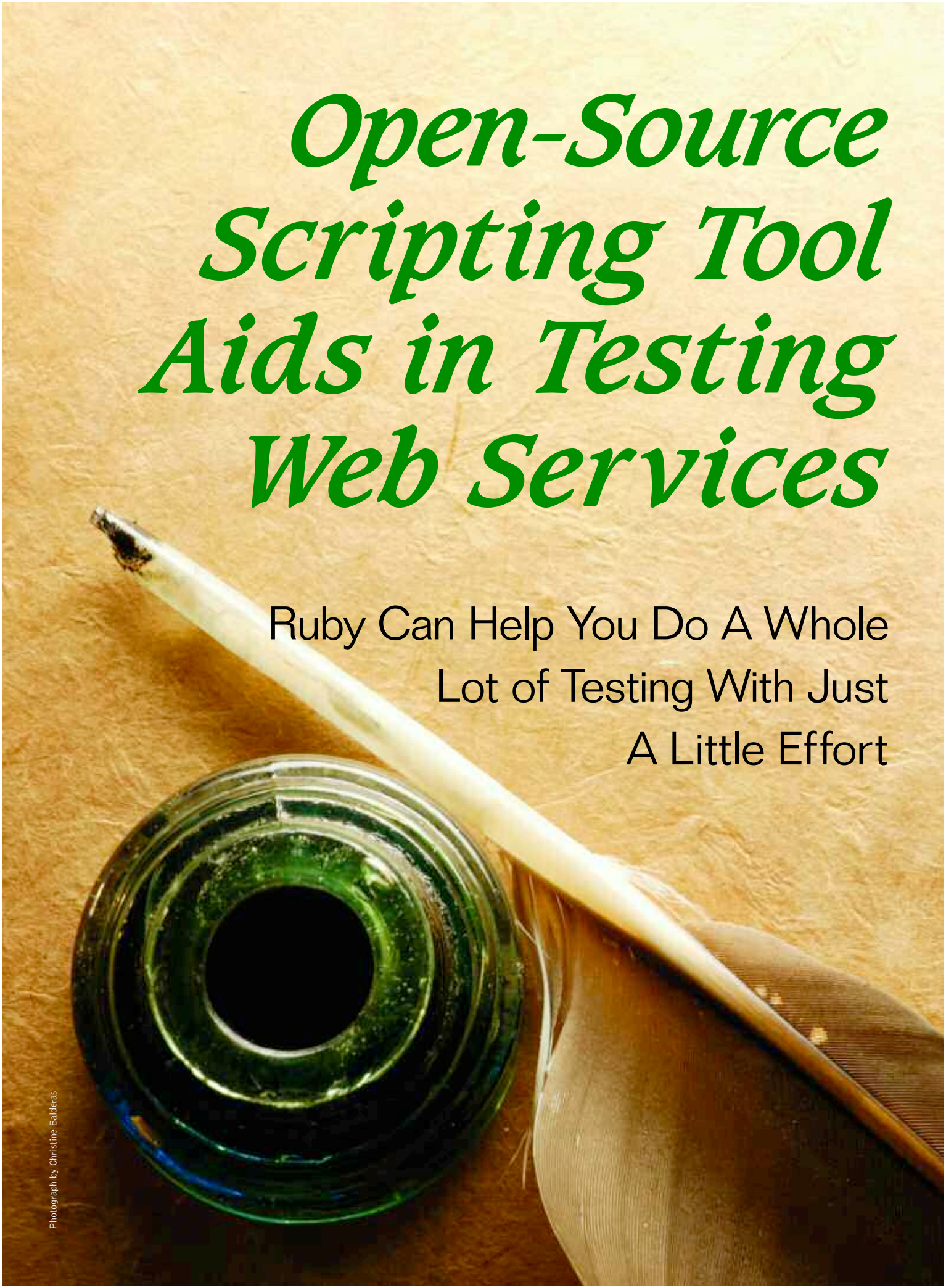
Open-Source Scripting Tool Aids in Testing Web Services

