ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

Optimizing Front End Applications and JavaScript

Vamsi Krishna Myalapalli[1], Srinivas Karri[2]

Software Engineer, Open Text Corporation, Mind Space IT Park, Hitec City, Hyderabad, India[1, 2]

ABSTRACT: In the contemporary world it is inevitable to develop Front End Applications without JavaScript. Many JavaScript programmers simply do code with less concern towards long term maintainability and optimized processing. Though JAVA Script offers sundry methods of coding systems to reach the same end, there exist some practices which are pre-requisite to ensure that consequences will be prolific. As such this paper proposes simple, miscellaneous, flexible, reliable and easy methodologies to make web pages efficient and faster, making the application robust. The exploration of this paper could serve as a benchmarking tool and code refactoring tool for overhauling coding practices as well as tuning web pages. Experimental fallouts of our investigation advocate that response time, flexibility, maintainability and reusability of the application are enriched.

KEYWORDS: JAVA Script Tuning; JAVA Script Best Practices; UI Tuning; JAVA Script Optimization; JAVA Script Code Refactoring; Web Page Tuning; Web Page Optimization.

I. INTRODUCTION

Initially, the Web was conceived as a collection of static HTML documents, tied together with hyperlinks. Soon, as the Web grew in popularity and size, the webmasters who were creating static HTML web pages felt they needed something more. They wanted the opportunity for richer user interaction, mainly driven by desire to save server round-trips for simple tasks such as form validation. JavaScript's instant popularity happened during the period of the Browser Wars. JAVA Script is a client side and dynamic programming language. It is the HTML and Web Programming language. JavaScript is categorized as prototype based script language with dynamic typing and first class functions. The combination of these amenities will make it an imperative, multi paradigm, object-oriented and functional programming styles. JavaScript is not a classic OO language, but a prototypal one.

JavaScript performance really begins with getting the code onto a page in the most efficient way possible. A large source of performance problems in JavaScript is poorly written code that uses inefficient algorithms or utilities.In effect, JavaScript forces the developer to perform the optimizations that a compiler would normally handle in other languages.This paper exposes proactive best practices or tuning techniques that the programmers should gaze at.

II. BACKGROUND AND RELATED WORK

High Performance JAVA Programming [1] explained several ways to tune programs at statements level. It explains statement tuning using profiling methodologies. Also it explained techniques to tune programs when the code is communicating with web pages and web protocols.

High Performance C Programming [2] expounded sundry logical methodologies to tune code in loops, methods and in other control structures. It also explained performance impact of variable scope.

This paper proposes Front End Application tuning i.e. tuning User Interface, Web pages etc., JavaScript tuning and enhancing response time that end user experiences. Hopefully this paper could serve as the benchmark to yield High Performance JavaScript and High Performance Web Pages.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

III. PROPOSED MODEL

This paper brings out several best programming practices in JavaScript language and serves as a Benchmarking tool. Tuning methods and best practices are explained in this section and their corresponding programmatic implementation is explained in Section 4.

Loading and Execution #(1-4): JavaScript performance in the browser is arguably the most important usability issue facing developers. The problem is complex because of the blocking nature of JavaScript, which is to say that nothing else can happen while JavaScript code is being executed. In fact, most browsers use a single process for both user interface (UI) updates and JavaScript execution, so only one can happen at any given moment in time. The longer JavaScript takes to execute, the longer it takes before the browser is free to respond to user input.

1) Script Positioning: The HTML 4 specification indicates that a <script> tag may be placed inside of a <head> or <body> tag in an HTML document and may appear any number of times within each. Traditionally, <script> tags that are used to load external JavaScript files have appeared in the <head>, along with <link> tags to load external CSS files and other meta-information about the page. It's best to keep as many style and behaviour dependencies together, loading them first so that the page will come in looking and behaving correctly.

2) Grouping Scripts: Each <script> tag blocks the page from rendering during initial download, it's helpful to limit the total number of <script> tags contained in the page. This applies to both inline scripts as well as those in external files. Every time a <script> tag is encountered during the parsing of an HTML page, there is going to be a delay while the code is executed; minimizing these delays improves the overall performance of the page. An inline script placed after a <link> tag referencing an external style sheet caused the browser to block while waiting for the style sheet to download. This is done to ensure that the inline script will have the most correct style information with which to work. Never put an inline script after a <link> tag for this reason.

