes test cases independent of each other and changes to the database are only visible within a single test.

When you want to test code that runs parallel transactions in threads, transactions can block each other because they are already nested under the test transaction.

You can disable transactions in a test case class by setting `self.use_transactional_tests = false`:

```
class WorkerTest < ActiveSupport::TestCase
  self.use_transactional_tests = false

  test "parallel transactions" do
    # start some threads that create transactions
  end
end
```

NOTE: With disabled transactional tests, you have to clean up any data tests create as changes are not automatically rolled back after the test completes.

Threshold to parallelize tests

Running tests in parallel adds an overhead in terms of database setup and fixture loading. Because of this, Rails won't parallelize executions that involve fewer than 50 tests.

You can configure this threshold in your `test.rb`:

```
config.active_support.test_parallelization_threshold = 100
```

And also when setting up parallelization at the test case level:

```
class ActiveSupport::TestCase
  parallelize threshold: 100
end
```

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.

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.

NOTE: 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`.

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.

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.

NOTE: 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 `bin/rails generate model` to create a new model, Rails automatically creates fixture stubs in this directory.

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:

```yaml
# 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, you can define a reference node between two different fixtures. Here's an example with a `belongs_to`/`has_many` association:

```yaml
# test/fixtures/categories.yml
about:
  name: About
```

```yaml
# test/fixtures/articles.yml
first:
  title: Welcome to Rails!
  category: about
```

```yaml
# test/fixtures/action_text/rich_texts.yml
first_content:
  record: first (Article)
```

```
  name: content
  body: <div>Hello, from <strong>a fixture</strong></div>
```

Notice the `category` key of the `first` Article found in `fixtures/articles.yml` has a value of `about`, and that the `record` key of the `first_content` entry found in `fixtures/action_text/rich_texts.yml` has a value of `first (Article)`. This hints to Active Record to load the Category `about` found in `fixtures/categories.yml` for the former, and Action Text to load the Article `first` found in `fixtures/articles.yml` for the latter.

NOTE: 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.

File attachment fixtures Like other Active Record-backed models, Active Storage attachment records inherit from ActiveRecord::Base instances and can therefore be populated by fixtures.

Consider an `Article` model that has an associated image as a `thumbnail` attachment, along with fixture data YAML:

```
class Article
  has_one_attached :thumbnail
end
```

```
# test/fixtures/articles.yml
first:
  title: An Article
```

Assuming that there is an image/png encoded file at `test/fixtures/files/first.png`, the following YAML fixture entries will generate the related `ActiveStorage::Blob` and `ActiveStorage::Attachment` records:

```
# test/fixtures/active_storage/blobs.yml
first_thumbnail_blob: <%= ActiveStorage::FixtureSet.blob filename: "first.png" %>
```

```
# test/fixtures/active_storage/attachments.yml
first_thumbnail_attachment:
  name: thumbnail
  record: first (Article)
  blob: first_thumbnail_blob
```

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 %>
```

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

TIP: 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).

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)
```

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`.

System Testing

System tests allow you to test user interactions with your application, running tests in either a real or a headless browser. System tests use Capybara under the hood.

For creating Rails system tests, you use the `test/system` directory in your application. Rails provides a generator to create a system test skeleton for you.

```
$ bin/rails generate system_test users
      invoke test_unit
      create test/system/users_test.rb
```

Here's what a freshly generated system test looks like:

```
require "application_system_test_case"

class UsersTest < ApplicationSystemTestCase
  # test "visiting the index" do
  #   visit users_url
  #
  #   assert_selector "h1", text: "Users"
  # end
end
```

By default, system tests are run with the Selenium driver, using the Chrome browser, and a screen size of 1400x1400. The next section explains how to change the default settings.

Changing the default settings

Rails makes changing the default settings for system tests very simple. All the setup is abstracted away so you can focus on writing your tests.

When you generate a new application or scaffold, an `application_system_test_case.rb` file is created in the test directory. This is where all the configuration for your system tests should live.

If you want to change the default settings you can change what the system tests are "driven by". Say you want to change the driver from Selenium to Cuprite. First add the `cuprite` gem to your `Gemfile`. Then in your `application_system_test_case.rb` file do the following:

```
require "test_helper"
require "capybara/cuprite"

class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
```

```
  driven_by :cuprite
end
```

The driver name is a required argument for `driven_by`. The optional arguments that can be passed to `driven_by` are `:using` for the browser (this will only be used by Selenium), `:screen_size` to change the size of the screen for screenshots, and `:options` which can be used to set options supported by the driver.

