Is your test automation actually agile? A Guardian Content API example.

09 August 2011 · agile, automation, exploratory testing, guardian, random, tools

In my last post (http://www.investigatingsoftware.co.uk/2011/08/test-automation-that-helps-guardian.html) I discussed how test automation could be used to do things that I couldn't easily do unaided. In that example, execute thousands of news 'content searches' and help me sort through them. With the help of some simple test automation I found some potential issues with the results returned by the REST API.

In that case, I started out with the aim of implementing a tool. But your testing might not lead you that way, often your own hands-on investigation can find an issue. But you don't know how widespread it is, is it a one-off curiosity? or a sign of something more widespread.?

Again, this is where test automation can help, and if done well, without being an implementation or maintenance burden. Many test automation efforts are blind to the very Agile idea of YAGNI (http://en.wikipedia.org/wiki/You_ain't_gonna_need_it) or You Ain't Gonna Need it . They often presume to know all that needs to be tested in advance, deciding to invest most of their time writing 'tests' blindly against a specification, that is as-yet un-implemented. This example shows how simple test automation based on your own feedback during testing can be very powerful.

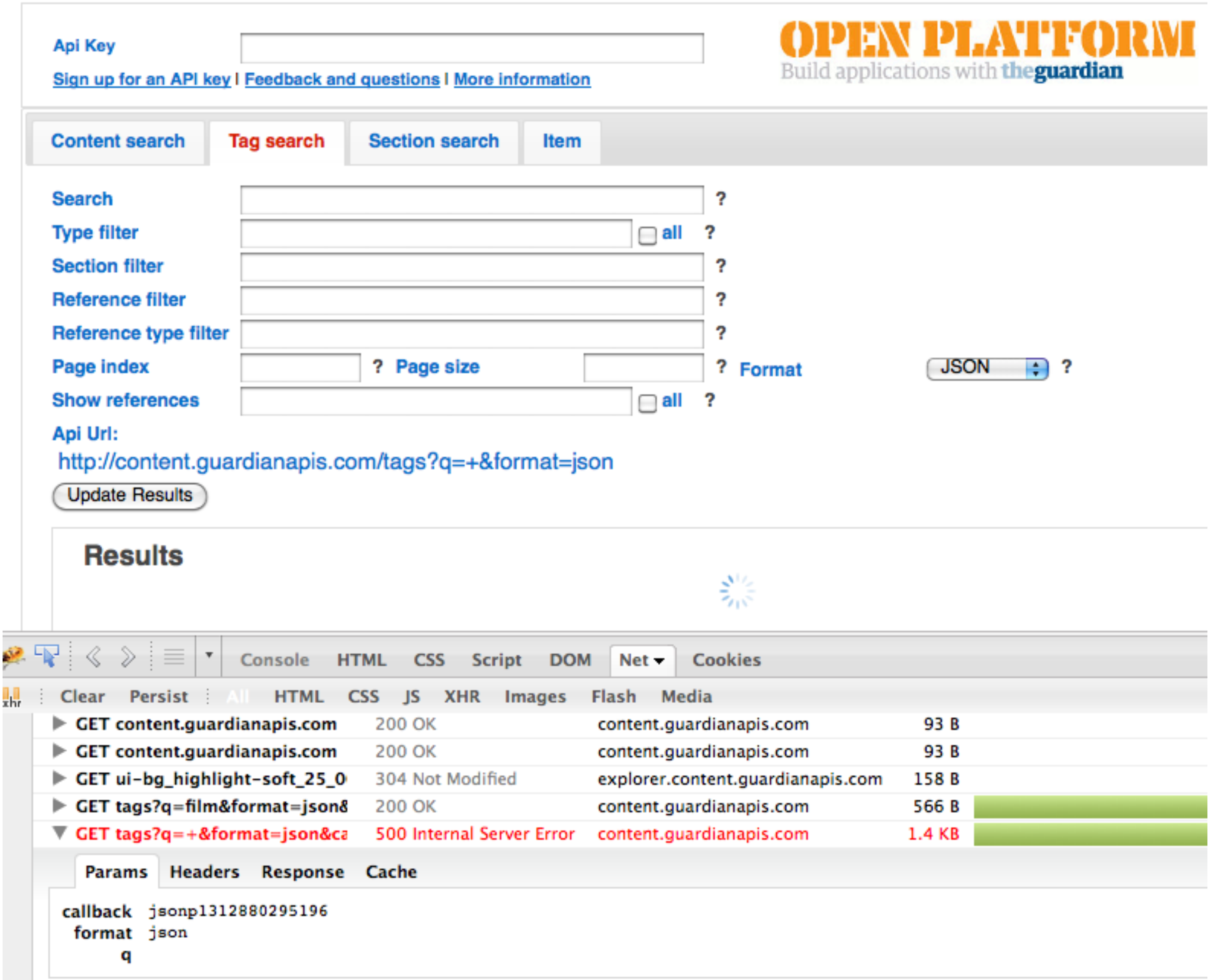
The test:

I had the idea that the Guardian Content API might be overly fussy with its inputs. Often software is written [and tested] using canned data, that's designed 'to work'. These data inputs usually confirm to a happy path, and even then, only a subset of the data that could be considered 'happy path' is used.

