

## Advance Ruby on Rails Topics

1. Rspec, Automation Testing
2. Cucumber
3. Capybara
4. Mocks and Stubs
5. Action Mailer

## RAILS: RSPEC AND CAPYBARA BY EXAMPLE

### How to setup and use two popular gems used in tests in Rails: RSpec and Capybara.

Post's example based on TDD approach.

Post based on example application to get context in testing and its purpose is a managing library with books. In this I focus only on the adding new books feature.

#### Setup Project

Ruby version: ruby 2.2.2p95

Rails version: rails 4.2.6

Let's start by creating project:

```
rails new my_app -d mysql
```

```
1 rails new my_app -d mysql / Postgres
```

If you prefer to use NO-SQL database, use following command:

```
rails new my_app
```

```
1 rails new my_app
```

If you choose MySQL database, you need to setup database access: username and password. It's better to keep this data secret, so I use config/secrets.yml. Add to this file 2 key-value pairs and replace root and password\_to\_database:

development:

secret\_key\_base: 6904e69fc...118

database\_username: root

database\_password: password\_to\_database

```
1 development:
2   secret_key_base: 6904e69fc...118
3   database_username: root
4   database_password: password_to_database
```

Open config/database.yml and add just created secret keys:

```
default: &default
  adapter: mysql2
  encoding: utf8
  pool: 5
  username: <%= Rails.application.secrets[:database_username] %>
  password: <%= Rails.application.secrets[:database_password] %>
  socket: /var/run/mysqld/mysqld.sock
```

```
1 default: &default
2   adapter: mysql2
3   encoding: utf8
4   pool: 5
5   username: <%= Rails.application.secrets[:database_username] %>
6   password: <%= Rails.application.secrets[:database_password] %>
7   socket: /var/run/mysqld/mysqld.sock
```

Before you push any changes to repository, tell GIT to ignore our secrets.

Open .gitignore file:

```
nano .gitignore
```

```
1 nano .gitignore
```

Add files to ignore:

```
config/secrets.yml
config/database.yml
```

```
1 config/secrets.yml
2 config/database.yml
```

And finally create database:

```
rake db:create
```

```
1 rake db:create
```

Let's run application:

```
rails s
```

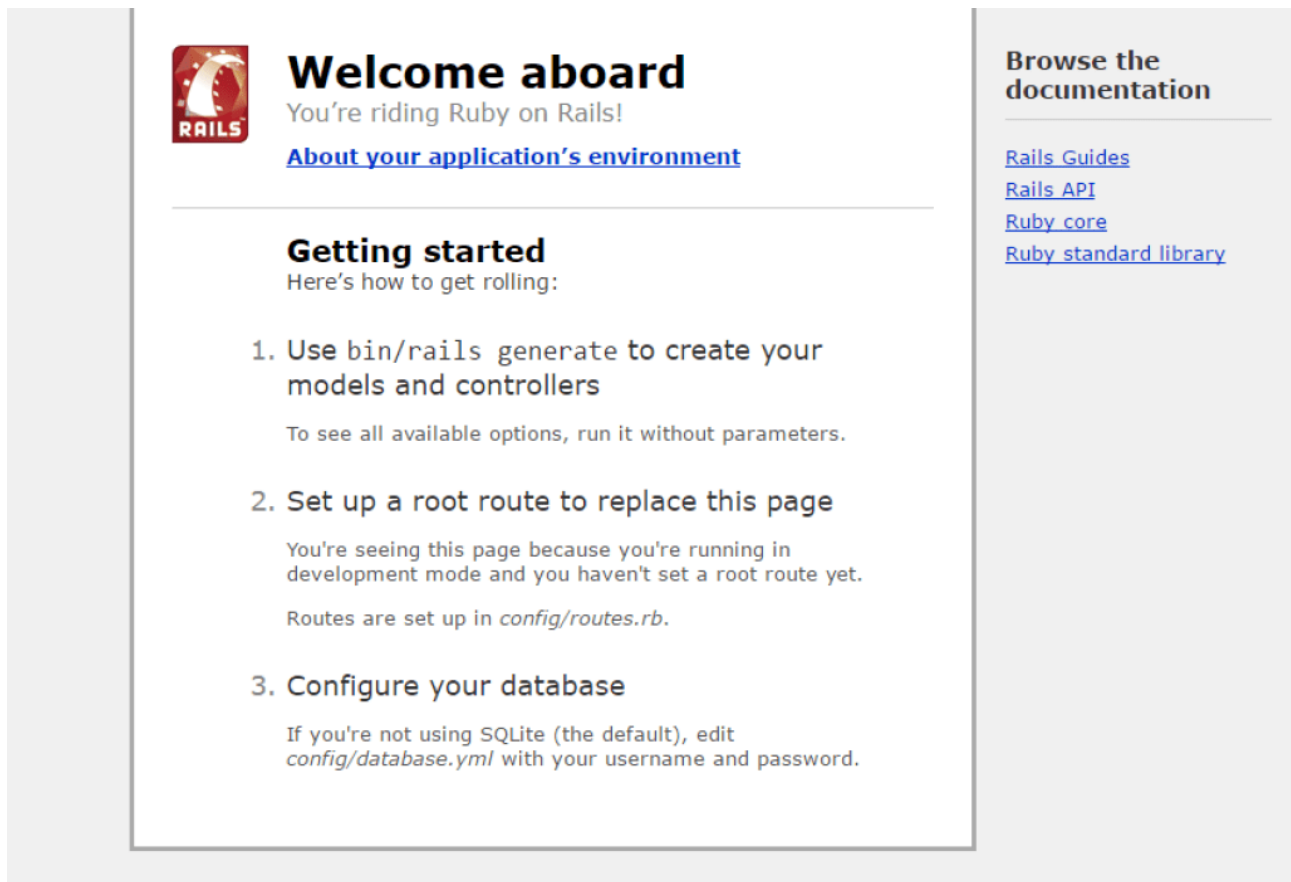
```
1 rails s
```


If you run application on external server (like VPS), probably you will need to run like below by adding IP and port:

```
rails s -b 12.34.567.891 -p 3000
```

```
1 rails s -b 12.34.567.891 -p 3000
```

Open application in web browser. If everything is OK, you should see following screen:



 **Welcome aboard**  
You're riding Ruby on Rails!  
[About your application's environment](#)

---

**Getting started**  
Here's how to get rolling:

- 1. Use `bin/rails generate` to create your models and controllers**  
To see all available options, run it without parameters.
- 2. Set up a root route to replace this page**  
You're seeing this page because you're running in development mode and you haven't set a root route yet.  
Routes are set up in `config/routes.rb`.
- 3. Configure your database**  
If you're not using SQLite (the default), edit `config/database.yml` with your username and password.

**Browse the documentation**

- [Rails Guides](#)
- [Rails API](#)
- [Ruby core](#)
- [Ruby standard library](#)

## Test Gems

- RSpec – testing framework for Rails
- Capybara – to testing web pages

## Installing Gems

Add to Gemfile:

```
group :development, :test do
  gem 'rspec-rails', '~> 3.0'
end
```

```
group :test do
  gem 'capybara'
end
```

```
1 group :development, :test do
2   gem 'rspec-rails', '~> 3.0'
3 end
4
5 group :test do
6   gem 'capybara'
7 end
```

Run:

bundle install

rails generate rspec:install

```
1 bundle install
2
3 rails generate rspec:install
```

Second command should create spec/spec\_helper.rb and spec/rails\_helper.rb files. Add following to spec/rails\_helper.rb:

```
require 'capybara/rails'
```

```
1 require 'capybara/rails'
```

Testing: RSpec and Capybara

The example application is a CMS-like application to managing library with books. First feature will be the adding new books to library.

I start from defining a scenario for this feature's test:

- # 1. Go to root path (there will be button to add new book)
- # 2. Click on "Add new book" button
- # 3. Fill out the form
- # 4. Submit form
- # 5. See 'show' page of created book

```
1 # 1. Go to root path (there will be button to add new book)
2 # 2. Click on "Add new book" button
3 # 3. Fill out the form
4 # 4. Submit form
5 # 5. See 'show' page of created book
```

Next, I'll implement it.

I create folder spec/books and file spec/books/creating\_book\_spec.rb:

```
require 'rails_helper'
```

```
feature 'Creating book' do
```

```
  scenario 'can create a book' do
```

- # 1. go to root where will be button to Add New Book:  
visit '/'
- # 2. click on Add New Book button  
click\_link 'Add New Book'
- # 3. Fill form - add title

```
fill_in 'book_title', with: 'Ulisses'  
# 4. Click on submit form button  
click_button 'Save Book'  
# 5. Then we should be redirected to show page with book details (book  
title)  
expect(page).to have_content('Ulisses')  
end  
end
```

```
require 'rails_helper.rb'  
  
feature 'Creating book' do  
  scenario 'can create a book' do  
    # 1. go to root where will be button to Add New Book:  
    visit '/'  
    # 2. click on Add New Book button  
    click_link 'Add New Book'  
    # 3. Fill form - add title  
    fill_in 'book_title', with: 'Ulisses'  
    # 4. Click on submit form button  
    click_button 'Save Book'  
    # 5. Then we should be redirected to show page with book details (book  
title)  
    expect(page).to have_content('Ulisses')  
  end  
end
```

To run all test:

```
rspec
```

```
1 rspec
```

To run specific test:

```
rspec spec/books/creating_book_spec.rb
```

```
1 rspec spec/books/creating_book_spec.rb
```

The test for creating book should failed, because nothing were implemented yet:

```

Failures:

  1) Creating book can create a book
     Failure/Error: visit '/'

     ActionController::RoutingError:
       No route matches [GET] "/"
     # ./spec/books/creating_book_spec.rb:6:in `block (2 levels) in <top (required)>'

Finished in 0.09645 seconds (files took 12.14 seconds to load)
1 example, 1 failure

Failed examples:

rspec ./spec/books/creating_book_spec.rb:4 # Creating book can create a book

```

Currently, the test fails due to `root_path "/"`.

I'll try to pass this test. To do that, I have to create new model `Book`, controller with (at least) `index`, `new`, `create` actions and views for specific actions. I have to set the route path to `Book` `index` action in `routes.rb`.

```
$ rails g model Book title:string
```

```
$ rake db:migrate
```

```
$ rails g controller books
```

```

1 $ rails g model Book title:string
2 $ rake db:migrate
3 $ rails g controller books

```

I add required actions to `app/controllers/book_controller.rb`:

```
class BooksController < ApplicationController
```

```
  def index
  end
```

```

  def new
  end
```

```

  def create
  end
```

```
end
```

```

1 class BooksController < ApplicationController
2   def index
3   end
4
5   def new
6   end
7
8   def create
9   end
10 end
11

```

I set routes in config/routes.rb:  
 Rails.application.routes.draw do

```

# root_path:
root 'books#index'

```

```

# It generates CRUD paths for Book:
resources :books

```

end

```

1 Rails.application.routes.draw do
2
3   # root_path:
4   root 'books#index'
5
6   # It generates CRUD paths for Book:
7   resources :books
8
9 end

```

Create views for Book index and new actions: app/views/books/index.html.erb and app/views/books/new.html.erb. Before I implement actions and views, I'll run a test again:

```
rspec spec/books/creating_book_spec.rb
```

```

1 rspec spec/books/creating_book_spec.rb

```

```

Failures:

  1) Creating book can create a book
     Failure/Error: click_link 'Add New Book'

     Capybara::ElementNotFound:
       Unable to find link "Add New Book"
     # ./spec/books/creating_book_spec.rb:8:in `block (2 levels) in <top (required)>'

Finished in 0.57706 seconds (files took 6.35 seconds to load)
1 example, 1 failure

Failed examples:

rspec ./spec/books/creating_book_spec.rb:4 # Creating book can create a book

```

This time the test fails because it couldn't find a button "Add New Book". It's (2) step in our test scenario. I'll add the "Add New Book" button on index page (app/views/books/index.html.erb):

```
<%= link_to 'Add New Book', new_book_path %>
```

```
1 <%= link_to 'Add New Book', new_book_path %>
```

In web browser you should see link:

[Add New Book](#)

Let's run test again:

```
rspec spec/books/creating_book_spec.rb
```

```
1 rspec spec/books/creating_book_spec.rb
```

And as I expected, test passed 2nd step (link was found) and failed on 3rd step (it didn't find a form):

```

Failures:

  1) Creating book can create a book
     Failure/Error: fill_in 'title', with: 'Ulisses'

     Capybara::ElementNotFound:
       Unable to find field "title"
     # ./spec/books/creating_book_spec.rb:10:in `block (2 levels) in <top (required)>'

Finished in 0.62064 seconds (files took 3.73 seconds to load)
3 examples, 1 failure, 2 pending

Failed examples:

rspec ./spec/books/creating_book_spec.rb:4 # Creating book can create a book

```

To pass 3rd step I need add form to app/views/books/new.html.erb:

```
<%= form_for Book.new do |f| %>
```

```
  <%= f.label :title %> <br/>
```



```
<%= f.text_field :title %>  
  
<%= f.submit 'Save Book' %>
```

```
<% end %>
```

```
1 <%= form_for Book.new do |f| %>  
2  
3   <%= f.label :title %> <br/>  
4   <%= f.text_field :title %>  
5  
6   <%= f.submit 'Save Book' %>  
7  
8 <% end %>
```

Let's open in web browser Home page and click on Add New Book link:

[Add New Book](#)

It should redirect you on new.html.erb page where you should see simple following form:

Title

If you click on "Save Book" button, you should get error about "No route matches [POST] /books/new", because the controller part is not ready. The same error you should get from test. So let's add controller stuff to create new book and show it. To app/controller/books\_controller.html.erb add following:

```
class BooksController < ApplicationController  
  def index  
  end  
  
  def new  
  end  
  
  def create  
    @book = Book.new(book_params)  
  
    if @book.save!
```

```
    redirect_to @book
  else
    render 'new'
  end
end
```

```
def show
  @book = Book.find(params[:id])
end
```

```
private
def book_params
  params.require(:book).permit(:title)
end
```

```
end
```

```
class BooksController < ApplicationController
  def index
  end

  def new
  end

  def create
    @book = Book.new(book_params)

    if @book.save!
      redirect_to @book
    else
      render 'new'
    end
  end

  def show
    @book = Book.find(params[:id])
  end
end
```

```
private
  def book_params
    params.require(:book).permit(:title)
  end
end
```

And add show.html.erb to app/views/books/ directory:

```
<p>Title: <%= @book.title %></p>
```

```
1 <p>Title: <%= @book.title %></p>
```

We should be able to add any book via web browser. Let's check the test result:

```
rspec spec/books/creating_book_spec.rb
```

```
1 rspec spec/books/creating_book_spec.rb
```

The result:

```
Finished in 0.94381 seconds (files took 4.4 seconds to load)
1 example, 0 failures
```

## Introduction to Writing Acceptance Tests with Cucumber

Improve your testing skills with acceptance testing. Cucumber makes you a better developer by helping you see your code through the eyes of the user.

