Test automation that helps, A Guardian Content API example.

05 August 2011 · automation, exploratory testing, ruby

Have you ever had to test an API that's accessible over the internet? or even one thats available internally within your organisation? They often take the form of a REST (http://en.wikipedia.org/wiki/Representational_State_Transfer) service (or similar) through which other software can easily access information in a machine readable form.

Even if you are not familiar with these APIs, you've probably heard-of or seen the results of them. Some examples of APIs are the Twitter API (https://dev.twitter.com/docs) , Flickr (http://www.flickr.com/services/api/) and the Guardian's Open Platform (http://www.guardian.co.uk/open-platform). Some examples of what people have built (http://www.flickr.com/services/) using the Flickr API are published on the flickr site. Despite being 'machine-readable' they are often human readable, greatly helping you test and debug them.

Companies use these APIs to ease the distribution of their content, encourage community and commercial development around their content or to simply provide a clear and documentable line between their role as data-provider and where the consumer's role begins.

When testing an API like the above, many teams slip into the test-automation-binary-world-view. That is, they discard the clever testing approaches they've used before with command-line or Graphical User Interfaces, and start to think only in terms of PASS and FAIL. The knowledge that the 'consumer' is another piece of software seems to cause many teams to assume that they no longer need to learn the application, and the 'testing' becomes a series of wrote steps. Read the specification, define some fixtures, write some PASS/FAIL tests and run.

That's good, if you want to create [literally] an executable specification. As well as your companies actual API implementation you now have a mirror image of that API in your test automation. If you have used a tool such as Cucumber (http://cukes.info/), you may now have an ambiguous natural language [style] definition of the rigidly implemented actual API. That might be useful in your organisation for communication, but as far as testing goes, its fairly limited.

What have you learned? What new avenues for finding problems have you uncovered? To make your tests work and work reliably, you've probably had to incorporate a range of canned data that 'works' with your tests. So once they do work, for this narrow range of code-paths, they will keep 'working' and not revealing any new information about your software, no matter how many times you run them.

As testers we know the vast wealth of bugs are yet to be found. We can write PASS/FAIL tests all day, trying to predict these bugs and barely scratch the surface. The real ‘gotchas’, the unexpected bugs, are not going to be in those canned tests. We have to use an explore the software to find them.

These APIs have a common theme, they are for communicating data. We are not just talking about simple unit-level methods that add/subtract/append etc. These methods work with human generated content, they try to represent the content in a form machines can work with and deliver. This makes these systems ‘messy’ as people don’t think like machines. Users don’t know or care what the executable specification says, They use Flickr, write content for the Guardian, or tweet. When software has to handle these messy real world situations bugs are born. The domain of possible inputs and range of outputs for these systems is huge, and as testers its our role to attempt to find the ambiguities, unknowns and bugs that affect the API and the data it needs to work with.

The tools we need to find these issues and bugs are readily accessible and generally free. This simple example, my initial look at the Guardian Content API, shows how to make test automation aid rather than hinder your testing. It isn’t a canned test and data, it doesn’t provide you with a reassuring green light, but does highlight the value of a more exploratory form of test automation. My example uses the live Guardian Content API, and took a matter of minutes to code.

Example:
The other day I read about the Guardian Content API (http://explorer.content.guardianapis.com/#/?format=json) I noticed that the queries can be a URL ‘GET’ request, just like a Google query, a simple example might be:
http://content.guardianapis.com/search?q=aerosol&page-size=50&format=json (http://content.guardianapis.com/search?q=aerosol&page-size=50&format=json)

This query returns a page of results that lists articles containing the word ‘aerosol’, in the Javascript based JSON format.
e.g.:

{
“response”:{
“status”:”ok”,
“userTier”:”free”,
“total”:500,
“startIndex”:1,
“pageSize”:50,
“currentPage”:1,
“pages”:10,
“orderBy”:”newest”,

“results”:[{
“id”:”artanddesign/2011/jul/27/leonardo-da-vinci-paintings-national-gallery-louvre”,
“sectionId”:”artanddesign”,
“sectionName”:”Art and design”,
“webPublicationDate”:”2011-07-27T18:33:30+01:00”,
“webTitle”:”Leonardo da Vinci works to be shared by National Gallery and the Louvre”,
“webUrl”:”http://www.guardian.co.uk/artanddesign/2011/jul/27/leonardo-da-vinci-paintings-national-gallery-louvre”,
“apiUrl”:”http://content.guardianapis.com/artanddesign/2011/jul/27/leonardo-da-vinci-paintings-national-gallery-louvre”
},{
“id”:”commentisfree/2011/jul/24/david-mitchell-old-masters-vandalism”,
“sectionId”:”commentisfree”,
“sectionName”:”Comment is free”,
“webPublicationDate”:”2011-07-24T00:06:00+01:00”,
“webTitle”:”Vandalising an old master is bad, but not quite as evil as queue-jumping | David Mitchell”,
“webUrl”:”http://www.guardian.co.uk/commentisfree/2011/jul/24/david-mitchell-old-masters-vandalism”,
“apiUrl”:”http://content.guardianapis.com/commentisfree/2011/jul/24/david-mitchell-old-masters-vandalism”
},{
…

As a tester, I noticed some good areas to investigate further. For example from experience I have noticed that time and dates are an area very prone to bugs. So I decided to take a closer look at the date-times returned by the API. They appear to be written in an ISO style (http://en.wikipedia.org/wiki/ISO_8601).