Ruby Can Help You Do A Whole Lot of Testing With Just A Little Effort



Photograph by Christine Balderas

By Michael Kelly

Many systems developed today involve the use of Web services. I will assume you know what Web services are, as well as some of the basic concepts behind them, but you may not know what makes testing Web services unique. The toughest challenge in testing Web services is the absence of a user interface, which often makes it difficult for testers who typically test applications only manually. This means that some level of automation, or the use of specialized tools, is necessary to perform the testing. There are many good tools that can help, but nothing beats creating your tests with a scripting language.

Testing Web services is exciting to me because it requires an intriguing mix of black-box testing skills and coding skills. It also adapts well to automation (no cumbersome GUIs getting in the way) and still requires you to think about the underlying business logic behind the service. In this article, I'll share some of the scripting I've done on a recent financial services project with the open-source, object-oriented programming language called *Ruby*. I used Ruby to create test XML files based on production data, submitted those files to the Web service I was testing, and parsed the resulting XML for validation. I was able to execute thousands of different XML test cases in only a few days of work.

The Ruby code in this article is simple and straightforward, but it's also powerful. It allowed me to do a lot of testing with very little effort. I'll start with some simple code (connecting to the Web service) and end with some more simple code (validating the response I get back). I'll add my disclaimer now: I'm not a developer. This is Ruby code written by a tester for a tester. I make no claim to the fastest or most elegant code—it just works.

We'll walk through the Ruby code a step at a time. Looking at code in this manner can sometimes be distracting for some people. Often, it helps to see all the code at once. For those of you who would like to see the sample code in its entirety, you can download the Ruby script at www.stpmag.com/downloads/stp-0607_galen.zip.

Using Ruby to Connect to A WSDL Service

The first thing you need to do is establish a connection to your Web service and get some data moving back and forth between your script and the service. Ruby offers many ways to do this, none of which is particularly well documented. In the end, I had to team up with a developer to get the final code working, but when all was said and done, it took us only a couple of days, part-time, to get it figured out.

The first service we connected to was a WSDL (Web Service Description Language) service. We used the Ruby Simple Object Access Protocol (SOAP) libraries to make our call. The basic logic goes as follows:

1. Take an XML file (for us, each XML file was a test case) and wrap it in a SOAP envelope. *Note: A SOAP envelope element is the root element of a SOAP message. It defines the XML document as a SOAP message.*
2. Create a connection to the WSDL service.
3. Send the SOAP envelope to the service (HTTP post).
4. Wait for a response.
5. Receive the response XML (or the error).

Following is the code we used to implement that logic. I'll walk you through it one section at a time, but if you're just getting started, Brian Marick has a better "first look" article on scripting with Ruby. I have a full reference to it below.

First, we need to include the various soap libraries we will be using:

```
require    'soap/element'
require    'soap/rpc/driver'
require    'soap/processor'
require    'soap/streamHandler'
require    'soap/property'
```

Next, we declare a constant for the endpoint (the *endpoint* is the location of the Web service) and we assign our XML to a variable:

```
LOCALHOST_ENDPOINT = "http://local-
host:8080/services/Service"
request_xml_string = 'xml...'
```

In the code snippet above, request_xml_string is assigned the XML request that the Web service is expecting. The next line creates a connection to the service using HTTPStream Handler:

```
stream = SOAP::HTTPStreamHandler.new
(SOAP::Property.new)
```

Once we've defined the connection, it's time to build the SOAP envelope, which is made up of a header and a body. The following creates the header and the body, using the service method getResponse and the XML stored in the variable above:

```
header = SOAP::SOAPHeader.new
body_item =
SOAP::SOAPElement.new('getResponse',
request_xml_string)
body = SOAP::SOAPBody.new(body_item)
envelope = SOAP::SOAPEnvelope.new(header, body)
```

