



Which practices should be modified or avoided altogether by developers for the mobile Web?

BY ALEX NICOLAOU

Best Practices on the Move: Building Web Apps for Mobile Devices

IF IT WAS not your priority last year or the year before, it is sure to be your priority now: bring your website or service to mobile devices in 2013 or suffer the consequences. Early adopters have been talking about mobile taking over since 1999—anticipating the trend by only a decade or so. Today, mobile Web

traffic is dramatically on the rise, and creating a slick mobile experience is at the top of everyone's mind. Total mobile data traffic is expected to exceed 10 exabytes per month by 2017, as shown in Figure 1 (in case your mind is not used to working in exabytes yet, that is 10 million terabytes per month, or almost four terabytes per second).¹

Of that data, iOS and Android devices consume the lion's share, which

suggests a focus for any immediate Web development efforts. Figure 2 shows a breakout of megabytes per month downloaded in 2011 and 2012, indicating these platforms are widely used and trending upward.

So, piece of cake, right? Just take some of the great literature on writing desktop Web sites and apply it to mobile. For example, Yahoo!'s excellent YSlow tool and corresponding perfor-

mance rules⁸ are an excellent starting point; Google's PageSpeed provides similar guidelines.⁵

The best practices for optimizing website performance, as articulated in YSlow and PageSpeed, were developed largely with the desktop world in mind. Now that mobile devices need to be accounted for as well, some of those practices might not have the intended benefits—or might even degrade performance on mobile.

Mobile users expect a more application-like experience. Smooth load experiences, fast and animated transitions, and application-centric error messages

are part of what makes using Internet services on mobile devices bearable and perhaps even enjoyable in spite of slow networks, tiny screens, and cold fingers (at least for those of us in the north). Mobile users demand that you deliver more with less, and the old rules of website design and implementation for the desktop need a lot of adaptation to create a slick mobile Web-app experience.

Which Recommendations Should Be Followed?

Although the recommendations in Google's PageSpeed and Yahoo!'s YSlow work well for desktop develop-

ment, some of them need to be reconsidered to provide maximum benefit for mobile development. Network connectivity, while far more ubiquitous than ever before, is still spotty; a mobile site needs to compensate for this problem so users generally feel the app is responsive even when the network is not. Older phones parse and execute JavaScript 100 times slower than desktops, and even the latest phones are still slower than desktop devices by a factor of 10—and this slowdown magnifies the smallest of sins. Handling JavaScript and network problems well will bring your performance within the expectations for any native mobile app. Then all that will remain is a host of fit-and-finish problems that require a solid understanding of when and why the browser paints and lays out to add the final polish.

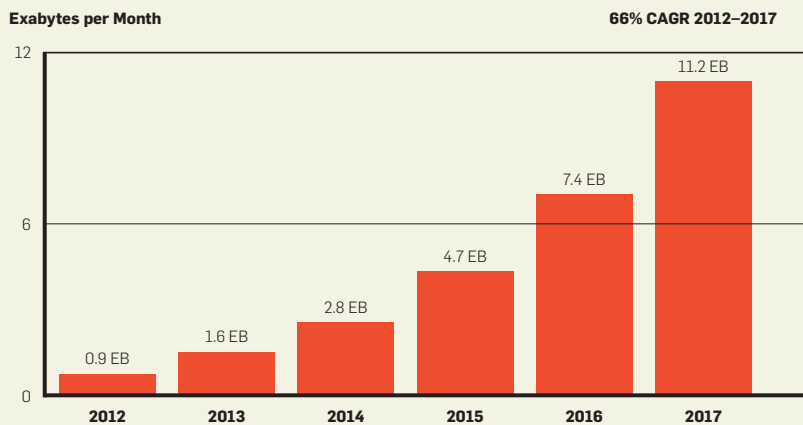
With these thoughts in mind, let's consider how YSlow and PageSpeed guidelines apply to the world of Android and iPhone Web-app development.

