

Fundamentals of Web Site Acceleration Performance Starts at the Web Server

By Port80 Software

Abstract

This paper outlines a common sense, cost-effective approach to lowering total cost of ownership and improving Web site and Web application performance according to two simple principles:

- **Send as little data as possible**
- **Send it as infrequently as possible**

We will explore "best practice" strategies that can be systematically employed in Web front-end source code and at the origin server in order to achieve performance improvements. These basic strategies, which all avoid expensive hardware solutions in favor of software and business process enhancements, include:

- [Code optimization](#)
- [Cache control](#)
- [HTTP compression](#)

We assume that our typical reader is responsible in some way for development and/or management of a Web site or application running on one or more Windows servers with Internet Information Services ([IIS](#)) and that he or she has an interest in improving its performance as much as possible without deploying additional hardware (such as dedicated acceleration appliances) or services (such as Content Distribution Networks).

As we examine each strategy, we will explore the potential benefits to a variety of different Web sites and applications in terms of three vital metrics:

- **Faster Web page loads and an improved user experience, translating into higher revenue and increased efficiencies**
- **Reduction of bandwidth utilization and increased, ongoing savings**
- **Consolidation of the number of server resources required to deliver existing sites and applications**

We will suggest relatively inexpensive software tools that will leverage common Web standards in order to maximize hardware and network resources, thus improving Web site and application performance while lowering the total cost of ownership of Web infrastructure.

Measuring Web Site Performance

The most valuable measurement of Web site performance from an end user's perspective is the amount of time needed to display a requested Web page. The most important metric of page load time is Time to First Byte ([TTFB](#)). Defined as the amount of time it takes to deliver the first byte of the requested page to the end user, TTFB represents the visitor's initial confirmation that a Web site or application is responding. Following Time to First Byte is the metric of [Throughput](#), or how many requests can be served by a Web site or application in a given time period. A user expects text, images, and other elements to load swiftly and methodically — failure in any of these metrics results in the perception of poor performance, which can very quickly lead to visitor frustration and abandonment of the site.

With a large enough budget for heavy infrastructure improvements, any server's connection to the Internet can always be improved. However, we are interested in cases in which these common measures of Web site performance degrade due to uncontrollable network conditions and in which expensive, complex hardware solutions are not feasible or desirable. As Web sites and applications grow increasingly complex with bulky code and third party applications, users — many of whom are still using dial-up connections — must download an increasing amount of data to access a Web site or application. Even when highly efficient applications are made available on fast Wide Area Networks ([WANs](#)) like corporate extranets, segments of the network will always be susceptible to bottlenecks, and the user may experience unacceptably long page load times. If the Web server's source code and management software are not optimized to keep pace with rising site traffic and application complexity, administrators will waste server resources and bandwidth, and users will be presented with slower, easier-to-abandon Web sites.

Optimizing content and content delivery has been proven to improve Web page delivery speed in case studies conducted by the [World Wide Web Consortium](#) and has been espoused by Web optimization experts such as [Andy King](#). Until recently, content optimization required difficult and time-consuming manual coding, but this can now be achieved through the implementation of inexpensive software tools and unobtrusive changes in development and deployment processes. Similarly, while optimizing content delivery once demanded expensive hardware and infrastructure investments, today it can be accomplished with affordable, easy-to-deploy server software tools.

[Code optimization](#), [cache control](#), and [HTTP compression](#) are strategies that focus on sending as little data as necessary as infrequently as possible to optimize performance in an existing Web application's front-end code and on the origin Web server delivering that content to Web browsers.

Code Optimization

Source code optimization is an important but often overlooked application of the "send as little data as possible" principle. As a Web site or application acceleration strategy, its chief benefit is that the technique can provide substantial reductions in network payloads without requiring any additional processing on the origin server. Source code optimization should be implemented as a pre-deployment step through which all additions and changes to the front-end source code (including [markup](#), [style sheets](#), and [client-side scripts](#)) normally pass before being uploaded to the "live" production site or application.

