Chapter 4 – Performance Testing

While the implementation works and can help in detecting the features of requesting UA’s, its inherent goal is to improve the performance of web pages across both existing and future devices and UA’s. This chapter presents the concept of Mobile Web Performance, as well as how we conducted our performance tests and their results.

# Mobile Web Performance

The performance of web pages has been steadily increasing along with the constantly improving network infrastructure and computing power of modern desktop computers. With this improved capacity Web developers have been able to create richer experiences on the web through increased use of resources such as CSS, JS, images and video. Because of this the size of Web pages has increased alongside the demand for more powerful computers to render them. With the emergence of mobile computing, Web pages again have to account for being viewed on devices with limited computing power, high network latency and reduced bandwidth. Web pages designed with desktop computers in mind can be orders of magnitude slower on a mobile device, sometimes to such a degree that the Web site is rendered unusable. This is in many ways a strange problem: on one hand mobile devices are much more powerful today than regular computers were ten years ago, but cellular networks are much akin to the regular wired networks in terms of speed as they were back then.

The web technologist, author and speaker Nicholas C. Zakas suggests in his article “The Evolution of Web Development for Mobile Devices” that the problems we face today with Mobile Web performance are rooted in two main concerns: network latency and mobile device hardware limitations. He bases his suggestions on the work of Steve Souders, who we mentioned in chapter 1, focusing on his list of best practices when it comes to Web performance, especially rule 1 (make fewer HTTP requests) and 11 (avoid redirects).

## Network latency

In the context of Web performance we define latency as the delay experienced from sending a request over the network to receiving a response, i.e. the round-trip time of the request. Bandwidth is defined as the amount of data a connection can receive over a specified amount of time, e.g. 20 Mb/s. Bandwidth may be limited by latency.

On regular wired connections, latency over short distances is minimal as the packets sent over the network propagate through physical cables. Barring network congestion, the only source of latency is electrical resistance in the wire material, or the speed of light in the case of optical fiber cables. Because the signals propagate at a finite speed, latency increases with transmission distance, but interference causing transmission loss is kept to a minimum. Desktop computers normally use a wired connection. This means that they will experience minimal latency when sending requests to Web servers, exacerbating the difference between making a request on a desktop computer versus a mobile device connected to a cellular network.

Wireless connections have several sources of interference and general signal loss that may increase latency when making requests to a Web server. Requests made over a wireless network propagate through the air, completely unshielded from any kind of external interference. Radios, microwaves, walls or any other form of electromagnetic or physical barrier may adversely impact the effective bandwidth of a wireless connection, giving wireless networks a much higher potential latency than wired networks.

Cellular networks are especially vulnerable to high latency by nature of their topology. A request from a mobile device has to first go to the nearest cellular tower, and then to a server using the General Packet Radio Service (GPRS) belonging to the mobile service provider, which functions as a gateway to the Internet. This server can then make the actual request to appropriate location (DNS, HTTP or other), the response from which then has to propagate the same way back. Currently these servers are few and centrally located, instead of distributed, giving the users proximity to them a measurable impact on the latency of a request. Going by Souders’ list of principles, rule 1 seems to be quite poignant here, as reducing the amount of requests will directly reduce the latency from the original request until the user sees a fully rendered Web page. Minimizing the number of redirects (rule 11) is also important in this regard, as each redirection requires a new DNS lookup request and an additional HTTP request. This will naturally increase the total request time substantially.

## Mobile device limitations

Modern desktop computers have extremely powerful hardware that have no problems with rendering even the most advanced Web pages. Even though modern mobile devices are quite powerful, especially compared to desktop computers from ten years ago, they do have limited processing power and memory in comparison. In this regard developers have to pay attention to how their Web pages utilize the capabilities of the device they are being viewed on. A Web page that is easily rendered in a desktop browser might cause severe problems or crashes in a mobile browser due to hardware limitations. The two things that are especially important is CPU and memory usage. CPU usage, as well as network access through cellular, Wi-Fi and Bluetooth antennas also impact the battery life of a device, which is already short on modern mobile devices.

Zakas mentions in his article that reducing the amount of JS on a Web page can reduce both the response time and the amount of battery drained as a result of the CPU having to execute less code. The JS engines running on mobile devices are orders of magnitude slower than those in desktop browsers. They also vary greatly in performance and can, by themselves, greatly reduce the user experience if a Web page uses a lot of JS. Keeping its use in mobile Web pages to a minimum should thus be a goal when it comes to improving performance.