Eliminate HTTP Requests and Round Trips. A core recommendation from both Google and Yahoo! is to minimize HTTP requests. This is a critical concept that needs special expertise and care to be effective on mobile. The good news is that existing recommendations on using CSS sprites to represent images embedded as inline data:URLs are techniques that work well on mobile.

Image maps should work as well, but their inflexibility in terms of layout options is a bit of a drawback for mobile. For example, for a truly slick mobile experience you will want to make good use of orientation changes that affect the horizontal dimension of your Web page. With image maps you would either have to use two of them or dice them up, in which case you might as well use CSS sprites and do your layout with markup as usual. So while image maps should work as a technique, on the whole sprites work better.

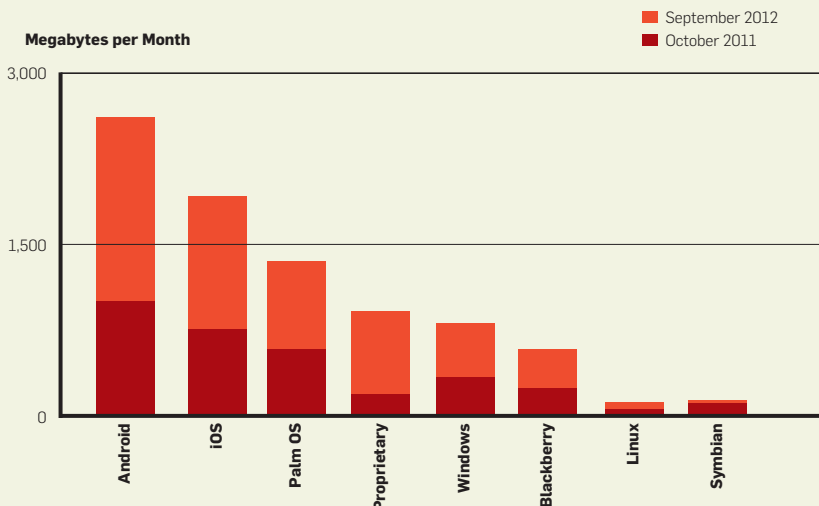
These existing techniques alone will not be enough to supercharge page load on mobile. On iOS, for example, pages are cached in memory only, and HTML files still appear to be limited to 25KB uncompressed (though other resource types can be much larger now). In addition, some iOS devices are still limited to a maxi-

Figure 1. Mobile data traffic projection.



Source: Cisco Global Mobile Data Traffic Forecast, 2012–2017

Figure 2. Megabytes per month by year and platform. Figure 2. Megabytes per month by year and platform.



Source: Cisco Global Mobile Data Traffic Forecast, 2012–2017

mum of four parallel connections to download your content—which means you cannot afford to rely on multiple external sources to load in a reasonable amount of time. Finally, the HTTP 1.1 specification recommends at most two parallel downloads per hostname. Of course these values are changing all the time; to find tools and data on the latest limits, Steve Souder's *Mobile Cache File Sizes*⁷ and Ryan Grove's *Mobile Browser Cache Limits, Revisited*⁶ are good starting points.

If you truly want to solve the cache issue, it is important to force the mobile browser to cache all the unchanging content on a site permanently, using HTML5 application cache, commonly referred to as app cache. Because app cache is still on the bleeding edge, it is best to use the minimal subset possible for your application.

The cache is controlled by a single manifest file that tells the browser where to get resources. The minimum functionality would be to make your website load as a single large Web page, and have the manifest file list that page and a hashCode as a comment that is updated when the Web page changes. Dynamic content can be loaded via XHR (`XMLHttpRequest`) and inserted into the document. This technique is most effective if you design the prefix of your page (up to about the 1,400th byte, in order to fit into a single packet across almost any TCP/IP network) to render a basic framework for the page before any script is executed.