Way Beyond White Space Removal

While traditional "white space removal" (the elimination of superfluous spaces, tabs, new lines, and comments) can result in modest savings of 10 to 15 percent in typical HTML and CSS files, a code optimizer modeled after a traditional software compiler can achieve a much higher level of optimization on all text-based files. Because it possesses a site-wide contextual awareness, such a tool can employ more aggressive tactics, including condensing function and variable names, re-mapping lengthy identifiers and [URL or URI](#) paths, and through the use of a real JavaScript parser, even streamline lines of code.

Until recently, this approach has been the exclusive domain of JavaScript experts with the time and patience to write very terse code. In addition, hand coding still presented limitations to site expansion and code readability because it maintains a single optimized code base for both development and deployment. It is both safer and more efficient to use next generation development tools like the [w3compiler](#) that preserve developer-friendly versions of Web site files as well as the highly optimized versions exclusively for delivery on the Web.

When integrated into the normal developer workflow as a final pre-deployment process, optimizations can be applied to growing Web sites without breaking the external references that abound in modern Web front-ends, including function and variable names defined or initialized in one file and referenced in another. This highly aggressive optimization approach can lead to average savings of 20 to 30 percent in JavaScript files and as high as 70 percent in tested, "real world" cases.

A sensible objection to aggressive source code optimization is that it diminishes the readability of the code when one "views source" in a Web browser. This is a particularly outdated objection as it adds essentially no value for end users and may, in fact, represent a security threat by making it trivially easy for competitors to copy unique client-side features. Even more troubling is the fact that un-optimized source code also clears the way for hackers to conduct reconnaissance on a Web application by "source sifting" for clues to its potential vulnerabilities. Readable and well-commented code is an obvious advantage during initial development, but it is not an appropriate format in which to deliver business-critical Web applications.

Code optimization represents the first step in improving Web site and application performance by reducing the initial network payload that a server must deliver to the end user. Once that content has been optimized on the developer side, attention shifts to how that content can be delivered in its most optimized and efficient form.

Cache Control

What Is Caching? How Does It Apply to the Web?

Caching is a well-known concept in computer science: when programs continually access the same set of instructions, a massive performance benefit can be realized by storing those instructions in RAM. This prevents the program from having to access the disk thousands or even millions of times during execution by quickly retrieving them from RAM. Caching on the Web is similar in that it avoids a roundtrip to the origin Web server each time a resource is requested and instead retrieves the file from a local computer's browser cache or a proxy cache closer to the user.

The most commonly encountered caches on the Web are the ones found in a user's Web browser such as Internet Explorer, Mozilla and Netscape. When a Web page, image, or JavaScript file is requested through the browser each one of these resources may be accompanied by HTTP header directives that tell the browser how long the object can be considered "fresh," that is for how long the resource can be retrieved directly from the browser cache as opposed to from the origin or proxy server. Since the browser represents the cache closest to the end user it offers the maximum performance benefit whenever content can be stored there.

The Common HTTP Request/Response Cycle

When a browser makes an initial [request](#) to a Web site — for example, a [GET request](#) for a home page — the server normally returns a 200 OK [response](#) code and the home page. Additionally, the server will return a 200 OK response and the specified object for each of the dependent resources referenced in that page, such as graphics, style sheets, and JavaScript files. Without appropriate freshness headers, a subsequent request for this home page will result in a "conditional" GET request for each resource. This conditional GET validates the freshness of the resource using whatever means it has available for comparison, usually the [Last-Modified](#) timestamp for the resource (if Last-Modified is not available, such as with dynamic files, an unconditional GET will be sent that circumvents caching altogether). Through a roundtrip to the server, it will be determined whether the Last-Modified value of the item on the server matches that of the item cached in the browser, and either a fresh version will be returned from the server with a 200 OK response, or a 304 Not-Modified response header will be returned, indicating that the file in the cache is valid and can be used again.

A major downside to validations such as this is that, for most files on the Web, it can take almost as long to make the roundtrip with the validation request as it does to actually return the resource from the server. When done unnecessarily this wastes bandwidth and slows page load time. With the dominant Internet Explorer browser under its default settings and without sufficient cache control headers, a conditional GET will be sent at the beginning of each new session. This means that most repeat visitors to a Web site or application are needlessly requesting the same resources over and over again.