```
require "test_helper"

class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
  driven_by :selenium, using: :firefox
end
```

If you want to use a headless browser, you could use Headless Chrome or Headless Firefox by adding `headless_chrome` or `headless_firefox` in the `:using` argument.

```
require "test_helper"

class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
  driven_by :selenium, using: :headless_chrome
end
```

If you want to use a remote browser, e.g. Headless Chrome in Docker, you have to add remote `url` through `options`.

```
require "test_helper"

class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
  options = ENV["SELENIUM_REMOTE_URL"].present? ? { url: ENV["SELENIUM_REMOTE_URL"] } : {}
  driven_by :selenium, using: :headless_chrome, options: options
end
```

In such a case, the gem `webdrivers` is no longer required. You could remove it completely or add `require:` option in `Gemfile`.

```
# ...
group :test do
  gem "webdrivers", require: !ENV["SELENIUM_REMOTE_URL"] || ENV["SELENIUM_REMOTE_URL"].empty
end
```

Now you should get a connection to remote browser.

```
$ SELENIUM_REMOTE_URL=http://localhost:4444/wd/hub bin/rails test:system
```

If your application in test is running remote too, e.g. Docker container, Capybara needs more input about how to call remote servers.

```
require "test_helper"

class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
```

```
  def setup
    Capybara.server_host = "0.0.0.0" # bind to all interfaces
    Capybara.app_host = "http://#{IPSocket.getaddress(Socket.gethostname)}" if ENV["SELENIUM
    super
  end
  # ...
end
```

Now you should get a connection to remote browser and server, regardless if it is running in Docker container or CI.

If your Capybara configuration requires more setup than provided by Rails, this additional configuration could be added into the `application_system_test_case.rb` file.

Please see Capybara's documentation for additional settings.

Screenshot Helper

The `ScreenshotHelper` is a helper designed to capture screenshots of your tests. This can be helpful for viewing the browser at the point a test failed, or to view screenshots later for debugging.

Two methods are provided: `take_screenshot` and `take_failed_screenshot`. `take_failed_screenshot` is automatically included in `before_teardown` inside Rails.

The `take_screenshot` helper method can be included anywhere in your tests to take a screenshot of the browser.

Implementing a System Test

Now we're going to add a system test to our blog application. We'll demonstrate writing a system test by visiting the index page and creating a new blog article.

If you used the scaffold generator, a system test skeleton was automatically created for you. If you didn't use the scaffold generator, start by creating a system test skeleton.

```
$ bin/rails generate system_test articles
```

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

```
      invoke  test_unit
      create    test/system/articles_test.rb
```

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

```
require "application_system_test_case"

class ArticlesTest < ApplicationSystemTestCase
```

```
  test "viewing the index" do
    visit articles_path
    assert_selector "h1", text: "Articles"
  end
end
```

The test should see that there is an `h1` on the articles index page and pass.

Run the system tests.

```
$ bin/rails test:system
```

NOTE: By default, running `bin/rails test` won't run your system tests. Make sure to run `bin/rails test:system` to actually run them. You can also run `bin/rails test:all` to run all tests, including system tests.

Creating Articles System Test Now let's test the flow for creating a new article in our blog.

