Chapter 5 – Discussion

Throughout this thesis we have presented the theories and concepts that underlie the idea of the Detector plugin. In this chapter we will look at the merits of the implementation and discuss its advantages in the context of improving performance of Web sites, with focus on the future friendly Web [[1](#_ENREF_1)].

# The advantages of using RESS and Detector

The growing popularity of using mobile devices to browse the Web has lead to innovations in Web development such as RWD. This concept has quickly become popular among Web developers, especially front-end developers, as it provides a simple way to do feature-detection on the front-end that makes it easy to tailor the style of Web pages to the viewport of each individual browser. The concept does have its problems, though, which has been pointed out by several prominent Web developers [[2](#_ENREF_2), [3](#_ENREF_3)]. The crux of the concept is that it relies entirely on CSS, and specifically Media Queries, which are situated entirely on the front-end. It ends up having the same monolithic markup for every device and also leads to all CSS, scripts and media being sent to every device, as the server is completely oblivious to the capabilities of the requestor. This can lead to performance issues, especially when dealing with larger web pages, scripts and other media. While the layout is changed and elements may be hidden from users on mobile devices by using media queries, all the content defined in the page’s markup is still downloaded by the browser. Large images and scripts that may never be visible to the user will make a page load slower and spend more bandwidth than necessary. Considering how bandwidth is still at a premium in the mobile context, it’s unfortunate that this widely adopted development method can be detrimental to the user experience.

To reduce the amount of bandwidth used, lighten the workload on the front-end and improve the user-experience further, it is necessary to delegate at least some of the responsibility to the back-end. Sending the same resources to a mobile device and desktop is counter-productive to what RWD is trying to achieve, namely a future friendly Web experience. Making entirely separate templates for different form-factors is often called “Device Experiences” because it changes the browsing experience of the page depending on the device. This has been standard fare for many years, and does reduce bandwidth use by limiting the amount of data sent by having it explicitly designed for the requesting device class. These kinds of pages can be made semi-responsive by utilizing certain RWD techniques, such as fluid grids, to adapt to small changes in screen size. They are not responsive in the sense Ethan Marcotte meant when he introduced the concept of RWD, though, and because of their mostly static design; they are definitely not future friendly. Even though RWD has become extremely popular since its introduction in 2010 [[4](#_ENREF_4)], the report that 82 percent of Alexa’s 100 top sites use server-side detection to tailor some amount of their content [[5](#_ENREF_5)] lends credence to the idea that using some kind of server-side detection is still useful. This is where the concept of RESS comes in.

As described in chapter 2, RESS suggests combining RWD with specific components of the markup rendered server-side to improve bandwidth usage, performance and user experience through optimizations done on the back-end. Luke Wroblewski, who came up with the concept, claims that it is effective amongst other things because it allows authors to only create one set of markup with components defined as templates, without having to worry about it working on different devices [[2](#_ENREF_2)]. This avoids splitting the code base, which would happen in the case of using Device Experiences. The theory surrounding the RESS concept is sound, and should definitely be taken seriously as a stepping-stone towards a solution to creating completely future friendly Web sites. While RESS is mostly a theoretical concept, Detector implements it practically. It also tackles a few of the problems mentioned by Wroblewski in his original article [[2](#_ENREF_2)], such as how to define device classes and improving upon the accuracy of device detection by combining server-side UA detection with client-side feature detection.

Detector improves server-side detection by including client-side feature detection when encountering unknown UA’s, in line with Alex Russel’s suggestion that feature tests only should be run in this case [[6](#_ENREF_6)]. The addition of these kinds of feature tests means that the server can be aware of exactly what capabilities the requesting UA has without any prior knowledge. It removes the need to maintain any kind of DDR, as the system itself is capable of figuring out the features of new UA’s dynamically. This is naturally limited by the feature tests that the system uses, but it is also obvious that updating the tests, which in this case is handled by Modernizr, is a lot less time consuming than updating a central DDR every time a UA is changed to support new features or a brand new type of UA is released. Considering this, the claim can be made that this kind of server-side detection is a leap forward in terms of creating a future friendly system which is not limited by the data stored in a DDR.