Fortunately, there is an easy way to eliminate these unnecessary roundtrips to the server. If a freshness indicator is sent with each of the original responses, the browser can pull the files directly from the browser cache on subsequent requests without any need for server validation until the expires time assigned to the object is reached. When a resource is retrieved directly from the user's browser cache, the responsiveness of the

Web site is dramatically improved, giving the perception of a faster site or application. This cache retrieval behavior can be seen clearly when a user revisits a Web page during the same browser session – images render almost instantaneously. When the correct cache control headers are incorporated into a server's responses, this performance improvement and reduction of network traffic will persist for any number of subsequent sessions as long as the object is not expired or until a user manually empties his or her cache.

Somewhere Between a Rotting Fish and Burning Currency

Web caching via HTTP headers is outlined in [HTTP/1.1, section 13 of RFC 2616](#). The language of the specification itself helps to define the desired effects and tradeoffs of caching on the Web. The specification offers suggestions for explicit warnings given by a user agent (browser) when it is not making optimal use of caching. For cases in which a browser's settings have been configured to prevent the validation of an expired object, the spec suggests that the user agent could display an image of a stale fish. For cases in which an object is unnecessarily validated (or re-sent, despite its being fresh) the suggested image is of burning currency, so that the user "[does not inadvertently consume excess resources or suffer from excessive latency](#)".

Cache control headers can be used by a server to provide a browser with the information it needs to accurately handle the resources on a Web site. By setting values to prevent the caching of highly volatile objects, one can help to ensure that files are not given the "rotting fish treatment." In the case of objects that are by design kept consistent for long periods of time, such as a Web site's navigation buttons and logos, setting a lengthy expiration time can avoid the "burning currency treatment." Whatever the desired lifetime of an object is, there is no good reason why each resource should not have cache-related headers sent along with it on every response.

A variety of cache-related headers are superior to the Last-Modified header because they can do more than merely provide a clue about the version of a resource. They can allow administrators to control how often a conditional GET, or validation trip to the Web server, occurs. Well-written cache control headers help to ensure that a browser always retrieves a fresh resource when necessary or tells the browser how long a validation check can be forestalled. This can dramatically reduce network chatter by eliminating superfluous validations and help keep the pipe clear for essential trips to retrieve dynamic or other files.

The [Expires header](#) (for HTTP 1.0) and the [Cache-control header](#) with its [max-age](#) directive (for HTTP/1.1) allow a Web developer to define how long each resource is to remain fresh. The value specifies a period of time after which a browser needs to check back with the server to validate whether or not the object has been updated.

By using a cache control header inspection tool like this [Cacheability Engine](#), it is easy to see by the Last-Modified header value how long many images, style sheets, and JavaScript files go unchanged. Oftentimes, a year or more has passed since certain images were last modified. This means that all validation trips to the Web server over the past year for these objects were unnecessary, thus incurring higher bandwidth expenditures and slower page loads for users.

Cache Control Does the Heavy Lifting in Web Server Performance

Formulating a caching policy for a Web site — determining which resources should not get cached, which resources should, and for how long — is a vital step in improving site performance. If the resource is a dynamic page that includes time sensitive database queries on each request, then cache control headers should be used to ensure that the page is never cached. If the data in a page is specific to the user or session, but not highly time sensitive, cache control directives can restrict caching to the user's private browser cache. Any other unchanging resources that are accessed multiple times throughout a Web site, such as logos or navigational graphics, should be retrieved only once during an initial request and after that validated as infrequently as is possible.

This begs the question: "Who knows best what the caching policies should be?" On Microsoft IIS Web servers, currently the Web site administrator has sole access to cache control-related headers through the MMC control panel, suggesting that the admin is best suited to determine the policies. More often, however, it is the application developer who is most familiar with the resources on the Web site, and is thus best qualified to establish and control the policies. Giving developers the ability to set cache control headers for resources across their site using a rules-based approach is ideal because it allows them to set appropriate policies for single objects, for a directory of objects, or based on MIME types. An added benefit of developer cache management is less work for site administrators. Try out [CacheRight](#) for Microsoft IIS cache control management and centralized cache rules management.

No matter what your Web site or application does, or what your network environment includes, performance can be improved by implementing caching policies for the resources on your site. By eliminating unnecessary roundtrips to the server, cache control can go a long way towards achieving the dual objectives of sending as little as possible as infrequently as possible. However, in cases where resources are dynamic or rapidly changing, a roundtrip to the server is unavoidable for each request. In these situations, the payload can still be minimized by using HTTP compression.