Now we're ready to create the actual request string using the envelope and send it to the service. Once it's sent, we wait for the response to be returned and store the response value in the variable resp_data:

```
request_string = SOAP::Processor.marshal(envelope)
request =
SOAP::StreamHandler::ConnectionData.new(request
_string)
resp_data = stream.send(LOCALHOST_ENDPOINT,
request, 'getResponse')
```

The above code creates the XML string from the Ruby SOAP objects, sends the string and stores the response. I know I glossed over some of that, but that's because I got some help from a real developer. There are several different ways to do this, and this is just one example. If, like me, you aren't a Ruby wizard, I recommend working with a developer to get your connection

Michael Kelly is an independent consultant who focuses on test automation and exploratory testing. Kelly also serves as a director at large for the Association for Software Testing. You can reach him by e-mail at Mike@MichaelDKelly.com

code working. It's the only tricky part in this entire article.

Once we had a working script that allowed us to send our XML test cases, we were ready to start testing. Initially, we validated all the result XML manually—nothing beats the power of the brain—but over time, that work became tedious and not very fruitful. We then started building in automated validation for certain XML elements as they came back from the service. Once that was done, we had only to check for test case–specific information when validating results.

Using Ruby to Generate XML

We used Ruby and JUnit (see the *Using Ruby With JUnit* sidebar for more information) for all of our traditional black-box functional testing, but we wanted more. All of the Web services we were testing were new interfaces to existing financial services applications. That meant we needed to do a very large amount of regression testing for a very large amount of data. If you can't tell from the process discussed above, each planned test case we executed was a fair amount of work.

We needed a faster way to regression test; something that would allow us to generate a lot of test cases, run them, do some basic validation, and then do manual testing in areas where problems were identified. So we decided to start using Ruby to generate XML test cases based on existing production data. That process turned out to be simpler than we thought it would be, going pretty much like this:

1. Open the file containing all the policy data that will be used to generate the XML files.
2. For each line in the file, create a new XML file with that policy data.
3. Close the file.

We could do this because the service we were testing was a simple query service. It would be more difficult if the service were more complex.

The XML we generated followed the ACORD standard (insurance.xml .org/standards), and all the data in the file dump was very basic policy data: policy number, contract number and version number. With those three values we could generate our request for the service. With the following code, we were able to generate around 500 test cases every second (depending on the size of the data dump we sent it).

We used Ruby and JUnit for all of our traditional black-box testing, but we wanted more.

The first line in the code opens the file containing all the policy data dumped from production:

```
myDataFile = File.open('C:/Service Testing/Service A Testing/12.22.05 - policy data dump.txt', "r")
```

Then, for each line in the file (myDataFile) we want to create a test case file (or XML request):

```
myDataFile.each { | line |
    #Build the XML file...
}
```

The .each method executes the block of code following it for each line in myDataFile. The current line is stored in the line object.

Following the creation of all the test cases, we close the data file:

```
myDataFile.close()
```

The rest of the code in this section goes inside the each block (where we have the comment *Build the XML file…*). The first thing to do once we read in each line is split up the data for ease of use. In Ruby, we have the .split() method, which does just that. The following line of code parses the data wherever it finds a space between the data elements. It then stores that data in an array:

```
policyInfo = line.split(' ')
```

That code splits the policy information, storing the policy number in policyInfo[0], the policy version in policyInfo[1], and the contract number in policyInfo[2]. Next, we need to create the new XML file for each line. Something similar to the following should work for that:

```
newXMLFile = File.open('C:/Service Testing/Service A Testing/Request XML/' + policyInfo[0] + '-rq.xml', "a")
```

