Making the Mobile Web Faster

Mobile performance issues? Fix the back end, not just the client.

Kate Matsudaira

Mobile clients have been on the rise and will only continue to grow. This means that if you are serving clients over the Internet, you cannot ignore the customer experience on a mobile device.

There are many informative articles on mobile performance, and just as many on general API design, but you'll find few discussing the design considerations needed to optimize the back-end systems for mobile clients. Whether you have an app, mobile Web site, or both, it is likely that these clients are consuming APIs from your back-end systems.

Certainly, optimizing the on-mobile performance of the application is critical, but software engineers can do a lot to ensure that mobile clients are remotely served both data and application resources reliably and efficiently.

MOBILE IS DIFFERENT

What is so special about mobile? If you were to go back in time and use the Internet, you would notice that most Web sites felt slower. The technology has now evolved to the point that clients can efficiently use and negotiate low-bandwidth channels. Mobile clients, however, don't have the computer power, storage, and high-bandwidth connections of desktops, so mobile needs to be thought about a little differently.

Here are some of the special considerations to take into account when building mobile-based applications:

- **Limited screen size.** There is less space for data and images.
- **Smaller number of simultaneous connections.** This one is important because, unlike Web browsers that can run many concurrent asynchronous requests, mobile browsers have a limited number of connections per domain at any given moment.
- **Slower network.** Network performance is heavily affected by poor signal reception and multiple cellular handovers (even though some clients are on Wi-Fi, some networks are congested and can require additional lookups if a user changes cell towers).
- **Slower processing power.** Extensive client-side computations, 3D graphics rendering, and heavy JavaScript usage can greatly affect performance.
- **Smaller caches.** Mobile clients are generally memory-restricted so it is best not to rely heavily on cached content for performance.
- **"Special" browsers.** In many ways the mobile browser ecosystem is reminiscent of the fragmented desktop browser scene of several years ago, with mobile vendors producing versions with fatal deficiencies and incompatibilities.

Although there are many ways to tackle these unique obstacles, this article focuses on what can be done from an API or back-end service to improve the performance (or the user's perception thereof) of mobile clients. The article is divided into two parts:

- Minimizing network connections and the need to transmit data—efficient media handling, effective caching, and employing longer data-oriented operations with fewer connections.
- Sending the “right” data across the network—designing APIs to return only the data that is needed/requested, and optimizing for the various types of mobile devices.

Although this article focuses on mobile, many of the lessons and ideas can be applied to other API client forms as well.

MINIMIZING CONNECTIONS AND DATA ACROSS THE NETWORK

Minimizing the number of HTTP requests required to render a Web page is undoubtedly one of the best ways of improving mobile performance. There are many ways to do this, but the exact approach may depend on your data and the architecture of your application.

In most cases you want to minimize how much information is sent across the network. Rendering on the server has its advantages (such as when the server sends back whole HTML pages) since it requires less compute and processing resources than doing so on the client. Of course, the downside of this approach is that the more code rendered server-side, the more likely that code will have display issues in client browsers (and dealing with browser compatibility is seldom fun). Still, the more that can be done on the client, the fewer trips across the network. After all, that is why “apps” have become so popular—if you could do everything in the Web browser with the network, this would be a mobile Web site world.

MINIMIZE IMAGE REQUESTS

In a standard browser, making a single request for each image on the page improves speed and allows you to take advantage of caching for each image. The browser is able to execute each request quickly and in parallel, so there isn’t a big performance hit for making many requests (and with the caching benefits there can even be performance gains). This approach, however, can be a killer on mobile.

Every request for data on a mobile device can require substantially more overhead, which can add significant latency to each request. Therefore, minimizing image requests can reduce the number of requests and in some cases the amount of data that needs to be sent (which can also help mobile performance).

Here are some strategies to consider:

Use image sprites. The use of image sprites can reduce the number of individual images that need to be downloaded from the server, but sprites can be cumbersome to maintain and difficult to generate in some circumstances (such as on product search results where you are showing thumbnail images for many products).