Another concern Zakas mentions is the limited memory available on mobile devices. Even the latest mobile devices have much less memory than desktop computers, and even less is available for use in Web browsers. While modern desktop computers will never encounter memory issues from loading a single Web page, a severely memory-intensive Web page might just cause problems on a mobile browser. The biggest reasons for memory issues on a Web page, Zakas claims, are images and hardware accelerated graphics in general. A Web page with a lot of images embedded in its DOM can quickly fill up the memory available to the browser, causing slowdown and possibly crashes. Modern browsers, including mobile browsers, also hardware accelerate graphics such as images, CSS transitions and animations. This is done because handling graphics on the GPU instead of the CPU will lead to a smoother experience for the user, but it also uses more memory, which is limited on mobile devices. Being aware of these issues and using images and graphics responsibly can go a long way in avoiding potential memory issues on a mobile Web page.

## Where Detector comes in

JS, images, CSS transitions and animations help in creating a rich Web experience for users on desktop computers. Simply cutting them out because they do not work properly on mobile browsers would be unfortunate, and this is where Detector comes in. Being able to detect UA features and tailor the Web pages on the server before sending them to the client can go a long way in allowing developers to get the best of both worlds. Developers can keep their rich Web page for desktops while altering the HTML for requesting UA’s that are known to have limitations such as those mentioned earlier.

Detecting the limitations on the server also assists developers in reducing the amount of requests needed for fully rendering their Web pages when a mobile UA is encountered. JS can be reduced, merged or removed. The resolution of images can be reduced or the number of images can be lowered. CSS files can be merged, or reduced in size to contain only what is absolutely necessary on mobile UA’s. Because these concerns are primarily relevant in the context of mobile devices, developers can choose to ignore these issues when developing the regular desktop version, while keeping the capability of tuning the mobile version later.

The plugin will increase the time a request takes to be handled, though. It adds additional database lookups and business logic to the server that it has to execute at least once per request it receives. To discuss the overall usefulness and effectiveness of our plugin, it is important to figure out just how much of a penalty is incurred on each request when Detector is active.

# Method

Measuring performance of web sites is a whole research field in its own right. There are many ways of doing it, each aimed at specific parts of request chain or the user experience. Some might target the performance on the backend, while others target the frontend exclusively, looking at the execution time of JS and the size of files sent in the response. Others may not look at response- or execution times at all, but rather do an analysis of the content of a web page as it loads to determine the web page’s performance as experienced by the user.

Because our plugin is situated primarily on the backend we focused on that as the common case, but we also had to consider the case where the system encounters an unknown UA and must do tests on the frontend. These two cases are quite different and measuring their performance had to be approached differently.

Enonic CMS also has its own device detection system built into it. Since our system is meant to replace it we also had to look at it and our system comparatively, to establish the performance impact of using our plugin as a replacement. This needed to be done for both cases mentioned above. Even though our system detects more features than the built-in system, a severe performance hit might be grounds to argue against using it.

## Measuring backend performance

The performance of the Detector plugin itself can be measured in two ways, either we can measure the time spent by the plugin on the server in its own code using timers, or we can look at the time it takes the browser from making a request until the initial HTML document is received in the response. The most important part is that the environment is kept static between tests, and that the tests reflect normal use cases. It is also important to minimize the possibility of measurement errors due to network irregularities by performing the tests in close succession and on the same network.

We chose to measure the performance of the backend by using the Google Chrome Developer Tools. This allows us to look at the time it takes from a request is sent to the server until a HTML document is received in the response. The Chrome Developer Tools have a module that illustrates the result of a request on a timeline that shows the individual round-trip times for each resource request to the server, as shown in Figure 1. The initial request is naturally for the HTML document, which is what we are interested in. This shows the amount of time it takes the server to handle the request, look up the UA, handle the UA if it is unknown and send a response.