Detector also suggests a system for handling and easily defining device classes in what Olsen calls “browser families” [[7](#_ENREF_7)], something that is only mentioned in passing in Wroblewski’s article, even though it is key to making a RESS system work [[2](#_ENREF_2)]. These families are what define the content the server renders in each individual component of a Web page using RESS. His method of defining families is both robust and user friendly both of which are important for a system to enjoy widespread use. It also makes it easy to define new families should the need arise, for example if a new kind of device class is introduced. The modular design of the browser family system allows it do be easily extensible, and thus future friendly by adapting alongside the client-side feature tests.

Performance has again become an issue in Web development when considering data traffic over cellular networks (3G, LTE, etc.). Not only is the bandwidth of cellular networks smaller than regular networks, devices also experience much higher latency on them because of routing, among other things [[8](#_ENREF_8)]. A report published by Juniper Research in April 2013 states that global data traffic on mobile devices will reach 90 000 petabytes a year by 2017. The same report also claims that about 60 percent of this will go through WLAN, not cellular networks, due to network providers building WLAN based “IP-zones” to cope with demand, along with improving their cellular networks [[9](#_ENREF_9)]. While the widespread usage of WLAN hotspots will lessen burden cellular networks, 40 percent of traffic will still go through them, which needs to be addressed.

To confront this problem, several issues need to be considered, most of which have to deal with the slow speeds, large distances and prices of data traffic over such networks. On top of this the devices themselves also have limited resources, such as processing power and memory. To improve performance on the mobile Web, several things need to be addressed: The amount of HTTP requests need to be reduced, images and other media needs to be minimized or eliminated, redirects should be avoided and JS needs to be optimized or its usage should be minimized [[10](#_ENREF_10)].

RESS can help with all of these issues, as it allows full control of what is delivered to the client after the first request is made. To reduce the amount of HTTP requests, developers can consolidate or reduce CSS and JS where it is needed, as well as reducing the amount of images and other media in the markup, all of which will trigger additional requests to be made to fetch them. This can be done dynamically when certain device classes are encountered. Redirects can be avoided entirely because components are changed on the server, which gives the ability to keep entirely different pages under the same URL. Because of the component-based approach RESS uses, the complexity of dynamically changing between layouts on the server is kept to a minimum. This in turn simplifies many of the steps needed to optimize for weaker devices, such as mobile phones, both through reducing HTTP requests in various ways and by reducing the performance requirements for viewing the site in terms of processing power and memory usage.

The concept of server-side feature detection, along with the browser family system allows Detector to adapt to changes and new UA’s without relying on DDR’s that need to be maintained. The components in RESS give Web pages built using it an inherently modular design, which allows them to be extended to support new families by creating a new family definition and component markup where needed. Olsen also mentions in his article that while these concepts come out of trying to solve mobile issues, they should not be pigeonholed as mobile solutions [[7](#_ENREF_7)]. RESS and Detector are not strictly mobile solutions and can help by offering a robust platform for building future friendly Web sites and apps, he believes. Considering the merits of these concepts, along with the server-side feature detection and browser family systems, Olsens’s claim does not seem too far-fetched.

## Disadvantages of using RESS and Detector

Most of the notable disadvantages of RESS come from the fact that it is so reliant upon device and feature detection. While the concept of delegating the rendering of components to the server and sharing the responsibility of adaptation between both the client and the server is a solid one, device and feature detection has several different potential problems that can hamper the functionality of a RESS Web site. It is reliant on uniquely identifying UA’s, which currently can only be done reliably by using the UA string sent with HTTP requests. UA strings, as mentioned in chapter 2, can be spoofed and otherwise misrepresent the actual UA making the request, be it through proxy servers or something else [[11](#_ENREF_11)]. This might cause the RESS system, such as Detector, to render components for the wrong device class, or conduct feature tests on a UA that is not the one it makes itself out to be. For instance: certain mobile browsers have settings that allow the user to send the desktop browser version of its UA string. This can, in the case of the Detector system, lead it to believe the desktop browser only has the features of the mobile version if it is the first time it encounters this UA string. These are edge-cases, but they are possible and have to be sorted out somehow.