Introduction

We're going to look at [Cucumber](#) as a tool for writing your [customer acceptance tests](#). More specifically, we're going to look at how a Cucumber acceptance test might look in practice. After reading this article, you should have a clearer picture of why Cucumber is a good candidate for writing your acceptance tests.

It's also worth mentioning that, [in a BDD fashion](#), you should start writing your acceptance test first, and it should drive your next steps, pushing you into deeper layers of testing and writing implementation code. So, as an example, your workflow should look similar to this:

1. Write your acceptance test
2. See it fail so you know what the next step is
3. Write a unit test for that next step
4. See it fail so you know what the implementation needs to be
5. Repeat steps 3 and 4 until you have everything you need, and all your tests (including the acceptance one) are passing
6. Rinse and repeat for every new feature.

If you would like to see the final code listed in the examples, there's a [GitHub repository available](#).

## Hello Cucumber Example

To get things started, we're going to look at a rather simple example, so you can familiarise yourself with the syntax and basic file structure:

```
# feature/hello_cucumber.feature
Feature: Hello Cucumber
As a product manager
I want our users to be greeted when they visit our site
So that they have a better experience
```

```
Scenario: User sees the welcome message
When I go to the homepage
Then I should see the welcome message
```

The first part starting with the keyword `Feature` is called a feature description. It needs to have a feature title, which is the string "Hello Cucumber" in our case. It can also have an optional description (the text underneath the title), which is meant to help the reader understand the feature and its context.

When writing your Cucumber features, it's good practice to follow the user story style, which looks like the following:

```
Feature: <feature title>
As a <persona|role>
I want to <action>
So that <outcome>
Steps and Step Definitions
```

The Cucumber feature we've written is readable, but how do we get it to do something? Does that plain text have anything to do with our code? Well,

those scenario instructions are called steps, and we're going to use them to drive our tests.

The way it works is, for each step in our scenario, we're going to provide a block of Ruby code to be executed. We're going to place our step definitions (the blocks of code) in a file called `hello_steps.rb`.

```
When(/^I go to the homepage$/) do
  visit root_path
end
```

```
Then(/^I should see the welcome message$/) do
  expect(page).to have_content("Hello Cucumber")
end
```

As you can see, we're simply associating each line in our Cucumber feature file, called a scenario step, with its corresponding step definition, matching the step definition string with the use of a regular expression.

So, in the first step definition, we're saying that, in order to go to the homepage, the user will visit the `root_path` (which is standard Rails terminology, and it's something we define in your `config/routes.rb` file). For the expectation step, we're going to check that the homepage contains the "Hello Cucumber" text.

```
# config/routes.rb
Rails.application.routes.draw do
  root 'welcome#index'
end
```

```
# app/controllers/welcome_controller.rb
```

```
class WelcomeController < ApplicationController
  def index
  end
end
# app/views/welcome/index.html.erb
<h1>Hello Cucumber</h1>
$ cucumber -s
Using the default profile...
Feature: Hello Cucumber
```

```
Scenario: User sees the welcome message
  When I go to the homepage
  Then I should see the welcome message
```

```
1 scenario (1 passed)
2 steps (2 passed)
0m0.168s
```

The `-s` flag tells Cucumber to hide the location of each step definition, which is the default behavior.

## Can I Test My JavaScript?

Cucumber lets you test your application from the user's perspective, and in some cases that means having to deal with JavaScript-driven user interface elements. Cucumber does this by starting a browser in the background, and doing what a real user would do by clicking on links, filling out forms, etc. You should not use Cucumber to unit test your JavaScript-driven code, but it's perfect for testing user interaction.

The default driver Cucumber uses through Capybara is called `:rack_test`. It has some limitations, mainly the fact that it does not support JavaScript, so

we'll need to add another driver that supports JavaScript, and use it for those features that require it.

We will use the `:rack_test` driver for all of our tests that don't depend on JavaScript because it's faster, as it doesn't have to open a web browser program. For tests that require JavaScript, we will use the Selenium driver. Selenium is based on launching and controlling an instance of your local Firefox browser, so you need to make sure you have Firefox installed.

Add the following line to your Gemfile's `:test` group:

```
gem 'selenium-webdriver'
```

Let's See a Cucumber and JavaScript Example

For this example, we're going to have a link that, when clicked, replaces the contents of the page with the string "Link Clicked" via Javascript. In order to differentiate between our regular (`rack_test` driven tests) and the ones that require JavaScript, we will use a [Cucumber tag](#). It looks like this: `@javascript`.

```
# features/link_click.feature
```

```
Feature: Link Click
```

```
@javascript
```

```
Scenario: User clicks the link
```

```
Given I am on the homepage
```

```
When I click the provided link
```

```
Then I should see the link click confirmation
```



Now that we have our feature, we need to add some step definitions. Note that in this case we have a new `Given` step, which sets the context in which the action (`When`) is triggered.

```
# features/step_definitions/link_click_steps.rb
Given(/^I am on the homepage$/) do
  visit root_path
end
```

```
When(/^I click the provided link$/) do
  click_on "js-click-me"
end
```

```
Then(/^I should see the link click confirmation$/) do
  expect(page).to have_content("Link Clicked")
end
```

For the `Given` step, we're going to do the same thing we did in our first example — we're going to visit the homepage. Next, we're going to click the link on the homepage, and finally we're going to check that the page contains the string "Link Clicked".

Let's also add the link to our homepage, so that we have something to click on:

```
<h1>Hello Cucumber</h1>
```

```
<%= link_to "Click Me", "", :id => "js-click-me" %>
```

The last missing piece of the puzzle is the JavaScript code that is going to listen for the `click` event on the link, and when it receives it, it will go ahead and replace the page contents with the "Link Clicked" string:

```
$(document).click("#js-click-me", function(event) {  
  event.preventDefault();  
  $("body").html("Link Clicked");  
});
```

Without the `@javascript` tag in the Cucumber feature file, this JavaScript code would never get executed, since it requires a JavaScript-aware browser.

## Why Not Use RSpec and Capybara Instead?

Just in case you were wondering, since [Capybara](#) seems to be a popular choice among Ruby on Rails developers, we're going to take a short detour and list a few things that make Cucumber a better choice for writing acceptance tests compared to using RSpec and Capybara directly together.

Choosing Cucumber over Capybara has a few benefits, some of which are less apparent at first sight. Note, though, that Cucumber uses Capybara behind the scenes, it's just that it provides a nice language abstraction layer on top of it.

- The most obvious reason for choosing Cucumber is that it seems to appeal to non-technical people, and it's said that, in an ideal world, the customer would be able to write the acceptance tests himself.
- The second, less obvious, but most important reason is the fact that it forces you (the developer) into business mode. It helps you switch gears for a second and look at your code architecture from a different point of view, one which helps you [plan and implement each feature systematically](#).

- Documentation is also another great benefit you can get as a side effect of writing your features in a language that is easier to read.

## Continuous Integration for Cucumber on Semaphore

By setting up [continuous integration](#) the tests you have written can run automatically on every git push you do.

[Semaphore](#) is a hosted CI service which comes with [all recent versions of Ruby preinstalled](#), so it takes minimum effort to get started.

First, [sign up for a free Semaphore account](#) if you don't have one already. All there's left to do is to [add your repository](#).

Semaphore will analyze your project and recommend commands for everything to run smoothly. Also, the cucumber job command we need will be added:

```
bundle exec rake cucumber
```

From now on, Semaphore will run your tests for you on every git push.

Capybara, aside from being [the largest rodent in the world](#), is also a fantastic tool to aid you in interacting with browser functionality in your code, either for testing or just to interact with or scrape data from a website.

Capybara isn't what actually interacts with the website — rather, it's a layer that sits between you and the actual web driver. This could be Selenium, PhantomJS, or any of the other drivers that Capybara supports. It provides a common interface and a large number of helper methods for extracting information, inputting data, testing, or clicking around.

Just like any abstraction, sometimes you need to go deeper, and Capybara won't stop you from doing that. You can easily bypass it to get at the underlying drivers if you need more fine-tuned functionality.

“Capybara is a fantastic tool to aid you in interacting with browser functionality in your code.”

## Testing with Capybara

Capybara integrates really nicely with all of the common test frameworks used with Rails. It has extensions for RSpec, Cucumber, Test::Unit, and Minitest. It's used mostly with integration (or feature) tests, which test not so much a single piece of functionality but rather an entire user flow.

“Capybara integrates really nicely with all of the common test frameworks used with Rails.”

You can use Capybara to test whether certain content exists on the page or to input data into a form and then submit it. This is where you try to ensure that the same key flows (such as registration, checkout, etc.) that your user will take work not just in isolation but flow nicely from one to another.

With RSpec, we need to first ensure that in our `rspec_helper.rb` file we include the line `require 'capybara/rails'`. Next, let's create a new folder called `features` where we'll put all of the tests which include Capybara.

Imagine that we have an application for managing coffee farms. In this application, creating a coffee farm is one of the most important functions you can perform, and therefore should be tested thoroughly.

```
# spec/features/creating_farm_spec.rb
require 'rails_helper'

RSpec.describe 'creating a farm', type: :feature do
  it 'successfully creates farm' do
    visit '/farms'
    click_link 'New Farm'

    within '#new_farm' do
      fill_in 'Acres', with: 10
      fill_in 'Name', with: 'Castillo Andino'
      fill_in 'Owner', with: 'Albita'
      fill_in 'Address', with: 'Andes, Colombia'
      fill_in 'Varieties', with: 'Colombia, Geisha, Bourbon'
      fill_in 'Rating', with: 10
    end
    click_button 'Create Farm'

    expect(page).to have_content 'Farm was successfully created.'
    expect(page).to have_content 'Castillo Andino'
  end
end
```

There are a few things to note with Capybara. The first is that it provides a ton of great helpers such as `click_link`, `fill_in`, `click_button`, etc. Many of these

helpers provide a variety of ways to actually find the HTML element that you're looking for.

In the example above, we see CSS selectors used with the `within` method. We also see selecting and filling in an `input` field by using the text in its `label`.

There's also a third way, not shown here, which allows you to select elements using `xpath`. While `xpath` is the most powerful for selecting, it's the least clear way. For the purposes of your own sanity, you should probably aim to include an `ID` or `class` property in your HTML to ensure that selecting is straightforward.

## Scraping with Capybara

Capybara isn't only for testing. It can also be used in web scraping. I'll admit that it isn't the fastest method, and if all you are doing is visiting a page to extract information without too much interaction with the DOM in terms of data input or clicking, it may not be the best approach. For that, you may want to investigate something like [mechanize](#) or even [nokogiri](#) if all you are doing is reading HTML to extract information from it.

["Capybara isn't only for testing." via @leighchalliday](#)

[CLICK TO TWEET](#)

But for the situation where you maybe have to first log in as a user, click on a tab, and then extract some information, this is the sweet spot for Capybara.

I've recently had to rent a car, and I ended up using [Hotwire](#) for this. Let's use Capybara to log in and retrieve my confirmation number. In this case, it would be more difficult to use a different scraping tool because it is an Angular SPA, so Capybara works perfectly.

I'll create a Rake task which will log in to my account and then loop through all of the confirmation codes and print them to the screen. I've used an `xpath` selector here to show that even if there isn't an easy CSS selector to

use, you can still find the element that you're looking for. This also demonstrates how to use Capybara outside of your testing environment.

```
namespace :automate do
  desc 'Grab hotwire confirmation code'
  task :hotwire => [:environment] do |t, args|
    session = Capybara::Session.new(:selenium)
    session.visit 'https://www.hotwire.com/checkout/#!/account/login'

    session.find('#sign-in-email').set(ENV.fetch('EMAIL'))
    session.find(:xpath, '//input[@type="password"]').set(ENV.fetch('PASSWORD'))
    session.find('.hw-btn-primary').click

    session.all('.confirmation-code').each do |code|
      puts code.text
    end
  end
end
```

On the screen, we get `Car confirmation 31233321CA3` outputted (not my real confirmation number, of course).

Any time we use the `find` method or `all`, we are given an instance of the `Capybara::Node::Element` object. This object allows us to `click` it, extract the `text`, ask for its DOM `path`, and interact with it in a variety of other ways.

One other interesting method is the `native` method, which returns us the underlying driver's object. In this case, it's an instance of `Selenium::WebDriver::Element` because we are using Selenium. As useful of an abstraction as Capybara is, there will always be times when you need to gain access to the underlying layer.