3) Non-Blocking Scripts: JavaScript's tendency to block browser processes, both HTTP requests and UI updates, is the most notable performance issue facing developers. Keeping JavaScript files small and limiting the number of HTTP requests are only the first steps in creating a responsive web application. The richer the functionality an application requires, the more JavaScript code is required, and so keeping source code small isn't always an option. Limiting our self to downloading a single large JavaScript file will only result in locking the browser out for a long period of time, despite it being just one HTTP request. To get around this situation, we need to incrementally add more JavaScript to the page in a way that doesn't block the browser. The secret to non-blocking scripts is to load the JavaScript source code after the page has finished loading. This means downloading the code after the window's load event has been fired. There are a few techniques for achieving this result.

a) Deferred Scripts
b) Dynamic Script Elements
c) XMLHttpRequest Script Injection

4) Non-Blocking Pattern: Loading a significant amount of JavaScript onto a page should be two-step process: first, include the code necessary to dynamically load JavaScript, and then load the rest of the JavaScript code needed for page initialization. Since the first part of the code is as small as possible, potentially containing just the load Script() function, it downloads and executes quickly, and so shouldn't cause much interference with the page. Once the initial code is in place, use it to load the remaining JavaScript.

Flow Control #(5-9): The overall structure of our code is one of the main determinants as to how fast it will execute. Having a very small amount of code doesn't necessarily mean that it will run quickly, and having a large amount of code doesn't necessarily mean that it will run slowly. A lot of the performance impact is directly related to how the code has been organized and how we are attempting to solve a given problem.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

5) For-in Loop vs. Other Loop: For-in loop is comparatively slower than other loops. Since, in each-iteration, the loop results in a property lookup either on the instance or on a prototype, thus it has considerably more overhead per iteration and is therefore slower than the other loops. For the same number of loop iterations, for in loop can end up as much as seven times slower than the other loop types. We should never use for-in to iterate over members of an array.

6) Memoization: Work avoidance is the best performance optimization technique. The less work our code has to do, the faster it executes. Along those lines, it also makes sense to avoid work repetition. Performing the same task multiple times is a waste of execution time. Memoization is an approach to avoid work repetition by caching previous calculations for later reuse, which makes memoization a useful technique for recursive algorithms.

7) Call Stack Limits: The amount of recursion supported by JavaScript engines varies and is directly related to the size of the JavaScript call stack. Even though it is possible to trap these errors in JavaScript, it is not recommended. No script should ever be deployed that has the potential to exceed the maximum call stack size. Apart from this stack overflow errors prevent the rest of the code from executing. If we run into a stack overflow error, change the method to an iterative algorithm or make use of memoization to avoid work repetition.

8) If-Else vs. Switch: Switch executes faster than If-Else. But If-else offers readability. So rewriting If-Else will optimize it. The rewritten if-else statement should have maximum number of four condition evaluations each time through. This is achieved by applying a binary-search-like approach, splitting the possible values into a series of ranges to check and then drilling down further in that section. The average amount of time it takes to execute this code is roughly half of the time it takes to execute the previous if-else statement when the values are evenly distributed between 0 and 10. This approach is best when there are ranges of values for which to test (as opposed to discrete values, in which case a switch statement is typically more appropriate).

9) Lookup Tables: Sometimes the best approach to conditionals is to avoid using if-else and switch altogether. When there are a large number of discrete values for which to test, both if-else and switch are significantly slower than using a lookup table. Lookup tables can be created using arrays or regular objects in JavaScript, and accessing data from a lookup table is much faster than using if-else or switch, especially when the number of conditions is large. When using a lookup table, the operation becomes either an array item lookup or an object member lookup. This is a major advantage for lookup tables: since there are no conditions to evaluate, there is little or no additional overhead as the number of possible values increase. Lookup tables are most useful when there is logical mapping between a single key and a single value. A switch statement is more appropriate when each key requires a unique action or set of actions to take place.

Data Access #(10-11): It has to be determined where data should be stored for optimal reading and writing, where data is stored is related to how quickly it can be retrieved during code execution.

10) Variables vs. Object Members: Literal values and local variables can be accessed very quickly, whereas array items and object members take longer.

Local variables are faster to access than out-of-scope variables because they exist in the first variable object of the scope chain. The further into the scope chain a variable is, the longer it takes to access. Global variables are always the slowest to access because they are always last in the scope chain.

Nested object members incur significant performance impact and should be minimized.