Another problem with the feature detection system is that it needs JS to work. JS is commonly activated in modern Web browsers, but users can choose to turn it off. In this case it is impossible to detect features on the client. If a new UA string is encountered and the requesting client has deactivated JS support, the system needs a way to handle the situation so that tests can be run at a later time on a client with JS switched on. This needs to be taken into account to be able to get complete data from every UA, even in the case where certain instances of a UA have JS deactivated.

# Making the Web Future Friendly with RESS

As mentioned in chapter 2, making a Web site Future Friendly consists of several things: thinking “content first”, being data-centric, structuring content in a way that is future-ready, as well as robust device and feature detection [[1](#_ENREF_1)]. Dave Olsen mentions in his article that he believes that RESS with systems such as Detector can create a robust platform for Future Friendly Web sites and apps [[7](#_ENREF_7)]. While we’ve demonstrated that the Detector concept has the potential to provide the latter, the three former are not something RESS or Detector enforce explicitly. It can be argued that RESS encourages both “content first” and data-centric mindsets as well, though.

## Content First

Thinking “content first” is central to creating Web sites that are Future Friendly. Considering the variety in devices used to view Web sites, it is evident that focusing on the content – the “meat” of the site – is the most important. High-resolution widescreen monitors have allowed desktop focused Web sites to have a lot of extraneous filler outside of the core content because of the extra available screen space. This is naturally a luxury that is not afforded mobile phone screens and other small devices, both current and future. Concepts such as Mobile First [[12](#_ENREF_12)] can help developers in creating Web sites that focus on the content and other key components, while filtering out all unnecessary containers and filler. Even though RESS should not be seen as a purely mobile concept, it does help enforce many of the same principles that concepts like Mobile First emphasize, such as focusing on the constraints of the least capable device, and designing the site thereafter.

RESS helps developers focus on the content of the site first and foremost from being a methodology based around adapting to different devices. The one common thing present on a site, no matter what version a user is viewing, is the content. It is what the user is there for, and it is thus natural to focus on the content and building the site around it. In RESS this is especially poignant, as each component on the site can be changed depending on the device class viewing it, but the content always has to be kept intact. Thus we can argue that RESS as a concept encourages thinking “content first”.

## Data-centric

Creating sites using RESS can also encourage data-centric approaches, as developers are given incentives to reuse content across components for different device classes. Concepts such as COPE (Create Once, Publish Everywhere), in which content is made independent from the display [[13](#_ENREF_13)] are natural extensions to the back-end of RESS sites. This is because components are meant to adapt and orbit around the content itself to create the best possible user experience, no matter the device. The basic layout of components in RESS sites means that having data that is interoperable between different components is highly desirable. The concept itself encourages reducing redundant code by reusing components where needed, it would then only seem natural to extend this to reusing data as well, adhering to the principles of COPE, for instance. Being able to reuse data means being data-centric from the start, to set up the data in such as way as to optimize the ability to utilize it, no matter the context it is being viewed in.

While RESS itself is not reliant upon a data-centric approach, it can benefit greatly from it. As Luke Wroblewski demonstrates in his blog post describing how all the data in his Bagcheck Web app can be accessed from a command line interface due to it being available through a generalized API [[14](#_ENREF_14)]. His Bagcheck blog is incidentally also the first site he developed using RESS, and he uses as an example in his article describing the concept [[2](#_ENREF_2)]. Based on this, we can argue that RESS as a concept can both encourage data-centric approaches as well as benefit greatly from them. A Web site built using RESS is highly modular in its design and reliant on being able to adapt its presentation based on the results from its device and feature detection. The ability to have the data be available through a generic API, for instance, would remove one point of uncertainty for developers, as well as making the site itself more Future Friendly.

# Choosing Enonic

CMS’s are widely used in larger Web sites to simplify the administration and usage of the sites. They allow content creators to easily create and publish content on a site without having to know all the underlying technical details. The basis of this thesis was to look into the concepts of Mobile First, RWD and how they tie into the problems we face with improving both performance and user experience on the mobile Web. The idea was to look into how we could improve performance of sites that are unsuited for mobile devices. This lead us to wanting to look at sites within the Norwegian public sector, as these have a tendency to be bloated with content and have layouts specifically made for desktops.