Some ‘tester’ questions that immediately ran through my head were: is that always the case? Are the times widely distributed throughout the day or is there a rounding bug?

To help investigate these questions, I wrote a short ruby script that takes a list of words, queries the Guardian’s content API and outputs a summary of the results. The results look like this:

```
200,aa,2011-08-03T06:45:36+01:00,world/2011/aug/03/china-calls-us-debt-manage
200,aa,2011-07-31T18:52:29+01:00,business/2011/jul/31/us-aaa-credit-rating
200,aa,2011-07-31T00:06:35+01:00,world/2011/jul/31/hong-kong-art-culture-china
200,aa,2011-07-29T15:36:00+01:00,world/2011/jul/29/spain-early-election
…
```

The results fields are:

1. [200] is the HTTP response code,
2. [aa] is the word that I queried,
3. [e.g. 2011-08-03T06:45:36+01:00] is the ‘webPublicationDate’ of the article,
4. [e.g. world/2011/aug/03/china-calls-us-debt-manage] is the ‘id’ of the article.

What words do I use in my queries?

For this test, I chose a list of ‘known to be good words’, that are definitely in the english language. They should bring back a large set of results, [from this index of english language articles]. This word list is easily found, most UNIX systems have a free word list (http://en.wikipedia.org/wiki/Words_(Unix)) built-in. On my Mac its here on the file-system: /usr/share/dict/web2

The word list contains 234936 entries, too many for a person to manually query, but trivial for our test automation to work with.

I started the script passing in the ‘words’ list file as an argument. The script takes each word, queries the server, records a summary of the first page of results and continues hour after hour. This is ideal grunt-work for test automation, using the computer to run tasks that a human can’t (thousands of boring queries) - when a human can’t (half the night).

The Results

When I return to the script a few hours later, it is still in the ‘a…’ section of the words but has already recorded over 125,000 results. If you remove the duplicates, that is still over 45,000 separate articles.

I looked through the summary results file, and decided that If I extract out the dates, and plot them on a graph I might get a better visualisation of any anomalies. I used a simple set UNIX commands, to output a list of times in year order. At this point I discovered an issue.

When sorted, it became clear there are some odd dates, for example, here are the first ten dates:

1917-07-07T00:02:25+01:00
1936-08-31T14:47:16+01:00
1954-05-25T15:59:16+01:00
1955-04-19T10:55:03+01:00
1981-06-30T13:07:30+01:00
1985-05-23T18:22:25+01:00
1985-05-23T22:21:29+01:00
1985-06-23T16:55:28+01:00
1994-05-30T18:22:00+01:00
1995-06-26T02:34:57+01:00

This doesn't match the claim on the Guardian's Open Platform website:

> "The Content API is a mechanism for selecting and collecting Guardian content. Get access to over 1M articles going back to 1999, articles from today's coverage, tags, pictures and video" http://www.guardian.co.uk/open-platform/faq (http://www.guardian.co.uk/open-platform/faq)

At this point I start checking some of these articles, for example the one dated 1954: http://www.guardian.co.uk/world/1954/may/25/usa.schoolsworldwide (http://www.guardian.co.uk/world/1954/may/25/usa.schoolsworldwide) . The articles appear to be genuine, and display a corresponding date in the web-page version of the article.

This is invaluable data, its opened up a whole series of questions and hypothesis for testing. e.g.:

- Does the 'webPublicationDate' refer to the actual publication date in the newspaper rather than the website? (the above examples might suggest that)
- Do the date-filter options in the API, correctly include/exclude these results?
- Will other features/systems of the site/API handle these less-expected dates correctly?
- Do we need to update the developer documentation to warn them of the large date range possible in the results?
- What dates are valid? How do we handle invalid dates etc?
- Are results dated from the future (as well as distant past) also returned? (Such as those subject to a news-embargo (http://en.wikipedia.org/wiki/News_embargo))
- Should I write a script to check all the dates reported match those in the web-page?

These questions are now the immediately the basis for the next set of tests, and provide some interesting lines of enquiry for the next discussion with our programmers and product owners.

The interesting thing here is that we used test-automation to improve our testing, by achieving something that couldn't be done practically by a person un-aided. Also it is highly unlikely these new questions would of been asked if we had merely created an executable specification. That process assumes that we know the details of the problems in advance. But as we saw in this example, we had no idea that this situation would occur in advance. To suggest we did, would likely be an instance of the hindsight bias (http://en.wikipedia.org/wiki/Hindsight_bias). That is, We believe we 'should' of known what to look for the issue, even though we and others failed given all the evidence present at the time.

Writing test automation that provides us with new information, new avenues for investigation, is clearly a better way to learn about your API. As opposed to focusing your time and resources on simple binary PASS / FAIL tests on a set of canned data. The very nature of its open ended investigation, lends itself to the tight loop of investigation, define hypothesis, and test at the heart of good testing.