
Behind

A skilled tester takes you on a guided exploration of a Web services interface.

BY BRIAN MARICK

the Screens

JAMES BACH HAS DEFINED EXPLORATORY TESTING AS "SIMULTAneous learning, test design, and test execution." Regrettably, this useful approach is often applied only to products with graphical user interfaces. Too few testers have the skills and tools needed to apply it more broadly. This article aims to give you a start on developing the skills you need to begin exploring beyond GUI products. It will also introduce you to a tool—the open source Ruby scripting language—and show you how to use that tool to explore Google's Web services interface. You don't need any programming experience to read this article.

There are some things this article will *not* teach you. This article will give you some feel for exploratory testing's flow and pacing, but it won't teach you exploratory testing or any aspect of test design. Go to James Bach, Elisabeth Hendrickson, Cem Kaner, or others for those skills. (See this issue's StickyNotes at bettersoftware.com.) This article will not teach you the crunchy details of Web services. Part of the message of this article is that

you don't need to know the details, given the proper tools. Oh, you'll test better if you do, so you should learn them eventually, but you don't have to *start* with the details; you can work your way into them.

Because of space limitations, I won't get very far into testing Google, and I'm sad to say I won't show you any Google bugs. But I hope that I'll take you to the point where you can clearly see how you could continue on with what I've started. To that end, I recommend you stop reading now and download Ruby onto your machine. (See this issue's StickyNotes for instructions.) C'mon! It'll be fun!

Introducing Narcissism Unlimited

Let's pretend I'm a tester working for Narcissism Unlimited, which runs a website catering to the deservedly famous (Motto: "All Things for the Self-Absorbed"). Recently, the marketing department tore its gaze away from the ceiling-height office mirrors long enough to conceive of a new feature for the NU home page. The NU home page is customized to each client (of course). The marketing department wants to add a feature to that home page so that each time a client visits it, he sees a different favorable quote about himself, culled from somewhere on the Web.

A technical team is put together to make this feature a reality. I'm assigned to the team as "chief tester." We know that Google provides a Web services interface to its search engine. Having this interface means that *programs* can do searches, not just people visiting the Google webpage. Our preliminary thought is that this request by marketing is possible: The results of a Google search on a client name could be massaged to pick a new favorable quote of the day—if, that is, the Google Web services interface is solid enough. It's my job as chief tester to see if it is.

Writing Programs

Web services are programs that are used from across the Internet by other programs. If I'm going to evaluate Google's Web service, I have to write little programs. The NU programmers are likely to write programs in Java or C#. I'm going to use a language called Ruby, because it makes it easier to do the kinds of things I want to do.

One nice thing about Ruby is that it has an *interpreter*. An interpreter lets me write little snippets of program and immediately see what they do. That makes exploration easy.

Ruby's interpreter is named "irb". You start it by typing those characters to a command prompt or shell or, in Windows, selecting Start→Program Files→Ruby→Interactive Ruby Shell. When you do, here's what you see:

```
irb(main):001:0>
```

All that gobbledygook is irb's prompt. Every character has meaning, but none of it matters for this article. In order to save horizontal space, I'll abbreviate the prompt from now on.

I can type something to the prompt, and Ruby tells me its value on the next line:

```
irb> 1+1
=> 2
```

Here's another example. It looks rather different:

```
irb> "hello".upcase
=> "HELLO"
```

What I'm doing here is sending the **upcase** *message* to the text *string* **"hello"**. The resulting value is the uppercase version of that string.

Working with the Google Web services will look like working with strings. Strings are *objects* that respond to messages. Ruby makes the Google Web service into an object that responds to messages. Those messages trigger much magic behind the scenes, magic that delivers information to a server somewhere, gets results back to me, then displays them. But I don't see any of that magic, because I don't need to.

You now know almost enough programming to understand the rest of this article. I'll explain a bit more as I go along.

Connecting to Google

I'll begin by skimming Google's Web services documentation (at api.google.com), reading just enough to get me started. I want to see Google in action as quickly as possible. After that, I'll step back and strategize. Who knows, I may discover immediately that Google is unsuitable. In that case, I'll be glad I did not spend much time on the documentation.

My first task is to find out how a program can connect to, and communicate with, Google's Web service. I quickly notice that Google uses WSDL (Web Service Definition Language), a standard way for Web services to advertise how programs should talk to them. Many programming languages and environments can take a WSDL document and, from it, construct objects and messages that a program can use. Ruby is one of those languages.