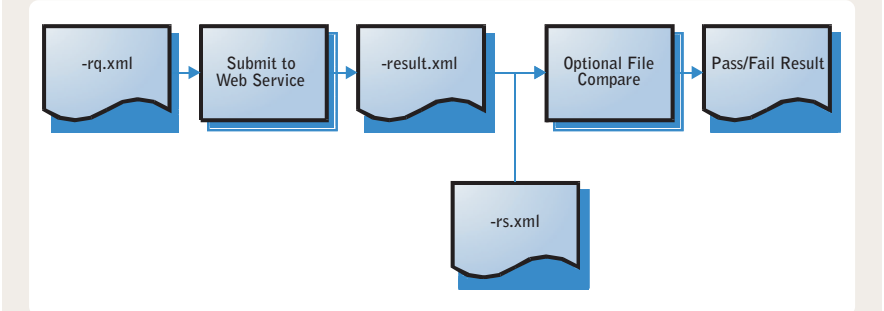
Notice that in the code above, we name the XML file according to the policy number. That's because the policy number is the only thing that's unique in the data. We also had an XML file-naming convention that included three possible extensions: –rq,–rs and –result.

The -rq file contains the request XML for the call to the Web service.

The -result file contains the actual result XML for the call to the Web service. This file can be compared to the expected result file (-rs) to see if the service returned the correct XML response. This file is automatically overwritten with every call to the Web service.

The -rs file contains the expected result XML for the call to the Web service. This file can be compared to the actual result file (-result) to see if the service returned the correct XML response. This file must be manually

FIG. 1: XML FILE-NAMING CONVENTION



LISTING 1

```
newXMLFile.puts('<?xml version="1.0" encoding="UTF-8"?>')
newXMLFile.puts('<ACORD
xmlns="http://www.ACORD.org/standards/PC_Surety/ACORD1.7.0/xml/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">')
newXMLFile.puts('      <SignonRq>')
newXMLFile.puts('                  <ClientDt>2005-12-22</ClientDt>')
newXMLFile.puts('                  <CustLangPref>en-US</CustLangPref>')
newXMLFile.puts('                  <ClientApp>')
newXMLFile.puts('                              <Org>Financial Services Company</Org>')
newXMLFile.puts('                              <Name>com.service.serviceA</Name>')
newXMLFile.puts('                              <Version>0.1</Version>')
newXMLFile.puts('                  </ClientApp>')
newXMLFile.puts('      </SignonRq>')
newXMLFile.puts('      <InsuranceSvcRq>')
newXMLFile.puts('                  <RqUID>00000000-0000-0000-0000-
000000000000</RqUID>')
newXMLFile.puts('                  <PolicySyncRq>')
newXMLFile.puts('                              <TransactionRequestDt>2005-12-
22T13:47:45.00000-05:00</TransactionRequestDt>')
newXMLFile.puts('                              <AsOfDt>2005-12-22</AsOfDt>')
newXMLFile.puts('                              <Producer>')
newXMLFile.puts('                                          <ProducerInfo>')
newXMLFile.puts('                                                      <ContractNumber>' +
policyInfo[2].chomp() + '</ContractNumber>')
newXMLFile.puts('                                          </ProducerInfo>')
newXMLFile.puts('                              </Producer>')
newXMLFile.puts('                              <PolicyNumber>' + policyInfo[0] +
'</PolicyNumber>')
newXMLFile.puts('                              <PolicyVersion>' + policyInfo[1] +
'</PolicyVersion>')
newXMLFile.puts('                  </PolicySyncRq>')
newXMLFile.puts('      </InsuranceSvcRq>')
newXMLFile.puts('</ACORD>')
```

overwritten or updated.

Figure 1, which depicts the XML file-naming convention and process for file use, shows how we used those files in our testing.

Building the XML file is actually a trivial exercise (or task) that amounts to little more than a bunch of puts commands for each line of the XML we want to write. Puts writes out a string to an I/O object (in this case, a File) and appends a new-line character if the string doesn't already end in one. We just copied and pasted the XML into our Ruby script, wrapped it with puts commands and inserted the few unique values we needed in the appropriate places. For example, look at Listing 1.