If you design your site in this way, you can also have the framework of the page fade in by providing an opacity transition from 0.0001 to 1.0 just after the framework of markup is loaded. This will create a very slick “app-like” startup experience. The opacity property is the method of choice for smooth transitions because the browser will have prerendered everything that is at opacity 0.0001, but the user cannot see it, and fading in is slick and smooth. Using opacity 0 will cause a white flash when the browser repaints, and using the display CSS attribute will be even worse because it will cause re-layout. If you cannot get the basic framework of your page to fit into the first packet, then you can instead load a spinner or a logo, again at opacity 0.0001, and conditionally de-

Figure 3. Reduce abandonment with a conditional spinner for non-approached loads.

```
<script>
if (window.applicationCache.status == 0) {
  // Page was loaded from the Network; reveal the spinner
} else {
  // Page was loaded from AppCache; heavier-weight startup applies
}
</script>
```

test whether the app cache is already populated before showing the spinner. The JavaScript for accomplishing this should appear in a script tag after the markup, as illustrated in Figure 3.

Using this pattern ensures the user will immediately get the feedback that your page is loading, thus reducing the chance of abandonment at the most critical moment: when the user first discovers your site.

To complete the effect, you also need to ensure your serving infrastructure flushes the framework of the page to the network before doing any heavy lifting on the server side, so that all overhead on first request is deferred until after the initial packet is sent. This will prove remarkably speedy even on spottier connections. It is also not a bad idea to make a request to the server at this point to track the page-load metric (more on this later in the section on JavaScript parse performance).

Following the recommended pattern means that even on the first load, all code, assets, and CSS for your site will load in a single round trip, and that will create the best possible mobile experience. Dynamic data will of course have to be loaded separately, and the same level of care and caching should apply there, too, except that HTML5 storage APIs will have to be used to cache follow-on content. Fewer than four round trips—one for the DNS lookup, one for the initial page content, and up to two for dynamic content—should produce a responsive mobile experience for your users.

Page Speed's best practices for minimizing round-trip times⁴ also include a number of techniques for avoiding round trips you might overlook, such as DNS lookups and redirects. Almost every recommendation relating to round-trip times is well worth implementing, as Google has provided good techniques for combining the content that forms all the components of your site.

The last item in that section of Google's recommendations focuses on parallelism achieved by serving from multiple hostnames; but parallel downloads are likely to prove ineffective for mobile devices. Combining your resources and components into single files and ensuring they are served from app cache will be the most effective technique.

Use Compression. Using compression on all content that would benefit (essentially everything except images and video) remains a great recommendation for mobile. Mobile CPUs are getting faster much more quickly than mobile networks, and their speed does not change when a user is visiting the cottage. Even if you have diligently used app cache and local storage to save all resources locally on the device, you can expect the user will be regularly loading dynamic content, and all these requests should be compressed. To speed initial load time, even cached content should be delivered compressed whenever possible, though the benefit is less.

Manage JavaScript Parse Time. After caching and compression, script loading is likely to be the single biggest source of performance degradation for a website, and it is the more difficult to address. The big issue on mobile is that parse and execute times of script are much slower than one would expect. In fact, the rule of thumb is that parse and execute times are 10 times slower than when testing on a desktop. A JavaScript payload of as little as 100KB can cost 100ms of startup time even on reasonably recent phones such as iPhone 4—and the previous generation of iPhone was 10 times slower!

Therefore, it is certainly worth putting the vast bulk of script at the bottom of the page. To evaluate parse time, you can use two script tags as shown in Figure 4.

The start checkpoint is parsed and evaluated before parsing for the sec-

Figure 4. Monitor the time required to parse and evaluate your JavaScript.

```

<script>
var start_checkpoint = new Date();
</script>
<script>
var parsed_checkpoint = new Date();
// more script here
var eval_checkpoint = new Date();
var parse_time = parse_checkpoint - start_checkpoint;
var eval_time = eval_checkpoint - parse_checkpoint;
</script>

```

Figure 5. Declaring a block of JavaScript for deferred parsing.

You can compute the load time of the page with:

```
window.performance.responseEnd - window.performance.navigationStart
```

and document load time with:

```
window.performance.loadEventEnd - window.performance.responseEnd.
```

ond script tag begins. The second script is then loaded and parsed, and the time is recorded in the parse checkpoint (which technically includes a tiny bit of execution of the second script tag, but this is neglected and billed to the parser). The eval checkpoint is recorded only after your business logic has all been evaluated.