Ruby's WSDL package has sample programs for four different Web services. Each program connects to its Web service the same way, so I begin exploring by copying that sequence of commands. Here's the first:

```
irb> require "soap/wsdlDriver"
=> true
```

The command tells Ruby that I'm going to use the WSDL package. The value **true** means that the package is available for use.

```
irb> wsdl = "http://api.google.com/GoogleSearch.wsdl"
=> "http://api.google.com/GoogleSearch.wsdl"
```

The long string names a location where my program can find a WSDL document that describes Google's Web service. (I got the location from the Google documentation.) I *assign* that name to a *variable* named **wsdl**. Whenever I need the string, I can use the shorter variable instead.

The next thing I type looks like this:

```
irb> factory = SOAP::WSDLDriverFactory.new(wsdl)
=> #<SOAP::WSDLDriverFactory>
```

The command looks alarming because of the colons and parentheses and periods and the way all the words are jammed together without spaces. First, let me translate it into English: "Make a *new factory* for *WSDL drivers* using the variable **wsdl** that I just created." Once you're used to programming, that kind of translation becomes second nature.

Translating the command only partly helps. What's a factory? Well, I happen to know that "factory" is programmer slang for an object that makes other objects. So the variable **factory** now names an object that I can ask to make me some "drivers." What are those? I suspect they're something you use to talk with (or "drive") Google. I'll find out for sure as I explore more.

The result of the command tells me that the factory is, specifically, a **WSDLDriverFactory**. (We'll see a more informative example of this kind of output later.)

The next command is the last one I copy from my examples. It looks like this:

```
irb> driver = factory.createDriver
=> #<SOAP::WSDLDriver>
```

Let's take stock. I went through four steps to connect to Google. I'd use the same four steps to connect to any Web service; the only difference is the string I give as the value for **wsdl** (the second step). Now that I'm connected to Google, it's time to explore.

Exploring Google

What can I do with **driver**? How, exactly, do I talk with Google? Again, I could read the documentation, but let's see how far I can get without it. Ruby has a nice feature: You can ask any object what messages it responds to by sending it a message named **methods**. (A *method* is the code snippet that responds to a message. Other languages call such things *functions*, *subroutines*, or *procedures*.) I do that for **driver**:

```
irb> driver.methods
```

The result is a list of about 100 message names. Many of them are ones you can send to any object, be it a factory, a number, or a string. But some others catch my eye because they seem Google-specific:

```
[…, "doGetCachedPage", … , "doSpellingSuggestion", …,
"doGoogleSearch", …]
```

I know from years of using Google through my Web browser that it caches pages so that they can be viewed even when the original site is unavailable, suggests alternate spellings for search terms, and—most of all—searches.

The message **doGoogleSearch** seems like a useful one to try. Monster of vanity that I am, I search for myself:

```
irb> driver.doGoogleSearch("Brian Marick")
ArgumentError: wrong number of arguments(1 for 10)
```

Rats. The message **doGoogleSearch** actually takes ten (ten!) arguments, not the single one I gave it. There's no way I'll guess all those arguments, so I'll have to browse the documentation. A document called the *API Reference* tells me what the ten arguments are. They are:

1. A key that identifies me to Google. I got this when I downloaded the documentation.

2. The search string.

3. The "index" of the first desired result. (I'll explain that next.)

4. The maximum number of results I want. It can be a number between one and ten. If I want more results, I have to repeat the search and use the third argument to tell Google I want the next ten.

5. Whether Google should filter the results to avoid near-duplicates. This value is either true or false.

6–10. A bunch of stuff I don't think I care about just now.

Armed with my newly acquired key, I can now do a search that works.

```
irb> result = driver.doGoogleSearch("mykey", "Brian
Marick", 0, 10, true, " ", false, " ", " ", " ")
```

If you're trying this at home, you'll have to substitute the key you get from Google for ***mykey*** above. (Be sure to keep the quote marks.)

Notice that the index of the first desired result is 0. Most computer languages start counting from zero. It's just something you have to get used to. Notice also that I had to give values even for the five parameters I don't care about. Google insists on it.

