Server Side Techniques for Client Side Optimization

Jason Huck, CTO, Core Five Creative

Overview

Web-based applications operate on the venerable client-server model, where the server does the bulk of the heavy lifting, and the client simply makes requests to the server and displays the results. Since most, if not all, of the business logic for an application is expressed in server-side processes, it's not uncommon for optimization efforts to focus on inefficiencies in the middleware, database, and other underlying layers of the server. After all, that's where things are the most complex.

Historically, client-side optimization has amounted to little more than media compression (static images, audio, and video). However, as web browsers have become more sophisticated, and client side scripting has matured, more of the processing burden has shifted to the client side, and thus the amount of data and processing instructions required to fulfill a single page request has grown tremendously. This paper examines the need for optimization of the client side components of a modern web-based application, and explores techniques which can be implemented on the server side to achieve that optimization.

Why Client Side Optimization Is Important

Client-side optimization is important for a number of reasons. First, we as developers must not forget that dial-up is still the prevalent means of accessing the internet in many parts of the world. In 2006, only about 17% of rural U.S. households had broadband access. Secondly, even as broadband availability improves, traditional desktop browsing is forced to share the stage with an increasing number of mobile devices — including the iPhone — which sometimes connect via slower networks such as EDGE, requiring cacheable components to be 25k or less. Finally, even though web apps today may be significantly more complex than they were a couple years ago, users certainly haven't grown more patient, and although your bandwidth may be cheaper, it certainly isn't free. So, if your app consumes less bandwidth and delivers its payloads quicker and more efficiently, everybody wins.

At the forefront of the latest research into client-side optimization is Yahoo!'s Exceptional Performance Team, who published a collection of 13 *Rules for Making Web Sites Fast*, which quickly became the seminal document on the subject. That document has since blossomed into a set of 34 best practices across 7 distinct categories.

This paper explores how Lasso can assist developers in implementing and automating these guidelines, focusing primarily on the original 13 rules. It presents an integrated system — adaptable to any framework or codebase — which will manage the optimization, compression, caching, and delivery of page components.

The Evolution of the Web Page

To fully appreciate the importance of optimization, it's helpful to review the path by which modern web applications became so complex in the first place.

HTML, as originally proposed by Tim Berners-Lee in the early 90's, was extremely basic compared to the options we have today. The first web pages were little more than plain text, with a few simple indications of paragraph breaks and headings, and of course the all-important hyperlinks. The purpose of HTML was to provide structure and hierarchy to a plain text document. The visual presentation (what little there was) was handled entirely by the user agent — and initially there was only one in widespread use: NCSA Mosaic. With only one browser, 22 HTML tags, and no scripts or stylesheets to interpret, early web authors didn't have to worry about compatibility or standards compliance. The only assets one could embed in a page were images. If a page didn't display correctly, it was almost certainly because the author had made a simple mistake.

However, that simplicity didn't last long. As the popularity of the world wide web as a publishing platform grew (thanks to faster modems, and tools like Adobe PageMill and Claris Homepage, which made it easier to author pages) people demanded more control over the presentation of their content. Netscape introduced JavaScript in 1995, and CSS 1.0 became an official recommendation in 1996, though neither would see widespread use for several more years. With the first browser wars well under way, proprietary extensions to HTML began to appear with each new release. Attempts to embed different types of media, each with several competing formats, led to a plethora of incompatible plugins. It was around this time that things started getting really ugly.

Tables, one of the very first additions to HTML, were intended to organize grids of tabular data, never for arranging columns in a page layout. But in the first few years of the 21st century, with CSS still in its infancy, there was little choice. Graphic HTML editors routinely created overly complex layout tables littered with `font` tags and the occasional, primitive inline style. Rudimentary JavaScript was plagued with cross-browser compatibility problems, but nonetheless injected directly into pages with inline event handlers. Consistent rendering across browsers was so difficult that authors began to target just one. The common term for this patchwork mess was *DHTML*.

It took quite a few years to sort everything out. Web development as a trade had to establish itself, and the community had to demand better standards, along with better compliance from browser vendors. Some proprietary concepts became standards (or de-facto standards, such as Flash), while others (`blink`, `marquee`, you know who you are) thankfully fell by the wayside. Since about 2005 or so, *most* web browsers (though not necessarily the most *widely used* ones, unfortunately) have been relatively standards-compliant, such that content authored primarily for one of them would render reasonably well in the others with only minor changes required for full compatibility.