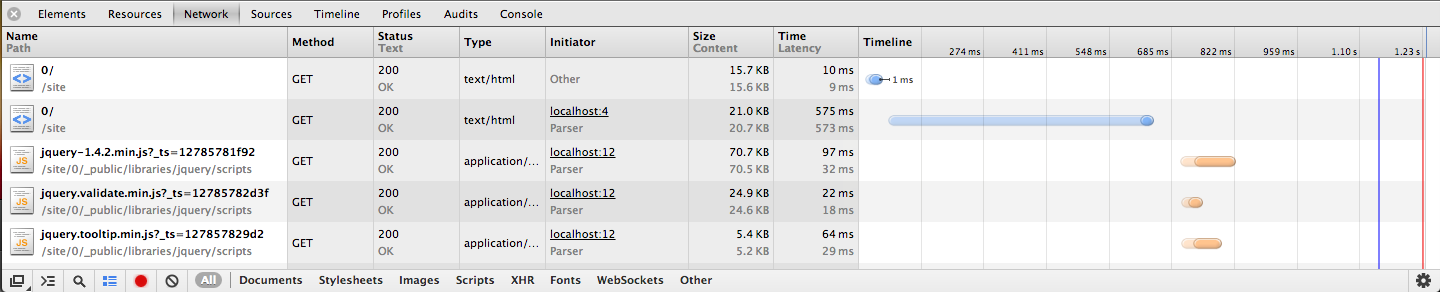


Figure 1: The Google Chrome Developer Tools Network pane. Showing our demo page making a request as an unknown UA to the server. The blue bars indicate requests for HTML documents. The first one is the redirect from the HttpInterceptor extension to do the Modernizr tests; the second is the actual page.

While the request round-trip time we get from Chrome is indicative of the actual user experience, we can also look at the actual processing and rendering time on the server by using Enonic’s administration tools. These tools also provide a page trace view after a page has been loaded in the browser. As shown in Figure 2, it displays the page and its portlets as XML, with each element containing attributes for the total rendering time of the page and portlets, their server side processing time as well as the time spent invoking each element’s datasource methods. This can be used to see the impact of having the Detector plugin specific datasources called, compared to when they are not, giving us data on the performance hit the plugin will inflict on the system’s page loading times. We do this to establish if there is a significant hit in the performance of a page using our plugin compared to one that only uses Enonic’s built in system.

There are two factors to look at on the backend: the performance of the HttpInterceptor extension, and of the FunctionLibrary extension. The performance of the HttpInterceptor extension is impacted by whether or not it encounters a new UA string, thus it is important to separate these two cases. The performance of the FunctionLibrary extension is wholly reliant on the structure of the UA family definition JSON file, so it needs to be kept static between tests.