As you can see, this could be an easy way to automate a task that has no other alternative than to use the "Screen Scraping" approach. Keep in mind that this is quite brittle, as a slight change to one of their classes or IDs means that the whole thing will stop working.

### Interacting with JavaScript

One of the things that Capybara gives you is the ability to interact with your webpages using JavaScript. You aren't limited to only using Ruby to find and interact with the DOM nodes on your page.

Capybara gives you two methods to invoke some JavaScript:

```
execute_script (does not return values)
evaluate_script (does return values)
```

These work great, but as usual you can bypass Capybara if needed and use the underlying driver to execute JavaScript. This allows us to pass arguments to our JavaScript code:

```
# example of returning values from javascript
classes = session.driver.browser.execute_script(
  "return document.getElementById(arguments[0]).className;",
  'sign-in-password'
)
puts classes
# => ng-pristine ng-untouched ng-invalid ng-invalid-required
```

The `execute_script` method allows you to return values from JS functions which are called. If you imagine that your code is being invoked like this:

```
var result =
(function(arguments) {
  return document.getElementById(arguments[0]).className;
})(['sign-in-password']);
```

You'll see that the arguments you pass in the second and higher parameter positions get placed into an array and passed to an anonymous function. The anonymous function contains, as its body, the code which was in the first parameter. This is why you must explicitly include a `return` statement if you want to use the value it returns.

Selenium takes care of how to convert what would end up being a JavaScript return value into something you can use in Ruby.

## Configuring Capybara

As I mentioned in the introduction, Capybara works by allowing you to work with a number of different web drivers. These could be lightweight headless



drivers such as [PhantomJS](#) (via [poltergeist](#)) or RackTest, but it could also be [Selenium](#) either running locally or connecting to a Selenium grid server remotely.

Here is an example of how you might configure Capybara to work with a remote Selenium server.

```
Capybara.register_driver(:firefox) do |app|
  Capybara::Selenium::Driver.new(
    app,
    browser: :remote,
    url: ENV.fetch('SELENIUM_URL'),
    desired_capabilities: Selenium::WebDriver::Remote::Capabilities.firefox
  )
end
```

Which would now allow you to set the driver to `:firefox` with the code `Capybara.default_driver = :firefox`.

## Debugging Capybara

With a headless driver, it is sometimes hard to see what the page looks like at the time you are interacting with it. Just because it is headless doesn't mean you need to be blind. You are able to request a screenshot of how the page looks and also extract the source code of the page.

```
File.open('/tmp/source.html', 'w') do |file|
  io = StringIO.new(session.driver.browser.page_source)
  file.write(io.read)
end
```

```
File.open('/tmp/screenshot.png', 'w', encoding: 'ascii-8bit') do |file|
  io = StringIO.new(session.driver.browser.screenshot_as(:png))
  file.write(io.read)
end
```

Another way to help with debugging is by placing a `binding.pry` call, which will pause the script and allow you to step into the code and perform commands that interact with the web page. You can even open up the Firefox/Chrome developer console and play with the JavaScript of the page in its current state.

## Conclusion

The truth is that I don't generally use Capybara when testing my Rails applications. It slows the tests down and potentially binds your tests quite closely to the DOM.

However, it does have its place, especially when you need to guarantee that certain important user flows work exactly as expected. It also finds the odd use in scraping when tools such as `mechanize` or `nokogiri` aren't enough.

Capybara is a great tool to have in your Ruby toolbelt. Give it a try the next time you want to make sure that a certain key user flow works or you need to automate a task via scraping. You can also read about Selenium on this [Codeship Selenium documentation](#) article.

## Mocking in Ruby with Minitest

Mocking is used to improve the performance of your tests. This tutorial will show you how to use mocks and stubs in Ruby with Minitest.

Introduction

In this tutorial, we will cover how to use mocks and stubs in Minitest to improve the performance of your tests and avoid testing dependencies.

## Prerequisites

To follow this tutorial, you'll need Ruby installed along with Rails. This tutorial was tested using Ruby version 2.3.1, Rails version 5.0, and Minitest version 5.9.1.

Currently, there is no known issue with using earlier or later versions of any of those, however there will be some differences. Models inherit from `ActiveRecord::Base` instead of `ApplicationRecord`, which is the new default in Rails 5.0. We'll also demonstrate that `assert_mock` can be used to verify mocks as of Minitest 5.9, but that will not work with earlier versions where `assert_mock.verify` was the method used to verify mocks.

To get started you can use `gem install rails`, and you should be good to go, provided you have Ruby installed.

```
gem install rails
```

## What is Minitest?

Minitest is a complete testing suite for Ruby, supporting test-driven development (TDD), [behavior-driven development \(BDD\)](#), mocking, and benchmarking. It's small, fast, and it aims to make tests clean and readable.

If you're new to Minitest, then you can take a look at our tutorial on [getting started with Minitest](#).

Minitest is the default testing suite which is included by default with new Rails applications, so no further setting up is required to get it to work. Minitest and RSpec are the two most common testing suites used in Ruby. If you'd like to learn more about RSpec, you can read our tutorial on [getting started with RSpec](#) as well as this tutorial on [mocking with RSpec: doubles and expectations](#).

## Test Doubles and Terminology

The terminology surrounding mocks and stubs can be a bit confusing. The main terms you might come across are stubs, mocks, doubles, dummies, fakes, and spies.

The umbrella term for all of these is double. A test double is any object used in testing to replace a real object used in production. We'll cover dummies, stubs, and mocks, because they are the ones used commonly in Minitest.

### Dummies

The simplest of these terms is a dummy. It refers to a test double that is passed in a method call, but never actually used. Much of the time, the purpose of these is to avoid `ArgumentError` in Ruby.

Minitest does not have a feature for dummies, because it isn't really needed. You can pass in `Object.new` (or anything else) as a placeholder.

## Stubs

Stubs are like dummies, except in that they provide canned answers to the methods which are called on them. They return hard coded information in order to reduce test dependencies and avoid time consuming operations.

## Mocks

Mocks are "smart" stubs, their purpose is to verify that some method was called. They are created with some expectations (expected method calls) and can then be verified to ensure those methods were called.

## Mocks and Stubs

The easiest way to understand mocks and stubs is by example. Let's set up a Rails project and add some code that we can use mocks and stubs to test.

For this example, we'll create user and subscription models with a subscription service which can be used to create or extend subscriptions.

Assuming you have Ruby and Ruby on Rails set up, we can start by creating our Rails application.

```
rails new mocking-in-ruby-with-minitest
```

Now, let's add our user model and tests by using the Rails generator:

```
rails g model user name:string
Running via Spring preloader in process 14377
  invoke active_record
  create db/migrate/20161017213701_create_users.rb
  create app/models/user.rb
  invoke test_unit
  create test/models/user_test.rb
  create test/fixtures/users.yml
```

Next, let's create the model for subscriptions which has a reference to the user model:

```
rails g model subscription expires_at:date user:references
Running via Spring preloader in process 15028
  invoke active_record
  create db/migrate/20161017214307_create_subscriptions.rb
  create app/models/subscription.rb
  invoke test_unit
  create test/models/subscription_test.rb
  create test/fixtures/subscriptions.yml
```

Then, migrate the database:

```
rake db:migrate
```

Finally, let's create a service which creates and manages subscriptions. Start by adding a reference from User to Subscription.

```
class User < ApplicationRecord
  has_one :subscription
end
```

Now, let's add our subscription service tests. To keep things simple, we don't test that the `expires_at` attribute is always correct.

```
# test/services/subscription_service_test.rb
```

```
require 'test_helper'
```

```
class SubscriptionServiceTest < ActiveSupport::TestCase
```

```
  test '#create_or_extend new subscription' do
```

```
    user = users :no_sub
```

```
    subscription_service = SubscriptionService.new user
```

```
    assert_difference 'Subscription.count' do
```

```
      assert subscription_service.apply
```

```
    end
```

```
  end
```

```
  test '#create_or_extend existing subscription' do
```

```
    user = users :one
```

```
    subscription_service = SubscriptionService.new user
```

```
    assert_no_difference 'Subscription.count' do
```

```
      assert subscription_service.apply
```

```
    end
```

```
  end
```

```
end
```

Let's also add a user fixture for the user which has no subscriptions. Add the following two lines to the user fixture file:

```
no_sub:
```

```
  name: No Subscription
```

Now, let's make our test pass by adding SubscriptionService.

```
# app/services/subscription_service.rb
```

```
class SubscriptionService
```

```
  SUBSCRIPTION_LENGTH = 1.month
```

```
  def initialize(user)
```

```
    @user = user
```

```
  end
```

```
  def apply
```

```
if Subscription.exists?(user_id: @user.id)
  extend_subscription
else
  create_subscription
end
end
```

```
private
```

```
def create_subscription
  subscription = Subscription.new(
    user: @user,
    expires_at: SUBSCRIPTION_LENGTH.from_now
  )
```

```
  subscription.save
end
```

```
def extend_subscription
  subscription = Subscription.find_by user_id: @user.id
```

```
    subscription.expires_at = subscription.expires_at +
SUBSCRIPTION_LENGTH
  subscription.save
end
end
```

Now, run the tests to make sure everything is passing.

```
rake
```

```
Running via Spring preloader in process 19998
Run options: --seed 23654
```

```
# Running:
```

```
..
```



Finished in 0.083658s, 23.9069 runs/s, 23.9069 assertions/s.

2 runs, 2 assertions, 0 failures, 0 errors, 0 skips

Note that `app/services` and `test/services` do not exist by default so you will have to create them.

Great! We're now ready to add some functionality, which we can benefit from by using mocks and stubs in the tests.

## Stubbing

Stubbing is useful when we want to replace a dependency method which takes a long time to run with another method that has the return value we expect.

However, it's usually not a good idea to do this if the method belongs to the class you are testing, because then you're replacing the method you should be testing with a stub. It's fine to do this for methods of other classes that have their own tests already, but are called from the class we are testing.

Let's add a method to `User` called `#apply_subscription`. This method will call `SubscriptionService` to apply the subscription. In this case, we have already tested the subscription service, so we don't need to do that again. Instead, we can just make sure it is called with a combination of stubbing and mocking.

In order to create mocks, we also need to load Minitest in `test_helper.rb`. Add this requirecall to the ones in `test_helper.rb`:

```
# test/test_helper.rb
```

```
require 'minitest/autorun'
```

Now, let's add tests where we use a mock to mock `SubscriptionService` and stub `#apply` to just return true without ever calling the real `SubscriptionService`.

```
# test/models/user_test.rb
```

```
require 'test_helper'
```

```
class UserTest < ActiveSupport::TestCase
```

```
  test '#apply_subscription' do
```

```
    mock = Minitest::Mock.new
```

```
    def mock.apply; true; end
```

```
    SubscriptionService.stub :new, mock do
```

```
      user = users(:one)
```

```
      assert user.apply_subscription
```

```
    end
```

```
  end
```

```
end
```

Since we have already tested `SubscriptionService`, we don't need to do it again. That way, we don't have to worry about the setup and the overhead of accessing the database, which makes our test faster and simpler.

Now, let's add the code to make the test pass.

```
# app/models/user.rb
```

```
class User < ApplicationRecord
  has_one :subscription

  def apply_subscription
    SubscriptionService.new(self).apply
  end
end
```

Although we have demonstrated how stubbing works here, we are not really testing anything, to do that we need to make full use of mocks.

## Mocking

One of the core functionalities of mocks is to be able to verify that we called a method that we stubbed. Sometimes this isn't something we want to, however a lot of the time, we want to make sure we called some method, but we don't care to test if it works or not, because it's already been tested.

Let's change our test to verify that `SubscriptionService#apply` was called, even though it calls our stub instead of the real thing.

```
# test/models/user_test.rb

require 'test_helper'

class UserTest < ActiveSupport::TestCase
  test '#apply_subscription' do
    mock = Minitest::Mock.new
    mock.expect :apply, true

    SubscriptionService.stub :new, mock do
      user = users(:one)
    end
  end
end
```

```
assert user.apply_subscription
end
```

```
assert_mock mock # New in Minitest 5.9.0
assert mock.verify # Old way of verifying mocks
end
end
```

Note how we tell our mock what method call we are expecting along with the return value. It's possible to pass in a third argument, which is an array of arguments that the method is expected to receive. This needs to be included if the method has any arguments passed to it in the method call.

## Stubbing Constants

Sometimes we want to be able to change the return value of calling a constant in a class from a test. If you're coming from RSpec, you might be used to having this feature in your toolbelt. However, Minitest doesn't ship with such a feature.

There's a gem which provides this functionality for Minitest called [minitest-stub-const](#).

It can be quite useful when you want to change the value of a constant in your class, e.g when you need to test some numerical limits. One common use is results per page in pagination. If you have 25 results per page set in a constant, it can be easier to stub that constant to return 2, reducing the setup required to test your pagination.

## Overusing Mocks or Stubs

It's possible to overuse mocks or stubs, and it's important to be careful and avoid doing that. For example, if we stubbed the test for `SubscriptionService` in order to just return some data instead of opening a real file and performing the search on it, we wouldn't actually know if `SubscriptionService` works.

This is a rather obvious case. However, there are more subtle scenarios where mocks and stubs should be avoided.

## RSpec Mocks

`rspec-mocks` is a test-double framework for `rspec` with support for method stubs, fakes, and message expectations on generated test-doubles and real objects alike.

### Install

```
gem install rspec      # for rspec-core, rspec-expectations, rspec-mocks
gem install rspec-mocks # for rspec-mocks only
```

Want to run against the master branch? You'll need to include the dependent RSpec repos as well. Add the following to your Gemfile:

```
%w[rspec-core rspec-expectations rspec-mocks rspec-support].each do |lib|
  gem lib, :git => "https://github.com/rspec/#{lib}.git", :branch => 'master'
end
```

### Contributing

Once you've set up the environment, you'll need to `cd` into the working directory of whichever repo you want to work in. From there you can run the specs and cucumber features, and make patches.