Use CSS instead of images. Avoiding images where possible and using CSS (Cascading Style Sheets) rendering for shadows, gradients, and other effects can reduce the number of bytes that need to be transmitted and downloaded.

Support responsive images. A popular way of delivering the right image to the right device is using responsive images. Apple does this by loading regular images and then replacing them with high-resolution ones using JavaScript.⁷ There are several other ways³ of approaching this problem, but the issue is far from solved.¹²

In these cases you should make sure that the server-side support and APIs are able to support different versions of the same image, and the exact way to do that will depend on the approach of

the clients. For example, one easy way of doing this with an API is to support a handful of image sizes as a parameter for the request, as shown here:

Request:

`http://yourdomain.net/api/objects.json?objectIds=18369542&imageSize=IMG_140x140`

Response:

```
objects: [
  { product: {
    id: "18369542",
    title: "Upright Freezer",
    brand: "Frigidaire",
    imageURL: "https://yourdomain.net/140x140/18369542-140x140.jpg",
  }
},
  { product: {
    id: "14958145",
    title: "Sony Bravia 32-inch LCD",
    brand: "Sony",
    imageURL: "https://yourdomain.net/140x140/14958145-140x140.jpg",
  }
}
]
```

These are the size options used in my last project:

```
{'IMG_ORIGINAL'|'IMG_70x70'|'IMG_80x80'|'IMG_85x85'|'IMG_90x90'|'IMG_100x100'|'IMG_140x140'|'IMG_160x160'|'IMG_170x170'|'IMG_180x180'|'IMG_200x200'|'IMG_312x312'}
```

To keep APIs simple, make this parameter optional and send back a default size. To pick your default size, select either the smallest size (to handle situations such as responsive images) or the most commonly used size on your Web site.

Use Data URIs for images inline to minimize extra requests. An alternative to sprites is to use data URIs (uniform resource identifiers) to embed images inline within the HTML itself. This makes the images part of the overall page, and while the URI-encoded images can be larger in terms of bytes, they compress better with gzip compression, which helps minimize the effect of transmitting additional data.

If using URIs, then make sure to:

- Resize images to the appropriate size before encoding them into the URI payload.
- Gzip responses (to take advantage of compression).
- Note that URI-encoded images are part of the CSS of the page. As a result, caching of individual images is more difficult, so don't use this approach if there are good reasons to cache the image locally (e.g., it is reused frequently on several pages).

LEVERAGE LOCALSTORAGE AND CACHING

Since mobile networks can be slow, HTML, CSS, and images can be stored in localStorage to make

the mobile experience faster. (There is a great case study on Bing's improvements using `localStorage` for mobile to reduce the size of an HTML document from about 200 KB to about 30 KB.¹¹)

Pulling data out of local storage can negatively impact performance,¹³ but the effect is typically much less than the latency incurred going across the network. In addition to `localStorage`, some apps are using other features in HTML5,⁶ such as `appCache`,¹ to improve performance and startup time.

One optimization that can be leveraged on the server involves being aware of what is on the device. By embedding CSS and JavaScript directly within a single Web request, then storing a reference to those files on the client, it is possible to track what has been downloaded and resides in the cache. Then, the next time the client makes a request to the server, it can pass the references to its cached files to the server via a cookie. The server then only has to send new files over the network, which prevents the client from downloading those assets again.

This trick to leverage local caching can save a lot of time. (For more details on how to embed directly and then reference these files, as well as other resources for more reading on the topic, see Mark Pilgrim's *Dive into HTML5*.⁸)

PREFETCH AND CACHE DATA

One great way to improve perceived performance is by prefetching data that will be used throughout the mobile experience so it can be loaded directly on the device without additional requests—for example, paginated results, popular queries, and user data. Thinking about these use cases and factoring them into your API design will allow you to create APIs designed for prefetching and caching data before the user interacts with it, increasing the perceived responsiveness.

If your client is an app, then for data that is not likely to change between updates (such as categories or main navigation) consider shipping the data inside the app so it never requires a trip across the network.

If you want to get sophisticated, ship the data inside the app but also create a versioning and expiration scheme; that way, the app can ping the server in the background and update the data only if the version on the device is out of date.