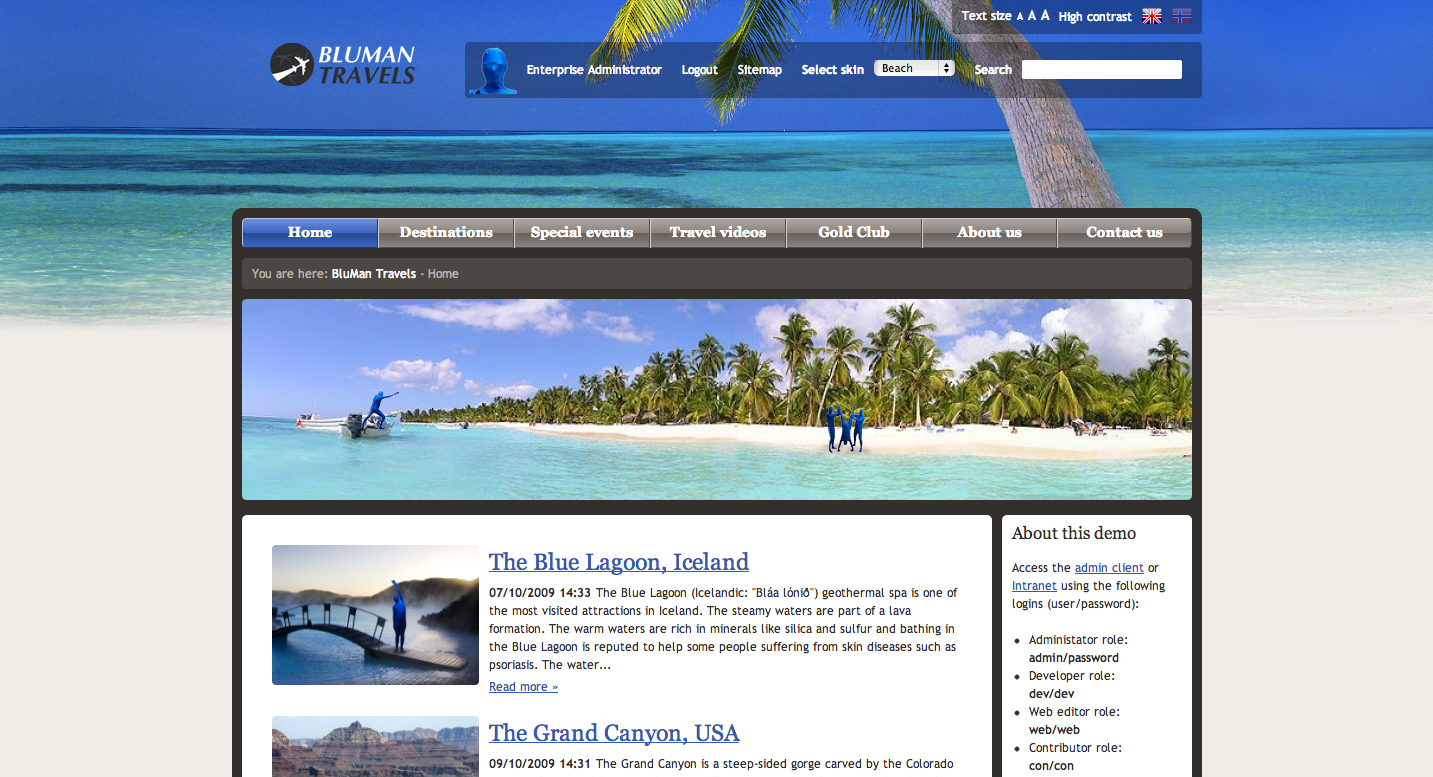


Figure : The Bluman Travels test site.



Figure 2: A page trace in Enonic. It shows the total rendering time and processing time of a page, as well as a breakdown of the time taken for each portlet and datasource function call.

Because the results from the HttpInterceptor extensions are used by a FunctionLibrary extension that is invoked by datasources, we needed to add a call to the datasources on he test page. A single datasource should suffice as Enonic caches the result on a per-session basis, and the FunctionLibrary extension will thus not be called twice.

The site we used for testing is the default demo page that is packaged with Enonic called Bluman Travels. It is a mock travel site with articles about various tourist locations around the world. It is a good example of a typical web site that uses CMS’s like Enonic, and will thus provide a good use-case example. For testing we used the site’s front page (Figure 3), with a datasource call to our FunctionLibrary extension method getUAFamily in the XSL page template it uses.

## Measuring frontend performance

Detector only ever does anything on the frontend if it encounters an unknown UA. When this occurs it sends a response to the client containing some JS code meant to test the features of the requesting UA. To measure the performance on the frontend means looking at the time it takes the JS to execute the tests, generate the resulting cookies and reload the page. In the case of no JS support on the UA, the page will simply be reloaded, which is not very relevant in our case, as we are interested in the worst-case scenario where all the JS is executed.

As previously mentioned, different browsers use different JS engines to execute code. These are all implemented differently and thus perform differently. On modern desktop browsers these variations are largely negligible. The JS engines on various mobile devices do have bigger differences in performance, though, and can thus have a significant impact on the request time and resource use on mobile devices. Because the JS based feature tests are core to the functionality of Detector, we cannot do much to improve the performance of these scripts, lest there arrives a more efficient alternative to using Modernizr. Despite this, it is still interesting to see what kind of performance impact these scripts have on the overall user experience.

Testing the performance of the JS code running on the client is a matter of measuring the time the code takes to execute. To do this we simply get the time using JS at the start of the actual Modernizr code, and then again after the cookie with the results has been generated. This is done using the getTime function, which is a part of JS’s Date prototype. The result of subtracting the end time with the start time is displayed in an alert box and written down in a spreadsheet. This process is repeated a number of times to get a reasonable dataset from which to calculate an average execution time. We do this in all the major desktop browsers and a few popular mobile browsers to get a comparison between their average execution time.

## Comparing the built-in system with our plugin

Enonic has its own device classification system built into it. This system is much simpler than our plugin, but might also be much more efficient when it comes to performance. While Detector has obvious advantages when it comes to the sheer amount of features it can detect, it has the potential of incurring a larger overhead on each request made to the server, due to the increase in business logic and database queries per request. This poses the question: is the increased overhead worth the additional flexibility provided by the plugin? This is highly relevant as a big increase in request round-trip time has the potential to negatively impact the user experience of Web pages using the plugin.