NOTE: You do not need to use `rspec-dev` to work on a specific RSpec repo. You can treat each RSpec repo as an independent project.

For information about contributing to RSpec, please refer to the following markdown files:

- [Build details](#)
- [Code of Conduct](#)
- [Detailed contributing guide](#)
- [Development setup guide](#)

## Test Doubles

A test double is an object that stands in for another object in your system during a code example. Use the `double` method, passing in an optional identifier, to create one:

```
book = double("book")
```

Most of the time you will want some confidence that your doubles resemble an existing object in your system. Verifying doubles are provided for this purpose. If the existing object is available, they will prevent you from adding stubs and expectations for methods that do not exist or that have an invalid number of parameters.

```
book = instance_double("Book", :pages => 250)
```

Verifying doubles have some clever tricks to enable you to both test in isolation without your dependencies loaded while still being able to validate them against real objects. More detail is available in [their documentation](#).

Verifying doubles can also accept custom identifiers, just like `double()`, e.g.:

```
books = []  
books << instance_double("Book", :rspec_book, :pages => 250)  
books << instance_double("Book", "(Untitled)", :pages => 5000)
```

```
puts books.inspect # with names, it's clearer which were actually added
```

## Method Stubs

A method stub is an implementation that returns a pre-determined value. Method stubs can be declared on test doubles or real objects using the same syntax. `rspec-mocks` supports 3 forms for declaring method stubs:

```
allow(book).to receive(:title) { "The RSpec Book" }
allow(book).to receive(:title).and_return("The RSpec Book")
allow(book).to receive_messages(
  :title => "The RSpec Book",
  :subtitle => "Behaviour-Driven Development with RSpec, Cucumber, and Friends")
```

You can also use this shortcut, which creates a test double and declares a method stub in one statement:

```
book = double("book", :title => "The RSpec Book")
```

The first argument is a name, which is used for documentation and appears in failure messages. If you don't care about the name, you can leave it out, making the combined instantiation/stub declaration very terse:

```
double(:foo => 'bar')
```

This is particularly nice when providing a list of test doubles to a method that iterates through them:

```
order.calculate_total_price(double(:price => 1.99), double(:price => 2.99))
```

## Stubbing a chain of methods

You can use `receive_message_chain` in place of `receive` to stub a chain of messages:

```
allow(double).to receive_message_chain("foo.bar") { :baz }
allow(double).to receive_message_chain(:foo, :bar => :baz)
allow(double).to receive_message_chain(:foo, :bar) { :baz }
```

# Given any of the above forms:

```
double.foo.bar # => :baz
```

Chains can be arbitrarily long, which makes it quite painless to violate the Law of Demeter in violent ways, so you should consider any use of `receive_message_chain` a code smell. Even though not all code smells indicate real problems (think fluent interfaces), `receive_message_chain` still results in brittle examples. For example, if you write `allow(foo).to receive_message_chain(:bar, :baz => 37)` in a spec and then the implementation calls `foo.baz.bar`, the stub will not work.

## Consecutive return values

When a stub might be invoked more than once, you can provide additional arguments to `and_return`. The invocations cycle through the list. The last value is returned for any subsequent invocations:

```
allow(die).to receive(:roll).and_return(1, 2, 3)
die.roll # => 1
die.roll # => 2
die.roll # => 3
die.roll # => 3
die.roll # => 3
```

To return an array in a single invocation, declare an array:

```
allow(team).to receive(:players).and_return([double(:name => "David")])
```

## Message Expectations

A message expectation is an expectation that the test double will receive a message some time before the example ends. If the message is received, the expectation is satisfied. If not, the example fails.

```
validator = double("validator")
expect(validator).to receive(:validate) { "02134" }
zipcode = Zipcode.new("02134", validator)
zipcode.valid?
```

## Test Spies

Verifies the given object received the expected message during the course of the test. For a message to be verified, the given object must be setup to spy



on it, either by having it explicitly stubbed or by being a null object double (e.g. `double(...).as_null_object`). Convenience methods are provided to easily create null object doubles for this purpose:

```
spy("invitation") # => same as `double("invitation").as_null_object`  
instance_spy("Invitation") # => same as  
`instance_double("Invitation").as_null_object`  
class_spy("Invitation") # => same as  
`class_double("Invitation").as_null_object`  
object_spy("Invitation") # => same as  
`object_double("Invitation").as_null_object`
```

Verifying messages received in this way implements the Test Spy pattern.

```
invitation = spy('invitation')
```

```
user.accept_invitation(invitation)
```

```
expect(invitation).to have_received(:accept)
```

# You can also use other common message expectations. For example:

```
expect(invitation).to have_received(:accept).with(mailer)
```

```
expect(invitation).to have_received(:accept).twice
```

```
expect(invitation).to_not have_received(:accept).with(mailer)
```

# One can specify a return value on the spy the same way one would a double.

```
invitation = spy('invitation', :accept => true)
```

```
expect(invitation).to have_received(:accept).with(mailer)
```

```
expect(invitation.accept).to eq(true)
```

Note that `have_received(...).with(...)` is unable to work properly when passed arguments are mutated after the spy records the received message. For example, this does not work properly:

```
greeter = spy("greeter")
```

```
message = "Hello"
```

```
greeter.greet_with(message)
```

```
message << ", World"
```

```
expect(greeter).to have_received(:greet_with).with("Hello")
```

## Nomenclature

### Mock Objects and Test Stubs

The names Mock Object and Test Stub suggest specialized Test Doubles. i.e. a Test Stub is a Test Double that only supports method stubs, and a Mock Object is a Test Double that supports message expectations and method stubs.

There is a lot of overlapping nomenclature here, and there are many variations of these patterns (fakes, spies, etc). Keep in mind that most of the time we're talking about method-level concepts that are variations of method stubs and message expectations, and we're applying to them to one generic kind of object: a Test Double.

### Test-Specific Extension

a.k.a. Partial Double, a Test-Specific Extension is an extension of a real object in a system that is instrumented with test-double like behaviour in the context of a test. This technique is very common in Ruby because we often see class objects acting as global namespaces for methods. For example, in Rails:

```
person = double("person")  
allow(Person).to receive(:find) { person }
```

In this case we're instrumenting Person to return the person object we've defined whenever it receives the find message. We can also set a message expectation so that the example fails if find is not called:

```
person = double("person")  
expect(Person).to receive(:find) { person }
```

RSpec replaces the method we're stubbing or mocking with its own test-double-like method. At the end of the example, RSpec verifies any message expectations, and then restores the original methods.

### Expecting Arguments

```
expect(double).to receive(:msg).with(*args)
expect(double).to_not receive(:msg).with(*args)
```

You can set multiple expectations for the same message if you need to:

```
expect(double).to receive(:msg).with("A", 1, 3)
expect(double).to receive(:msg).with("B", 2, 4)
```

## Argument Matchers

Arguments that are passed to `with` are compared with actual arguments received using `===`. In cases in which you want to specify things about the arguments rather than the arguments themselves, you can use any of the matchers that ship with `rspec-expectations`. They don't all make syntactic sense (they were primarily designed for use with `RSpec::Expectations`), but you are free to create your own custom `RSpec::Matchers`.

`rspec-mocks` also adds some keyword Symbols that you can use to specify certain kinds of arguments:

```
expect(double).to receive(:msg).with(no_args)
expect(double).to receive(:msg).with(any_args)
expect(double).to receive(:msg).with(1, any_args) # any_args acts like an arg
splat and can go anywhere
expect(double).to receive(:msg).with(1, kind_of(Numeric), "b") #2nd argument
can be any kind of Numeric
expect(double).to receive(:msg).with(1, boolean(), "b") #2nd argument can be
true or false
expect(double).to receive(:msg).with(1, /abc/, "b") #2nd argument can be any
String matching the submitted Regexp
expect(double).to receive(:msg).with(1, anything(), "b") #2nd argument can
be anything at all
expect(double).to receive(:msg).with(1, duck_type(:abs, :div), "b") #2nd
argument can be object that responds to #abs and #div
expect(double).to receive(:msg).with(hash_including(:a => 5)) # first arg is a
hash with a: 5 as one of the key-values
expect(double).to receive(:msg).with(array_including(5)) # first arg is an array
with 5 as one of the key-values
expect(double).to receive(:msg).with(hash_excluding(:a => 5)) # first arg is a
hash without a: 5 as one of the key-values
```

## Receive Counts

```
expect(double).to receive(:msg).once
expect(double).to receive(:msg).twice
expect(double).to receive(:msg).exactly(n).times
expect(double).to receive(:msg).at_least(:once)
expect(double).to receive(:msg).at_least(:twice)
expect(double).to receive(:msg).at_least(n).times
expect(double).to receive(:msg).at_most(:once)
expect(double).to receive(:msg).at_most(:twice)
expect(double).to receive(:msg).at_most(n).times
```

## Ordering

```
expect(double).to receive(:msg).ordered
expect(double).to receive(:other_msg).ordered
# This will fail if the messages are received out of order
This can include the same message with different arguments:
```

```
expect(double).to receive(:msg).with("A", 1, 3).ordered
expect(double).to receive(:msg).with("B", 2, 4).ordered
```

## Setting Responses

Whether you are setting a message expectation or a method stub, you can tell the object precisely how to respond. The most generic way is to pass a block to receive:

```
expect(double).to receive(:msg) { value }
```

When the double receives the `msg` message, it evaluates the block and returns the result.

```
expect(double).to receive(:msg).and_return(value)
expect(double).to receive(:msg).exactly(3).times.and_return(value1, value2,
value3)
# returns value1 the first time, value2 the second, etc
expect(double).to receive(:msg).and_raise(error)
# error can be an instantiated object or a class
```

```
# if it is a class, it must be instantiable with no args
expect(double).to receive(:msg).and_throw(:msg)
expect(double).to receive(:msg).and_yield(values, to, yield)
expect(double).to receive(:msg).and_yield(values, to, yield).and_yield(some,
other, values, this, time)
# for methods that yield to a block multiple times
Any of these responses can be applied to a stub as well
```

```
allow(double).to receive(:msg).and_return(value)
allow(double).to receive(:msg).and_return(value1, value2, value3)
allow(double).to receive(:msg).and_raise(error)
allow(double).to receive(:msg).and_throw(:msg)
allow(double).to receive(:msg).and_yield(values, to, yield)
allow(double).to receive(:msg).and_yield(values, to, yield).and_yield(some,
other, values, this, time)
```

## Arbitrary Handling

Once in a while you'll find that the available expectations don't solve the particular problem you are trying to solve. Imagine that you expect the message to come with an Array argument that has a specific length, but you don't care what is in it. You could do this:

```
expect(double).to receive(:msg) do |arg|
  expect(arg.size).to eq 7
end
```

If the method being stubbed itself takes a block, and you need to yield to it in some special way, you can use this:

```
expect(double).to receive(:msg) do |&arg|
  begin
    arg.call
  ensure
    # cleanup
  end
end
```

## Delegating to the Original Implementation

When working with a partial mock object, you may occasionally want to set a message expectation without interfering with how the object responds to the message. You can use `and_call_original` to achieve this:

```
expect(Person).to receive(:find).and_call_original
Person.find # => executes the original find method and returns the result
```

## Combining Expectation Details

Combining the message name with specific arguments, receive counts and responses you can get quite a bit of detail in your expectations:

```
expect(double).to receive(:<<).with("illegal
value").once.and_raise(ArgumentError)
```

While this is a good thing when you really need it, you probably don't really need it! Take care to specify only the things that matter to the behavior of your code.

## Stubbing and Hiding Constants

See the [mutating constants README](#) for info on this feature.

Use `before(:example)`, not `before(:context)`

Stubs in `before(:context)` are not supported. The reason is that all stubs and mocks get cleared out after each example, so any stub that is set in `before(:context)` would work in the first example that happens to run in that group, but not for any others.

Instead of `before(:context)`, use `before(:example)`.

Settings mocks or stubs on any instance of a class

`rspec-mocks` provides two methods, `allow_any_instance_of` and `expect_any_instance_of`, that will allow you to stub or mock any instance of a class. They are used in place of `allow` or `expect`:

```
allow_any_instance_of(Widget).to receive(:name).and_return("Wibble")
expect_any_instance_of(Widget).to receive(:name).and_return("Wobble")
```

These methods add the appropriate stub or expectation to all instances of `Widget`.

This feature is sometimes useful when working with legacy code, though in general we discourage its use for a number of reasons:

- The `rspec-mocks` API is designed for individual object instances, but this feature operates on entire classes of objects. As a result there are some semantically confusing edge cases. For example in `expect_any_instance_of(Widget).to receive(:name).twice` it isn't clear whether each specific instance is expected to receive `name` twice, or if two receives total are expected. (It's the former.)
- Using this feature is often a design smell. It may be that your test is trying to do too much or that the object under test is too complex.
- It is the most complicated feature of `rspec-mocks`, and has historically received the most bug reports. (None of the core team actively use it, which doesn't help.)

## Action Mailer

Action Mailer is the Rails component that enables applications to send and receive emails. In this chapter, we will see how to send an email using Rails. Let's start creating an emails project using the following command.

```
tp> rails new mailtest
```

This will create the required framework to proceed. Now, we will start with configuring the ActionMailer.

### Action Mailer - Configuration

Following are the steps you have to follow to complete your configuration before proceeding with the actual work –

Go to the config folder of your emails project and open `environment.rb` file and add the following line at the bottom of this file.

```
config.action_mailer.delivery_method = :smtp
```

It tells ActionMailer that you want to use the SMTP server. You can also set it to be `:sendmail` if you are using a Unix-based operating system such as Mac OS X or Linux.