We can enhance the performance of JavaScript code by storing frequently used object members, array items, and out-of-scope variables in local variables. We can then access the local variables faster than the originals.

11) Prototype Chaining: The deeper into the prototype chain that a property or method exist, the slower it is to access. Avoid the 'with' statement because it augments the execution context scope chain. The catch clause of a try-catch statement will have the same effect.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

DOM Scripting #(12-16): DOM (Document Object Model) scripting is expensive, and it's a common performance bottleneck in rich web applications. The DOM is a language-independent application interface for working with XML and HTML documents. Even though the DOM is a language-independent API, in the browser the interface is implemented in JavaScript.

12) Minimize DOM Access: Minimize DOM access, and try to work as much as possible in JavaScript.Use local variables to store DOM references that are accessed repeatedly.Leaving the DOM alone is big JavaScript optimization. Example: Appending an array of list items: if we append each of these to the DOM individually, it is considerably slower than appending them all at once. This is because DOM operations are expensive.

13) ***HTML Collections:*** HTML collections represent the live, underlying document. Cache the collection length into a variable and use it when iterating, and make a copy of the collection into an array for heavy work on collections.

14) ***Event Delegation:*** Using event delegation minimizes the number of event handlers.When there are a large number of elements on a page and each of them has one or more event handlers attached (such as on-click), this may affect performance. Attaching every handler comes at a price—either in the form of heavier pages (more mark-up or Java- Script code) or in the form of runtime execution time. The more DOM nodes we need to touch and modify, the slower our application, especially because the event attaching phase usually happens at the on-load (or DOMContentReady) event, which is a busy time for every interaction-rich web page. Attaching events takes processing time, and, in addition, the browser needs to keep track of each handler, which takes up memory.

A simple and elegant technique for handling DOM events is event delegation. It's based on the fact that events bubble up and can be handled by a parent element. With 'event delegation', we attach only one handler on a wrapper element to handle all events that happen to the children descendant of that parent wrapper.

15) Building DOM Node and All its Sub-Nodes Offline: When adding complex content such as tables to a site, performance is improved by adding complex sub-trees offline.

16) Minimizing Reflow in DOM: DOM operations are resource-heavy because of reflow. Reflow is basically the process by which the browser re-renders the DOM elements on the screen. For instance, if we change the width of a div with JavaScript, the browser has to refresh the rendered page to account for this change. Any time an element is added or removed from the DOM, reflow will occur.

The solution for this is a very handy JavaScript object, "Document Fragment". Document Fragment is basically a document-like fragment that isn't visually represented by the browser. Having no visual representation provides a number of advantages; mainly we can append nodes to a 'documentFragment' without incurring any browser reflow.

String Processing #(17-20): Pragmatically all JavaScript programs are intimately tied to strings. A typical program deals with numerous tasks like these that require to merge, split, rearrange, search, iterate over, and otherwise handle strings; and as web applications become more complex, progressively more of this processing is done in browser.

17) String Concatenation: For String Concatenation we use either 'Array.prototype.join' or 'Concat' or '+' operator. Array joining is slower than other methods of concatenation in most browsers, but this is efficient in IE7 and earlier. So based on the browser prefer concatenation method. On the other hand 'concat' is a little slower than simple + and += operators in most cases.

18) Regular Expressions: The following are sundry ways to advance Regular Expression efficiency:

a) Focus on Failing Faster.
b) Start regexes with simple, required tokens.
c) Making quantified patterns and their following token mutually exclusive.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

d) Reduce the amount and reach of alternation.
e) Use non-capturing groups: Capturing groups spend time and memory remembering back-references and keeping them up to date. If we don't need a back-reference, avoid this overhead by using a non-capturing group—i.e., (?:…) instead of (…).
f) Capture interesting text to reduce post-processing.
g) Expose required tokens.
h) Use appropriate Quantifiers.
i) Reuse regexes by assigning them to variables.
j) Split complex regexes into simpler pieces.

19) When Not to Use Regular Expressions: When used with care, regexes are very fast. However, they usually overkill when we are merely searching for literal strings. This is especially true if we know in advance which part of a string we want to test. Backtracking is both a fundamental component of regex matching and a frequent source of regex inefficiency.