Ideally, you want to transfer data when needed by the client and preload data when advantageous to do so (i.e., when the network or other required resources are not in use). Therefore, if an end user will not view the image or content, don't send it (this is particularly important for responsive sites, since some just “hide” elements). Design your APIs to be flexible and support sending smaller payloads to the client.

A great use case for prefetching images is a gallery of image results, such as a list of products on an e-commerce site. In these situations it is worth downloading the previous and next image(s) to speed up interactions and browsing. Be careful, however, not to go overboard and fetch too far ahead; otherwise, you could end up requesting data that may not be seen by the user.

USE NONBLOCKING I/O

With client optimizations, developers know to watch out for blocking JavaScript execution,¹⁴ which can have a big impact on the perception of performance. This is even more important for APIs. If there is a longer API call, such as one that could rely on a third party and might time out, it is important to implement this call as nonblocking (or even long-waiting) and instead choose a polling or triggering model:

- **Polling API** (pull-based model). The client makes a request and then periodically checks for the results of that request, periodically backing off if required.
- **Triggering API** (push-based model). The call makes the request and then listens for a response from the server. The server is provided with a call back so it can trigger an event letting the caller know the results are available.

Triggering APIs are typically harder to implement, as connections on mobile clients are unreliable. Therefore, polling is a much better option in most cases.

For example, in Decide.com's mobile app,⁴ each product page shows availability and pricing at stores close to a user's location. Since a third party delivers those results, the developers did not want the local pricing to take as long as the partner's API did to deliver results to the client. To work around this, Decide.com created its own wrapper API that allows users to pass a flag for any product query (a set of APIs support retrieving product data in various ways) that would signal the server to retrieve local prices for that product. Those prices would be stored in the server's cache. Then, in the event the user wanted the local pricing for the product, those prices would have a higher probability of being in the cache and wouldn't incur the longer wait times from the third-party partner.

This method is a lot like prefetching on the client but is instead done on the server side with APIs and data. Here are sample requests to show how this works:

Request for product:

`http://example.com/api/products.json?productId=18369542&local=true`

Response:

```
{“product”: {  
  “id”: “18369542”,  
  “title”: “Upright Freezer”,  
  “brand”: “Frigidaire”  
}}
```

Request for prices:

`http://example.com/api/prices.json?productId=18369542`

As shown in figure 1, this call looks in the cache first, and if the cache doesn't contain prices for that product, it calls the third-party API and waits.

In general, you want to make sure that APIs return quickly and don't block while waiting for results, since mobile clients have a limited number of connections. In cases where some components are significantly slower than others on the server side, it can be worth breaking the API into separate calls, using typical response time as a factor. That way the client can start rendering pages from the initial fast response calls while waiting for the slower ones. The goal is to minimize the time-to-text rendering on the screen.

You should avoid chatty APIs, and in slow network situations it is important to avoid excess API calls. A good rule of thumb is to have all the data needed to render a page returned in a single API call.

AVOID REDIRECTS AND MINIMIZE DNS LOOKUPS

When it comes to requests, redirects can harm performance, especially if they cross domains and require a DNS lookup.

FIGURE 1
An API Call

For example, many sites handle their mobile sites using client-side redirects. For example, when a mobile client goes to the main site URL (e.g., <http://katemats.com>), it redirects the client to the mobile site (<http://m.katemats.com>). This is especially common when the sites are built on different technology stacks. Here is an example of how this works:

1. A user Googles “yahoo” and clicks on the first link in the results.
2. Google captures the click using its own tracking URL and then redirects the phone to <http://www.yahoo.com> [redirect].
3. Google’s redirect response goes through the cell tower and then back to the phone.
4. Then there is a DNS lookup for www.yahoo.com.
5. The IP resulting from the DNS lookup is sent through the cell tower and back to the phone.
6. When the phone hits <http://www.yahoo.com>, it is recognized as a mobile client and is redirected to <http://m.yahoo.com> [redirect].
7. The phone then has to do another DNS lookup for that subdomain (<http://m.yahoo.com>).
8. The IP resulting from the DNS lookup is sent through the cell tower and back to the phone.
9. The resulting HTML and assets are finally sent back through the cell tower and then to the phone.
10. Some of the images on pages of the mobile site are served via a CDN (content delivery network), referencing yet another domain, <http://l2.yimg.com>.
11. The phone then has to do another DNS lookup for that subdomain, <http://l2.yimg.com>.
12. The IP resulting from the DNS lookup is sent through the cell tower and back to the phone.
13. The images are rendered, completing the page.