HTTP Compression

Low Cost, High Impact Performance Tuning Should Not Be Overlooked

HTTP compression is a long-established Web standard that is only now receiving the attention it deserves. Its implementation was sketched out in HTTP/1.0 ([RFC 1945](#)) and completely specified by 1999 in HTTP/1.1 (RFC 2616 covers [accept-encoding](#) and [content-encoding](#)). Case studies conducted as early as 1997 by the WC3 proved the ability of compression to significantly reduce bandwidth usage and to improve Web performance (see their [dial-up](#) and [LAN](#) compression studies).

In the case of HTTP compression, a standard gzip or deflate encoding method is applied to the payload of an HTTP response, significantly compressing the resource before it is transported across the Web. Gzip (RFC 1952) is a lossless compressed data format that has been widely used in computer science for decades. The deflation algorithm used by gzip (RFC 1951) is shared by common libraries like zip and zlib and is an open-source, patent-free variation of the LZ77 (Lempel-Ziv 1977) algorithm. Because of the proven stability of applying compression and decompression, the technology has been supported in all major browser implementations since early in the 4.X generation (for

Internet Explorer and Netscape).

While the application of gzip and deflate is relatively straightforward, two major factors limited the widespread adoption of HTTP compression as a performance enhancement strategy for Web sites and applications. Most notably, isolated bugs in early server implementations and in compression-capable browsers created situations in which servers sent compressed resources to browsers that were not actually able to handle them.

When data is encoded using a compressed format like gzip or deflate, it introduces complexity into the HTTP request/response interaction by necessitating a type of content negotiation. Whenever compression is applied, it creates two versions of a resource: one compressed (for browsers that can handle compressed data) and one uncompressed (for browsers that cannot). A browser needs only to accurately request which version it would like to receive. However, there are cases in which some browsers express, in their HTTP request headers, the ability to handle a gzip compressed version of a certain MIME type resource when they effectively cannot. These browser bugs have given rise to a number of third-party compression tools for Microsoft IIS Web servers that give administrators the ability to properly configure their servers to handle the exceptions necessary for each problematic browser and MIME type.

The second factor that has stunted the growth of HTTP compression is something of a historical and economic accident. Giant, pre-2001 technology budgets and lofty predictions of widespread high-bandwidth Internet connections led to some much more elaborate and expensive solutions to performance degradation due to high latency network conditions. Despite the elements being in place to safely implement HTTP compression, [Akamai](#) and other Content Distribution Networks employing edge caching technology captured the market's attention. While HTTP compression should not be viewed as a direct alternative to these more expensive options, with the help of a reliable configuration and management tool, compression should be employed as a complementary, affordable initial step towards performance enhancement.

Compression Will Deliver Smaller Web Transfer Payloads and Improved Performance

Most often, HTTP compression is implemented on the server side as a filter or module which applies the gzip algorithm to responses as the server sends them out. Any text-based content can be compressed. In the case of purely static content, such as markup, style sheets, and JavaScript, it is usually possible to cache the compressed representation, sparing the CPU of the burden of repeatedly compressing the same file. When truly dynamic content is compressed, it usually must be recompressed each time it is requested (though there can be exceptions for quasi-dynamic content, given a smart enough compression engine). This means that there is trade-off to be considered between processor utilization and payload reduction. A highly configurable compression tool enables an administrator to adjust the tradeoff point between processor utilization and compressed resource size by setting the compression level for all resources on the Web site, thereby not wasting CPU cycles on "over-compressing" objects which might compress just as tightly with a lower level setting as with a higher one. This also allows for the exclusion of binary image files from HTTP compression, as most images are already optimized when they are created in an editor such as Illustrator. Avoid the needless recompression of images as this may actually increase their file size or introduce distortion.

Compression results in significant file size savings on text-type files. The exact percentage saved will depend on the degree of redundancy or repetition in the character sequences in the file, but in many cases, individual files may be reduced by 70 or even 80 percent. Even relatively lumpy or less uniform text files will often shrink by 60 percent. Keep in mind that when looking at overall savings on the site, these extremely high compression rates will be counterbalanced by the presence of image MIME types that cannot usefully be compressed. Overall savings from compression is likely to be in the range of 30 to 40 percent for the typical Web site or application. This free online tool will help you to [see the file size savings and transfer improvement garnered from compressing any Web resource](#).