Finally, before we finish with the new XML file, we want to close it:

```
newXMLFile.close()
```

Executing the above code yields an XML request file for each policy in the data dump. Once we had the test data, all we needed to do was run it through the Web service using our earlier script. But that's not all we wanted to do. Since we're talking thousands of test cases with multiple points where each can fail, we also wanted simple validation and detailed logging of results. User-friendly logging is a must. We wrote our results to csv files for easy formatting and sorting.

Using Ruby to Validate XML and Log Results

Once we had our bazillion test cases, we needed to do the following:

1. Delete the old results file before running the test and create a new one (I could have just created a new one with a time-date stamp, but I took care of versioning a different way).
2. Get all the files in the directory and sort them.
3. For each file in the directory, execute the service call, check for errors, and validate the results that are returned.
4. Close the results file.

First, we want to delete the old results file and create a new one stored the path to the file in a variable to make the code a bit more readable:

```
bulkTestResults = 'C:/Service Testing/Service A
Testing/Test Results.csv'
if File.exists?(bulkTestResults) then
File.delete(bulkTestResults) end
myResultsFile = File.open(bulkTestResults, "a")
```

Next, we need to find all the files in the directory. I like to sort them by name so that my results are sorted by policy number (that's how I named them when writing the XML). In the following code, the glob method returns an array of file names matching the specified wildcard pattern (I use * just to capture all the files in the directory).

```
eachFileInDirectory = Dir.glob('C:/Service
Testing/Service A Testing/Request XML' + '/*').each
{ | file | file.downcase }.sort
```

For each file returned we change all the characters in the name to lowercase and then once all of them have been changed to lowercase we sort the array eachFileInDirectory. Changing the file names to all lowercase renders the results easier to read in the csv because it makes all the file names consistent.

Again, we'll need to use the each method. For each file in the directory, we want to process each test case file:

```
eachFileInDirectory.each do | testCaseFile |
        #Do something with the file...
end
```

At the end of the script, we need to close the results file:

```
myResultsFile.close()
```

Within the testCaseFile loop, we use a flag for test case failures. That way, we know to stop processing the other checks we perform on the response when a failure occurs. So the first thing we do when we enter the loop is reset that flag for the test case being executed:

```
testCaseFail = false
```

Next, we use the code covered above to send the request to the service and capture the response. The response then gets written out to the "–Result" file. That's done for debugging and archiving purposes. Next, we'll look at how to open the results file and validate the results.

In the code that follows, I'll show an example of validating only one type of element (the policy number). You can repeat that code as much as you like for any elements you'd like to validate.

They all look the same; just change the XML element and/or the value you're looking for. In this example, we're going to verify that the results we got back were the results for the correct policy number. If they aren't, then we'll log a message indicating that we didn't get the policy number we expected (or that we couldn't find the XML element containing the policy number).

The first thing we do is create a flag that indicates whether or not we found the XML element <PolicyNumber>:

```
tagFound = false
```

Then, we check to see if there was a failure in an earlier test validation check. If there was, we don't continue:

```
if not testCaseFail then
        #Rest of code here...
end
```

Inside that if statement, we open the results file for the test we just ran:

```
myTestCaseFile =
File.open(testCaseFile.chomp.split('\-')[0] +'-
results.xml')
```

Then we check each line in the results file for the <PolicyNumber> element:

```
myTestCaseFile.collect do | line |
        if line =~ /<PolicyNumber>/ then
                #More code here...
        end
end
```