Measuring the difference in performance of the two systems can be done using the same method for measuring the performance of the backend. The testing environment should be kept as similar between systems as is practically possible. The references to the result of the Enonic device classification system should be equal in number to the amount of calls datasources make to the Detector FunctionLibrary extension. The difference in performance between the two systems should then present itself in the form of the request round-trip time found in the request timeline in the Chrome Developer Tools. Intuition dictates that the plugin will perform worst in terms of speed, the question is simply: how much worse?

# Results

The following section contains the results from the various performance tests described above. The tests were run on an Apache Tomcat webserver running Enonic, and all requests to it were done from the same machine addressing localhost except for the JS performance tests conducted on mobile browsers. The specs of the computer and mobile device are as follows:

The server/computer is a MacBook Pro running OS X 10.8 with a 2.66 GHz Intel Core 2 Duo CPU, 4 GB RAM and a NVIDIA GeForce 9400M GPU.

The mobile device is a Samsung Galaxy S running Android 2.3.3 with a 1 GHz Cortex-A8 CPU, 512 MB RAM and a PowerVR SGX540 GPU.

Both of these are representative of devices among the most widely used in their respective categories around the world at time of writing.

## JavaScript Performance

We conducted the performance tests of the JS code as mentioned under the “method” section of this chapter. The three leftmost browsers are desktop browsers, while the three rightmost are mobile browsers. The time represents the average of 40 executions. Testing on mobile browsers was done on a wireless local area network (WLAN) through a Linksys E4200 router. The actual data can be viewed in the appendices.

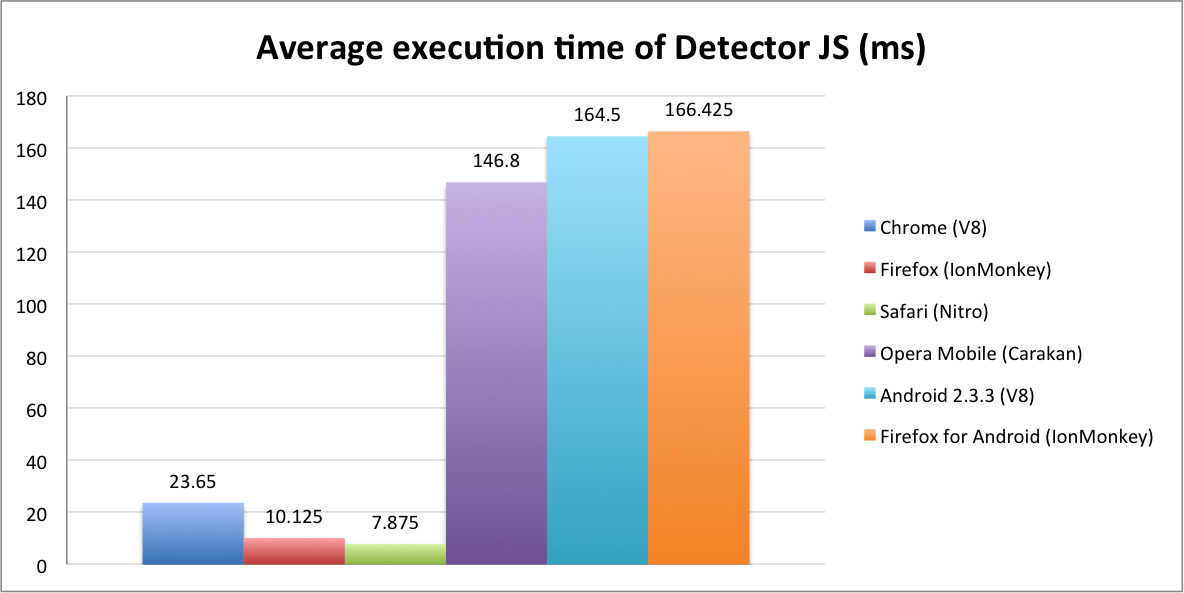


Figure 4: Average execution time in milliseconds of Detector JS code on various desktop and mobile browsers. The JS engine used by the browsers is written in parentheses.

There is an obvious difference in the execution time on desktop browsers compared to mobile browsers, which was expected considering the limited system resources on the mobile device. The choice of browsers to test on was simply a matter of choosing three popular browsers among the devices used that also used different JS engines. There are also others, such as Opera for desktops and Chrome for Android, but for our purposes we decided that three browsers per device sufficed.

