Chapter No. 8
"Testing Rake Tasks"
In this package, you will find:

A Biography of the author of the book
A preview chapter from the book, Chapter NO.8 "Testing Rake Tasks"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Andrey Koleshko had his first touch with programming while at school, when he worked on Pascal. He had been solving basic algorithmic tasks. The first programming language he used at the beginning of his career was Java. He worked with this language for a year and then migrated to the Ruby language, which he worked with for the next four years. Of these four years, he worked with Altoros for three. He had an amazing time there, learning the language and technologies deeply.

Currently, he works at a local cloud hosting company. The company change provided him with the opportunity to deal with a lot of challenges concerning application architecture, code testing, debugging, and deployment processes. As a result, he has been able to contribute to some famous Ruby libraries. More detailed information about his contributions can be found on GitHub at http://github.com/ka8725.

He mostly works with the Rails framework. He openly shares all of his thoughts and his most interesting experiences through his blog at http://railsguides.net. He has recently started to learn the Python programming language.

He lives in Minsk, Belarus, and likes to watch and play sports such as soccer, ping-pong, and volleyball. He also likes travelling to tropical countries with his wife. Teaching people gives him immense pleasure.

For More Information:

Rake Task Management Essentials

Maybe every Ruby developer who is familiar with Rails knows what Rake is. However, many of them are unaware of the complete power of this tool and its real aim. The goal of this book is to improve this situation.

Have you ever had to perform boring, repetitive tasks while deploying your project? I assume here that a project is not only something written in Ruby or another programming language, but it can also consist of operations with files. For example, it might be a book or the documentation of a project that you are writing in Markdown and then compiling into HTML. Or it can be compiling a lot of files to one package. Have you ever wished to build a project or run tests on a project whenever it undergoes a change? All this stuff is easily made possible by programs called software management tools. Rake is one such program.

Rake was initially implemented as a Ruby version of Make—a commonly used build utility. However, calling Rake a build utility undermines its true power. Rake is actually an automation tool—it’s a way to put all those tasks that you perform under the project into one neat and tidy place.

Basically, build automation includes the following processes:

- Compiling the computer source code into binary code
- Packaging the binary code
- Running tests
- Deployment to production systems
- Creating documentation and/or release notes

Rake can be used in all these situations, and this book shows you how Rake performs all the steps. After reading this book, you will know Rake better and be able to write more clear and robust Rake code.

What This Book Covers

Chapter 1, The Software Task Management Tool – Rake, introduces you to the basic usage of Rake and its command-line utilities. You will learn what a rake task is and how to set dependencies between rake tasks, what a default rake task is, Rakefile, and the global Rakefile. This chapter also contains information about the Rake project structure and how to organize the code.

For More Information:
Chapter 2, Working with Files, explains the foundational features of Rake that help us work with files. This is mandatory information because of Rake’s orientation—it is built to be an automation tool. You will see that there is a special rake task for file processing called file. The main part of the chapter contains the explanation of utilities that are offered by Rake to work with the files: FileList and FileUtils. At the end, you will be given a real-world example on how to apply the acquired knowledge.

Chapter 3, Working with Rules, will show how knowing a rule task may allow you to write more robust and precise code.

Chapter 4, Cleaning Up a Build, describes one of the useful features of Rake—the capability to clean the build of your project with the clean standard task.

Chapter 5, Running Tasks in Parallel, helps us figure out how to speed up the resulting task execution with multitask. We will learn which basic problems may arise while implementing parallelism and how to avoid them.

Chapter 6, Debugging Rake Tasks, provides the basic knowledge to debug Rake projects. You will be provided with an example on how to debug rake tasks inherent to Rake techniques and also to Ruby projects in general.

Chapter 7, Integration with Rails, provides an overview of how Rake is integrated into the famous Ruby web framework, Rails. The chapter shows how to write custom rake tasks in a Rails project and run them manually or automatically on schedule.

Chapter 8, Testing Rake Tasks, details the reasons we should test rake tasks. Also, you will see an example of how to write the tests with MiniTest—a built-in Ruby test framework.

Chapter 9, Continuous Integration, briefly introduces you to Jenkins—a continuous integration software. You will see how to configure it and run rake tasks with its help.

Chapter 10, Relentless Automation, doesn’t introduce any new Rake terms, but you will find useful examples of the Rake appliance by popular programs. You will be introduced to the Thor utility, which can replace Rake in some circumstances. Then we will compare both of these frameworks. Finally, we will briefly gather all the information that was provided throughout the book.