Chrome on Android now offers a much more powerful technique to debug startup time: using `window.performance` to read performance-related timestamps in the code. Figure 5 illustrates how to compute the load time of the page. Whether you are using JavaScript Date objects or the new `window.performance` timestamps, it is a good idea to record this number by making a request to your server side so that you can track page-load metrics as a basic performance indicator of your site—it is so easy to regress that without continuous monitoring, performance is almost sure to degrade over time.

It is highly instructive to look at these numbers to appreciate how bad parse time is, and then to apply tricks to avoid parse latency. The most powerful of these tricks is allowing the browser to load your JavaScript without recognizing it as script and defer parsing and initial evaluation to a time of your choosing.

There are two methods for accomplishing this. The older one is com-

menting out your JavaScript and using another block of script to uncomment the content of the tag only when it is needed, as described in the Google mobile blog post from 2009.³ More recently, a cleaner technique has been used for the same purpose. It involves setting the type of the script to an unrecognized type and changing it to text/JavaScript later. In this approach, you start with script blocks that look like this:

```

<script type="deferred" id="module1">
var start_checkpoint = new Date();
</script>

```

When you are ready to use the script, the JavaScript finds the script tag and modifies the type, which will cause the browser to parse and evaluate it at that time.

By putting your deferred script at the bottom and uncommenting it only when needed, you can very effectively amortize the cost of using JavaScript across the runtime of your application. If your modules are small enough, this technique is cheap enough to be applied at the moment the user first clicks a button or link that needs the script, which can be uncommented, parsed, evaluated, and executed with no noticeable latency.

Avoid Layout and Style Calculation.

One of the easiest ways of introducing unexpected latency into a site is inadvertently causing the browser to lay out

the document. Seemingly innocuous JavaScript can trigger style recomputation in the midst of execution, while a rearrangement of the same code can be much more efficient.

The rule of thumb is to avoid reading properties that rely on the position of elements on the page to be returned, because any such property could cause the browser to recalculate styles on demand and return the value to your script.

The easiest way to inspect examples of this problem is to use the developer tools in Chrome. Ryan Fioravanti's excellent Google I/O presentation² details how to do this by using adb (Android Debug Bridge) to forward a port on a desktop machine to the development-tool port on the Android Chrome browser. Once that is set up, the Events timeline can be used to spot Layout events followed by large gaps of time that represent style recalculation. In the example in Fioravanti's talk, paying attention to when style recalculation was happening made the event handler in question twice as fast as previously.

Monitor Request Size as Well. The focus so far has mainly been on the size of responses to the client and how long the client takes to process data on the client side. Request size can also be a problem, particularly for sites using a large number of cookies alongside every request. This is easiest to monitor on the server side, and ideally all requests made back to the server should be much smaller than a single TCP packet.

Preloading Components is most effective when the user's workflow is almost certain to cause a predictable action just after the initial page loads. For example, in a news site certain articles will have a large click-through rate caused by a combination of placement, content, and imagery, so it makes sense to preload the article page as soon as there is no other work to be done. This can give users a positively delightful experience where the site seems to load instantly.

Another example of this is the AJAX version of the mobile Google Calendar website, where the next day's events are loaded as the user clicks through each day. The effect is the user can easily walk through the week and see the data load instantly. You can compare this to the non-preloaded case by


using the month view to jump ahead a few days at a time, exposing how slow it can be to take a round trip to the server to serve the same quantity of data. The round trip involves a noticeable spinner even over Wi-Fi, while the precached data appears instantly, even on old devices.

Optimize Images. Particularly for dynamic images, applying the best compression available will pay off handsomely in overall page speed. You can test image quality on a device to determine if there is a noticeable difference between 50% JPEG quality and 80% or 90%, and note the significant reductions in size that can be achieved. Even dramatic reductions in JPEG quality are often not visible, and the savings can be as much as 40% of the file size. Similar advice applies to .PNG and .GIF files, though these are used less for dynamic content (and therefore they are served from app cache and are not as sensitive to size).