Add the following lines of code at the bottom of your `environment.rb` as well.

```
config.action_mailer.smtp_settings = {  
  address:      'smtp.gmail.com',  
  port:        587,  
  domain:      'example.com',  
  user_name:   '<username>',  
  password:    '<password>',  
  authentication: 'plain',  
  enable_starttls_auto: true  
}
```

Replace each hash value with proper settings for your Simple Mail Transfer Protocol

SMTP server. You can take this information from your Internet Service Provider if you already don't know. You don't need to change port number 25 and authentication type if you are using a standard SMTP server.

You may also change the default email message format. If you prefer to send email in HTML instead of plain text format, add the following line to `config/environment.rb` as well –

```
ActionMailer::Base.default_content_type = "text/html"
```

`ActionMailer::Base.default_content_type` could be set to `"text/plain"`, `"text/html"`, and `"text/enriched"`. The default value is `"text/plain"`.

The next step will be to create a mailer

### Generate a Mailer

Use the following command to generate a mailer as follows –

```
tp> cd emails
```

```
emails> rails generate mailer Usermailer
```

This will create a file `user_mailer.rb` in the `app\mailer` directory. Check the content of this file as follows –

```
class Emailer < ActionMailer::Base
```

```
end
```



Let's create one method as follows –

```
class UserMailer < ApplicationMailer
  default from: 'notifications@example.com'

  def welcome_email(user)
    @user = user
    @url = 'http://www.gmail.com'
    mail(to: @user.email, subject: 'Welcome to My Awesome Site')
  end
end
```

end

- default Hash – This is a hash of default values for any email you send from this mailer. In this case we are setting the :from header to a value for all messages in this class. This can be overridden on a per-email basis
- mail – The actual email message, we are passing the :to and :subject headers in.

Create a file called welcome\_email.html.erb in app/views/user\_mailer/. This will be the template used for the email, formatted in HTML –

```
<html>
```

```
<head>
  <meta content = 'text/html; charset = UTF-8' http-equiv = 'Content-Type' /
>
</head>
```

```
<body>
  <h1>Welcome to example.com, <%= @user.name %></h1>
```

```
<p>
  You have successfully signed up to example.com,your username is:
  <%= @user.login %>.<br>
</p>
```

```
<p>
  To login to the site, just follow this link:
  <%= @url %>.
</p>
```

```
<p>Thanks for joining and have a great day!</p>
```

```
</body>
```

```
</html>
```

Next we will create a text part for this application as follow –

```
Welcome to example.com, <%= @user.name %>
```

```
=====
```

```
You have successfully signed up to example.com,  
your username is: <%= @user.login %>.
```

```
To login to the site, just follow this link: <%= @url %>.
```

```
Thanks for joining and have a great day!
```

```
Calling the Mailer
```

```
First, let's create a simple User scaffold
```

```
$ bin/rails generate scaffold user name email login
```

```
$ bin/rake db:migrate
```

Action Mailer is nicely integrated with Active Job so you can send emails outside of the request-response cycle, so the user doesn't have to wait on it –

```
class UsersController < ApplicationController
```

```
  # POST /users
```

```
  # POST /users.json
```

```
  def create
```

```
    @user = User.new(params[:user])
```

```
    respond_to do |format|
```

```
      if @user.save
```

```
        # Tell the UserMailer to send a welcome email after save
```

```
        UserMailer.welcome_email(@user).deliver_later
```

```
        format.html { redirect_to(@user, notice: 'User was successfully  
created.') }
```

```
        format.json { render json: @user, status: :created, location: @user }
```

```
      else
```

```
        format.html { render action: 'new' }
```

```
        format.json { render json: @user.errors,  
status: :unprocessable_entity }
```

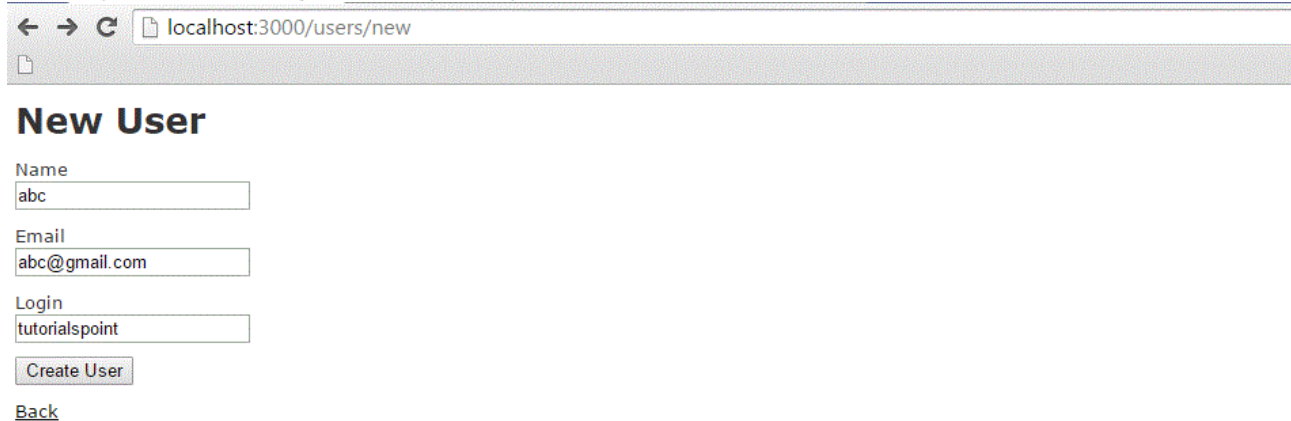
```
      end
```

```
    end
```

```
  end
```

```
end
```

Now, test your application by using `http://127.0.0.1:3000/users/new`. It displays the following screen and by using this screen, you will be able to send your message to anybody.



← → ↻ localhost:3000/users/new

## New User

Name

Email

Login

[Back](#)

This will send your message and will display the text message "Message sent successfully" and output as follow –

```
sent mail to surendra.panpaliya@gmail.com (2023.Sms)
[ActiveJob] [ActionMailer::DeliveryJob] [2cfde3c-260e-4a33-1a6ada13a9b]
Date: Thu, 09 Jul 2015 11:44:05 +0530
From: notification@example.com
To: surendra@gmail.com
Message-Id: <559e112d63c57_f1031e7f23467@kiranPro.mail>
Subject: Welcome to My Awesome Site
Mime-Version: 1.0
Content-Type: multipart/alternative;
boundary="--mimepart_559e112d601c8_f1031e7f20233f5";
charset=UTF-8
Content-Transfer-Encoding:7bit
```

## 1 Introduction

Action Mailer allows you to send emails from your application using mailer classes and views. Mailers work very similarly to controllers. They inherit from `ActionMailer::Base` and live in `app/mailers`, and they have associated views that appear in `app/views`.

## 2 Sending Emails

This section will provide a step-by-step guide to creating a mailer and its views.

### 2.1 Walkthrough to Generating a Mailer

#### 2.1.1 Create the Mailer

```
$ bin/rails generate mailer UserMailer
create  app/mailers/user_mailer.rb
create  app/mailers/application_mailer.rb
invoke  erb
create  app/views/user_mailer
create  app/views/layouts/mailer.text.erb
create  app/views/layouts/mailer.html.erb
invoke  test_unit
create  test/mailers/user_mailer_test.rb
create  test/mailers/preview/user_mailer_preview.rb
```

```
# app/mailers/application_mailer.rb
class ApplicationMailer < ActionMailer::Base
  default from: "from@example.com"
  layout 'mailer'
end

# app/mailers/user_mailer.rb
class UserMailer < ApplicationMailer
end
```

As you can see, you can generate mailers just like you use other generators with Rails. Mailers are conceptually similar to controllers, and so we get a mailer, a directory for views, and a test.

If you didn't want to use a generator, you could create your own file inside of `app/mailers`, just make sure that it inherits from `ActionMailer::Base`:

```
class MyMailer < ActionMailer::Base
end
```

### 2.1.2 Edit the Mailer

Mailers are very similar to Rails controllers. They also have methods called "actions" and use views to structure the content. Where a controller generates content like HTML to send back to the client, a Mailer creates a message to be delivered via email.

`app/mailers/user_mailer.rb` contains an empty mailer:

```
class UserMailer < ApplicationMailer
end
```

Let's add a method called `welcome_email`, that will send an email to the user's registered email address:

```
class UserMailer < ApplicationMailer
  default from: 'notifications@example.com'

  def welcome_email
    @user = params[:user]
    @url = 'http://example.com/login'
    mail(to: @user.email, subject: 'Welcome to My
    Awesome Site')
  end
end
```

Here is a quick explanation of the items presented in the preceding method. For a full list of all available options, please have a look further down at the [Complete List of Action Mailer user-settable attributes](#) section.

- **default Hash** - This is a hash of default values for any email you send from this mailer. In this case we are setting the `:from` header to a value for all messages in this class. This can be overridden on a per-email basis.

- mail - The actual email message, we are passing the :to and :subject headers in.

Just like controllers, any instance variables we define in the method become available for use in the views.

### 2.1.3 Create a Mailer View

Create a file called `welcome_email.html.erb` in `app/views/user_mailer/`. This will be the template used for the email, formatted in HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <meta content='text/html; charset=UTF-8' http-
equiv='Content-Type' />
  </head>
  <body>
    <h1>Welcome to example.com, <%= @user.name %></h1>
    <p>
      You have successfully signed up to example.com,
      your username is: <%= @user.login %>.<br>
    </p>
    <p>
      To login to the site, just follow this link: <%=
@url %>.
    </p>
    <p>Thanks for joining and have a great day!</p>
  </body>
</html>
```

Let's also make a text part for this email. Not all clients prefer HTML emails, and so sending both is best practice. To do this, create a file called `welcome_email.text.erb` in `app/views/user_mailer/`:

```
Welcome to example.com, <%= @user.name %>
=====

You have successfully signed up to example.com,
your username is: <%= @user.login %>.

To login to the site, just follow this link: <%= @url
%>.

Thanks for joining and have a great day!
```

When you call the mail method now, Action Mailer will detect the two templates (text and HTML) and automatically generate a multipart/alternative email.

#### 2.1.4 Calling the Mailer

Mailers are really just another way to render a view. Instead of rendering a view and sending it over the HTTP protocol, they are just sending it out through the email protocols instead. Due to this, it makes sense to just have your controller tell the Mailer to send an email when a user is successfully created.

Setting this up is painfully simple.

First, let's create a simple User scaffold:

```
$ bin/rails generate scaffold user name email login
$ bin/rails db:migrate
```

Now that we have a user model to play with, we will just edit the `app/controllers/users_controller.rb` make it instruct the `UserMailer` to deliver an email to the newly created user by editing the create action and inserting a call to `UserMailer.with(user: @user).welcome_email` right after the user is successfully saved.

Action Mailer is nicely integrated with Active Job so you can send emails outside of the request-response cycle, so the user doesn't have to wait on it:

```

class UsersController < ApplicationController
  # POST /users
  # POST /users.json
  def create
    @user = User.new(params[:user])

    respond_to do |format|
      if @user.save
        # Tell the UserMailer to send a welcome email
        after save
          UserMailer.with(user:
@user).welcome_email.deliver_later

        format.html { redirect_to(@user, notice: 'User
was successfully created.' ) }
        format.json { render json: @user,
status: :created, location: @user }
      else
        format.html { render action: 'new' }
        format.json { render json: @user.errors, status:
:unprocessable_entity }
      end
    end
  end
end
end
end

```

Active Job's default behavior is to execute jobs via the `:async` adapter. So, you can use `deliver_later` now to send emails asynchronously. Active Job's default adapter runs jobs with an in-process thread pool. It's well-suited for the development/test environments, since it doesn't require any external infrastructure, but it's a poor fit for production since it drops pending jobs on restart. If you need a persistent backend, you will need to use an Active Job adapter that has a persistent backend (Sidekiq, Resque, etc).

If you want to send emails right away (from a cronjob for example) just call `deliver_now`:



```
class SendWeeklySummary
  def run
    User.find_each do |user|
      UserMailer.with(user:
user).weekly_summary.deliver_now
    end
  end
end
```

Any key value pair passed to `with` just becomes the params for the mailer action. So `with(user: @user, account: @user.account)` makes `params[:user]` and `params[:account]` available in the mailer action. Just like controllers have params.

The method `welcome_email` returns an `ActionMailer::MessageDelivery` object which can then just be told `deliver_now` or `deliver_later` to send itself out. The `ActionMailer::MessageDelivery` object is just a wrapper around a `Mail::Message`. If you want to inspect, alter or do anything else with the `Mail::Message` object you can access it with the `message` method on the `ActionMailer::MessageDelivery` object.

## 2.2 Auto encoding header values

Action Mailer handles the auto encoding of multibyte characters inside of headers and bodies.

For more complex examples such as defining alternate character sets or self-encoding text first, please refer to the [Mail](#) library.

## 2.3 Complete List of Action Mailer Methods

There are just three methods that you need to send pretty much any email message:

- `headers` - Specifies any header on the email you want. You can pass a hash of header field names and value pairs, or you can call `headers[:field_name] = 'value'`.
- `attachments` - Allows you to add attachments to your email. For example, `attachments['file-name.jpg'] = File.read('file-name.jpg')`.

- mail - Sends the actual email itself. You can pass in headers as a hash to the mail method as a parameter, mail will then create an email, either plain text, or multipart, depending on what email templates you have defined.

### 2.3.1 Adding Attachments

Action Mailer makes it very easy to add attachments.

- Pass the file name and content and Action Mailer and the [Mail gem](#) will automatically guess the mime\_type, set the encoding and create the attachment.