## Backend performance

The backend performance tests were split into two parts, as mentioned under the “method” section earlier in this chapter. One for measuring the round trip time of a request made to the test home page, and one for measuring the rendering and processing time on the server itself. Both of these tests are done both with and without the plugin installed, as to measure the impact of using the plugin against the native device classification system.

The request round-trip time also took into account the extra request that occurs when the interceptor makes an extra redirect to execute the Modernizr tests on the client prior to loading the actual page.

### Request Round-trip time

The round-trip time of a request reflects the time it takes from a user enters a URL to the actual web site is displayed. In our instance it was interesting to see the average round-trip time of requests made with, and without the plugin being installed. For the plugin we measured the average round-trip time for both the worst case and the common case. The worst case is when it encounters an unknown UA and has to gather data, create a UA object and store it in the database. The common case is when it encounters an existing UA by a single database lookup. The measurements can be seen in the appendices.

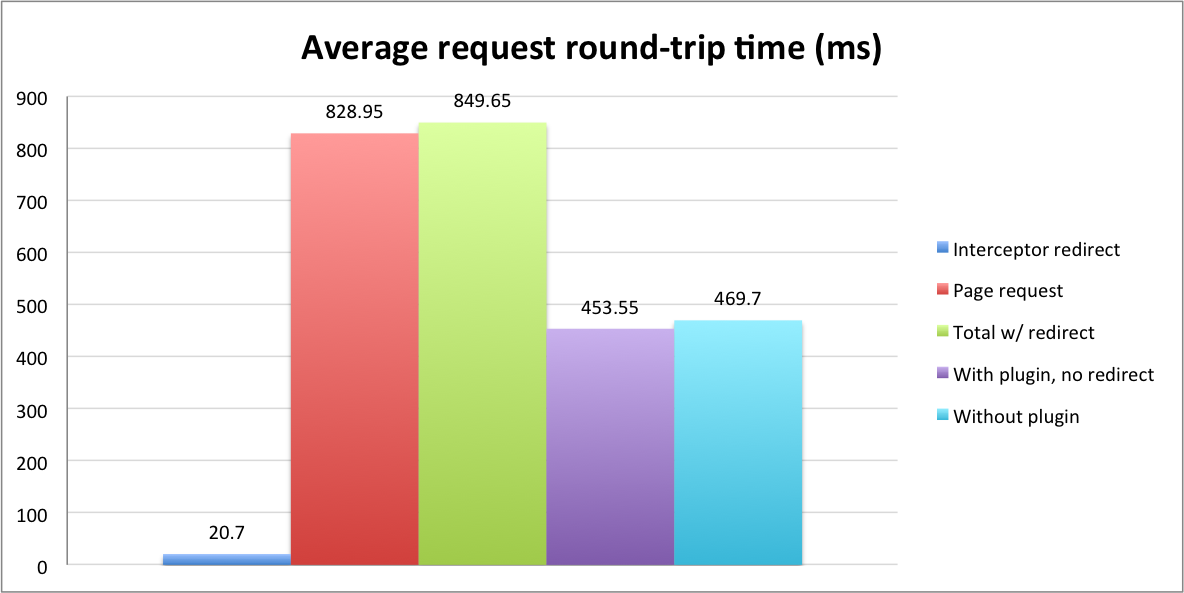


Figure 5: The average round-trip time of requests made using the plugin in both the common, and worst case, along with the time when not having the plugin installed.

The results suggest that the only time the plugin has a significant impact on the request round-trip time is in the worst case, where it is increased by 81 percent. The differences between the common case and not having the plugin installed are so small that they can be attributed to performance fluctuations on the machine we used for testing. This test was conducted using Google Chrome, as mentioned earlier, and the interceptor redirect time, which includes the rendering of the Detector JS, is quite close to the average execution time we measured for Chrome’s V8 JS engine. The biggest chunk of time spent in the worst case is in the main request, after the redirect. This includes parsing the Modernizr result cookie, instantiating a UA object with the results and marshaling it into a MongoDB document, before storing it in the database. An unknown factor is how long it takes to resolve a UA family when it is invoked through an Enonic datasource. This can be tested using Enonic’s built-in page trace tool in its administration interface.