Today, despite a much higher degree of complexity overall, we are seeing an increasing trend back towards the clean, valid, semantically correct markup of the original web, driven by three separate but highly sympathetic motivating forces: standards advocacy (the vision of the semantic web and emerging standards such as microformats), accessibility (ADA and Section 508 compliance), and of course search engine optimization. This trend is supported by an increasingly competitive array of A-Grade browsers, as well as the need to repurpose content for entire new categories of user agents, including mobile browsers, console browsers, and RIA's.

At the same time, DHTML has matured (and we don't call it that any more). AJAX and JavaScript effects are now the norm, and popular JavaScript libraries have, for the most part, solved the browser compatibility problems for us. CSS-based layouts have supplanted table-based layouts for precision and efficiency. And thanks to unobtrusive binding techniques, both scripts and styles can now be completely externalized, just like any other type of asset, cleanly separating the content (markup), presentation (CSS), and behavior (JavaScript) layers.

Challenges of Modern Web Design

Of course, managing all of the components of a modern web application is not without its unique challenges. Even as our markup is getting lighter, our scripts and styles are getting heavier. As projects increase in size and complexity, stylesheets quickly grow long and unwieldy, and can easily become the largest components of a page. It becomes difficult to keep them organized in a consistent fashion.

JavaScript libraries such as jQuery, MooTools, and Script.aculo.us have adopted a modular approach so that developers can include only the parts they need. But in dynamic and decentralized systems where multiple teams are working on the same project, it may be difficult to know whether a given module's dependencies will already be available at the time your component is loaded. Thus, unfortunately, it's not uncommon for core scripts and styles to be included multiple times for a single page request.

Faced with these complications, and in the rush to meet a looming deadline, it can be tempting to add just a little inline javascript or css into the markup. A little style block here, a little `document.write` there...no one will ever know. Except you (and hopefully it's you), a year later, when you're trying to re-skin the project and you suddenly realize why some parts of the page refuse to render in the new style.

We're faced with a dilemma: keeping all your site assets together in a single file, i.e. a single global stylesheet, is efficient to serve, but difficult to maintain. Splitting assets up into separate files improves maintainability at the expense of performance, since each file requires a separate HTTP request. Frameworks and libraries add to the complexity by introducing dependency trees into the mix.

Fortunately, we can use Lasso to help us strike a balance between maintainability and performance, allowing us to keep our applications fast and efficient, adhering to those aforementioned best practices, without abandoning a dynamic workflow for a build->publish cycle. We'll dissect the components of the system which allows us to do that later. But first, let's get specific about these so-called "best practices." What are they, exactly, and why do we care?

Best Practices (A Subset)

The original 13 best practices for client side optimization, as put forth by the Yahoo! Exceptional Performance Team, are as follows:

(1) Make Fewer HTTP Requests

Although this list is not strictly ordered, this rule is first for a reason: it's one of the most important things to consider when optimizing performance. Why? When a browser requests a page, after the page itself is downloaded, every reference to an asset in that page must be evaluated. It goes something like this:

> *Is it in the cache? No? What domain is it on? Okay. Do we have any cookies for that domain? Okay, load them up and fire off another request. Now let's evaluate the response (more cookies? mmm!) and make sure it's okay. All good? On to the next request!*

This process is repeated for every JavaScript, stylesheet, image, and/or other asset in every `script`, `link`, `anchor`, and `embed` tag in the page. This is often, easily, dozens and dozens of files. The request and response headers sent back and forth can add an extra 1600 or more bytes *per request*, costing bandwidth and time.

The simplest way to reduce the number of requests is to combine files. Put all your stylesheets into one big file. Same with all your scripts. You can even do it with your images, creating what are known as *CSS Sprites*, where CSS controls which portion of a single larger image is shown in different areas of the layout, making it appear as if each clipped area is displaying a completely separate image.

(2) Use A Content Delivery Network

A *Content Delivery Network*, or *CDN*, is a specialized hosting provider which only serves certain portions of your application. Specifically, the completely static portions such as images. Static assets can be served much faster from dedicated CDN's because the requirements for serving are so simple, and the domains from which they are served never set cookies or do anything to inflate the request and response headers used in the handshaking between the server and client.

Also, if you have many assets to serve, splitting the load up between multiple domains can actually work around restrictions in most browsers which limit the number of items that can be simultaneously retrieved from the same host. (The HTTP 1.1 specification recommends no more than two simultaneous downloads per host.) It doesn't take very many assets to realize the performance benefits of moving from sequential to parallel downloading.