As is obvious from this example, a lot of overhead is involved in these requests. They can be avoided by using redirects on the server side (routing via the server and keeping DNS lookups and redirects to a minimum on the client) or by using responsive techniques.² If DNS lookups are unavoidable, try using DNS prefetching for known domains to save time.

USE HTTP PIPELINING AND SPDY

Another useful technique is HTTP pipelining, which allows combining multiple requests. If I were to implement an optimization translation layer, however, I would opt for SPDY, which essentially optimizes HTTP requests to make them much more efficient. SPDY is getting traction in places such as Amazon’s Kindle browser, Twitter, and Google.

SENDING THE “RIGHT” DATA

Depending on the client, the experience may require different files, CSS, JavaScript, or even a different number of results. Creating APIs in a way that supports different permutations and versions of results and files provides the most flexibility for creating amazing client experiences.

USE LIMIT AND OFFSET TO GET RESULTS

As with regular APIs, fetching results using `limit` and `offset` allows clients to request ranges of the data that make sense for the client’s use case (thus, fewer results for mobile). The `limit` and `offset` notation is more common (than, say, `start` and `next`), well understood in most databases, and therefore easy to build on:

```
/products?limit=25&offset=75
```

You should choose a default that caters to either the lowest or highest common denominator, depending on which clients are more important to your business: smaller if mobile clients are your biggest users; bigger if users are likely to be on their desktops, such as a B2B Web site or service.

SUPPORT PARTIAL RESPONSE AND PARTIAL UPDATE

Design your APIs to allow clients to request just the information they need. This means that APIs should support a set of fields, instead of returning the full resource representation each time. By avoiding the need for clients to collect and parse unnecessary data, you can simplify the requests and improve performance.

Partial update allows clients to do the same thing with data they are writing to the API (thereby avoiding the need to specify all elements within the resource taxonomy).

Google supports partial response by adding optional fields in a comma-delimited list as follows:

```
http://www.google.com/calendar/feeds/zachpm@google.com/private/full?fields=entry(title,gd:when)
```

For each call, specifying `entry` indicates that the caller is requesting only a partial set of fields.

AVOID OR MINIMIZE COOKIES

Every time a client sends a request to the domain, it will include all the cookies that it has from that domain—even duplicated entries or extraneous values. This means that keeping cookies small is another way to keep payloads down and performance up. Don’t use or require cookies unless necessary. Serve static content that doesn’t require permissions from a cookieless domain, such as images from a static domain or CDN. (The Google Developers site provides some best practices for cookies and performance.⁵)

ESTABLISH DEVICE PROFILES FOR APIs

With the many different screen sizes and resolutions on desktops, tablets, and mobile phones, it is helpful to establish a set of profiles you plan to support. For each profile you can deliver different images, data, and files so they suit each device; you can do this using media queries on the client.¹⁰

If each profile is tailored to a device, then it has the opportunity to offer a better user experience. Each different function and scenario supported by each profile, however, makes it more difficult to

maintain (since devices are constantly changing and evolving). As a result, the smartest approach is to support only as many profiles as absolutely necessary for your particular business. (The mobiForge Web site provides more information on some of the tradeoffs and options for creating great experiences on different devices.⁹)

For most applications three profiles will be sufficient:

- Mobile phone—smaller images, touch enabled, and low bandwidth.
- Tablet—larger images designed for lower bandwidth, touch enabled, more data per request.
- Desktop—larger, high-resolution images designed for desktop browsers or tablets with high resolution and Wi-Fi.