```
test "should create Article" do
  visit articles_path

  click_on "New Article"

  fill_in "Title", with: "Creating an Article"
  fill_in "Body", with: "Created this article successfully!"

  click_on "Create Article"

  assert_text "Creating an Article"
end
```

The first step is to call `visit articles_path`. This will take the test to the articles index page.

Then the `click_on "New Article"` will find the "New Article" button on the index page. This will redirect the browser to `/articles/new`.

Then the test will fill in the title and body of the article with the specified text. Once the fields are filled in, "Create Article" is clicked on which will send a POST request to create the new article in the database.

We will be redirected back to the articles index page and there we assert that the text from the new article's title is on the articles index page.

Testing for multiple screen sizes If you want to test for mobile sizes on top of testing for desktop, you can create another class that inherits from SystemTestCase and use in your test suite. In this example a file called `mobile_system_test_case.rb` is created in the `/test` directory with the following configuration.

```ruby
require "test_helper"

class MobileSystemTestCase < ActionDispatch::SystemTestCase
  driven_by :selenium, using: :chrome, screen_size: [375, 667]
end
```

To use this configuration, create a test inside `test/system` that inherits from `MobileSystemTestCase`. Now you can test your app using multiple different configurations.

```ruby
require "mobile_system_test_case"

class PostsTest < MobileSystemTestCase

  test "visiting the index" do
    visit posts_url
    assert_selector "h1", text: "Posts"
  end
end
```

Taking it further The beauty of system testing is that it is similar to integration testing in that it tests the user's interaction with your controller, model, and view, but system testing is much more robust and actually tests your application as if a real user were using it. Going forward, you can test anything that the user themselves would do in your application such as commenting, deleting articles, publishing draft articles, etc.

Integration Testing

Integration tests are used to test how various parts of our 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:

```ruby
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.

Helpers Available for Integration Tests

In addition to the standard testing helpers, inheriting from `ActionDispatch::IntegrationTest` comes 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`.

When performing requests, we will have `ActionDispatch::Integration::RequestHelpers` available for our use.

If we need to modify the session, or state of our integration test, take a look at `ActionDispatch::Integration::Session` to help.

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.

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.

NOTE: 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.

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.

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.

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 appropriate message displayed to the user in the view?
- was the correct information displayed in the response?

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 URI of the controller action you are requesting. This can be in the form of a string or a route helper (e.g. `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.

All of these keyword arguments are optional.

Example: Calling the `:show` action for the first `Article`, passing in an `HTTP_REFERER` header:

```
get article_url(Article.first), headers: { "HTTP_REFERER" => "http://example.com/home" }
```

Another example: Calling the `:update` action for the last `Article`, passing in new text for the `title` in `params`, as an Ajax request:

```
patch article_url(Article.last), params: { article: { title: "updated" } }, xhr: true
```

One more example: Calling the `:create` action to create a new article, passing in text for the `title` in `params`, as JSON request:

```
post articles_path, params: { article: { title: "Ahoy!" } }, as: :json
```

NOTE: 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.

NOTE: If you followed the steps in the Basic Authentication section, you'll need to add authorization to every request header to get all the tests passing:

```
post articles_url, params: { article: { body: "Rails is awesome!", title: "Hello Rails" } }
```

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.

NOTE: 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.

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.media_type
end
```

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]
```

Instance Variables Available

You also have access to three instance variables in your functional tests, after a request is made:

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

```
class ArticlesControllerTest < ActionDispatch::IntegrationTest
  test "should get index" do
    get articles_url

    assert_equal "index", @controller.action_name
    assert_equal "application/x-www-form-urlencoded", @request.media_type
    assert_match "Articles", @response.body
  end
end
```

Setting Headers and CGI variables

HTTP headers and CGI variables can be passed as headers:

```
# setting an HTTP Header
get articles_url, headers: { "Content-Type": "text/plain" } # simulate the request with cus

# setting a CGI variable
get articles_url, headers: { "HTTP_REFERER": "http://example.com/home" } # simulate the req
```

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 articles_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_tes
--- 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
```

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
```

```ruby
  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.

Test helpers

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

```ruby
# 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
```

Using Separate Files If you find your helpers are cluttering `test_helper.rb`, you can extract them into separate files. One good place to store them is `test/lib` or `test/test_helpers`.