### Server processing time

As mentioned earlier, Enonic contains a page trace tool in its administration interface, which gives a breakdown of where the processing and rendering time is spent on the server. We used this to find the averages in the common case of the plugin, where the UA is known and it only needs to resolve the UA family when the getUAFamily method is invoked from a datasource. To test the method we needed a UA family definition file to that getUAFamily should use to match the best fitting UA family. The JSON file we used is shown in Code Snippet 1. The actual data can be viewed in the appendices.

{

"chrome" : {

"uaFamily" : "Chrome",

"isMobile" : false,

"features" : {

"video" : {

"h264" : true,

"webm" : true

}

}

},

"notmobile" : {

"isMobile" : false,

"features" : {

"flexbox" : true

}

},

"mobile" : {

"isMobile" : true

}

}

Code Snippet : The UA family definition JSON used for testing getUAFamily.

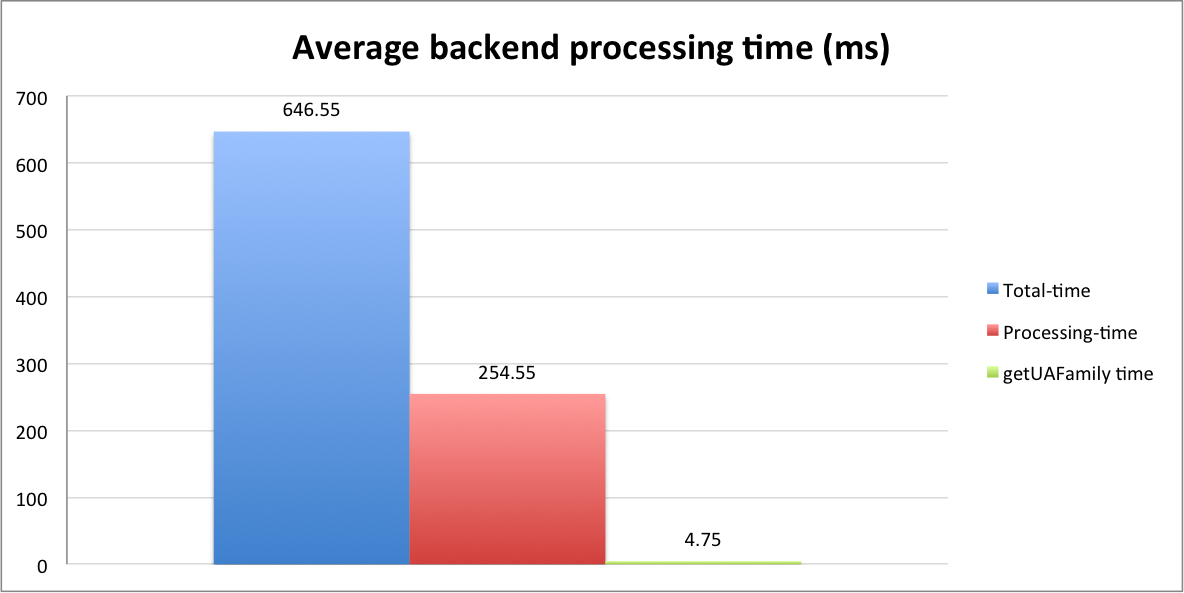


Figure 6: The average backend rendering and processing time spent, as well as the average time spent executing the getUAFamily method.

From our results shown in Figure 6 it is evident that the time spent executing the getUAFamily method is so small as to almost be irrelevant when looking at the processing time, and even more so when looking at the total rendering time. getUAFamily takes up 1.9 percent of the average processing time, and 0.7 percent of the average total rendering time.

From these results we can see that most of the performance hit from the plugin is in the case of encountering an unknown UA. In the common case it has a negligible impact on the performance of the request round-trip times and backend processing time. In the common case the plugin does lookups in the database and resolves the UA family each time a datasource invokes getUAFamily, which we can see from our results does not impact the performance in any meaningful way. Database queries can be expensive, but they are not in our case when using MongoDB. Its caching scheme uses all available memory on the machine as a cache, and it will thus cache most results, leading to fast database lookups in the common case. MongoDB documents are more similar to objects than relational data, meaning the process of mapping between POJO’s and database data is potentially less expensive for MongoDB. It unknown, but likely that using a relational database can increase the time of both getUAFamily and the HttpInterceptor itself because of the difference in caching schemes and data structures.