# Making sense of the user agent string

30 March 2017

Niels Basjes

Ever since I've started working for a WebAnalytics company in 2005 I've been working on problems related to making sense of web data. One of the most difficult elements in this type of analysis is making sense of the user agent.

Very often the raw web data I work with is stored in Apache HTTPD access log files that have been compressed using gzip. Because I'm a software developer at heart, it will not surprise you that over the years I've written several tools to work more effectively with these types of files. Think of easily extracting the various elements of log lines (Apache HTTPD & NGINX access log parser) and speeding up the analysis of multi-GiB text files in an Apache Hadoop cluster (Hadoop Splittable Gzip Codec). This blog post is about the tool Yauaa (Yet Another User Agent Analyzer) that I created to make it easier to make sense of the user agent.

In his post on Better device profiling my colleague Hans van Buuren describes how bol.com uses Yauaa. My post digs deeper into the background and the details on the how and why of this system.

# The User-Agent request header

When a web browser does an HTTP request to a website, one of the headers in such a request is the User-Agent header as described in RFC 2616:

```
The User-Agent request-header field contains information about the user
agent originating the request. This is for statistical purposes, the
tracing of protocol violations, and automated recognition of user agents
for the sake of tailoring responses to avoid particular user agent
limitations.
```

An example of a user agent is this:

```
Mozilla/5.0 (Android 4.4; Tablet; rv:41.0) Gecko/41.0 Firefox/41.0
```

It clearly indicates the type and version of the operating system, the device type and the browser name and version. Nice!

that tried to give users the best possible usability resorted to using features supported in some web browsers only. As a consequence they needed to include code that effectively enabled or disabled features to make the websites 'look better' if the browser supported it.

Because client-side code support (like javascript) was limited, many websites chose to use very simple server-side code to make the choice between sending back the 'basic' or the 'enhanced usability' version of their website.

They essentially did something like this in their web server:

```
if (useragent.contains("Mozilla/")) {

// Use the Netscape feature we need.

} else {

// Fallback to the simpler site

}
```

Some sites improved on this by actually checking if the version is high enough.

After a while the competing web browser caught up and released a new version which implemented the features needed to show the 'enhanced usability' versions of that website. Unfortunately, the website would still be served in its basic version as their user agent couldn't match the pattern that would trigger the better version. The solution many have chosen was simple and effective: add extra information to the User-Agent header to indicate the browsers you are compatible with.

# Lies, damned lies and user agents

At first glance it makes a lot of sense to add this extra compatibility information. The reality, however, is that this is the reason the current user agents are such a mess. Let me illustrate with this real user agent pulled from our production access log files of February 2017:

```
Mozilla/5.0 (Windows Phone 10.0; Android 6.0.1; Microsoft; Lumia 650)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.79 Mobile
Safari/537.36 Edge/14.14393
```

Let's examine the various parts we find and assess them:

```
Mozilla/5.0              No, this is not Mozilla Netscape 5.0
Windows Phone 10.0       Yes, a Windows Phone device
Android 6.0.1            No, not Android
Microsoft; Lumia 650     Yes, a Microsoft (Nokia) Lumia 650 device
AppleWebKit/537.36       No, not AppleWebKit
KHTML                    No, not KHTML (Konqueror)
like Gecko               No, not Gecko based
Chrome/51.0.2704.79      No, not Chrome
Mobile Safari/537.36     No, not Mobile Safari ...
Edge/14.14393            Yes, this is the Microsoft Edge browser
```

So out of the 10 elements we see here, there are 7 lies to indicate the tools they're compatible with. Now, don't blame Microsoft this time; almost all of the browsers do this.

# But what do I really need to know?

Before we start digging into the user agents, we first need to answer the fundamental question: what do we really need to know and what do we intend to do with it? In this context I have found two fundamental situations:

1. High level:
   - What class of device is this?
   - What operating system, browser, etc. is this? This is useful for analytics purposes and for situations (like we have at bol.com) where we have a 'full-size' site and a 'light-weight' mobile site.
2. Detailed:
   - What screen resolution does this device have and does this browser support the latest javascript/css/... feature?