Looking into this we found Enonic, a CMS that is used in several large Web sites in the Norwegian public sector, such as The Norwegian Labor and Welfare Service (NAV), The Norwegian Public Roads Administration (Statens Vegvesen) and The Norwegian Agency for Development Cooperation (NORAD) [[15](#_ENREF_15)]. Looking at how the system underlying these sites worked, we could establish what is possible with regards to improving their performance on mobile devices and otherwise.

We decided that it would be interesting to look into the merits of the RESS concept, and thus we needed to look into the capability of doing this in Enonic. The CMS has its own system for adapting pages to different devices, but it is very basic and does not provide much in terms of feature detection and accurate definitions of device classes [[16](#_ENREF_16)]. To be able to use RESS it needed a more accurate detection scheme that allowed it to dynamically detect features on requesting clients and resolve device classes.

It has been noted that many modern CMS’s, Enonic included, have attributes that are poorly suited for mobile, such as not separating text, images and video so that assets can be provided in different sizes, editors that allow content to not be republished everywhere, and HTML templates, JS and CSS that assume desktop bandwidth, CPU and caching behavior [[17](#_ENREF_17)]. Many of these problems come from the fact that a lot of CMS’s conflate content management with Web publishing, which would be entirely at odds with the concept of COPE [[13](#_ENREF_13)]. This comes back to Sæterås’s article, where he states that the entire value chain in the Web infrastructure has to be made responsive, not just the front-end [[18](#_ENREF_18)]. We saw RESS as a possible solution to some of these problems, and decided that implementing a Detector-like system to improve Enonic’s device classification system would be a good way to look into how to improve many of these observed problems with modern CMS’s on mobile devices and otherwise.

Enonic’s support for plugins, along with its already existing device classification system presented the possibility of making a comparative analysis of our implementation and the native system, giving us a good metric for the possible improvements our system provides. It also meant that it was possible to create such a new detection system for the CMS without hacking the source code and making users reliant on a code base modified from the original system. The plugin support coupled with its usage in large Web sites within the Norwegian public sector; along with its less than ideal device classification system are the main reasons why we chose to use Enonic for our implementation. That is not to say that the concept could not work, or be relatively easily ported to other Java-based CMS’s, such as dotCMS – a popular open-source web CMS – which also supports plugins built much the same way as Enonic, with OSGi bundles and Spring [[19](#_ENREF_19)].

# The plugin

Our plugin is for the most part a direct implementation of Dave Olsen’s own concept as he describes it in his article [[7](#_ENREF_7)] and from the source code of his Detector project, which is written in PHP [[20](#_ENREF_20)]. The implementation is meant to contain the necessary features needed to support a RESS solution in the Enonic plugin. This implies that it must have a robust feature detection system along with functionality for defining device classes that can be resolved when needed. There are several minor differences between our and Olsen’s implementation, but they are mostly similar in execution and functionality, even though the languages, libraries and technologies used are often vastly different. For example: Olsen uses PHP and stores the detected UA features in the file system, while we use Java in Enonic and store data in a NoSQL database. These are underlying technologies, though, and do not change the actual functionality, which apart from minor differences is the same.

## Performance impact

The whole point of Detector is to improve the rendering time and general latency of loading Web pages on all devices, as well as increasing the flexibility and accuracy of detecting which device is making the request, and what features it supports. This is all done in the interest of giving developers the tools to create content-centric Web sites without having to worry about creating entirely new Web pages for every device imaginable. This all has the effect of improving rendering time and latency by cutting away all unnecessary fluff from the versions that are sent to less capable device on slow connections, such as mobile phones on cellular networks. Since the point of the plugin is to improve performance, it was necessary to establish the impact the plugin had on the performance of Enonic.