Using the Guardian's own HTML GUI (API Explorer) that allows you to easily query the REST API, by hand, I tried a few quick tests. These included, some random text (http://www.investigatingsoftware.co.uk/2011/07/random-text-tool.html) as well as a few typical characters that are likely to occur in text but I suspect would not be present in the usual canned test data. For example, a single SPACE character.

That quick test, of the SPACE character highlighted an issue. Entering a single SPACE character into the Tags search API explorer, appeared to cause the HTML GUI to not return a response. The API Explorer appeared to hang. At that point, I didn't know the cause, the issue could be a problem with this developer-GUI, and not the API itself.

A closer examination using Firebug, clearly identified the cause as a HTTP 500 (http://en.wikipedia.org/wiki/List_of_HTTP_status_codes#5xx_Server_Error) Error from the server.



Failing query using Guardians Tags Search API - Firebug showing the HTTP 500 Error.

I could have reported this one issue. That despite the documentation stating any free text (http://www.guardian.co.uk/open-platform/content-api-tag-search-reference-guide) is ok for this field a simple space character can expose a failure. But using some simple automation I was better able to define the extent and distribution of this issue. For example, is it a general problem with entering single characters? Does it only affect one part of the API?

With a minor script change, my previous Ruby API-tool could report error responses and details of whether a JSON response object was returned by the server. (Though a simple cURL (http://curl.haxx.se/) based shell script could have just as easily done the job.) I also wrote a little script to output every ASCII character:

```
lower_bound=0x0000
upper_bound=0x007F
```

```
lower_bound.upto(upper_bound) do |codepoint|
  puts  ’’ << codepoint
end # end upto
```

The output of the above was directed to a file, and used as the input for my API-script. The script now reads one ASCII character at a time and uses that character to query the Guardians Content API (http://www.guardian.co.uk/open-platform/content-api-content-search-reference-guide). As I had found this issue in the Tags Search (http://www.guardian.co.uk/open-platform/content-api-tag-search-reference-guide) I also ran this script against the Guardians Tags search API.

This is ideal ground for test automation, there are 128 ASCII characters, and I’m examining 2 services, making 256 queries. That’s too many to do by hand, but trivial for a simple test automation script. Common characters that are not available in ASCII, are nonetheless very common in English. And therefore will be present in the body and headlines of the Guardians content. A simple example is the € (euro) symbol. The script would also allow me to query these many [millions of] non-ASCII characters if my current testing suggested that it might be fruitful.

The results of this initial run, were quite interesting:

Tags Search API			**Content Search API**			**Query decoded**
Response Code	**Query used**	**Was a JSON response object returned?**	**Response Code**	**Query used**	**Was a JSON response object returned?**	
500	%09	No response object				Horizontal tab
500	%20	No response object				Space
500	!	No response object				
500	%22	No response object	500	%22	No response object	" (Double Quotes)
500	(	No response object				
500	)	No response object				
500	*	No response object				
500	%2B	No response object	500	%2B	No response object	+ (Plus sign)
400	%2C	No response object	400	%2C	No response object	, (Comma)
500	-	No response object	500	-	No response object	
500	%3A	No response object				: (Colon)
500	%3F	No response object				? (Question mark)
500	%5B	No response object				[(Square bracket)
500	%5C	No response object				\ (Backslash)
500	%5D	No response object				] (Square bracket)
500	%5E	No response object				^ (Caret)
500	%7B	No response object				{ (Curly bracket)
500	%7D	No response object				} (Curly bracket)
500	~	No response object				

Table indicates non-HTTP 500 response codes, and the corresponding queries.

The results when filtered to only show non-HTTP 200 (http://en.wikipedia.org/wiki/List_of_HTTP_status_codes#2xx_Success) results clearly indicate the Tags API is less robust that the Content Search API, over these inputs. For example the space character produced no error in Content Search but did in the Tags Search. The same is true of the Horizontal tab, both characters that might be present in ‘any free text’ (http://www.guardian.co.uk/open-platform/content-api-tag-search-reference-guide).

The lack of consistency between the two APIs is the most striking factor, to me as a tester. The two systems clearly handles these inputs differently. That information is invaluable to testers. We can instantly use this information in our next round of testing as well as discussions with programmers and product owners. Asking such questions as:

1. What code is shared between the two services? There are clearly some differences.
2. How do the two APIs handle these characters in combination? with each other or ‘typical’ english words.
3. As error handling code is itself often flaky, what new bugs can I find in the Tags search API?
4. How badly will the APIs, in particular the Tags API, handle non-ASCII characters? Should the APIs handle all UTF-8 codepoints without exposing failures?
5. As far as the product owner is concerned, What is expected behaviour when a character is not-accepted?

Light weight, easy to build test automation that lets the team quickly get a mental-model of how the software actually-works is clearly valuable. I’m using the computer to do the laborious work its good at, extending my reach as a tester to help me see the bigger picture. Showing that there is more than just one isolated character not being handled well, but in fact the Tags Search API is generally a lot more prone to failure. This, more exploratory, automation is freeing me to do analysis and face to face communication with team members and product owners. Allowing me to adapt quickly to those discussions and how I see the software behaving, a fundamentally more Agile (and agile) approach (http://agilemanifesto.org/principles.html).