If bandwidth savings is the primary goal, the strategy should be to compress all text-based output. Ideally, this should include not only static text files (such as HTML and CSS), but files that produce output in text media MIME types (such as ASP and ASP.NET files), as well as files that are text-based but of another media type (such as external JavaScript files). This free online tool will help you to [gauge the bandwidth cost savings to be realized from compressing any Web resource](#).

[Case studies like this one from IBM](#) have proven the performance improvement that can be realized by using compression, especially in WAN environments that include areas of high traffic and high latency. The general principle is clear: the more crowded the pipe and the more hops in the route, the greater the potential benefits of sending the smallest possible payloads. When the size of the resource is drastically reduced, as a text file is when gzip compression is applied, the number of packets sent is also reduced, decreasing the likelihood of lost packets and retransmissions. All of these benefits of HTTP compression lead to much faster and more efficient transfers.

On Microsoft IIS 4.0 and 5.0 Web servers, [httpZip](#) is the best solution for compression as it addresses a number of [shortcomings in functionality, customization, and reporting on Windows NT/2000](#) that gave rise to a third party tools market. With the launch of Windows Server 2003 and IIS 6.0, Microsoft chose to make compression functionality a priority, and their internal IIS 6.0 compression software works — though you must delve into the IIS metabase to manage it beyond a simple "on/off" decision. You can use a tool like [ZipEnable](#) to unlock and greatly enhance the configuration and management options for IIS 6.0 built-in compression.

Make Your Web Server a Faster, More Efficient Resource

Having explored these three techniques for Web server performance optimization, how can you put this learning into practice? We have mentioned a few of our Port80 Software tools above, but the following guidelines will help you find the right solutions for your own Web server performance strategy.

Selecting a Code Optimization Tool

An effective source code optimizer should combine an awareness of Web standards with the ability to undertake the aggressive optimizations that can only be done if the entire site is treated as a whole. The capability to fully parse the code is the key here. This is particularly true in the case of JavaScript and DHTML optimizations, in which external references can easily be broken if optimization is done file by file.

Selecting a Cache Control Tool

A cache control tool should provide the ability to implement cache control policies in a rule-based approach so that each resource is cached and validated in the most appropriate and efficient manner. It should also include a mechanism that allows site and application developers to contribute to the generation of those rules without requiring intervention by server administrators.

Selecting a HTTP Compression Tool

An origin server compression solution should be, above all else, highly configurable. Although compression is widely supported by modern browsers, several known bugs remain. Such potential problems are best addressed on the server side by properly configuring exceptions for the MIME types problematic for each specific browser. Similar exceptions may be made for files that are very small or very large, or which, for other specialized reasons, should not be compressed. It is also helpful to be able to adjust the level of compression for different file types and to be able to monitor and log the results of compression.

Strong Performance is the Foundation of Your Web Presence

In order to achieve maximum Web site and application performance, it is vital to view the complete chain — from source code development, server-side processes, and the connection between server and end user, all the way to the end user's Web browser — and to examine each link in that chain for both potential pitfalls and opportunities for improvement. Implementing the three-fold strategy of code optimization, cache control, and compression will dramatically reduce the amount of Web site and application data transmitted and ensures that your data is sent in the most efficient manner possible. Leverage the resources of your existing Web infrastructure with the strategies presented here, none of which are particularly expensive. Implementing these unobtrusive changes in development and deployment practices will aid your Web developers, site administrators, and users. The results will speak for themselves: a more efficient Web server, lower bandwidth expenses, and a faster experience for your users, all helping to drive revenue growth, lower costs, and modernize your Web systems.

About Port80 Software

Port80 Software, Inc. is an innovative developer of software products for Microsoft Internet Information Services (IIS) focused Web site administrators, developers and owners. Port80 products enhance IIS functionality, security, performance, and user experience, augmenting IIS with on-par or better features than those provided by the Apache server. Port80 also develops the w3compiler desktop tool for next generation markup and code optimization. Port80 Software is a Microsoft Certified Partner located in San Diego, CA. Additional information about the company is available on the Internet at www.port80software.com.