Avoid Inefficient CSS Selectors. Almost every recommendation in Google's "Optimize Browser Rendering" section applies perfectly to mobile, particularly the concerns about needlessly inefficient CSS. There is a trade-off to consider, however: using a complex CSS selector can avoid a Document Object Model (DOM) modification. While complex CSS selectors are considered inefficient by Google, they are fast compared with doing the equivalent work in JavaScript on a mobile device. For other cases where the CSS selector is not being used to avoid JavaScript execution, Google's recommendations apply.

Reduce DNS Lookups. Ideally, a site would load with 0 DNS lookups, as each represents a round trip with the potential to block all other activity. Minimally, however, a DNS lookup will fetch the manifest file, which is done in parallel with page load as the browser brings in resources from the application cache. So it is fairly safe to use the name of your host site and assume it will be in the system's DNS cache whenever the device is online.

If you rely on using multiple hosts to parallelize content loading, you may need one more hostname lookup. There is no point using more than two hosts to parallelize content loading since in the best of cases you



By putting your deferred script at the bottom and uncommenting it only when needed, you can very effectively amortize the cost of using JavaScript across the runtime of your application.



are likely to be limited to four simultaneous connections. A second DNS lookup is probably preferable to hard-coding the IP address in your application. This would cause all content to download every time your servers moved and would make it impossible to make good use of any CDN (content delivery network) you may be using to serve data. On balance, the recommendation here is to stick with a single hostname for serving all resources, as the benefit from multiple hostnames to get more parallel downloads is small relative to the implementation complexity, and actual speed gains are unlikely to be significant over poor networks.

Minify JavaScript and CSS. Minification is beneficial for mobile, though the biggest effect is seen in comparing uncompressed sizes. If minification poses a challenge, be sure to compare the sizes only after applying gzip to get a realistic sense of the gains. They may be below 5% and not worth the extra serving and debugging complexity.

Which Recommendations Should Be Used Only Occasionally?

Some recommendations depend on the context of the particular project and should be used only on certain sites.

Use Cookie-Free Domains for Sub-components. Using cookie-free domains applies to mobile, except that all static content should already be served by the app cache, so the effect is not as strong. If you are not using app cache, then follow this recommendation.

Avoid Empty Image SRC. If you need to create image tags that contain a dummy image, then you can use a data URL to encode a small image in place of an empty string until you can replace the image source.

Keep Components Under 25KB. The 25KB restriction does not apply to resources in the app cache but does apply to everything else. If older devices are important to your site, you may still want to respect this recommendation. If you are designing primarily for future devices, however, only HTML pages still need to be lightweight: the rest of your resources will stay cached in memory until the user restarts the iOS device or forcibly exits the process; and on Android, your components will stay in persistent cache until they expire.

Which Recommendations Should Be Ignored?


Some of the guidelines for desktop development do not apply when building mobile sites.

Do Not Make JavaScript and CSS External. This is one recommendation that cannot really be followed on mobile. Because of a healthy distrust of the mobile device's browser cache policy, it will almost always turn out best if you cache your JavaScript and CSS manually; thus, inlining it all into the main page will deliver the best results in terms of speed. For maintenance reasons, or perhaps if a site has seldom-visited areas where it would be better not to download until first use, you can use XHRs to fetch the JavaScript or CSS and the HTML5 database to store the resource for later reuse.


Redirects are a source of round trips and are to be avoided at all costs. Regularly monitoring the response codes from your servers should make it possible to catch redirect mistakes, which are commonly caused by poor URL choices, authentication choices, ads, or external components. It is better to update the site's script and resources and let app cache download and cache all the new locations than to deliver even one redirect return code per user visit. Redirects can also cause unexpected effects with app cache, causing the browser to download the same resources multiple times and store them in the database. This is easiest to test for on the desktop where the `sqlite3` command-line tool can be used to look directly at the app-cache database and see what is stored there. On Chrome, an even easier method is to use `chrome://appcache-internals` to inspect the state of app cache.