This works and produces a blizzard of what looks like gobbledygook:

```
=> #<SOAP::Mapping::Object:0x623a28 @searchQuery="Brian
Marick", @searchTime=0.11519, @documentFiltering=true,
@startIndex=1, @searchTips=" ", @resultElements=
[#<SOAP::Mapping::Object:0x621ebc @summary=…
```

What you're looking at is Ruby's way of displaying an object when no programmer told it a better way. (Earlier we saw it display numbers, strings, factories, and drivers. In each case, someone told Ruby how to display them.) I call this the **#<*stuff*>** notation. It has two parts. The first identifies what type of object it is (a **SOAP::Mapping::Object**), together with a peculiar number (**0x623a28**) that is almost always irrelevant. (It's the location of the object in memory.) The rest of the ***stuff*** shows the components that make up the object. The first is **@searchQuery**, whose value is the string **"Brian Marick"**. (The proper Ruby term for

"component" is *instance variable*. I'll use "component" for this article.)

There's interesting information in that solid block of text, but it's hard to pick out. Fortunately, a kind programmer named Akira Tanaka has made it easier. He wrote a package named "pp" (for "pretty print"). I'll use that to put each component on its own line, nicely indented. You can see the first part of the results in Figure 1. To save space, I've omitted uninteresting components like **@searchQuery** and **@searchTime**.

What have I got here? The **result** is some object that has a component named **@resultElements**. What's that? Its value has this form, spread out over several lines:

@resultElements=[*SOAP::Mapping::Object, SOAP::Mapping::Object, …*]

```
irb> require "pp"
=> true
irb> pp result
#<SOAP::Mapping::Object:0x62c010
    @resultElements=
        [#<SOAP::Mapping::Object:0x627b3c
            @URL="http://www.testing.com/",
            @snippet=
                "Testing Foundations Consulting in Software Testing <b>Brian</b>
<b>Marick</b>. <b>...</b> I&#39;m <b>Brian</b><br> <b>Marick</b>. Welcome.
This is my web site devoted to software testing. <b>...</b> ",
            @summary=
                "Home page of <b>Brian</b> Marrick, author of the book 'The Craft Of
Software Testing'.",
            @title="Testing Foundations - <b>Brian</b> <b>Marick</b>">,
        #<SOAP::Mapping::Object:0x623fb4
            @URL="http://www.testing.com/cgi-bin/blog", …
```

Figure 1: Prettier search results

That's the Ruby way of printing lists or arrays. Each object in the array is evidently a single search result. (I asked for up to ten of them.) I look at the first one.

The **@URL** is www.testing.com. That's great! It's my main website. The following **@snippet** is fine— I recognize the text from my webpage. The **@summary** is reasonable, but it spells my name wrong. Where did *that* come from? It's certainly not from www.testing.com!

I go back to the *API Reference* to see what it tells me about the summary component of a search result: "If the search result has a listing in the ODP directory, the ODP summary appears here as a text string." That's not enormously helpful because the document nowhere defines "ODP". However, it's short work (using Google) to find that ODP is the Open Directory Project, which aims to be "the most comprehensive human-reviewed directory of the Web." Some kind person submitted my site, but misspelled my name. Oh well. I make a note that NU might want to include the ODP in our new personalization feature, then move on.

Making Exploring Easier

At this point, I want to do some more searches, but it's annoying that **doGoogleSearch** requires ten arguments. I don't want to type those all the time (and retype them when I make the inevitable mistakes). What I'd rather do is type **search("Brian Marick")** and have that **search** method handle the details for me. That's easy to do with a *method definition* that tells Ruby how to take an argument to **search** and substitute it into a use of **doGoogleSearch**:

```
def search(for_string)
    driver.doGoogleSearch("mykey", for_string, 0, 10, true, " ", false,
    " ", " ", " ")
end
```

For convenience, I'll put that definition into a file named "goog.rb" so that I can use it each time I start irb. Unfortunately, however, **search** won't work. Let's see it not work:

```
irb> load "goog.rb"
=> true
irb> search("Brian Marick")
NameError: undefined local variable or method 'driver' for main:Object
    from ./goog.rb:2:in 'search'
    …
```

The error message tells me that **driver** isn't available within the function **search**. The next line tells me the problem happened on line 2 of "goog.rb". A simple fix is to make **driver** into a *global* variable (one that's available inside every definition) by prepending a dollar sign: **$driver**. Also, my four lines of setup have to use **$driver** instead of **driver**. It seems easiest to put those four lines in "goog.rb" so that loading it does everything I need. Here's the result:

```
require "soap/wsdlDriver"

wsdl = "http://api.google.com/GoogleSearch.wsdl"
factory = SOAP::WSDLDriverFactory.new(wsdl)
$driver = factory.createDriver

def search(string)
    $driver.doGoogleSearch("mykey", string, 0, 10, true, " ", false,
    " ", " ", " ")
end
```

I also decide the pretty printed results, while nicer than the raw results, are still not convenient enough to read. So I add a **show** function that takes the results and prints them to my taste. See Figure 2. For the sake of brevity, I won't explain that method here.