```
# test/test_helpers/multiple_assertions.rb
module MultipleAssertions
  def assert_multiple_of_forty_two(number)
    assert (number % 42 == 0), 'expected #{number} to be a multiple of 42'
  end
end
```

These helpers can then be explicitly required as needed and included as needed

```
require "test_helper"
require "test_helpers/multiple_assertions"

class NumberTest < ActiveSupport::TestCase
  include MultipleAssertions

  test "420 is a multiple of forty two" do
    assert_multiple_of_forty_two 420
  end
end
```

or they can continue to be included directly into the relevant parent classes

```
# test/test_helper.rb
require "test_helpers/sign_in_helper"
```

```ruby
class ActionDispatch::IntegrationTest
  include SignInHelper
end
```

Eagerly Requiring Helpers You may find it convenient to eagerly require helpers in `test_helper.rb` so your test files have implicit access to them. This can be accomplished using globbing, as follows

```ruby
# test/test_helper.rb
Dir[Rails.root.join("test", "test_helpers", "**", "*.rb")].each { |file| require file }
```

This has the downside of increasing the boot-up time, as opposed to manually requiring only the necessary files in your individual tests.

Testing Routes

Like everything else in your Rails application, you can test your routes. Route tests reside in `test/controllers/` or are part of controller tests.

NOTE: 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`.

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:

```ruby
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.

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.

Assertion	Purpose
`assert_select_encoded`	Allows you to make assertions on encoded HTML. It does this by unencoding the contents of each element and then calling the block with all the unencoded elements.

Assertion	Purpose
`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
```

Testing Helpers

A helper is just a simple module where you can define methods which are available in 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 UsersHelper
  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 UsersHelperTest < 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_use
  end
end
```

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

Testing Your Mailers

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

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

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.

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.

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.

If you generated your mailer, the generator does not create stub fixtures for the mailers actions. You'll have to create those files yourself as described above.

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
    # 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 create 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.

NOTE: `email.body.to_s` is present when there's only one (HTML or text) part present. If the mailer provides both, you can test your fixture against specific parts with `email.text_part.body.to_s` or `email.html_part.body.to_s`.

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`).

NOTE: 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`

Functional and System Testing

Unit testing allows us to test the attributes of the email while functional and system testing allows us to test whether user interactions appropriately trigger the email to be delivered. For example, you can check that the invite friend operation is sending an email appropriately:

```
# Integration Test
require "test_helper"

class UsersControllerTest < ActionDispatch::IntegrationTest
  test "invite friend" do
    # Asserts the difference in the ActionMailer::Base.deliveries
    assert_emails 1 do
      post invite_friend_url, params: { email: "friend@example.com" }
    end
  end
end

# System Test
require "test_helper"
```

```ruby
class UsersTest < ActionDispatch::SystemTestCase
  driven_by :selenium, using: :headless_chrome

  test "inviting a friend" do
    visit invite_users_url
    fill_in "Email", with: "friend@example.com"
    assert_emails 1 do
      click_on "Invite"
    end
  end
end
```

NOTE: The `assert_emails` method is not tied to a particular deliver method and will work with emails delivered with either the `deliver_now` or `deliver_later` method. If we explicitly want to assert that the email has been enqueued we can use the `assert_enqueued_emails` method. More information can be found in the documentation here.

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.

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:

```ruby
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 got 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.

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`.

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
  include ActiveJob::TestHelper

  test "billing job scheduling" do
    assert_enqueued_with(job: BillingJob) do
      product.charge(account)
    end
  end
end
```

Testing Action Cable

Since Action Cable is used at different levels inside your application, you'll need to test both the channels, connection classes themselves, and that other entities broadcast correct messages.

Connection Test Case

By default, when you generate new Rails application with Action Cable, a test for the base connection class (`ApplicationCable::Connection`) is generated as well under `test/channels/application_cable` directory.

Connection tests aim to check whether a connection's identifiers get assigned properly or that any improper connection requests are rejected. Here is an example:

```
class ApplicationCable::ConnectionTest < ActionCable::Connection::TestCase
  test "connects with params" do
    # Simulate a connection opening by calling the `connect` method
    connect params: { user_id: 42 }

    # You can access the Connection object via `connection` in tests
    assert_equal connection.user_id, "42"
  end
```

```
  test "rejects connection without params" do
    # Use `assert_reject_connection` matcher to verify that
    # connection is rejected
    assert_reject_connection { connect }
  end
end
```

You can also specify request cookies the same way you do in integration tests:

```
test "connects with cookies" do
  cookies.signed[:user_id] = "42"

  connect

  assert_equal connection.user_id, "42"
end
```

See the API documentation for `ActionCable::Connection::TestCase` for more information.