```
• attachments['filename.jpg'] = File.read('/path/to/  
filename.jpg')
```

When the mail method will be triggered, it will send a multipart email with an attachment, properly nested with the top level being multipart/mixed and the first part being a multipart/alternative containing the plain text and HTML email messages.

Mail will automatically Base64 encode an attachment. If you want something different, encode your content and pass in the encoded content and encoding in a Hash to the attachments method.

- Pass the file name and specify headers and content and Action Mailer and Mail will use the settings you pass in.

```
• encoded_content = SpecialEncode(File.read('/path/to/  
filename.jpg'))  
attachments['filename.jpg'] = {  
  mime_type: 'application/gzip',  
  encoding: 'SpecialEncoding',  
  content: encoded_content  
}
```

If you specify an encoding, Mail will assume that your content is already encoded and not try to Base64 encode it.

### 2.3.2 Making Inline Attachments

Action Mailer 3.0 makes inline attachments, which involved a lot of hacking in pre 3.0 versions, much simpler and trivial as they should be.

- First, to tell Mail to turn an attachment into an inline attachment, you just call `#inline` on the attachments method within your Mailer:

```
def welcome
  attachments.inline['image.jpg'] = File.read('/path/to/image.jpg')
end
```

- Then in your view, you can just reference attachments as a hash and specify which attachment you want to show, calling `url` on it and then passing the result into the `image_tag` method:

```
<p>Hello there, this is our image</p>
<%= image_tag attachments['image.jpg'].url %>
```

- As this is a standard call to `image_tag` you can pass in an options hash after the attachment URL as you could for any other image:

```
<p>Hello there, this is our image</p>
<%= image_tag attachments['image.jpg'].url, alt: 'My Photo', class: 'photos' %>
```

### 2.3.3 Sending Email To Multiple Recipients

It is possible to send email to one or more recipients in one email (e.g., informing all admins of a new signup) by setting the list of emails to the `:to` key. The list of emails can be an array of email addresses or a single string with the addresses separated by commas.

```
class AdminMailer < ApplicationMailer
  default to: -> { Admin.pluck(:email) },
           from: 'notification@example.com'

  def new_registration(user)
    @user = user
    mail(subject: "New User Signup: #{@user.email}")
  end
end
```

The same format can be used to set carbon copy (Cc:) and blind carbon copy (Bcc:) recipients, by using the `:cc` and `:bcc` keys respectively.

### 2.3.4 Sending Email With Name

Sometimes you wish to show the name of the person instead of just their email address when they receive the email. The trick to doing that is to format the email address in the format "Full Name" <email>.

```
def welcome_email
  @user = params[:user]
  email_with_name = %("#{@user.name}" <#{@user.email}>)
  mail(to: email_with_name, subject: 'Welcome to My
  Awesome Site')
end
```

### 2.4 Mailer Views

Mailer views are located in the `app/views/name_of_mailer_class` directory. The specific mailer view is known to the class because its name is the same as the mailer method. In our example from above, our mailer view for the `welcome_email` method will be in `app/views/user_mailer/welcome_email.html.erb` for the HTML version and `welcome_email.text.erb` for the plain text version.

To change the default mailer view for your action you do something like:

```

class UserMailer < ApplicationMailer
  default from: 'notifications@example.com'

  def welcome_email
    @user = params[:user]
    @url = 'http://example.com/login'
    mail(to: @user.email,
         subject: 'Welcome to My Awesome Site',
         template_path: 'notifications',
         template_name: 'another')
  end
end

```

In this case it will look for templates at `app/views/notifications` with name `another`. You can also specify an array of paths for `template_path`, and they will be searched in order.

If you want more flexibility you can also pass a block and render specific templates or even render inline or text without using a template file:

```

class UserMailer < ApplicationMailer
  default from: 'notifications@example.com'

  def welcome_email
    @user = params[:user]
    @url = 'http://example.com/login'
    mail(to: @user.email,
         subject: 'Welcome to My Awesome Site') do |
format|
      format.html { render 'another_template' }
      format.text { render plain: 'Render text' }
    end
  end
end

```

This will render the template `'another_template.html.erb'` for the HTML part and use the rendered text for the text part. The render command is the same one used inside of Action Controller, so you can use all the same options, such as `:text`, `:inline` etc.

### 2.4.1 Caching mailer view

You can perform fragment caching in mailer views like in application views using the `cache` method.

```
<% cache do %>
  <%= @company.name %>
<% end %>
```

And in order to use this feature, you need to configure your application with this:

```
config.action_mailer.perform_caching = true
```

Fragment caching is also supported in multipart emails. Read more about caching in the [Rails caching guide](#).

## 2.5 Action Mailer Layouts

Just like controller views, you can also have mailer layouts. The layout name needs to be the same as your mailer, such as `user_mailer.html.erb` and `user_mailer.text.erb` to be automatically recognized by your mailer as a layout.

In order to use a different file, call `layout` in your mailer:

```
class UserMailer < ApplicationMailer
  layout 'awesome' # use awesome.(html|text).erb as the
  layout
end
```

Just like with controller views, use `yield` to render the view inside the layout.

You can also pass in a `layout_name` option to the `render` call inside the format block to specify different layouts for different formats:

```
class UserMailer < ApplicationMailer
  def welcome_email
    mail(to: params[:user].email) do |format|
      format.html { render layout: 'my_layout' }
      format.text
    end
  end
end
```

Will render the HTML part using the `my_layout.html.erb` file and the text part with the usual `user_mailer.text.erb` file if it exists.

## 2.6 Previewing Emails

Action Mailer previews provide a way to see how emails look by visiting a special URL that renders them. In the above example, the preview class for `UserMailer` should be named `UserMailerPreview` and located in `test/mailers/previews/user_mailer_preview.rb`. To see the preview of `welcome_email`, implement a method that has the same name and call `UserMailer.welcome_email`:

```
class UserMailerPreview < ActionMailer::Preview
  def welcome_email
    UserMailer.with(user: User.first).welcome_email
  end
end
```

Then the preview will be available in [http://localhost:3000/rails/mailers/user\\_mailer/welcome\\_email](http://localhost:3000/rails/mailers/user_mailer/welcome_email).

If you change something in `app/views/user_mailer/welcome_email.html.erb` or the mailer itself, it'll automatically reload and render it so you can visually see the new style instantly. A list of previews are also available in <http://localhost:3000/rails/mailers>.

By default, these preview classes live in `test/mailers/previews`. This can be configured using the `preview_path` option. For example, if you want to change it to `lib/mailer_previews`, you can configure it in `config/application.rb`:

```
config.action_mailer.preview_path = "#{Rails.root}/lib/mailer_previews"
```

## 2.7 Generating URLs in Action Mailer Views

Unlike controllers, the mailer instance doesn't have any context about the incoming request so you'll need to provide the `:host` parameter yourself.

As the `:host` usually is consistent across the application you can configure it globally in `config/application.rb`:

```
config.action_mailer.default_url_options = { host:
  'example.com' }
```

Because of this behavior you cannot use any of the `*_path` helpers inside of an email. Instead you will need to use the associated `*_url` helper. For example instead of using

```
<%= link_to 'welcome', welcome_path %>
```

You will need to use:

```
<%= link_to 'welcome', welcome_url %>
```

By using the full URL, your links will now work in your emails.

### 2.7.1 Generating URLs with `url_for`

`url_for` generates a full URL by default in templates.

If you did not configure the `:host` option globally make sure to pass it to `url_for`.

```
<%= url_for(host: 'example.com',
            controller: 'welcome',
            action: 'greeting') %>
```

### 2.7.2 Generating URLs with Named Routes

Email clients have no web context and so paths have no base URL to form complete web addresses. Thus, you should always use the `"_url"` variant of named route helpers.

If you did not configure the `:host` option globally make sure to pass it to the url helper.



```
<%= user_url(@user, host: 'example.com') %>
```

non-GET links require [rails-ujs](#) or [jQuery UJS](#), and won't work in mailer templates. They will result in normal GET requests.

## 2.8 Adding images in Action Mailer Views

Unlike controllers, the mailer instance doesn't have any context about the incoming request so you'll need to provide the `:asset_host` parameter yourself.

As the `:asset_host` usually is consistent across the application you can configure it globally in `config/application.rb`:

```
config.action_mailer.asset_host = 'http://example.com'
```

Now you can display an image inside your email.

```
<%= image_tag 'image.jpg' %>
```

## 2.9 Sending Multipart Emails

Action Mailer will automatically send multipart emails if you have different templates for the same action. So, for our `UserMailer` example, if you have `welcome_email.text.erb` and `welcome_email.html.erb` in `app/views/user_mailer`, Action Mailer will automatically send a multipart email with the HTML and text versions setup as different parts.

The order of the parts getting inserted is determined by the `:parts_order` inside of the `ActionMailer::Base.default` method.

## 2.10 Sending Emails with Dynamic Delivery Options

If you wish to override the default delivery options (e.g. SMTP credentials) while delivering emails, you can do this using `delivery_method_options` in the mailer action.

```

class UserMailer < ApplicationMailer
  def welcome_email
    @user = params[:user]
    @url = user_url(@user)
    delivery_options = { user_name: params[:company].smtp_user,
                        password: params[:company].smtp_password,
                        address: params[:company].smtp_host }

    mail(to: @user.email,
         subject: "Please see the Terms and Conditions attached",
         delivery_method_options: delivery_options)
  end
end

```

## 2.11 Sending Emails without Template Rendering

There may be cases in which you want to skip the template rendering step and supply the email body as a string. You can achieve this using the `:body` option. In such cases don't forget to add the `:content_type` option. Rails will default to `text/plain` otherwise.

```

class UserMailer < ApplicationMailer
  def welcome_email
    mail(to: params[:user].email,
         body: params[:email_body],
         content_type: "text/html",
         subject: "Already rendered!")
  end
end

```

## 3 Receiving Emails

Receiving and parsing emails with Action Mailer can be a rather complex endeavor. Before your email reaches your Rails app, you would have had to configure your system to somehow forward emails to your app, which needs to be listening for that. So, to receive emails in your Rails app you'll need to:

- Implement a `receive` method in your mailer.
- Configure your email server to forward emails from the address(es) you would like your app to receive to `/path/to/app/bin/rails runner`

- 'UserMailer.receive(STDIN.read)'.

Once a method called receive is defined in any mailer, Action Mailer will parse the raw incoming email into an email object, decode it, instantiate a new mailer, and pass the email object to the mailer receive instance method. Here's an example:

```
class UserMailer < ApplicationMailer
  def receive(email)
    page = Page.find_by(address: email.to.first)
    page.emails.create(
      subject: email.subject,
      body: email.body
    )

    if email.has_attachments?
      email.attachments.each do |attachment|
        page.attachments.create({
          file: attachment,
          description: email.subject
        })
      end
    end
  end
end
```

#### 4 Action Mailer Callbacks

Action Mailer allows for you to specify a before\_action, after\_action and around\_action.

- Filters can be specified with a block or a symbol to a method in the mailer class similar to controllers.
- You could use a before\_action to populate the mail object with defaults, delivery\_method\_options or insert default headers and attachments.

```

class InvitationsMailer < ApplicationMailer
  before_action { @inviter, @invitee = params[:inviter],
params[:invitee] }
  before_action { @account = params[:inviter].account }

  default to:      -> { @invitee.email_address },
    from:          -> { common_address(@inviter) },
    reply_to:      ->
{ @inviter.email_address_with_name }

  def account_invitation
    mail subject: "#{@inviter.name} invited you to their
Basecamp (#{@account.name})"
  end

  def project_invitation
    @project      = params[:project]
    @summarizer =
ProjectInvitationSummarizer.new(@project.bucket)

    mail subject: "#{@inviter.name.familiar} added you
to a project in Basecamp (#{@account.name})"
  end
end

```

- You could use an `after_action` to do similar setup as a `before_action` but using instance variables set in your mailer action.