In chapter 4 we went through how we executed our performance tests and their results. We described why performance is important, especially in the mobile context, and how the Detector plugin might impact the performance of Enonic when it is used on a Web site. The results showed that the plugin had little to no impact on the performance of the Enonic demo Web site in the common case, while it did have an impact in the case of having to perform feature tests on the client. In this case the culprit was the execution time of the JS, which only showed a significant execution time on mobile browsers, and was largely negligible on desktop browsers. Based on the results of these tests we claim that the performance impact of using the plugin is negligible compared to not using it. Especially when considering that the only time it showed a significant performance hit was in the rare case of encountering a new UA. In these cases the rendering time of the Web page was increased by 81 percent. This increase in rendering time will ideally only happen to the very first user of that UA that makes a request to the server using the plugin. All subsequent requests from that UA will not experience any significant lag compared to an identical server not using the plugin, meaning all users except the first one will experience the rendering times of the Web pages without delay.

Based on these results we conclude that using the plugin will improve the UA feature detection capabilities of Enonic, as well as provide a more flexible device classification system without impacting the loading times of its Web pages in any significant way.

## Detector versus Enonic device classification

Enonic provides its own device classification system [[16](#_ENREF_16)], which is completely reliant upon analyzing the UA string provided by the requesting UA. On top of this the script which checks the string has to be defined by the administrator of the page itself, and is based on XSLT, a language that is not always easy to write, read, or maintain. The system is based on writing regular expressions (regex) that are matched against the provided UA string, which subsequently outputs a string that is defined along with the matched regex. This string represents the detected device class.

Our plugin provides a more detailed detection scheme that not only checks the UA string, but also checks the requesting client for all supported features. In addition it gives administrators the ability to define the device classes based on their detected features and capabilities in a JSON file, which is easy to write, read and maintain. Our system is also independent of knowing the structure of the UA string, as our client side feature tests will be able to run no matter what the string looks like, and is thus Future Friendly in a way that the Enonic system cannot be. Being able to detect the features of new, unknown UA’s without any manual input from users or administrators. The Detector device classification system is naturally dependent on having a device class definition file that is maintained, but it does not have to be updated every time a new UA comes along. With well thought out class definitions a system can be left unmodified for long periods of time without being in danger of leaving new devices completely unsupported.

## Using Detector in Enonic Web pages

It is not just a matter of installing the plugin to be able to use it. To use it on Enonic Web pages, certain changes need to be made in terms of structure and usage of datasources. Firstly, every page that is made using RESS principles needs to invoke the getUAFamily method from the FunctionLibrary extension. Upon page load this will resolve a device class and inject it into the XML that Enonic generates before rendering a page template. The injected XML element with the device class string can be checked through conditional statements in the XSLT page templates. This way each block of conditional statements can be considered an individual template for a specified device class, much in the same way Olsen does using Mustache, as described in chapter 2. This is not entirely dissimilar to how it is done with Enonic’s own system. The difference is that instead of checking the native XML element under “context”, we check our own custom XML element provided by the datasources through the FunctionLibrary extension.

For the plugin to work on every page, a datasource that invokes the getUAFamily method has to be defined for every page template, and for every necessary portlet template. Once this is done, the conditional statements within the XSLT template should function as intended. This means that some work has to be done in order to move an existing Web site over to our plugin, and the amount of work correlates directly to the number of page templates on the given site. The added flexibility provided by Detector thus comes at a price of increased complexity throughout the site.

## Potential problems

There are several potential problems that may be encountered when using the plugin in a production setting. The first one we have already mentioned earlier in this chapter: false or misrepresented UA strings. The plugin has no way of knowing if a UA string attached to a request actually belongs to the requesting UA. It can be changed by the user, the UA itself or by a proxy somewhere on the line between the UA and the server. In these cases the system has to trust that the UA is telling the truth, and store whatever features it detects. There are currently no way to fix this problem outright, as there are no certificates or other way of verifying the validity of a UA string in relation to the UA that sends it.

One way to lessen the problem might be to attach a Time To Live (TTL) on each entry in the database, and rerun the feature tests on the requesting UA of entries that are expired. This way we can lessen the risk of having a misrepresented UA stored in the database forever. There is also a possibility of being able to verify the UA name and version on the client using additional JS. The navigator global object in JS contains information about the UA it is running in [[21](#_ENREF_21)]. Though it is apparently unreliable, as the various attributes, such as appName returns the wrong name for various UA’s, such as returning “Netscape” for Gecko- and WebKit-based browsers like Firefox and Chrome, respectively [[22](#_ENREF_22)].