Channel Test Case

By default, when you generate a channel, an associated test will be generated as well under the `test/channels` directory. Here's an example test with a chat channel:

```
require "test_helper"

class ChatChannelTest < ActionCable::Channel::TestCase
  test "subscribes and stream for room" do
    # Simulate a subscription creation by calling `subscribe`
    subscribe room: "15"

    # You can access the Channel object via `subscription` in tests
    assert subscription.confirmed?
    assert_has_stream "chat_15"
  end
end
```

This test is pretty simple and only asserts that the channel subscribes the connection to a particular stream.

You can also specify the underlying connection identifiers. Here's an example test with a web notifications channel:

```
require "test_helper"

class WebNotificationsChannelTest < ActionCable::Channel::TestCase
  test "subscribes and stream for user" do
```

```
    stub_connection current_user: users(:john)

    subscribe

    assert_has_stream_for users(:john)
  end
end
```

See the API documentation for `ActionCable::Channel::TestCase` for more information.

Custom Assertions And Testing Broadcasts Inside Other Components

Action Cable 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 `ActionCable::TestHelper`.

It's a good practice to ensure that the correct message has been broadcasted inside other components (e.g. inside your controllers). This is precisely where the custom assertions provided by Action Cable are pretty useful. For instance, within a model:

```
require "test_helper"

class ProductTest < ActionCable::TestCase
  test "broadcast status after charge" do
    assert_broadcast_on("products:#{product.id}", type: "charged") do
      product.charge(account)
    end
  end
end
```

If you want to test the broadcasting made with `Channel.broadcast_to`, you should use `Channel.broadcasting_for` to generate an underlying stream name:

```
# app/jobs/chat_relay_job.rb
class ChatRelayJob < ApplicationJob
  def perform(room, message)
    ChatChannel.broadcast_to room, text: message
  end
end
```

```
# test/jobs/chat_relay_job_test.rb
require "test_helper"

class ChatRelayJobTest < ActiveJob::TestCase
  include ActionCable::TestHelper

  test "broadcast message to room" do
```

```
    room = rooms(:all)

    assert_broadcast_on(ChatChannel.broadcasting_for(room), text: "Hi!") do
      ChatRelayJob.perform_now(room, "Hi!")
    end
  end
end
```

Testing Eager Loading

Normally, applications do not eager load in the `development` or `test` environments to speed things up. But they do in the `production` environment.

If some file in the project cannot be loaded for whatever reason, you better detect it before deploying to production, right?

Continuous Integration

If your project has CI in place, eager loading in CI is an easy way to ensure the application eager loads.

CIs typically set some environment variable to indicate the test suite is running there. For example, it could be `CI`:

```
# config/environments/test.rb
config.eager_load = ENV["CI"].present?
```

Starting with Rails 7, newly generated applications are configured that way by default.

Bare Test Suites

If your project does not have continuous integration, you can still eager load in the test suite by calling `Rails.application.eager_load!`:

minitest

```
require "test_helper"

class ZeitwerkComplianceTest < ActiveSupport::TestCase
  test "eager loads all files without errors" do
    assert_nothing_raised { Rails.application.eager_load! }
  end
end
```

RSpec

```
require "rails_helper"
```

```ruby
RSpec.describe "Zeitwerk compliance" do
  it "eager loads all files without errors" do
    expect { Rails.application.eager_load! }.not_to raise_error
  end
end
```

Additional Testing Resources

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` helper:

```ruby
# 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
  assert user.applicable_for_gifting?
end
assert_equal Date.new(2004, 10, 24), user.activation_date # The change was visible only ins
```

Please see `ActiveSupport::Testing::TimeHelpers` API Documentation for in-depth information about the available time helpers.