Unfortunately, commercial CDN's can be costly, and their low-cost alternatives unreliable. However, we can realize most of the benefits of using a CDN by setting up *asset subdomains*, which are simply additional virtual hosts on your existing server set up to only serve plain, static, cookie-free content. Since the host name changes, browsers will still make parallel requests, even though everything is still being served from the same location.

(3) Add An Expires Header
When assets are sent to a browser, the browser will cache them unless specifically instructed otherwise (in fact, some items will be cached regardless). The web server can indicate within the response header how long a particular asset should be considered "fresh." Adding an Expires Header is important because without it, the browser has to check to see whether the item needs to be refreshed by asking the server. For the smallest assets, the cost of doing that can outweigh the cost of sending the asset itself over the wire.

(4) GZip Components
This one is pretty straightforward. Obviously a compressed asset requires less bandwidth to transfer than its uncompressed counterpart. Most browsers will accept GZip-compressed content, and most web servers will compress content before sending it. We just need to make sure the server is configured to do so.

(5) Put Stylesheets At The Top
Browsers will begin displaying pages as soon as they have enough information about the layout and appearance of the page to do so. This is called *progressive rendering* and can vastly improve the *perceived* performance of larger/longer pages. Since CSS files now contain the bulk of the layout and presentational information about a page, loading the CSS as early as possible, by putting it in the document's head, ensures the browser has all of the information it needs to begin progressive rendering as quickly as possible.

(6) Put Javascripts At The Bottom
JavaScript controls the behavioral layer of a web application. It manipulates the page after its initial internal representation (a.k.a. the *Document Object Model* or *DOM*) has been established. Thus, in the interest of progressive rendering, it's best to defer the loading of JavaScript until as late in the page as possible.

(7) Avoid CSS Expressions
This one's a no-brainer. CSS Expressions are non-standard and supported only in, you guessed it, Internet Explorer. Performance considerations aside, just don't use these. Period.

(8) Make Scripts and Styles External
Purely in terms of performance, the key to this rule is that externalized assets are separately cacheable. Thus, if you use the same assets across multiple pages, it's best to keep those assets external to the page. External files also help keep the content, presentation, and behavior layers distinctly separate, making it easier to repurpose content for other media.

(9) Reduce DNS Lookups
When retrieving a page, if the browser encounters a new hostname, it has to do a DNS lookup to resolve it in order to request whatever assets are being served from the new host. DNS lookups take time which could otherwise be eliminated, so it's best to keep the number of hostnames referenced by a single document as low as possible. This rule conflicts somewhat with the previous rule about using asset subdomains, so it's best to experiment with different combinations to see what provides the best results for your particular project.

(10) Minify Javascript (and CSS)
Minification is a process in which a given section of code is shortened as much as possible without altering its functionality. Long variable and function names are replaced with shorter ones (typically one character), and all extraneous white space, including tabs and newlines, is removed. The result is a file which is smaller (in the case of YUI Compressor, the minifier we'll be using later, 20% smaller on average) even before GZip compression is applied. Smaller files equates to faster load times. Minification usually targets JavaScript, but CSS files can be minified as well.

(11) Avoid Redirects

Any time a redirect is encountered, precious time is wasted spooling up an additional set of requests. Often they are unavoidable, but beyond the obvious META redirects and even Lasso's `redirect_url` tag, you'll want to make sure that links to directories on your site, whether real or virtual, end with a trailing slash, especially when using Apache. Without the trailing slash, Apache has to determine whether you are requesting a directory or an extensionless file, and then, upon discovering that you are indeed requesting a directory, it will redirect you to the same URL with a trailing slash appended. Quicker and simpler to skip all of that and just go straight to the version with the slash already appended.

(12) Remove Duplicate Scripts

When multiple developers are working on separate components of a larger project, they may not know whether some assets will already have been loaded at the time their particular components are included. To be safe, they may include these assets without checking to see if it's even necessary. As a result, these scripts and styles have to be retrieved and evaluated multiple times for a single page view, adding unnecessary overhead and potentially creating additional problems due to load order and cascading.

(13) Configure ETags

An *ETag* (the "E" stands for "Entity" and refers to an individual asset) is simply an additional header containing a checksum which can be used to verify whether a cached item is still "fresh" in the absence of other indications such as an Expires header. Because web servers haven't standardized on how to create ETags so that they are globally unique, and their functionality is essentially moot when Expires headers are used, this is one of the more controversial optimization tips. As long as you aren't running a server cluster, where a browser may attempt to validate an ETag against a different server than the one which created the tag, it won't hurt anything to use them, but take this one with a grain of salt.