For More Information:
In this chapter, you will find information on how to test the rake tasks. It is important to understand that if you don't perform the tests, your rake tasks may fail just like a usual program. This is why the chapter starts by providing cases when the tasks fail, along with the reasons for their failure. Here, we will provide you with an example on how to test rake tasks using Ruby's embedded unit test framework — MiniTest.

In this chapter, we will cover the following topics:

- The need for tests
- Writing tests for rake tasks

The need for tests

The rake tasks are not run as often as regular code from the application. Consider a basic situation when you have a web application with rake tasks that have not been tested. At first glance, after deploying the code to a server and manually testing the application through the web interface, you can be confident that the application works. However, because the rake tasks are usually run by a scheduler, it may be a time bomb. Finally, when the time comes to execute a rake task, it doesn't work because it wasn't tested! Such issues might often occur at the start of your career as a Ruby programmer.

Testing Rake Tasks

For example, each Rails application has a `db:seed` rake task. This task is often used to initialize the application using some essential data. For example, it could contain the code to create an administrator in our application. The code is defined in the `db/seeds.rb` folder that is related to the root of the application. The following is a basic Ruby code to create an administrator in our application:

```ruby
User.create!({
  :admin => 'example@email.com',
  :password => 'password'
})
```

Suppose one particular day we created this code and manually ran the rake task with the `rake db:seed` command. Several days passed and we decided to add validation to the user model for the `name presence`. The previous code won't work in this case, but as we don't test the `db:seed` tasks, all other tests are passed and we are ready to deploy the application because everybody has already forgotten about this piece of code and nobody knows that it doesn't work at all. The issue will be detected when the application is initially deployed to a production server. At that moment, we may have a lot of outdated tasks. The problem keeps cropping up again and again until we finally decide to write the tests for rake tasks. So, in order to avoid mistakes such as these, writing tests may provide us with an airbag.

Writing tests for rake tasks

Now, when you are prepared to write the tests, it's time to figure out how to do it. To demonstrate this, assume that we have to write a rake task named `send_email`, which has these optional arguments: `subject` and `body`. The task should write the given strings to a `sent_email.txt` file (this is for the purpose of simplicity to demonstrate the tests; in reality, you may want to use a real mailer).

Start to solve the problem statement. The following is a basic class `Mailer` that will be used in the rake task (place this code in the `mailer.rb` file):

```ruby
class Mailer
  DESTINATION = 'sent_email.txt'.freeze
  DEFAULT_SUBJECT = 'Greeting!
  DEFAULT_BODY = 'Hello!'

  def initialize(options = {})
    @subject = options[:subject] || DEFAULT_SUBJECT
    @body = options[:body] || DEFAULT_BODY
  end

  def send
```

For More Information:

puts 'Sending email...'  
File.open(DESTINATION, 'w') do |f|
  f << "Subject: #{@subject}\n"  
  f << @body  
end
puts 'Done!'
end

The interface of the class is very simple. Its initializer accepts `subject` and `body` as parameters of hash, and then we are ready to use the `send` method on the object of this class to create the `sent_email.txt` file. This is a simulation to send e-mails for easier the demonstration of the tests.

The following is Rakefile (it's assumed that this file is in the folder where you created `mailer.rb`):

```ruby
require 'rake/clean'
require_relative './mailer'

CLOBBER.include(Mailer::DESTINATION)

desc "Sending email. The email is saved to the file #{Mailer::DESTINATION}"
task :send_email, :subject, :body do |t, args|
  Mailer.new({
    :subject => args.subject,
    :body => args.body
  }).send
end
```

As you can see, there are no complications here. It's quite a simple task. Besides the `send_email` task, it also defines the `clobber` task to remove the generated file `sent_email.txt`. The `send_email` task has arguments that may be passed through the command line. At this point, we are able to check the rake task using the following commands:

```
$ rake send_email
Sending email...
Done!
$ cat sent_email.txt
Subject: Greeting!
Hello!
$ rake "send_email[Test, Hi]"
```
Testing Rake Tasks

Sending email...
Done!
$ cat sent_email.txt

Subject: Test

Hi

$ rake clobber

Now we will talk about the final file that you will see in this chapter. There are many test frameworks for Ruby; here, we will take a look at the tests within the MiniTest framework that is in-built in Ruby since Version 1.9. So, you don't have to install any additional software except Ruby 1.9 to run these Ruby tests.