You can find a complete explanation in this issue's StickyNotes.

After all that, I load the new definitions and use them:

```
irb> load "goog.rb"
=> true
irb> show(search("Brian Marick"))
```

These nested parentheses may be confusing, so think of what happens in two steps. First, "Brian Marick" is given to the **search** function. The **search** function uses it as one of the arguments to **doGoogleSearch**, which produces the complicated result we saw before. That result is given to the **show** function, which converts it into pleasant text. Part of the outcome is shown in Figure 3.

Getting Stuck

So now I have tools that make exploration easier. The last item in Figure 3 catches my eye. NU's clients like to have their names linked to other people. What would happen if I searched for all three names in this list?

```
irb> show(search("Mark Swanson
Brian Marick Ralph Johnson"))
XSD::ValueSpaceError: {http://www.w3.org/
2001/XMLSchema}string: cannot accept
'<b>...</b> Ron Jeffries <b>Ralph</b>
<b>Johnson</b>, University of Illinois
Joshua <b>...</b> Tony Gould, John
Isner,<br> <b>Brian</b> <b>Marick</b>,
Ralf Reissing, John Salt, <b>Mark</b>
<b>Swanson</b>, Dave Thomas Âíå
<b>...</b>'.
    from /usr/local/lib/ruby/1.8/xsd/
    datatypes.rb:184:in '_set'
    …
```

```
def show(results)
    for result in results.resultElements
        show_one(result)
    end
    puts "There were #{results.resultElements.length} results."
end

def show_one(result)
    puts("URL is #{result.URL}")
    puts("title is #{result.title}")
    puts("summary is #{result.summary}")
    puts("snippet is #{result.snippet}")
    puts("--------------")
end
```

Figure 2: The show function and its helper

```
URL is http://www.testing.com/
title is Testing Foundations - <b>Brian</b> <b>Marick</b>
summary is Home page of <b>Brian</b> Marrick, author of the book 'The Craft Of
Software Testing'.
snippet is Testing Foundations Consulting in Software Testing <b>Brian</b>
<b>Marick</b>. <b>...</b> I&#39;m <b>Brian</b><br> <b>Marick</b>. Welcome.
This is my web site devoted to software testing. <b>...</b>
--------------
URL is http://www.testing.com/cgi-bin/blog
title is Exploration Through Example
summary is
snippet is Exploration Through Example. Agile testing, context-driven testing, agile
programming,<br> Ruby, and other things of interest to <b>Brian</b>
<b>Marick</b>. <b>...</b> About <b>Brian</b> <b>Marick</b>. <b>...</b>
--------------
URL is http://c2.com/cgi/wiki?BrianMarick
title is <b>Brian</b> <b>Marick</b>
summary is
snippet is <b>Brian</b> <b>Marick</b>. I used to write for MarkSwanson and The
Wild Hunt. I took the first<br> course in Smalltalk that RalphJohnson taught at the
University of Illinois. <b>...</b>
```

Figure 3: Results from the Google Web service

Crud. An error. And deep within a Ruby package, too, judging by where the message comes from. What's going on?

Notice anything odd in the message? What's that funny text after the name Dave Thomas? There's an easy way to find out—try the same search with Google's browser interface. I see odd text like that in the first, fifth, sixth, and seventh search results. And all four point to Japanese language webpages—each of them a translation of the preface from Martin Fowler's fine book *Refactoring*. When I click through, I discover those mysterious characters are the translation of "and", judging by their position between the next-to-last and last of a series of names. Oho! I think I've got multilingual character-set issues, something I know nothing about.

(Honesty compels me to admit that the previous paragraph is only the path I *should* have taken. In reality, I wasted about fifteen minutes because I'm not as observant as an exploratory tester should be. I missed the Japanese language connection but got to the same realization via another route.)

I try some more searches, and it turns out that many Google results contain odd characters that break my little program. How do I go forward? Are my tools useless? If not, how can I fix them? Fortunately, there's a mailing list for Ruby enthusiasts. I post a question and four hours later have a reply from Hiroshi Nakamura. He gives two ways to fix my problem. I choose to set the global variable **$KCODE** to "UTF8" (the correct character set), and my problem goes away. After

Why Ruby?