An Asset Management System for Lasso-Powered Sites

I wanted to create a system that would automate as many of these optimization techniques as possible, while at the same time not require a drastic change in workflow, and I believe I have struck a reasonable balance, providing intuitive helper functions while still keeping things loosely coupled. This system provides support for rules 1, 2, 4, 5, 6, 8, 10, and 12, but can easily be disabled for debugging.

Getting Organized

Scripts and styles are needed at three distinct levels of a web site, going from general to specific:

- **Global**: These assets are applied to every page within the site.

- **Template or Page**: These assets apply only to a specific page or page type within the site.

- **Sub-Page Components or "Modules"**: These assets apply only to specific areas within a page, though they may be repeated on multiple pages within the site.

Based on this observation, I decided to designate a few special folders to aid in automation. The specific paths are configurable, and you can opt not to use this feature if it doesn't appeal to you, but this is what worked for me:

- **"Base" folders**: One for scripts, and one for styles. Anything placed here is included globally. For CSS, this might be your reset.css file, typography, or a grid framework. For JavaScript, this might be a base library like jQuery or MooTools. Items in base folders are loaded first, in alphabetical order.

- **“Cache” folders**: Again, one each for scripts and styles. This is where the asset management system will store the minified, compressed versions.

- Within my own projects, I also designate folders for **“templates”** and **“modules”**, corresponding to the levels described above, but I decided not to enforce that within the system for fear of making it too specific to my particular habits.

Adding Automation

The basic idea behind the system is simple. At the beginning of each page request, we create two unique arrays. One array is for scripts, and the other for stylesheets. As the page is processed, we insert the paths to the scripts and styles needed for each component at each level, along with the modification date of each file.

After everything else has been processed, we create a unique checksum of that information and look for matching cached files to serve. If matches are found, the appropriate HTML is inserted into the response: a `<link>` tag goes into the document `<head>` and a `<script>` tag goes right before the closing `</body>`.

If no matches are found, depending on what options have been configured, a number of things happen. First, all of the files are concatenated into a single file. Next, that file is optimized and minified using the YUI Compressor. The result is optionally further compressed using GZip compression (only if your web server doesn’t do that for you automatically, which is preferable), and the resulting concatenated, minified, compressed file is written to disk. Then, just as if it had already been there, the HTML response is updated to include a link to it.

All of this happens on the fly in a few seconds’ time. Once the cached versions are created, that time is reduced to a few milliseconds. Using this “lazy” caching technique, there is no need for a separate build process when deploying sites, and the caches are automatically updated whenever assets are added, edited, or removed. You can programmatically force a refresh of the caches, and if you encounter problems, a single flag tells the system not to combine or compress anything, and it will simply insert all of the `<link>` and `<script>` tags for each resource individually for easier debugging.

With this system in place, there is only one stylesheet and one script linked to each page, regardless of how many you started with, so it helps you reduce the number of HTTP requests (rule 1). These links are automatically inserted in the appropriate places in the HTML response: CSS at the top (rule 5) and JS at the bottom (rule 6). It requires external files to work, so by using it you’re forced to use external JS and CSS files (rule 8). With all options enabled, the resulting files are minified (rule 10) and compressed (rule 4). And, even if you insert the same asset into the system multiple times, it will only be included in the final result once, eliminating duplicate scripts (rule 12).

Using Asset Subdomains With Expires Headers

The system also includes primitive support for automating the use of asset subdomains via a series of string replacments performed after the page has been processed. You have the option of supplying a list of paths to asset folders within your web root, along with the subdomain that should be used for each one, as a pair of find/replace regular expressions. So, for example, if you keep all of your images in */assets/images/*, you can set up a subdomain like *images.mydomain.com* to point directly to that folder, and then tell the system to replace all references before serving the page. Thus, markup like this:

```
<img src=”/assets/images/logo.gif” />
```

...could become:

```
<img src=”http://images.mydomain.com/logo.gif” />
```

In Apache 2.x, the virtual host entry for the above example would be as follows:

```
# Asset Subdomain for Static Content
<VirtualHost *:80>
     DocumentRoot /Library/WebServer/Documents/mydomain.com/assets/images
     ServerName images.mydomain.com

        <Files ~ “^.*$”>
               Order allow,deny
               Deny from all
               Satisfy All
        </Files>
        <Files ~ “\.(gif|jpe?g|png)$”>
               Order allow,deny
               Allow from all
               Satisfy All
        </Files>

        # “Far Future” expires header for static content
        ExpiresActive On
        ExpiresDefault “now plus 3 days”

        # Disable ETags (optional)
        FileETag none
</VirtualHost>
```