Check out the online documentation for the MiniTest framework at http://goo.gl/elz2hH.

Now let's try to test the tasks from the previous Rakefile. These are tests for send_email and the clobber rake tasks (place this code in the file with the send_mail_test.rb name in the test folder, which should be created in the folder where you have the previous two files):

```ruby
require 'minitest/autorun'
require 'rake'

class TestSendEmail < MiniTest::Unit::TestCase
  def setup
    Rake.application.init
    Rake.application.load_rakefile
    @task = Rake::Task[:send_email]
    @task.reenable
  end

  def teardown
    if File.exists?(Mailer::DESTINATION)
      File.delete(Mailer::DESTINATION)
    end
  end

  def test_sending_email_with_default_params
    @task.invoke
    assert_equal email, "Subject: Greeting!
Hello!"
  end
end
```

For More Information:
def test_sending_email_with_custom_subject_and_body
  @task.invoke('Test', 'Hi!')
  assert_equal email, "Subject: Test\nHi!"
end

def test_clobber_task_deletes_email
  @task.invoke
  Rake::Task[:clobber].invoke
  refute File.exists?(Mailer::DESTINATION)
end

private

def email
  File.readlines(Mailer::DESTINATION).join
end

end

Let's figure out what is going on here. In the beginning, we check the setup method. This is a cornerstone to test the rake tasks. To get ready to test a task, we have to initialize the Rake application first. Kindly note that we don't use the rake utility in these tasks because the tests would be rather inefficient if we test them by invoking commands with the code. For example, in this case, we won't have access to the rake task's internals and so we won't be able to make stubs, but sometimes, it's reasonable to replace real classes with their fake replacements in the tests.

The technique that is explained here is called Test-driven Development (TDD). If you are new to this, please follow the wiki page at http://en.wikipedia.org/wiki/Test-driven_development for more details.

After initializing the Rake application, we get the rake task and save it to the @task variable. This variable is then made accessible in each test. The last line of the setup method allows us to run the rake task several times. By default, the Rake counts the task starts and doesn't allow us to invoke the tasks again. This line of code resets the counter and provides us with an opportunity to run a task with the invoke method as many times as we want per one test. To avoid this behavior, a method named reenable is defined in Rake::Task. The next method, teardown, will be executed after each test run.

For More Information:
The lines after the teardown method define the tests themselves. They are the same as the ones for general Ruby unit tests. There are two tests for the send_email task and one for the clobber task:

- The first one tests the execution of the rake task without parameters
- The second one tests the rake task for the acceptance of the command-line arguments, subject and body
- The third test tests the clobber task that should delete the generated file

Note that the test file is placed to the separated subdirectory — test. We have done this to separate the tests from the main code. In real life, we have to split the code into many files because of its complexity. Following the known programming paradigm divide and conquer, we get huge benefits from this improvement.

Having the main logic in classes but not in the rake task actions is a good practice and allows us to test the classes in isolation. This is much easier than testing the rake tasks themselves.

Now, let's stop the theory for a while and just try to run the tests that are written. To run the tests, use the following command:

$ ruby test/send_mail_test.rb

The output of the command should look like the following:

MiniTest::Unit::TestCase is now Minitest::Test. From test/send_email_test.rb:4:in `<main>'
Run options: --seed 19500

# Running:

Sending email...  
Done!  
.Sending email...  
Done!  
Sending email...  
Done!  
.Sending email...  
Done!  
Sending email...  
Done!
Sending email...
Done!
.

Finished in 0.013278s, 225.9376 runs/s, 225.9376 assertions/s.

3 runs, 3 assertions, 0 failures, 0 errors, 0 skips

We can see that there are three tests and all of them have passed. Also, the output includes the outgoing messages from the send_mail task.

If you are interested in working with Rails, there is a useful article about how to test rake tasks in Rails with RSpec at http://goo.gl/baLy0R. RSpec is another test framework for Ruby (http://rspec.info).

**Summary**

In this chapter, we explained to you why we should test the rake tasks. Then, you saw an example of how to write tests for the rake tasks with the MiniTest testing framework.

In the next chapter, you will learn how Rake can be useful for continuous integration with an example of the Jenkins tool. You will also see how to install the Rake plugin for Jenkins.

For More Information:
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.