Gathering Performance Information While Executing Everyday Automated Tests

By Michael Kelly.

First published on informit.com

When you're building an application, gathering up-to-date performance info as you go along isn't all that easy. Michael Kelly shows how his team combined a spreadsheet with a simple timer mechanism in the automation framework to provide the details for which management was salivating.

Have you ever worked on a project for which you needed to provide your management team with a constant stream of performance information about the application being tested? Or perhaps a project where you were unsure about the reliability of your performance tests, and you wanted some way to prove that those tests were actually *doing* what you hoped they were doing? Possibly you've been debugging a performance problem with an environment and wished for some quick and current data you could use to compare multiple environments as you tried to debug the problem. A few months ago, I was looking at all three of those problems. This article explains the solution that my team and I developed.

Needed: Rapid Feedback on Environmental Performance

To provide a little context, the project team was developing a web-enabled financial application written in Java. The testing team for the project was using IBM Rational for functional automated testing and Mercury Interactive for performance testing. During the first release of the application, we encountered many of the growing pains and uncertainty that most projects go through. Management was very concerned about performance; for various political reasons, application performance gained a lot of visibility early in the project. To compound this problem, we deployed the application on a regular basis to several different environments for testing. Supposedly, each environment had the same configuration, but each returned significantly different performance results. We needed a way to provide rapid feedback on environmental performance.

Because traditional performance tests typically require detailed setup and a good portion of time to execute, we decided to take a different approach. We were already developing a data-driven framework for regression testing in the Rational tool. We had a significant number of smoke tests, executed several times a day, and traditional functional tests, executed on a regular basis. We decided to include a simple timer mechanism into our automation framework to gather page-load times. We then took that information and wrote it out to a spreadsheet, detailing the page that was loading and the environment in which the script was executing.

This decidedly simple solution solved all of our problems (well, all of *those* problems, anyway):

- We gained a spreadsheet—management's favorite type of document—that we could send to management, with detailed timer information for every page and every call to every external web service in the application.
- Because this information was gathered every time we ran a smoke test or any other type of automated test, we always had up-to-date information to help us debug the differences in all of the deployment environments—a task that would have been almost impossible without this rapid feedback.
- This data also served to audit our existing performance test scripts (which turned out to be working just fine). It's worth noting that this type of audit would work for most single or low-volume performance test results, but might not be appropriate for larger tests.

That's the story; let's take a look at what we actually did.

Simple Solution to a Complex Problem

The following example was written for the open-source testing tool WATIR (pronounced *water*), short for Web Application Testing in Ruby. This article was written using version 1.0.1.

NOTE

I used WATIR for my example because it's an excellent tool for web-based testing and is quickly becoming my tool of choice, but more importantly because this way everyone can run the sample code, regardless of the testing tools you use at your organization. See the end of this article for links for the source code and macros used in this example.

For a sample application, we'll look at Bookpool.com. This is a great intuitive example of an everyday e-commerce web site.

Let's begin by looking at the finished product and work our way backward from there. Figure 1 shows a sample of the spreadsheet that we used to debug the deployment environment and eventually sent to management.

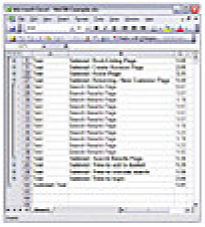

Figure 1

Let's break this apart and look at each piece in isolation:

- Columns A, B, and C represent deployment environment, application page (or function), and time in seconds, respectively.
- Pages and functions are formatted to be hidden using the +/- buttons on the left.
- Each page or function contains a subtotal for the function, showing the average time (in seconds) for that page/function, as shown in Figure 2.

Figure 2

- If you expand a page/function, you can see all of the actual data that was collected for that page or function (see Figure 3).

Figure 3

- The average time (in seconds) for all pages and functions for the environment is then listed, as shown in Figure 4.

Figure 4

It's fairly simple. At the lowest level is always an average. If you want to display the source of that data, click the expand (+) button. We typically sent this information to management on a weekly basis. Our average spreadsheet would have around 2,200 rows of data and contain information on no less then five deployment environments.