I picked Ruby for this article because it's an easy language to learn and use, yet powerful enough to do most anything you want. Other languages would do fine, so long as they have an interpreter and enough of a user base that the language's packages keep pace with new technologies like Web services. The Python language is a fine alternative. Were I testing Java APIs, I'd probably choose it. There's a variant named Jython that's particularly well suited for such testing.

making a note to investigate character set issues more carefully later, I move on.

When I repeat the search, the results are pleasing. The first search result is this:

```
URL is http://www.aw-bc.com/catalog/academic/product/
0,4096,0201485672-PRE,00.html
title is Refactoring: Improving the Design of Existing Code - Addison-
<b>...</b>
summary is
snippet is <b>...</b> Ron Jeffries; <b>Ralph</b> <b>Johnson</b>,
University of Illinois; Joshua <b>...</b> Tony Gould, John Isner,<br>
<b>Brian</b> <b>Marick</b>, Ralf Reissing, John Salt,
<b>Mark</b> <b>Swanson</b>, Dave Thomas <b>...</b>
```

That's Addison-Wesley's webpage for *Refactoring*. Narcissist that I am, it pleases me to find my name linked to that of a luminary like Martin Fowler. It seems there's something to my idea of using repeated searches to further inflate our clients' egos. I jot down a note about it and move on. It may seem odd for a tester to be thinking about product features, but I don't think it should be.

Moving Forward

Although I've barely begun my imaginary mission of evaluating Google, I'll stop my exploration here. I hope I've convinced you that interpreter-assisted exploration can be as productive, fun, and easy as manual exploration.

But none of that's true—at first. Imagine a super-intelligent toddler doing exploratory testing against a product's GUI. She would become extraordinarily frustrated because her ideas would greatly outstrip her manual dexterity. Every thought would be such a labor to put into practice that she'd never achieve a "flow" state, never reach the point of effortless creativity.

Something similar will happen to you as you use even a friendly, interpreter-based programming language like Ruby or Python. If you don't know programming, you won't have the right reflexes. Things will work fine when you follow—exactly!—a script like the one in this article. But when you make a mistake, the interpreter will spit out some strange result. It might make immediate sense to an experienced programmer, but it will mean nothing to you. You have to stop and puzzle it out. Bang! There goes the flow.

Toddlers learn to manipulate the world with skill because they persevere. You'll have to persevere too, until it becomes nearly effortless to turn your ideas into Ruby.

It's hard to persevere alone. I suggest you learn in pairs. Each of you will keep the other from giving up too soon. And, since so much of learning programming is having "Aha!" moments where something completely obscure suddenly becomes clear, you'll learn faster when you share than you would alone.

Perhaps the best pairing would be a tester with a programmer, where each trades knowledge with the other. The tester teaches exploratory testing; the programmer teaches programming ideas. (See Jonathan Kohl's article in the January issue of *Better Software*.)

Some of your beginner's frustration will be increased if you're not used to open source products targeted to programmers. Languages like Ruby consist of a stable core surrounded by packages contributed by members of the community. Some of those packages will be unfinished, though still useful. That's especially true of those that track new technologies like Web services. And it's the documentation that's likely to be the most unfinished.

Because of that, your early exploration of something like Web services may also be an exploration of your language's Web services package. Some people find that maddening. You have to be prepared to experiment, search the Web for examples of use, or poke around in the package source code for hints. (These packages are almost always written in the same language you're using, which makes it much easier.)

Open source languages have active user communities; historically, Ruby's has been especially kind to newcomers. If you ask a question that shows you've done some diligent exploration for the answer, you're likely to get help quickly. And only a hugely expensive support contract would get you better handling of bugs. While writing this article, I found a problem in the WSDL package. Its author, Hiroshi Nakamura, posted a fix *twenty-three minutes* after I reported the bug. Five minutes later, my problem had gone away.

In sum, interpreter-assisted exploration isn't a silver bullet. There is a learning curve. Because of the interpreter, and because modern programming languages are more friendly than those of the past, it's not incredibly steep—and you can get useful work done along the way. Once you've learned, I believe you'll find exploration via the interpreter as rewarding as exploration via the GUI. **{end}**

Brian Marick has worked in testing since 1981. He is the author of The Craft of Software Testing, *and is the technical editor for* Better Software *magazine. Contact Brian at marick@testing.com. Read other writings at testing.com.*

Sticky Notes

For more on the following topics go to www.stickyminds.com/bettersoftware

- Getting Ruby
- Exploratory testing background
- Kohl's pair-testing article