YSlow also recommends avoiding duplicate script. Certainly after all the effort made to minify, obfuscate, and manually cache your script tags, it will be worth ensuring the browser is not parsing and evaluating the same piece of script twice. Removing duplicate scripts remains a solid piece of advice for mobile sites.

Do Not Configure ETags. In the case of entity tags (ETags), the removal advice is the bit that applies. As you are already storing all the resources in app cache, and app cache is separate per host domain, there is no benefit to us-



Redirects are a source of round trips and are to be avoided at all costs. Regularly monitoring the response codes from your servers should make it possible to catch redirect mistakes.



ing ETags—and the mobile browser cache cannot be trusted to retain the components, anyway.

Do Not Make AJAX Cacheable. Rather than trusting the browser to cache AJAX responses, it is best to build either a full-blown write-through cache layer or an XHR caching layer on top of the HTML5 local storage facilities.

Do Not Split Components Across Domains. I have already discussed parallel downloads, and the opportunity for leveraging high-bandwidth connections on mobile is too limited to make this a core technique.

Reducing cookie sizes applies to mobile just as it does to the desktop. Specifically, be sure the cookies for a page do not, in aggregate, cause any requests to be split across multiple packets. Aim for 1,000 bytes or less of cookie data.

Do Not Choose `<link>` over `@Import`. Despite the existing recommendation stated in the title here, for absolute speed on mobile, neither `link` nor `@import` is appropriate. Instead, inline styles in the main body of the page create the fastest load time and the most app-like Web experience. If separate resources are a must to simplify serving, the preference should be `<link>` to avoid rendering content with missing style information.

Do Not Pack Components into a Multipart Document. As mobile devices do not support multipart documents, app cache is required.

What Still Applies?

With these extensive modifications to the guidelines, the YSlow and Page Speed recommendations may seem to have very little application to mobile Web development. Quite a few recommendations can still be used effectively for mobile, however, depending on your exact requirements. Here is a list of recommendations from YSlow that can still be useful for mobile sites:

- ▶ Use a CDN
- ▶ Add an expires or cache-control header
- ▶ Put style sheets at the top
- ▶ Avoid redirects
- ▶ Remove duplicate scripts
- ▶ Flush the buffer early
- ▶ Use GET for AJAX requests
- ▶ Postload components

- ▶ Reduce the number of DOM elements
- ▶ Minimize the number of iframes
- ▶ No 404s
- ▶ Reduce cookie size
- ▶ Optimize CSS sprites
- ▶ Do not scale images in HTML
- ▶ Make favicon.ico small and cacheable

Similarly, Google's PageSpeed recommendations, except those that have been specifically modified in this article, can apply to mobile. These tools from Yahoo! and Google should be the first line of defense against a slow site.

Putting It All Together

Let's spend a moment on how to measure performance. One technique already referenced is using adb to connect to Chrome running on your mobile device. This enables you to apply the full set of Web development tools in Chrome to the instance running on your phone. For interactive debugging, this cannot be beat. It is worth spending some time to understand the Network, Timeline, Profile, and Resources screen so you can master debugging and optimizing using Chrome. Since Chrome and Mobile Safari are very similar rendering engines, this will pay dividends on both major mobile platforms.

For the long run, do not forget to implement server-side statistics. Using JavaScript timestamps or `window.performance`, you can measure and track all the key load-time latency metrics so you know when there is a regression and in which part of the system it is. Frequently authentication, domain name changes, or third-party components wind up introducing additional redirects or network traffic that are not noticeable on good connections but that show up for your end users' real-world networks.

For AJAX, it is generally valuable to build a generic AJAX fetch layer that knows how to cache AJAX requests persistently. For example, a simple technique for a news or discussion board site would be to use unique hashes for each piece of content and a generic AJAX fetch layer that records all incoming data in an LRU (least recently used) cache in local storage. This general pattern of unique URLs and a sim-

ple-minded cache scheme will make dynamic content caching easy to understand and easy to implement and reuse across multiple sites.