20) String Trimming: Removing leading and trailing whitespace from a string is a simple but common task. Trimming strings is not a common performance bottleneck, but it serves as a decent case study for regex optimization since there are lot of ways to implement it. Trimming can be performed with and without using regular expressions. Using two simple regexes (one to remove leading whitespace and another for trailing whitespace) offers a good mix of brevity and cross-browser efficiency with varying string contents and lengths. Looping from the end of the string in search of the first non-whitespace characters, or combining this technique with regexes in a hybrid approach, offers a good alternative that is less affected by overall string length.

Responsive Interfaces #(21-26): There's nothing more frustrating than clicking something on a web page and having nothing happen. This problem goes back to the origin of transactional web applications and resulted in the now-ubiquitous "please click only once" message that accompanies most form submissions.

21) Browser UI Threading: No JavaScript task should take longer than 100 milliseconds to execute. Longer execution times cause a noticeable delay in updates to the UI and negatively impact the overall user experience.

Browsers behave differently in response to user interaction during JavaScript execution. Regardless of the behavior, the user experience becomes confusing and disjointed when JavaScript takes a long time to execute.

22) Yielding with Timers: Timers can be used to schedule code for later execution, which allows us to split up long-running scripts into a series of smaller tasks.

23) Array Processing with Timers: One common cause of long-running scripts is loops that take too long to execute. The approach is to split up the loop's work into a series of timers.

The actual amount of time to delay each timer is largely dependent on our use case. It's best to use at least 25 milliseconds because smaller delays leave too little time for most UI updates.

One side effect of using timers to process arrays is that the total time to process the array increases. This is because the UI thread is freed up after each item is processed and there is a delay before the next item is processed. Nevertheless, this is a necessary trade-off to avoid a poor user experience by locking up the browser.

24) Splitting Up Tasks: One task can often be broken down into a series of subtasks. If a single function is taking too long to execute, check to see whether it can be broken down into a series of smaller functions that complete in smaller amounts of time. This is often as simple as considering a single line of code as an atomic task, even though multiple lines of code typically can be grouped together into a single task.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

25) Timers and Performance: Timers can make a huge difference in the overall performance of our JavaScript code, but overusing them can have a negative effect on performance. Performance issues start to appear when multiple repeating timers are being created at the same time. Low-frequency repeating timers-those occurring at intervals of one second or greater have little effect on overall web application responsiveness. The timer delays in this case are too large to create a bottleneck on the UI thread and are therefore safe to use repeatedly. When multiple repeating timers are used; with a much greater frequency (between 100 and 200 milliseconds), the application will become noticeably slower and less responsive.To limit the number of high-frequency repeating timers in our web application create a single repeating timer that performs multiple operations with each execution.

26) Enforce Web Workers: Web workers introduce an interface through which code can be executed without taking time on the browser UI thread; Thus, preventing UI locking. Web workers represent a potentially huge performance improvement for web applications because each new worker spawns its own thread in which to execute JavaScript. That means not only will code executing in a worker not affect the browser UI, but it also won't affect code executing in other workers.

Web workers are suitable for any long-running scripts that work on pure data and that have no ties to the browser UI. This may seem like a fairly small number of uses, but buried in web applications there are typically some data-handling approaches that would benefit from using a worker instead of timers.

AJAX #(27-29): Ajax is a cornerstone of high-performance JavaScript. It can be used to make a page load faster by delaying the download of large resources. It can prevent page loads altogether by allowing for data to be transferred between the client and the server asynchronously. It can even be used to fetch all of a page's resources in one HTTP request. By choosing the correct transmission technique and the most efficient data format, we can significantly improve how our users interact with our site.

Once we have selected the most appropriate data transmission technique and data format, we can start to consider other optimization techniques. These can be highly situational, so be sure that our application fits the profile before considering them.

27) Cache Data: The fastest Ajax request is one that we don't have to make. There are two main ways of preventing an unnecessary request:

a) On the server side, set HTTP headers which ensure that response will be cached in the browser.
b) On the client side, store fetched data locally so that it doesn't have to be requested again.

The first technique is the easiest to set up and maintain, whereas the second gives us the highest degree of control.
Improve perceived loading time of page by using Ajax to fetch less important files after rest of the page has loaded.

28) Setting HTTP Headers: If Ajax responses should be cached by the browser, we must use GET to make the request. But simply using GET isn't sufficient; we must also send the correct HTTP headers with the response. The Expires header tells the browser how long a response can be cached. The value is a date; after that date has passed, any requests for that URL will stop being delivered from cache and will instead be passed on to the server.