Enabling GZip Compression

Although Lasso can compress files on the fly using Fletcher’s `[compress_gzip]` tag, it’s generally easier and more efficient to enable GZip compression within the web server itself. To enable GZip compression in Apache 2.x, edit your http.conf file by uncommenting the following line:

```
LoadModule deflate_module libexec/apache2/mod_deflate.so
```

...and add the following lines (thanks to Bil Corry for the browser-specific tweaks):

```
# mod_deflate
<Location />
    # Insert filter
    SetOutputFilter DEFLATE

    # Netscape 4.x has some problems...
    BrowserMatch ^Mozilla/4 gzip-only-text/html

    # Netscape 4.06-4.08 have some more problems
    BrowserMatch ^Mozilla/4\.0[678] no-gzip

    # MSIE masquerades as Netscape, but it is fine
    # BrowserMatch \bMSIE !no-gzip !gzip-only-text/html

    # NOTE: Due to a bug in mod_setenvif up to Apache 2.0.48
    # the above regex won’t work. You can use the following
    # workaround to get the desired effect:
    BrowserMatch \bMSI[E] !no-gzip !gzip-only-text/html

    # Don’t compress images
    SetEnvIfNoCase Request_URI \
    \.(?:gif|jpe?g|png)$ no-gzip dont-vary

    # Make sure proxies don’t deliver the wrong content
    Header append Vary User-Agent env=!dont-vary
</Location>
```

Using The System

The interface to the system is primarily through the `[asset_manager]` custom type, although that type is simply a wrapper for several lower-level tags with more specific functions. If you decide the way I've bundled everything in `[asset_manager]` is not for you, you can create your own workflow using the other tags.

Initializing The Asset Manager

Near the beginning of the code for a given page request, whether in a global include or just at the top of a page, you must call the `->options` member tag to initialize the system. All parameters are optional, so if the default values work for you, all you need to do is this:

```
[asset_manager->options]
```

However, most likely, you'll want to make some changes. The options are:

-usecache: boolean, default true, whether or not to do anything at all that results in cached files. Setting this to false turns off concatenation, minification, compression, and caching, and it just writes all the links to your scripts and styles out individually, though still in the correct places.

-minify: boolean, default true, whether or not to run the concatenated files through the YUI Compressor.

-compress: boolean, default false, whether or not to GZip compress the files. It's off by default because it's better to do this at the web server level.

-refresh: boolean, default false. Setting this to true will cause new files to be generated, even if the checksums haven't changed.

-paths: dictionary, the set of paths which specifies where everything goes. The specific paths are:
yui - The path to yuicompressor.jar. Default: /lib/tools/yuicompressor.jar.
scriptcache - Where scripts are cached. Default: /lib/scripts/cache/.
stylecache - Where styles are cached. Default: /lib/styles/cache/.
scriptbase - Where base scripts are stored. Default: /lib/scripts/base/.
stylebase - Where base styles are stored. Default: /lib/styles/base/.

-subdomains: an array of pairs which will be used in a find/replace regular expression after all other processing has finished. Support for this is experimental. The first value in the pair will be captured for use in the second. For example:

```
-subdomains = array('/lib/[^"]+?' = 'http://static.x.com/\\1')
```

...would cause the following conversions:

```
<img src="/lib/images/logo.gif" />
-> <img src="http://static.x.com/lib/images/logo.gif" />

<img src="/lib/scripts/jquery.js" />
-> <img src="http://static.x.com/lib/scripts/jquery.js" />

<img src="/lib/styles/reset.css" />
-> <img src="http://static.x.com/lib/styles/reset.css" />
```

Once these options have been configured, the simplest usage is to add an individual script or style like so:

```
[asset_manager->add('/path/to/file.js')]
```

The provided path will be added to the correct list based on its file extension. Duplicates and nonexistant paths will be quietly ignored.

Depending on how you like to organize your files, you can also take advantage of a little more automation using the `->loadmodule` tag. Say you have some lasso code which creates a navigation bar at */includes/navigation.inc*. Instead of just including it, if instead you do this:

```
[asset_manager->loadmodule('/includes/navigation.inc')]
```

...then the following two paths will be checked, and automatically added if they exist:

/includes/navigation.css
/includes/navigation.js

I use this extensively in my projects so that I can keep related assets bundled together with the content which references them.