Implementing the Solution in a Simple Automation Script

Keep in mind that while my example shows implementation in a single test script, integrating this type of information-gathering in an automation framework is much more practical and powerful. The following Ruby code is for a WATIR script that opens Internet Explorer to http://www.bookpool.com, performs a search on *Ruby*, and verifies that seven books are returned:

```
#includes
require '../watir.rb'  # the controller

#variables
testSite = 'http://www.BookPool.com'

#open the IE browser to http://www.BookPool.com
$ie = IE.new
$ie.goto(testSite)

puts 'Step 2: enter "Ruby: in the search text field under Simple Search'
$ie.textField(:name, "qs").set("Ruby") # qs is the name of the search field

puts 'Step 3: submit the search form.'
$ie.form(:index, "1").submit #submitting first form found in the page.
```

```
puts 'Expected Result: '
puts ' - 7 results should be shown.'
a = $ie.pageContainsText("All 7 results for Ruby:")
if !a
  puts "Test Failed! Could not find test string: 'All 7 results for Ruby:'"
else
  puts "Test Passed. Found the test string: 'All 7 results for Ruby:.'"
end

puts 'Close the browser'
$ie.close()
```

Note that if you download the sample code, there will be more comments.

The `puts` command in Ruby simply outputs the following string to the console window. Everything else should be fairly self-explanatory even if you don't know Ruby. We create a new browser (`$ie`), navigate to http://www.bookpool.com, enter our search criteria into the search field with the name `qs`, click Search, and then check the results using an assertion.

If we take the same script and simply wrap each of the page transitions with a timer, we get the following:

```
#includes
require '../watir.rb'  # the controller

#variables
testSite = 'http://www.BookPool.com'
executionEnvironment = 'Test'  #This could be read from a configuration file at runtime
beginTime = 0
endTime = 0

#open spreadsheet
timeSpreadsheet = File.new( Time.now.strftime("%d-%b-%y") + ".csv", "a")  #Note this
creates a new file every day...

#open the IE browser to http://www.BookPool.com
$ie = IE.new
beginTime = Time.now
$ie.goto(testSite)
endTime = Time.now

#Log the time for the Home Page to load
timeSpreadsheet.puts executionEnvironment + ",Home Page," + (endTime - beginTime).to_s

puts 'Step 2: enter "Ruby: in the search text field under Simple Search'
$ie.textField(:name, "qs").set("Ruby") # qs is the name of the search field

puts 'Step 3: submit the search form.'
beginTime = Time.now
$ie.form(:index, "1").submit #submitting first form found in the page.
endTime = Time.now

#Log the time for the search
timeSpreadsheet.puts executionEnvironment + ",Time to execute search," + (endTime -
beginTime).to_s

#All 7 results for Ruby:
puts 'Actual Result: Check that the "All 7 results for Ruby:" message actually appears'
a = $ie.pageContainsText("All 7 results for Ruby:")
if !a
  scriptLog.puts "Test Failed! Could not find test string: 'All 7 results for Ruby:'"
else
  scriptLog.puts "Test Passed. Found the test string: 'All 7 results for Ruby:'"
```

```
end

puts 'Close the browser'
$ie.close()

#close the file
timeSpreadsheet.close
```

Note that if you download the sample code, there will be more comments.

By simply wrapping the page transitions with start and end times, we can calculate the load times for each page and create a makeshift timer. Implementation for this tool is almost identical in Rational, as I imagine it would be with any tool. Notice at the start of the script that we set the environment in which the script is running. This information could be read into the script at runtime, providing the greatest flexibility (at the very least, you could use a global variable in most tools). It might also be possible to use a `gettime()` function, parse it into a variable, and use simple subtraction to calculate the actual load time.

Executing this script produces a spreadsheet similar to the one in Figure 5.

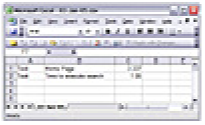

Figure 5

When you're ready to format the data, you can load the macros included at the end of the file, or simply copy the data into a spreadsheet that has the macros.

All said and done, gathering this performance data allowed us to fix problems, better leveraging our automation framework, and gave us a gold star with project management. I've rarely seen so little work pay off so richly. If you happen to be in an environment where implementing this type of performance testing makes sense, I encourage you to try it. Implementation was trivial; four hours for one person, in my case, but it will depend on how your scripts are structured. The benefits are substantial in both practical ways (fix problems) and political ways (wow management). While this isn't a complete performance test, it's a first step that ensures that major performance problems are detected early, leaving the performance testing team to focus on load, stability, and concurrency.

For more information, check out the following:

- Sample code and spreadsheets for this article
- WATIR project home page
- RubyForge project for the tool
- Web Testing with Ruby (WTR) Wiki, where it all started (I think)

About the author

Michael Kelly is a Senior SQA Specialist for CTI Group with experience in software development and testing. Mike has published numerous articles on topics software testing and has presented at several conferences on software testing. Mike is currently serving as the Program Director for the Indianapolis Quality Assurance Association and the Membership Chair for the Association for Software Testing. You can reach Mike by email at Mike@MichaelDKelly.com.