An Expires header is the easiest way to make sure that Ajax responses are cached on the browser. We don't have to change anything in the client-side code, and can continue to make Ajax requests normally; knowing that the browser will send the request on to the server only if the file isn't in cache. It's also easy to implement on the server side, as all languages allow us to set headers in one way or another. This is the simplest approach to ensure that data is cached.

29) Storing Data Locally: Instead of relying on the browser to handle caching, we can also do it in a more manual fashion, by storing the responses we receive from the server. This can be done by putting the response text into an object, keyed by the URL used to fetch it.

A local cache also works well for users browsing on mobile devices. Most of the browsers on such devices have small or non-existent caches, and a manual cache is the best option for preventing unnecessary requests.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

Programming Practices #(30-52): Every programming language has pain points and inefficient patterns that develop over time. JavaScript presents unique performance challenges related to the way we organize our code.

30) Avoid Double Evaluation: JavaScript allows us to take a string containing code and execute it from within running code. There are 4 standard ways to accomplish this: eval(), the Function() constructor, setTimeout(), and setInterval(). Each of these functions allows us to pass in a string of JavaScript code and have it executed.

Whenever we're evaluating JavaScript code from within JavaScript code, we incur a double evaluation penalty. This code is first evaluated as normal, and then, while executing, another evaluation happens to execute the code contained in a string. Double evaluation is a costly operation and takes much longer than if the same code were included natively.

Avoiding double evaluation is the key to achieve the most optimal JavaScript runtime performance possible. Optimizing JavaScript engines often cache the result of repeated code evaluations using eval().

31) Using Object/Array Literals: There are multiple ways to create objects and arrays in JavaScript, but nothing is faster than creating object and array literals. Literals are evaluated faster apart from taking up less space in code, so the overall file size is smaller. As the number of object properties and array items increases, so too does the benefit of using literals.

Using Fast Parts of Language #(32-33): Hardware friendly code will always get executed faster.

32) Using Bitwise Operators: Bitwise Operators are incredibly fastest operations. Hence the possibility of using Bitwise operators should be explored in the code.

33) Using Native Methods: No matter how optimal our JavaScript code is, it will never be faster than the native methods provided by the JavaScript engine. The reason for this is native parts of JavaScript—those already present in the browser before we write a line of code—are all written in a lower-level language such as C++. i.e. these methods are compiled down to machine code as part of browser and hence don't have same limitations as our JavaScript code.

34) Avoid Eval Functions: It is the most misused feature of JavaScript. Eval will be slower because it needs to run the compiler just to execute a trivial assignment statement. On the other hand eval function compromises security of application, because it grants too much authority to eval'd text. And it compromises the performance of the language as a whole in the same way that the 'with' statement does.

35) Avoid 'new' Operator: JavaScript's new operator creates a new object that inherits from the operand's prototype member, and then calls the operand, binding the new object to this. This gives the operand (which had better be a constructor function) a chance to customize the new object before it is returned to the requestor.

If we forget to use the new operator, we instead get an ordinary function call, and this is bound to the global object instead of to a new object. That means that our function will be clobbering global variables when it attempts to initialize the new members. That is a very bad thing. There is no compile-time warning. There is no runtime warning.

By convention, functions that are intended to be used with new should be given names with initial capital letters, and names with initial capital letters should be used only with constructor functions that take the new prefix. This convention gives us a visual cue that can help spot expensive mistakes that the language itself is keen to overlook.

An even better coping strategy is to not use new at all.Avoid 'new Object' and 'new Array'. Use {} and [] instead.

36) Function Statement vs. Function Expression: JavaScript has a function statement as well as a function expression. This is confusing because they can look exactly the same. A function statement is shorthand for a var statement with a function value.

The statement function foo( ) {}means about the same thing as

var foo = function foo( ) {};// foo is a variable containing a function value.

To use the language well, it is important to understand that functions are values.

Function statements are subject to hoisting. This means that regardless of where a function is placed, it is moved to the top of the scope in which it is defined. This relaxes the requirement that functions should be declared before used,

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

which I think leads to sloppiness. It also prohibits the use of function statements in if statements. It turns out that most browsers allow function statements in if statements, but they vary in how that should be interpreted. That creates portability problems.