There are projects like BrowserScope and Apache DeviceMap that try (tried) to maintain a database of supported detailed features. As you can understand this is a lot of work to maintain because of the high change rate of new devices and browsers. In general these databases will always be 'too old' for some applications.

I have found that if you need to know the detailed information the only *workable* way of doing this is by using some scripting (usually javascript) to determine the support for these features in the browser itself. There are tools available that support this idea (like Modernizr).

In my work I only have the user agent string in some kind of log file and I only need the high-level information. At that point in time, running any JavaScript is no longer possible and it is also not needed for my use case.

rules in a (proprietary) pattern matching language and the first rule that matched was used for the output. This worked quite well for the attributes I needed (things like Operating System and Browser).

Yet after a while I found that the variations in the user agent strings was so large it became a maintenance nightmare. The main issue is that the ordering of some elements varied and putting N! rules for the required combination of N explicit elements quickly becomes a problem.

In 2008 I started working at bol.com and after a while I found UA-Parser for which I wrote the Apache Pig UDF. UA-Parser effectively does the same as what I wrote back in 2005. The main difference is that it uses regular expressions, making the set of rules easily portable to many programming languages. The downside is that it uses regular expressions which makes it a bit harder to maintain.

The systems that work based on an ordered list of rules suffer from a fundamental performance problem. The exceptions to the normal (common) pattern are at the start of the list that is checked, the 'clean' situations at the bottom. So in practice, the most common situations are at the bottom so that in most cases a very large number of rules needs to be checked.

## Can it be done differently?

Back in 2011 I started thinking: can this be done differently? Can I actually do this without sequential lists of rules? Can it be made so that anyone can add new rules?

I quickly realized that perhaps the user agent could be parsed using a real parser. But ... there is only a very unclear language specification and many browsers don't follow this specification at all. So this parser must be able to handle some parse errors without failing too hard.

Worst of all: this is also an example I want to be able to properly classify:

```
Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:36.0) Gecko/20100101
Firefox/36.0')waitfor delay'0:0:20'--
```

Yes, this is an SQL injection attack using the user agent. So no "Firefox" or "Linux" but "Hacker".

At first I did some experiments with Antlr3 to see if I could create a parser. I wanted to transform the user agent string into a predictable tree structure on which I could turn a pattern analyzer loose. Using Antlr3 I couldn't get it do do what I wanted, probably caused by my own limited knowledge at the time.

than reading the manual) is that Antlr4 is capable of creating a 'mostly right' tree even in cases where the input doesn't fully follow the described pattern. So some of the 'catching errors' is done by Antlr4.

## So now I have a tree, let's match some patterns

Having seen the high rate of change in the available devices and browsers I chose to limit the number of lookup tables and browser/device specific rules. So the main focus of the rules I did write, is on 'extracting' information from the user agent, not 'looking up information'.

Also, the pattern matching system has been designed to handle a very large number of rules as efficiently as possible.

One of the key differences with other systems is that a tree structure is available for the user agent. It becomes possible to look for a pattern and walk through all the elements of the tree to find the value that needs to be extracted. At startup each rule set (called Matcher) has registered itself in a HashMap indicating which nodes of the tree are of interest.

Essentially Yauaa now works as follows:

1. Parse the user agent
2. Walk through all the elements of the tree
   - Fire each tree element against the rules that have indicated interest in that element.
3. Ask each rule set if it has enough input
   - If enough input evaluate the rules
   - If all rules success return the result
4. Aggregate all returned results into a final answer

For a very common 'Chrome on Windows' user agent we can now extract all relevant information from over 1800 rule sets in about 0.3 milliseconds. It can run even faster if only specific fields are requested.

## And then ...

in the summer of 2016 I asked some of my colleagues what they thought of my hobby project. It happened to be exactly what they were looking for because the Apache DeviceMap project (on which some of the tools were running) was no longer being updated. My colleague Hans van Buuren has already written in his blog about that transition.

The answer today is: yes, it surely can and it is very efficient too.

I made this project open source under the Apache 2.0 licence. So if this solves a problem for you then go to https://github.com/nielsbasjes/yauaa and read how you can use and run it.

## Tags

Java Open Source