In the code above, /<PolicyNumber>/ is a regular expression. If <PolicyNumber> is anywhere in the line, then =~ evaluates to true. If that regular expression evaluates to true, we'll want to check to make sure that the <PolicyNumber> element contains the correct policy number. If you remember from above, the name of the test case file is also the policy number. The next chunk of code uses the .include? method (which evaluates to a Boolean value) to check to see if the policy number is there. Notice that we perform some operations on the testCaseFile variable. All we're doing is isolating the actual file name from the full path to the file. We're stripping off both the path and the file extension. If the policy number isn't found, we log an error to our .csv results file and set the testCaseFail flag to true.

```
if not line.include?(testCaseFile.split('/')[-
1].chomp.split('\-')[0]) then
myResultsFile.puts(testCaseFile.split('/')[-1].chomp
+ ', FAIL, -result.xml, Could not find
<PolicyNumber> element that matched the policy
number requested.')
                testCaseFail = true
end
```

Whether we found the correct policy number value or not, before we exit our if statement we also need to set our tagFound flag to true. That way, we won't log an error indicating that we couldn't find the tag.

```
tagFound = true
```

Next, we want to close the test case results file:

```
myTestCaseFile.close()
```

And then we perform a check to see if we actually found the element we were looking for. If tagFound is false, we log an error and set testCaseFail to true:

```
if not tagFound then
myResultsFile.puts(testCaseFile.split('/')[-1].chomp
+ ', FAIL, -result.xml, Could not find
<PolicyNumber> element in response.')
        testCaseFail = true
end
```

We then repeat all that code for each verification we want to perform on the XML. I've thought about optimizing the code to make it more modular, but it works, and I currently perform a handful of checks on each file. At some point (once I have to start scrolling my script more than a couple of times) I'll rework the code to make it more modular.

That's all there is to it. These scripts made it possible to create several hundred request test case files in a few seconds and then execute them all and perform simple verifications on the results within about 45 minutes. All the results were logged in a .csv file that we could auto-filter based on result or error message.

This made it easy to identify all the affected policies when submitting defects. We were able to do all our traditional functional testing using hand-coded XMLs and submitting them using Ruby and JUnit, and then we did some high volume automation and found more than a handful of very obscure and hard-to-anticipate issues.

Scripting With Ruby Resources

As you begin your journey with Ruby, you can find advice, new ideas and friendly support at several rich resources.

First, check out a couple of articles by Brian Marick: "Behind the Screens" (www.testing.com/writings/behind-the-screens.pdf) and "Bypassing the GUI" (www.testing.com/writings/bypassing-the-gui.pdf). Marick is also working on a book titled "Scripting for Testers: Using Ruby" (Pragmatic Bookshelf, 2006), which will be a must-read for testers.

Another great place for Ruby tips and tricks for testers is www.Watir.com, which stands for Web Application Testing in Ruby. WATIR is an open-source test tool for automated testing with Ruby. They have a strong mailing list (which is searchable) and a lot of helpful people.

Finally, grab a copy of "Ruby in a Nutshell" by Yukihiro Matsumoto (O'Reilly Media, 2001). I've found it a must-have for writing Ruby code.

SMOKE TESTING WITH RUBY AND OUTLOOK

One little trick we picked up along the way was to have Outlook kick off our smoke test scripts (or build verification tests—depending on your preferred nomenclature). We were testing several services, and each of them ran in several different environments. So we wanted our smoke test scripts to be available to the project team for any service, in any environment, at any time. The problem with that is that only a couple of us had Ruby installed (it was not an "approved" technology at the time).

We needed a way for others to execute tests on our machines. Enter Outlook. All we needed to do was add an Outlook rule that would launch the correct script on the local machine based on the subject line of the e-mail. The requestor would send a headline of "Smoke Test" followed by the service name and environment, and Outlook would launch a batch file (I don't believe that Outlook can run Ruby scripts) that would launch the Ruby script. The test would execute and e-mail them the results.

It would also run in the background—so if I was working, I wouldn't even know it had happened (unless I was unusually observant that day). If you use the rubyw.exe instead of ruby.exe to execute your scripts from the batch file (rubyw.exe doesn't launch a DOS shell when it runs), you shouldn't even notice it running.