Selecting the right profile can be handled by the client, which means that on the server side, APIs just need to support this configuration. You should design APIs to take these profiles as input, or parameters, and send different information based on the device making the request. Depending on the application, this may mean sending smaller images, fewer results, or inline CSS and JavaScript.

For example, if one of your APIs returns search results to the client, each profile might behave differently as follows:

```
/products?limit=25&offset=0
```

This would use the default profile (desktop) and serve up the standard page, making a request for each image so subsequent product views could be loaded from cache.

```
/products?profile=mobile&limit=10&offset=0
```

This would return 10 product results and use the low-resolution images encoded as URIs with the same HTTP request.

```
/products?profile=tablet&limit=20&offset=0
```

This would return 20 product results using the larger-size low-resolution images encoded as URIs with the same HTTP request.

You can even create special profiles for devices such as feature phones.

Unlike smartphones, feature phones can cache files on only a per-page basis, so it is better to send CSS and JavaScript with each request for these clients. Using profiles is an easy way to support that functionality on the server side.

You should use profiles instead of partial responses when the response from the server is drastically different for each profile—for example, if the response has inline URI images and compact layout in one case but not the other. Of course, profiles could be specified using a “partial response,” although typically partial responses are used to specify a part (or portion) of a standard schema (such as a subset of a larger taxonomy), not a whole different set of data, format, etc.

CONCLUSION

There are many ways to make the Web faster, including mobile. This article is meant to be a useful reference for API developers who are designing the back-end systems that support mobile clients—and to this end, ultimately enabling and preserving a positive mobile-application user experience.

REFERENCES

1. Bidelman, E. 2011. A beginner's guide to using the application cache. HTML5 Rocks; <http://www.html5rocks.com/en/tutorials/appcache/beginner/>.
2. Breheny, R., Jung, E., Zürcher, M. 2012. Responsive design—harnessing the power of media queries. Google Webmaster Central Blog; <http://googlewebmastercentral.blogspot.com/2012/04/responsive-design-harnessing-power-of.html>.
3. Coyier, C. 2012. Which responsive images solution should you use? CS-tricks; <http://css-tricks.com/which-responsive-images-solution-should-you-use/>.
4. Decide.com. <https://www.decide.com/>.
5. Google Developers. 2012. Make the Web faster; <https://developers.google.com/speed/docs/best-practices/request>.
6. Graham, A. 2010. Google APIs + HTML5 = a new era of mobile apps. Google code; <http://googlecode.blogspot.com/2010/04/google-apis-html5-new-era-of-mobile.html>.
7. Grigsby, J. 2012. How Apple.com will serve retina images to new iPads. Cloud Four Blog; <http://blog.cloudfour.com/how-apple-com-will-serve-retina-images-to-new-ipads/>.
8. Pilgrim, M. 2011. The past, present and future of local storage for Web applications. In *Dive into HTML5*; <http://diveintohtml5.info/storage.html>.
9. Rieger, B. 2009. Effective design for multiple screen sizes. mobiForge; <http://mobiforge.com/designing/story/effective-design-multiple-screen-sizes>.
10. Smus, B. 2012. A nonresponsive approach to building cross-device Webapps; <http://www.html5rocks.com/en/mobile/cross-device/>.
11. Souders, S. 2011. Storaiger case study: Bing, Google; <http://www.stevesouders.com/blog/2011/03/28/storager-case-study-bing-google/>.
12. W3C Responsive Images Community Group; <http://www.w3.org/community/respimg/>.
13. Zakas, N. C. 2011. localStorage read performance. Performance Calendar; <http://calendar.perfplanet.com/2011/localstorage-read-performance/>.
14. Zakas, N. C. 2010. What is a nonblocking script? NCZonline; <http://www.nczonline.net/blog/2010/08/10/what-is-a-non-blocking-script/>.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

KATE MATSUDAIRA is an experienced software engineer and has spent the past seven years immersed in the startup world as an architect or CTO. Prior to that she spent time as a software engineer and technical lead/manager at Amazon and Microsoft. She has a passion for mobile and a lot of experience in building large-scale distributed Web systems, big data, cloud computing, and engineering leadership. She maintains a blog at <http://katemats.com>.