```

class UserMailer < ApplicationMailer
  before_action { @business, @user = params[:business],
params[:user] }

  after_action :set_delivery_options,
              :prevent_delivery_to_guests,
              :set_business_headers

  def feedback_message
  end

  def campaign_message
  end

  private

  def set_delivery_options
    # You have access to the mail instance,
    # @business and @user instance variables here
    if @business && @business.has_smtp_settings?
      mail.delivery_method.settings.merge!
(@business.smtp_settings)
    end
  end

  def prevent_delivery_to_guests
    if @user && @user.guest?
      mail.perform_deliveries = false
    end
  end

  def set_business_headers
    if @business
      headers["X-SMTPAPI-CATEGORY"] = @business.code
    end
  end
end

```

- Mailer Filters abort further processing if body is set to a non-nil value.

## 5 Using Action Mailer Helpers

Action Mailer now just inherits from AbstractController, so you have access to the same generic helpers as you do in Action Controller.

## 6 Action Mailer Configuration

The following configuration options are best made in one of the environment files (environment.rb, production.rb, etc...)

<b>Configuration</b>	<b>Description</b>
logger	Generates information on the mailing run if available. Can be set to <code>nil</code> for no logging. Compatible with both Ruby's own <code>Logger</code> and <code>Log4r</code> loggers.

smtp\_settings

Allows detailed configuration for :smtp delivery method:

- :address - Allows you to use a remote mail server. Just change it from its default "localhost" setting.
- :port - On the off chance that your mail server doesn't run on port 25, you can change it.
- :domain - If you need to specify a HELO domain, you can do it here.
- :user\_name - If your mail server requires authentication, set the username in this setting.
- :password - If your mail server requires authentication, set the password in this setting.
- :authentication - If your mail server requires authentication, you need to specify the authentication type here. This is a symbol and one of :plain (will send the password in the clear), :login (will send password Base64 encoded) or :cram\_md5 (combines a Challenge/Response mechanism to exchange information and a cryptographic Message Digest 5 algorithm to hash important information)
- :enable\_starttls\_auto - Detects if STARTTLS is enabled in your SMTP server and starts to use it. Defaults to true.
- :openssl\_verify\_mode - When using TLS, you can set how OpenSSL checks the certificate. This is really useful if you need to validate a self-signed and/or a wildcard certificate. You can use the name of an OpenSSL verify constant ('none' or 'peer') or directly the constant (OpenSSL::SSL::VERIFY\_NONE or OpenSSL::SSL::VERIFY\_PEER).

<p>sendmail_settings</p>	<p>Allows you to override options for the <code>:sendmail</code> delivery method.</p> <ul style="list-style-type: none"> <li>• <code>:location</code> - The location of the sendmail executable. Defaults to <code>/usr/sbin/sendmail</code>.</li> <li>• <code>:arguments</code> - The command line arguments to be passed to sendmail. Defaults to <code>-i</code>.</li> </ul>
<p>raise_delivery_errors</p>	<p>Whether or not errors should be raised if the email fails to be delivered. This only works if the external email server is configured for immediate delivery.</p>
<p>delivery_method</p>	<p>Defines a delivery method. Possible values are:</p> <ul style="list-style-type: none"> <li>• <code>:smtp</code> (default), can be configured by using <code>config.action_mailer.smtp_settings</code>.</li> <li>• <code>:sendmail</code>, can be configured by using <code>config.action_mailer.sendmail_settings</code>.</li> <li>• <code>:file</code>: save emails to files; can be configured by using <code>config.action_mailer.file_settings</code>.</li> <li>• <code>:test</code>: save emails to <code>ActionMailer::Base.deliveries</code> array.</li> </ul> <p>See <a href="#">API docs</a> for more info.</p>
<p>perform_deliveries</p>	<p>Determines whether deliveries are actually carried out when the <code>deliver</code> method is invoked on the Mail message. By default they are, but this can be turned off to help functional testing.</p>
<p>deliveries</p>	<p>Keeps an array of all the emails sent out through the Action Mailer with <code>delivery_method :test</code>. Most useful for unit and functional testing.</p>
<p>default_options</p>	<p>Allows you to set default values for the mail method options (<code>:from</code>, <code>:reply_to</code>, etc.).</p>



For a complete writeup of possible configurations see the [Configuring Action Mailer](#) in our Configuring Rails Applications guide.

## 6.1 Example Action Mailer Configuration

An example would be adding the following to your appropriate config/environments/\$RAILS\_ENV.rb file:

```
config.action_mailer.delivery_method = :sendmail
# Defaults to:
# config.action_mailer.sendmail_settings = {
#   location: '/usr/sbin/sendmail',
#   arguments: '-i'
# }
config.action_mailer.perform_deliveries = true
config.action_mailer.raise_delivery_errors = true
config.action_mailer.default_options = {from: 'no-reply@example.com'}
```

## 6.2 Action Mailer Configuration for Gmail

As Action Mailer now uses the [Mail gem](#), this becomes as simple as adding to your config/environments/\$RAILS\_ENV.rb file:

```
config.action_mailer.delivery_method = :smtp
config.action_mailer.smtp_settings = {
  address:           'smtp.gmail.com',
  port:             587,
  domain:           'example.com',
  user_name:        '<username>',
  password:         '<password>',
  authentication:   'plain',
  enable_starttls_auto: true }
```

Note: As of July 15, 2014, Google increased [its security measures](#) and now blocks attempts from apps it deems less secure. You can change your Gmail settings [here](#) to allow the attempts. If your Gmail account has 2-factor authentication enabled, then you will need to set an [app password](#) and use that instead of your regular password. Alternatively, you can use another ESP to send email by replacing 'smtp.gmail.com' above with the address of your provider.

## 7 Mailer Testing

You can find detailed instructions on how to test your mailers in the [testing guide](#).

## 8 Intercepting Emails

There are situations where you need to edit an email before it's delivered. Fortunately Action Mailer provides hooks to intercept every email. You can register an interceptor to make modifications to mail messages right before they are handed to the delivery agents.

```
class SandboxEmailInterceptor
  def self.delivering_email(message)
    message.to = ['sandbox@example.com']
  end
end
```

Before the interceptor can do its job you need to register it with the Action Mailer framework. You can do this in an initializer file `config/initializers/sandbox_email_interceptor.rb`

```
if Rails.env.staging?
  ActionMailer::Base.register_interceptor(SandboxEmailIn
  terceptor)
end
```

The example above uses a custom environment called "staging" for a production like server but for testing purposes. You can read [Creating Rails environments](#) for more information about custom Rails environments.

## Sending Emails in Rails Applications

### Introduction

In this article we will walk through a simple app to demonstrate how to send emails through a Rails application with ActionMailer, ActionMailer Preview, and through a third party email service provider such as Gmail or Mailgun. We will also demonstrate how to use Active Job to send the email with a background processor.

You can find the code for this tutorial [here](#)

### Sending Emails Using ActionMailer and Gmail

Now we will build a rails application which will send an email to the user when a new user is created. Let's create a new rails application.

```
1 $ rails new new_app_name  
2 $ rails g scaffold user name:string  
3 email:string  
$ rake db:migrate
```

We now have a basic application, let's make use of ActionMailer. The mailer generator is similar to any other generator in rails.

```
1 $ rails g mailer example_mailer
2 create app/mailers/example_mailer.rb
3 invoke erb
4 create app/views/example_mailer
5 invoke test_unit
6 create test/mailers/example_mailer_test.rb
7 create test/mailers/preview/example_mailer_preview.rb
```

Our application is currently using Rails 4.2.0.beta4 so the rails generator has created preview files for our application by default as `test/mailers/preview/example_mailer_preview.rb` which we will be using later.

```
app/mailers/example_mailer.rb
```

```
1 class ExampleMailer < ActionMailer::Base
2   default from: "from@example.com"
3 end
```

Now let's write methods and customized email. First you should change the default email address from `from@example.com` to the email address you want use as the sender's address.

```
1 class ExampleMailer < ActionMailer::Base
2   default from: "from@example.com"
3
4   def sample_email(user)
5     @user = user
6     mail(to: @user.email, subject: 'Sample
7 Email')
8   end
end
```

`sample_email` takes user parameters and sends email using method `mail` to email address of user. In case you want to know about more features like attachment and multiple receivers, you can check out rails guide in the reference section. Now let's write the mail we want to send to our users, and this can be done in `app/views/example_mailer`. Create a file `sample_email.html.erb` which is an email formatted in HTML.

`app/views/example_mailer/sample_email.html.erb`

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta content='text/html; charset=UTF-8'
5     http-equiv='Content-Type' />
6   </head>
7   <body>
8     <h1>Hi <%= @user.name %></h1>
9     <p>
10      Sample mail sent using smtp.
11    </p>
12  </body>
</html>
```

We also need to create the text part for this email as not all clients prefer HTML emails. Create `sample_email.text.erb` in the `app/views/example_mailer` directory.

```
app/views/example_mailer/sample_email.text.erb
```

```
1 Hi <%= @user.name %>
2 Sample mail sent using smtp.
```

In the development environment we can use ActionMailer Preview to test our application. We are going to use the `test/mailers/previews/example_mailer_preview.rb` file created while generating mailers. We will just call any user (first user in this case) to preview the email.

```
test/mailers/previews/example_mailer_preview.rb
```

```

# Preview all emails at http://localhost:3000/
1 rails/mailers/example_mailer
2 class ExampleMailerPreview <
3   ActionMailer::Preview
4     def sample_mail_preview
5       ExampleMailer.sample_email(User.first)
6     end
end

```

When you visit `http://localhost:3000/rails/mailers/example_mailer/sample_mail_preview` you will see preview of the email. By default email previews are placed in `test/mailers/previews`. You can change this by setting up different a path in `/config/environments/development.rb`. Just set `config.action_mailer.preview_path` to the desired path and add preview file to the corresponding location.

## Sending emails using ActionMailer and Gmail

By default rails tries to send emails via [SMTP](#). We will provide SMTP configuration in environment settings `/config/environments/production.rb`. Let's first look at the configuration you need to send emails with Gmail. Before we proceed we need to save sensitive information such as username and password as environment variables. We will do so by using the gem `figaro`. For detailed information on how to manage environment variables in rails refer to [Manage Environment Configuration Variables in Rails](#).

`/config/application.yml`

```
1 gmail_username: 'username@gmail.com'  
2 gmail_password: 'Gmail password'
```

/config/environments/production.rb

```
config.action_mailer.delivery_method = :smtp  
1 # SMTP settings for gmail  
2 config.action_mailer.smtp_settings = {  
3   :address          => "smtp.gmail.com",  
4   :port             => 587,  
5   :user_name        =>  
6   ENV['gmail_username'],  
7   :password         =>  
8   ENV['gmail_password'],  
9   :authentication   => "plain",  
10  :enable_starttls_auto => true  
}
```

Note here we are setting the app to send out emails with Gmail from the production environment. This is typically what you want because you don't want to accidentally send out emails while working locally. If you run into errors like `Net::SMTPAuthenticationError` while using gmail for sending out emails, visit your [gmail settings](#) and enable less secure apps to get the application working.

Now let's edit the `UsersController` to trigger the event that will send an email to a user. We just need to add `ExampleMailer.sample_email(@user).deliver` to the `create` method in `app/controllers/users_controller.rb`. The `create` method in `users_controller.rb` should look something like:



```

def create
  @user = User.new(user_params)

  1  respond_to do |format|
  2
  3    if @user.save
  4
  5      # Sends email to user when user is
  6      created.
  7
  8      ExampleMailer.sample_email(@user).deliver

  9
 10      format.html { redirect_to @user, notice:
 11      'User was successfully created.' }
 12      format.json { render :show,
 13      status: :created, location: @user }
 14    else
 15      format.html { render :new }
 16      format.json { render json: @user.errors,
 17      status: :unprocessable_entity }
 18    end
 19  end
 20 end
 21 end

```

When a new user is created we are sending out an email via the `sample_email` method in mailer `ExampleMailer`.

Sending emails using ActionMailer and Mailgun through SMTP

Let's see how to use mailgun to send out emails. First [create a free account on Mailgun](#). Once done you will be redirected to the dashboard. We will be using mailgun subdomains. Click on the sandbox, you should see something like this:



We are going to need information listed to get mailgun working for our application. Let's store the credentials in `/config/application.yml`.

`/config/application.yml`

```
1 api_key: 'API Key'
2 domain: 'Domain'
3 username: 'Default SMTP Login'
4 password: 'Default Password'
5 gmail_username: 'username@gmail.com'
6 gmail_password: 'gmail password'
```

Replace the corresponding credentials received from domain information of sandbox server. We must also change SMTP settings as we are now using Mailgun instead of Gmail.

/config/environments/production.rb

```
1 config.action_mailer.delivery_method = :smtp
2 # SMTP settings for mailgun
3 ActionMailer::Base.smtp_settings = {
4   :port          => 587,
5   :address       => "smtp.mailgun.org",
6   :domain       => ENV['domain'],
7   :user_name    => ENV['username'],
8   :password     => ENV['password'],
9   :authentication => :plain,
10 }
```

Sending emails using ActionMailer and Mailgun through Mailgun's APIs

[The official ruby library](#) of Mailgun `mailgun-ruby` empowers users and developers to take advantage of the Mailgun APIs. To use it first add `gem 'mailgun-ruby', '~>1.0.2'`, `require: 'mailgun'` to your Gemfile and run `bundle install`. Finally make changes to `app/mailers/example_mailer.rb`.

```

class ExampleMailer < ActionMailer::Base
  def sample_email(user)
    @user = user
    mg_client = Mailgun::Client.new
ENV['api_key']
    message_params = {:from =>
ENV['gmail_username'],
                    :to => @user.email,
                    :subject => 'Sample Mail
using Mailgun API',
                    :text => 'This mail
is sent using Mailgun API via mailgun-ruby'}
    mg_client.send_message ENV['domain'],
message_params
  end
end

```

Mailgun::Client.new initiates mailgun client using the API keys. In message\_params we are providing custom email information and .send\_message takes care of sending emails via Mailgun API. You should change from@example.com to desired sender's email address.

### Sending Emails with a Background Processor through Active Job

While testing the application you might have noticed that it takes more time than usual to create a new user. This happens because we have to hit an external API to send out emails. This can be an issue if you are sending multiple emails or sending emails to multiple users. This problem can be easily resolved by moving the email sending part to background jobs. In our

application we will make use of [Active Jobs](#) and [Delayed Job](#) to send emails in the background.

Active Job is an adapter that provides a universal interface for background processors like Resque, Delayed Job, Sidekiq, etc. Note that for using Active Job you will need Rails 4.2 or above.

```
1 $ rails g job send_email
2   invoke  test_unit
3   create  test/jobs/send_email_job_test.rb
4   create  app/jobs/send_email_job.rb
5 $
```

Now let's write the job to be performed by workers. Active Job is integrated with ActionMailer so you can easily send emails asynchronously. For sending emails through Active Job we use `deliver_later` method.

```
/app/jobs/send_email_job.rb
```

```

1 class SendEmailJob < ActiveJob::Base
2   queue_as :default
3
4   def perform(user)
5     @user = user
6     ExampleMailer.sample_email(@user).deliver_late
7   end
8 end

```

Now let's make changes to our user creation process. Instead of sending email while creating the user we enqueue the email sending job to be performed later.

app/controllers/users\_controller.rb

```

1 def create
2   ...
3   SendEmailJob.set(wait:
4   20.seconds).perform_later(@user)
5   ...
6 end

```

Now we need to configure the backend for our background process. We have selected `delayed_jobs` as our backend but you can choose your own backend depending on your needs. Active Job has built-in adapters for multiple queueing backends.