Under normal circumstances, there is no further code required for the asset manager to run. A `[define_atend]` block is created which kicks off the compression and caching routines. However, if you wish to manually trigger the process, you may do so by calling the `->cache` tag:

```
[asset_manager->cache]
```

The following code illustrates the bare minimum steps required to use the asset manager on a basic Lasso page with all options specified:

```
[//lasso
	// must be authorized for file tags and os_process
	// in actual production, use an inline
	auth_admin;

	// make sure shell.lasso is loaded from lassostartup
	// load required tags
	library('tags/array_unique.inc');
	library('tags/asset_manager.inc');
	library('tags/cache_assets.inc');
	library('tags/compress_gzip.inc');
	library('tags/server_webroot.inc');
	library('tags/url_normalize.inc');

	// initialize the asset manager
	asset_manager->options(
		-usecache=true,
		-minify=true,
		-compress=false,
		-refresh=true,
		-paths=map(
			'yui'=server_webroot + '/assetmgr/yuicompressor.jar',
			'scriptcache'='/assetmgr/scripts/cache/',
			'stylecache'='/assetmgr/styles/cache/',
			'scriptbase'='/assetmgr/scripts/base/',
			'stylebase'='/assetmgr/styles/base/'
		),
        -subdomains=(: ‘/[^”]+?’ = ‘http://local.dev\\1’)
    );

    // add some assets
    asset_manager->add(‘scripts/jquery.corner.js’);
    asset_manager->add(‘scripts/global.js’);
    asset_manager->add(‘styles/global.css’);
]
<html>
    <head>
        <title>Asset Manager Example</title>
    </head>
    <body>
        <h1>Hello, world.</h1>
    </body>
</html>
```

Creating Your Own Workflow

If you’d rather organize things differently, no problem. The `[asset_manager]` custom type is mostly a wrapper for a custom tag called `[cache_assets]`. Its options are the same, except you pass it an array of filepaths explicitly. Even more basic, if all you want to do is play with YUI Compressor, you can call the `[yui_compress]` tag directly. This tag accepts a source path and an optional target path for output. It requires the `[shell]` tag, and thus also `[os_process]`. There are also tags that convert URL’s from relative to absolute. For complete documentation of these tags, visit tagSwap.net.

Caveats

There are a few caveats to using this system. Most notably, since the files that will ultimately be served to the browser will be in a different location from their various source files, it’s important that all paths, particularly in the `url()` attributes of CSS files, be absolute instead of relative. Although the system will convert the paths in CSS files which are directly included for you, there is no way to do the same for JavaScript files. Fortunately, it’s not uncommon for scripts which include other files to include a base URL variable you can configure as a workaround. Also, you should avoid the use of `@import` to import one stylesheet from within another. Doing so will spawn additional HTTP requests, which partially defeats the purpose of using the system. From a security standpoint, the system requires a Lasso user with permission to use the file tags and `[os_process]`. Finally, it’s important to keep the load order in mind. Since the assets in the base folders are loaded in alphabetical order, you’ll need to make sure to rename files if that order breaks any dependency chains.

Measuring The Results

In addition to the bare-bones code shown above, another example is included with this paper which is intended to simulate real-world usage. It is a single-page, pastebin-style application called snppt. It’s intentionally heavy, including many different scripts and stylesheets (some of which still spawn their own requests), to help illustrate the benefits of using the asset management system. The total client-side response time, number of requests, and bytes transferred when loading snppt were measured using YSlow and Firebug, which are plugins available for Firefox. The table below shows the results with and without the asset manager enabled:

	Disabled	**Enabled**	**Savings**
Response Time*	3.61s	1.90s	47.1%
Transferred	205kb	61kb	70.2%
Requests	20	9	55.0%

** averaged over 10 loads each*

As you can see, the benefits of employing these techniques can be significant. I encourage you to run the provided samples and try incorporating these tools into your own projects.

Tools & Reference

Firebug for Firefox
http://www.getfirebug.com/

YSlow for Firebug
http://developer.yahoo.com/yslow/

Jiffy for Firebug
http://looksgoodworkswell.blogspot.com/2008/06/measuring-user-experience-performance.html

YUI Compressor
http://developer.yahoo.com/yui/compressor/

Writing Efficient CSS
http://developer.mozilla.org/en/docs/Writing_Efficient_CSS

Yahoo! Best Practices for Speeding Up Your Site
http://developer.yahoo.com/performance/rules.html