Another potential solution is to test several instances of the same UA string, and discard UA data with outlying results after a set amount of tests have been conducted. This would require a substantial amount of extra business logic in the interceptor extension, such as a scheme for deciding which test results are true and which should be discarded. In the long run this could provide a more robust and trustworthy result compared to the current scheme, though, despite it increasing the number of worst-case requests made that trigger a full client-side test suite to be run.

Another problem that is slightly related to the one mentioned earlier is the case of the UA being tested lacking JS support or having it turned off by the user. It is impossible for the server to distinguish between these two cases, and because JS support is lacking, there is no way to check on the client either. To alleviate the potential problem of redirecting the client to the feature tests every single request, it is necessary to store some amount of information about the requesting UA. Currently this is done by storing the UA object normally, but with a “nojs” attribute set to “true”. The problem with this is that the UA might in fact support JS, but the user has turned it off. Because of this the stored information might need to have some kind of TTL, like in the case of the fake UA string. Another possible solution may be to store the UA information on a per-session basis in the case of no JS support, having everything be discarded when a session expires.

We have done several performance tests using the plugin, but they have been done on a small scale on a single server running on a regular MacBook Pro, as stated in chapter 4. There is no way of knowing for sure how the plugin will perform in a production context running on an enterprise-scale server with thousands of requests per day made for hundreds of different pages. In these cases there is no sure way of saying what will happen, and if it will perform as we expect from the performance tests we did. The only way to know this for sure is to actually perform tests on a Web site of this scale.

Such a problem could be in the form of caching on a reverse proxy. A reverse proxy is a type of proxy server that gets resources from one or more servers on behalf of a requesting client. The reverse proxy then acts as the origin of the content it gets from the underlying servers [[23](#_ENREF_23)]. Caching on reverse proxies means that it stores any file that it gets from the underlying infrastructure, and returns it directly upon subsequent requests. Since our RESS solution manipulates HTML markup based on what class of device is making the request, it is important to disable caching of HTML files in reverse proxies when using RESS systems, as this would lead to the same markup being served all devices, thus negating the effect of using it. Caching of HTML can be kept on a per-session basis, as this would not interfere with Detector, though this would only improve performance in cases where the cache on the client is either invalidated or turned off entirely.

## Future work

In its current state the plugin works as intended, but as mentioned earlier it has not been tested in a production environment, and may encounter several problems when subjected to rigorous usage. Future work on the system should include attempting to find stable solutions to these problems, along with actually testing it in a realistic setting.

The next step for the Detector concept should probably be to make it work independently of superfluous systems, such as CMS’s. There are several ways this can be done, the most sustainable and Future Friendly of which is to provide it as en external service through an API, for example using REST principles. We could for instance have the Detector plugin be standalone under a regular Java servlet application. Because the system is already written in Java, it would not take much work to convert it to run under a regular servlet. The function library extension methods could then either keep returning XML, or be changed to return JSON if that is deemed as more practical.

While the lookup part of the application could work just fine as a separate service, the interceptor extension part of it raises several concerns. One of the largest advantages of Detector over a regular DDR is that it dynamically detects features of a UA as it encounters it, without the need for human interference. As a third party service this is not as simple because it is reliant on sending the tests as JS to the requesting Web server. An issue of trust then arises in the sense that the requesting Web server in no way can be sure if the JS it gets from the service is not malicious. The service also has to have systems in place to handle the results from the requesting Web server, as the results sent back could be altered before being forwarded to the service. Having it be a third party API openly available on the Web can thus be deemed to be unsafe in its current form. It could be used in an internal system, though, where the API is situated within the back-end architecture. In this case the server providers would have complete control over what is being sent, and can thus trust what is being sent to and from the Detector service.

# Summary

The Detector system is an intriguing proposition and a step in the right direction towards having a Future Friendly RESS system that can make the Web available and more importantly tailored to all devices. By looking at concepts such as COPE we have discussed the merits of RESS and how it can provide a platform for a Future Friendly Web that sets content first and orbits around its data, instead of its presentation.