For the final bit of polish, it is worth thinking hard about using CSS tricks to prerender layers of your site and transition between them. By rendering dialogs, for example, at a low opacity and fading or sliding them in as needed, you can create application-like experiences that run smoothly in response to user actions.

For the ultimate fit and finish, the site should never let the browser paint elements in an order of its choosing, but instead use layers of divs that are revealed in the right order and only when completely rendered. For transition durations, values of 150ms–300ms strike a nice balance between a snappy transition and a slick looking one—slower transitions look better but cost too much time, and faster ones look choppy.

A final note to consider is whether to use any application meta tags to give your website a special icon on the desktop, a splash screen, or to load full screen. For some sites these little touches add significantly to the fit and finish. The main drawback of these methods is that on iOS the Web view used for full-screen mode does not take advantage of the JavaScript JIT (just-in-time) compiler in Webkit, so JavaScript compute-intensive code runs slower. This will almost never affect the site, but you will need to test for it. In addition, the cookies are separate from the browser cookies, which could mean authenticating multiple times for different bookmarks.


On iOS, Mobile Safari gets special treatment and can start up significantly faster than a full-screen bookmark. Given these considerations, if you want to use bookmarking, there is sample code that provides a nice example of how to show users a different screen on first load, encouraging them to bookmark your site to their home screens.

Conclusion

With diligent attention to existing desktop recommendations and a few additions that take into account mobile network and CPU speed challenges, it is possible to create very fast, very slick website experiences for users. If

you cannot follow all the recommendations, this article has presented the most valuable techniques first—namely, HTML5 app cache and deferred JavaScript techniques. If you can follow every recommendation, then your users can look forward to a fast mobile experience that they will come back to again and again.

Acknowledgments

Almost every idea in this article is not mine, but rather the ideas of many Google teammates working on mobile websites over the years. They are Matthew Bolohan, Bikin Chiu, Ryan Fioravanti, Andrew Grieve, Alex Kennberg, Pavel Kobayakov, Robert Kroeger, Peter Mayo, Joanne McKinley, Derek Phillips, Keith Stanger, Neil Thomas, and Ilia Tulchinsky. 

Related articles on queue.acm.org

Streams and Standards: Delivering Mobile Video

Tom Gerstel

<http://queue.acm.org/detail.cfm?id=1066067>

Mobile Media: Making It a Reality

Fred Kitson

<http://queue.acm.org/detail.cfm?id=1066066>

Mobile Devices in the Enterprise: CTO Roundtable Overview

Mache Creeger

<http://queue.acm.org/detail.cfm?id=2019556>

References

1. Cisco. Cisco Visual Networking Index: global mobile data traffic forecast update, 2012–2017 (2013); http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html.
2. Fioravanti, R. Building high-performance mobile Web applications; <http://rjf-io2012.appspot.com/#5>; video: http://www.youtube.com/watch?v=jD_-r6y558o.
3. Google Code Blog. Gmail for Mobile HTML5 Series: reducing startup latency; <http://googlecode.blogspot.ca/2009/09/gmail-for-mobile-html5-series-reducing.html>.
4. Google Developers. Minimize round-trip times (2012); <https://developers.google.com/speed/docs/best-practices/rtt>.
5. Google Developers. Web performance best practices (2012); https://developers.google.com/speed/docs/best-practices/rules_intro.
6. Grove, R. Mobile browser cache limits, revisited; <http://www.yuiblog.com/blog/2010/07/12/mobile-browser-cache-limits-revisited/>.
7. Souders, S. Mobile cache file sizes; <http://www.stevesouders.com/blog/2010/07/12/mobile-cache-file-sizes/>.
8. Yahoo! Developer Network. Best practices for speeding up your Web site (2013); <http://developer.yahoo.com/performance/rules.html>.

Alex Nicolaou is Chrome Engineering Manager at the Google office in Waterloo, Ontario where he works on building ChromeOS for ARM platforms. Prior to joining Google in 2006, he was president of aruna.ca Inc., a startup developing RDBMS based on text-search algorithms and data structures and was part of LiquiMedia Inc.

© 2013 ACM 0001-0782/13/08