## Gemfile

```
1 gem 'delayed_job_active_record'
```

```
1 $ bundle
2 $ rails generate delayed_job:active_record
3 $ rake db:migrate
```

Set up queueing backend for the production environment.

```
/config/environments/production.rb
```

```
1 config.active_job.queue_adapter = :delayed_job
```

Everything is configured now, for testing the application just start the rails server and create a new user. A new job will be added to the queue and you will notice the time required for creating a new user is drastically decreased. You can start running the jobs in queue by:

```
1 $ bundle exec rake jobs:work
```

## Conclusion

In the article we went over basic configuration and tools used for sending emails through a rails application. We covered the basics of ActionMailer, Gmail & Mailgun (as email sending services), ActionMailer Previews(for previewing emails) and `mailgun-ruby` gem for the Mailgun APIs. In the end, we showed how to send out emails with a background processor through Active Job

## Capbara using Sinatra App tes

### Step 1: Building the App

We're going to create an incredibly simple Sinatra app to test. For starters, let's create a project folder and throw this in a Gemfile:

```
1 source :rubygems
2
3 gem "sinatra"
4 gem "shotgun"
5 gem "cucumber"
6 gem "capbara"
7 gem "rspec"
```

Now, run `bundle install` in that directory.

So, open a file called `myapp.rb`; here's our super simple app; it just simulates a site that might let you sign up for a newsletter.



```
01 require "sinatra/base"
02
03 class MyApp < Sinatra::Base
04   get "/" do
05     erb :index
06   end
07
08   post "/thankyou" do
09     @name = params["name"]
10     @email = params["email"]
11     erb :thankyou
12   end
13
14   get "/form" do
15     erb :form
16   end
17 end
```

If you're not familiar with Sinatra, check out Dan Harper's excellent sessions [Singing with Sinatra](#); that'll get you up and running with the basics in no time.

If you are familiar with Sinatra, you'll see that we're creating three paths here; on the home page (`/`), we just render the `index.erb` template (more on the templates in a minute). If we get a post request to the path `/thankyou`, we take the values of the `name` and `email` parameters and assign them to instance variables. Instance variables will be available inside whatever template we render, which happens to be `thankyou.erb`. Finally, at `/form`, we render the `form.erb` template.

Now, let's build these templates. Sinatra will look inside a 'views' folder for the templates, so let's put them there. As you saw in `myapp.rb`, we're using ERB to render the templates, so they'll, of course, be ERB templates. If we have a `layout.erb` template, it will wrap all our other templates. So, let's do this:

`layout.erb`

```
01 <!DOCTYPE html>
02 <html>
03 <head>
04   <meta charset='UTF=8' />
05   <title>THE APP</title>
06 </head>
07 <body>
08
09   <h1>THE APP</h1>
10
11   <%= yield %>
12
13 </body>
14 </html>
```

That call to `yield` will be where the other templates are inserted. And those other templates are pretty simple:

`index.erb`

```
1 <p>This is the home page</p>
2 <p><a id="link" href="/form">Sign up for our
  newsletter!</a></p>
```

`form.erb`

```
01 <form method="post" action="/thankyou">
02   <p>
03     Fill out this form to receive our newsletter.
04   </p>
05   <p>
06     <label for="name">Name:</label>
07     <input type="text" name="name" id="name" />
08   </p>
09   <p>
10     <label for="email">Email:</label>
11     <input type="text" name="email" id="email" />
12   </p>
13   <p>
14     <button type="submit">Sign Up!</button>
15   </p>
16 </form>
```

thankyou.erb

```
1 <p>Hi there, <%= @name %>. You&#39;ll now receive our
  email at <%= @email %></p>
```

So, there's our app. To test it manually, you can put this in a `config.ru` file:

```
1 require "./myapp"
2 run MyApp
```

And then run `shotgun` in the terminal. This will start up a webserver, probably on `port 9393`. You can now poke around and test our web app. But we want to automate this testing, right? Let's do it!

Step 2: Setting our our Test Environment

[Cucumber](#) bills itself as “behaviour driven development with elegance and joy.” While joy seems a bit far-fetched to me, I think we’ll both agree that elegance, well, Cucumber’s got it.

Because behaviour driven development is partly about understanding what the client wants before you begin coding, Cucumber aims to make its tests readable by clients (AKA, non-programmers). So, you’ll see here that all your tests are written in what appears to be plain text (it’s actually [Gherkin](#)).

Remember how, with Rspec, we have separate spec files to describe different functionalities? In Cucumber-speak, those are features, and they all belong in a “features” folder. Inside that folder create two more folders called “support” and “step\_definitions.”

Inside the “support” folder, open an `env.rb`. This code will set up our testing environment. Here’s what we need:

```
01 require_relative "../..../myapp"
02
03 require "Capybara"
04 require "Capybara/cucumber"
05 require "rspec"
06
07
08 World do
09   Capybara.app = MyApp
10
11   include Capybara::DSL
12   include RSpec::Matchers
13 end
```

This requires the different libraries that we need, and uses `include` to load their methods into our environment. What's this Capybara that we're using? Basically, it's the functionality that allows us to use our web app, so that we can test it. It's important to set `Capybara.app` to our app. I should mention that, were we doing this with a Rails app, most of this setup would be done automatically for us.

(Note: in the screencast, I `include RSpec::Expectations` unnecessarily; leave it out.)

Okay, so, let's write some tests!

### Step 3 Writing the Tests

Let's start with our home page. Open the file `home_pages.feature` (in the "features" folder) and start with this:

```
1 Feature: Viewer visits the Home Page
2   In order to read the page
3   As a viewer
4   I want to see the home page of my app
```

This is a common way to start a feature file starts; Doesn't really look like code, does it? Well, it's [Gherkin](#), a domain-specific languages (DSL) that "lets you describe software's behaviour without detailing how that behaviour is implemented." What we're written so far doesn't run in any way, but it explains the purpose of the feature. Here's the general structure:

```
1 In order to [goal]
2 As a [role]
3 I want [feature]
```

You don't have to follow that template: you can put whatever you want; the purpose is to describe the feature. However, this seems to be a common pattern.

Next comes a list of scenarios that describe the feature. Here's the first:

```
1 Scenario: View home page
2   Given I am on the home page
3   Then I should see "This is the home page."
```

Each scenario can have up to three parts: Givens, Whens, and Thens:

- Given - Given lines describe what pre-condition should exist.

- When - When lines describe the actions you take.
- Then - Then lines describe the result.

There are also And lines, which do whatever the line above them does. For example:

```
1 Given I am on the home page
2 And I am signed in
3 Then I should see "Welcome Back!"
4 And I should see "Settings"
```

In this case, the first And line acts as a Given line, and the second one acts as a Then line.

We'll see a few When lines shortly. But right now, let's run that test. To do that, run cucumber in the terminal. You'll probably see something like this:

```
Terminal — zsh — 78x23
→ cucumber features/home_page.feature
Feature: Viewer visits the home page
  In order to read the page
  As a viewer
  I want to see the home page of my app

  Scenario: View home page # features/home_page.feature:6
    Given I am on the home page # features/home_page.feature:7
    Then I should see "This is the home page." # features/home_page.feature:8

1 scenario (1 undefined)
2 steps (2 undefined)
0m0.075s

You can implement step definitions for undefined steps with these snippets:

Given /^I am on the home page$/ do
  pending # express the regexp above with the code you wish you had
end

Then /^I should see "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

Cucumber feature files are written to be readable to non-programmers, so we have to “implement step definitions for undefined steps.” Thankfully, Cucumber gives us some snippets to start with.

Looking at these snippets, you can see how this will work. Each step is matched with a regular expression. Any quoted values will be captured and passed as a block parameter. Inside the block, we do whatever we expect to happen as a result of that step. This might be set-up code in **Given** steps, some calculations or actions in **When** steps, and a comparison in **Then** steps.

Cucumber will load any files in the folder “features/step\_definitions” for steps, so let’s create “sinatra\_steps.rb” file and add these two steps:



```

1 Given /^I am on the home page$/ do
2   visit "/"
3 end
4
5 Then /^I should see "([^"]*)"$/ do |text|
6   page.should have_content text
7 end

```

In this little snippet here, we're using Cucumber, RSpec, and Capybara. Firstly, we've got the cucumber Given and Then method calls. Secondly, we're using the Capybara methods visit (to visit a URL) and has\_content?. But you don't see the call to has\_content? because we've loaded the RSpec matchers, so we can make our tests read as they would with RSpec. If we wanted to leave RSpec out, we would just write page.has\_content? text.

Now, if you run cucumber again, you'll see that our tests pass:

```

Terminal — zsh — 88x15
→ cucumber features/home_page.feature
Feature: Viewer visits the home page
  In order to read the page
  As a viewer
  I want to see the home page of my app

Scenario: View home page # features/home_page.feature:6
  Given I am on the home page # features/step_definitions/sinatra.rb:2
  Then I should see "This is the home page." # features/step_definitions/sinatra.rb:23

1 scenario (1 passed)
2 steps (2 passed)
0m12.892s

mothership:~/Desktop/myapp

```

Let's add two more Scenarios for our home page:

```
1 Scenario: Find heading on home page
2   Given I am on the home page
3   Then I should see "MY APP" in the selector "h1"
4
5 Scenario: Find the link to the form
6   Given I am on the home page
7   Then I should see "Sign up for our newsletter." in a
  link
```

These require two more `Then` steps, as you'll find if you try to run this. Add these to `sinatra_steps.rb`:

```
1 Then /^I should see "([^"]*)" in the selector
2 "([^"]*)"$/ do |text, selector|
3   page.should have_selector selector, content: text
4 end
5
6 Then /^I should see "([^"]*)" in a link$/ do |text|
7   page.should have_link text
  end
```

You should be able to tell what these are doing: the first looks for text within a certain element on the page. The second looks for a link with the given text (yes, you could have done `Then I should see "Sign up ..." in the selector "a"`, but I wanted to show you another Capybara/Rspec method)

Again, run `cucumber`; you'll see all our tests passing:

```
Terminal — zsh — 103x21
→ cucumber features/home_page.feature
Feature: Viewer visits the home page
  In order to read the page
  As a viewer
  I want to see the home page of my app

  Scenario: View home page # features/home.
    Given I am on the home page # features/step.
    Then I should see "This is the home page." # features/step.

  Scenario: Find heading on home page # features/
    Given I am on the home page # features/
    Then I should see "MY APP" in the selector "h1" # features/

  Scenario: Find the link to the form # features/
    Given I am on the home page # features/
    Then I should see "Sign up for our newsletter." in a link # features/

3 scenarios (3 passed)
6 steps (6 passed)
0m12.923s
```

Let's now open "features/form\_page.feature"; throw this in there:

```
Feature: Viewer signs up for the newsletter
  In order to receive the newsletter
  As a user of the website
  I want to be able to sign up for the newsletter
```

```
Scenario: View form page
  Given I am on "/form"
  Then I should see "Fill out this form to receive our newsletter."
```

```
Scenario: Fill out form
  Given I am on "/form"
  When I fill in "name" with "John Doe"
  And I fill in "email" with "john@doe.com"
```

And I click "Sign Up!"

Then I should see "Hi there, John Doe. You'll now receive our email newsletter at john@doe.com"

The first scenario here is pretty simple, although we need to write the Given step for it. You can probably figure out how to do that by now:

```
1 Given /^I am on "([^"]*)"$/ do |path|
2   visit path
3 end
```

The second one is a little more in depth. For the first time, we're using When steps (remember, the And steps that follow the When step are also When steps). It's pretty obvious what those When steps should do, but how do we do that in the Ruby code? Thankfully, Capybara has a few handy methods to help up:

```
1 When /^I fill in "([^"]*)" with "([^"]*)"$/ do |
2   element, text|
3   fill_in element, with: text
4 end
5
6 When /^I click "([^"]*)"$/ do |element|
7   click_on element
8 end
```

We're using the fill\_in method, which takes the name or id attribute of an element on the page. We're also using click\_on, which will click on the element with the given text, id, or value. There are also the more

specific `click_link` and `click_button`. To see more, check out [the Capybara Readme](#). Browse around the “DSL” section to see more of the methods that Capybara offers.

When you run `cucumber` now, you should get all our tests, passing:

```
→ cucumber
Feature: Viewer signs up for the newsletter
  In order to receive the newsletter
  As a user of the website
  I want to be able to sign up for the newsletter

Scenario: View form page # features/form_page.feature:1
  Given I am on "/form" # features/step_definitions/sinatra.rb:2
  Then I should see "Fill out this form to receive our newsletter." # features/step_definitions/sinatra.rb:23

Scenario: Fill out form
  Given I am on "/form"
  When I fill in "name" with "John Doe"
  And I fill in "email" with "john@doe.com"
  And I click "Sign Up!"
  Then I should see "Hi there, John Doe. You'll now receive our email newsletter at john@doe.com"

Feature: Viewer visits the home page
  In order to read the page
  As a viewer
  I want to see the home page of my app

Scenario: View home page # features/home_page.feature:6
  Given I am on the home page # features/step_definitions/sinatra.rb:2
  Then I should see "This is the home page." # features/step_definitions/sinatra.rb:23

Scenario: Find heading on home page # features/home_page.feature:10
  Given I am on the home page # features/step_definitions/sinatra.rb:2
  Then I should see "MY APP" in the selector "h1" # features/step_definitions/sinatra.rb:28

Scenario: Find the link to the form # features/home_page.feature:14
  Given I am on the home page # features/step_definitions/sinatra.rb:2
  Then I should see "Sign up for our newsletter." in a link # features/step_definitions/sinatra.rb:28

5 scenarios (5 passed)
13 steps (13 passed)
0m22.125s
```