We have looked at the potential problems of RESS and Detector, which is primarily focused on the uncertainty that is inherent in relying on UA strings for identification. Unfortunately, as we have seen, this is currently the only option available to us. But while this is a potential problem we have made a few suggestions as to how this can be fixed when discussing the plugin itself.

We have described why we chose to implement Detector generally and for Enonic specifically. Looking at the relevance of RESS in the context of CMS’s is important, especially in the light of arguments that many of the things RESS provides can be used to solve many of the problems we face when using CMS’s for mobile [[17](#_ENREF_17)]. We also discussed the functionality of the plugin, how it stands up to the native device classification system and how existing sites can be converted to using our plugin, with some work.

# Bibliography

[1] L. J. Wroblewski, Scott; Frost, Brad; Keith, Jeremy; Gardner, Lyza D.; Jehl, Scott; Rieger, Stephanie; Grigsby, Jason; Rieger, Bryan; Clark, Josh; Kadlec, Tim; Leroux, Brian; Trasatti, Andrea. (2013). *Future Friendly*. Available: <http://futurefriend.ly/>

[2] L. Wroblewski, "RESS: Responsive Design + Server Side Components," in *LukeW* vol. 2012, ed, 2011.

[3] J. Grigsby, "CSS Media Query for Mobile is Fool’s Gold," in *Cloud Four Blog* vol. 2012, ed, 2010.

[4] E. Marcotte, "Responsive Web Design," in *A List Apart* vol. 2012, ed, 2010.

[5] R. Cremin, "Server-Side Mobile Web Detection Used by 82% of Alexa Top 100 Sites," in *CircleID* vol. 2013, ed, 2012.

[6] A. Russel, "Cutting The Interrogation Short," in *Infrequently Noted* vol. 2012, ed, 2011.

[7] D. Olsen, "RESS, Server-Side Feature-Detection and the Evolution of Responsive Web Design," in *Mobile in Higher Ed* vol. 2012, ed, 2012.

[8] I. Nir, "Latency in Mobile Networks – The Missing Link," in *Performance Calendar* vol. 2013, ed, 2012.

[9] N. Bhas, "Mobile Data Offload & Onload - Wi-Fi, Small Cell & Carrier-Grade Strategies 2013-2017," 2013.

[10] N. C. Zakas, "The Evolution of Web Development for Mobile Devices," *Queue,* vol. 11, pp. 30-39, 2013.

[11] C. Krycho, "User Agent Detection Will Get You In Trouble " in *Designgineering* vol. 2012, ed, 2012.

[12] L. Wroblewski, *Mobile First*, 2011.

[13] D. Jacobson, "COPE: Create Once, Publish Everywhere," in *Programmable Web* vol. 2012, ed, 2009.

[14] L. Wroblewski, "Bagchecking in the Command Line," in *Bagcheck* vol. 2013, ed, 2011.

[15] Enonic. (2013). *Listen to our customers*. Available: https://enonic.com/en/home/reference-customers

[16] Enonic. (2013). *Device Classification*. Available: https://enonic.com/en/docs/enonic-cms-47?page=Device+Classification

[17] J. Grigsby, "Our Content Management Systems are the Mainframes of the Mobile Era," in *Cloud Four Blog* vol. 2012, ed, 2012.

[18] J. A. Sæterås, "Next steps of Responsive Web Design," in *mpulp* vol. 2012, ed, 2011.

[19] dotCMS. (2013). *dotCMS Home Page*. Available: <http://dotcms.com/>

[20] D. Olsen, "Detector," 0.8.5 ed. GitHub, 2011.

[21] Mozilla. (2013). *window.navigator*. Available: https://developer.mozilla.org/en-US/docs/DOM/window.navigator

[22] Mozilla. (2012). *window.navigator.appName*. Available: https://developer.mozilla.org/en-US/docs/DOM/window.navigator.appName

[23] Apache. (2011). *Forward and Reverse Proxies*. Available: [http://httpd.apache.org/docs/2.0/mod/mod\_proxy.html - forwardreverse](http://httpd.apache.org/docs/2.0/mod/mod_proxy.html#forwardreverse)