37) ***The Equality Operators:*** JavaScript has two sets of equality operators: === and !==, and their evil twins == and !=. The good ones work the way we would expect. If the two operands are of the same type and have the same value, then === produces true and !== produces false. The evil twins do the right thing when the operands are of the same type, but if they are of different types, they attempt to coerce the values. The rules by which they do that are complicated and unmemorable. These are some of the stimulating cases:

'' == '0'	// false	0 == ''	// true
0 == '0'	// true		
false == 'false'	// false	false == '0'	// true
false == undefined	// false	false == null	// false
null == undefined	// true	' \t\r\n ' == 0	// true

The lack of transitivity is alarming. We should not use the evil twins. Instead, always use === and !==. All of the comparisons just shown produce false with the === operator.

38) Evaluating Local Variables: Local variables are found based on the most to the least specific scope and can pass through multiple levels of scope, the look-ups can result in generic queries. When defining the function scope, within a local variable without a preceding var declaration, it is important to precede each variable with var in order to define the current scope in order to prevent the look-up and to speed up the code.

39) Manipulate Element Fragments before adding them to DOM: Before placing the elements to the DOM, we have to ensure that all tunings are performed in order to improve JavaScript performance. This will eliminate the need to set aside Prepend or AppendjQuery APIs.

40) Using .js file to Cache Scripts: By utilizing this method, increased performance can be achieved because it allows the browser to load the script once and will only recall it from cache should the page be reloaded or revisited.

41) Curtail Scope Chains: Global scopes can be slow, because each time a function executes, it cause a temporary calling scope to be created. JavaScript searchers for the first item in the scope chain, and if it doesn't find the variable, it swells up the chain until it hits the global object.

42) Create Shortcut Codes to Speed Up Coding: For codes that are constantly being processed, speeding up the coding process can be achieved by creating shortcuts for longer codes, for example, 'document.getElementById'. By creating a shortcut, longer scripts will not take as long to code and will save time in the overall process.

43) Enhancing Speed of Object Detection: The efficient method of using Object Detection is to use a code created dynamic object detection, rather than performing object detection inside of a function.

44) Function In-lining: This helps in eliminating call costs, and replaces a function call with the body of the called function. In JavaScript, performing a function call is an expensive operation because it takes several preparatory steps to perform: allocating space for parameters, copying the parameters, and resolving the function name.

45) Implement Common Sub-expression Elimination: Common sub-expression elimination is a performance-targeted compiler optimization technique that searches for instances of identical expressions and replaces them with a single variable holding the computed value. We can expect that using a single local variable for a common sub-expression will always be faster than leaving the code unchanged.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

46) Deter Global Variables: The scripting engine needs to look through the scope, when referencing global variables from within function or another scope, the variable will be destroyed when the local scope is lost.

47) Primitive Functions vs. Function Calls: Performance in critical loops and functions can be enhanced by using equivalent primitive functions instead of function calls.

48) Don't Retain Alive References of other Documents: Do not retaining alive references of other documents after the script has finished with them. This is because any references to those objects from that document are not to be kept in its entire DOM tree, and the scripting environment will not be kept alive in memory. Thus the document itself is no longer loaded.

49) Using XMLHttpRequest: XMLHttpRequest assists to reduce the amount of content coming from the server and avoids the performance impact of destroying and recreating the scripting environment in between page loads. It is important to ensure that XMLHttpRequest is supported, or otherwise it can lead to glitches.

50) Evade 'try-catch-finally' Clauses: Whenever the catch clause is executed, where the caught exception object is assigned to a variable, "try-catch-finally" creates a new variable in the current scope at runtime. A number of browsers do not handle this process efficiently because the variable is created and destroyed at runtime.

51) Using Closures Sparingly: Closures are a very powerful and useful aspect of JavaScript, but they come with their own performance drawbacks. Closures can basically be thought of as the new in JavaScript, and we use one whenever we define a function on the fly, for instance:document.getElementById('foo').onclick = function(ev) { };

The problem with closures is that by definition, they have a minimum of three objects in their scope chain: 'the closure variables' 'the local variables', and 'the globals'. This last item leads to all the performance problems.

52) Don't Dig too Deep into Arrays: Deeper we dig, the slower the operation, because array item lookups are slow. If we dig three levels into an array, that's three array item lookups instead of one. So if we constantly reference foo.bar we can get a performance boost by defining var bar = foo.bar;

Dealing with CSS #(53-57): Inefficient handling of CSS will degrade the performance by consuming more time.

53) Avoid CSS Expressions: CSS expressions are a powerful (and dangerous) way to set CSS properties dynamically. The problem with expressions is that they are evaluated more frequently.Not only are they evaluated whenever the page is rendered and resized, but also when the page is scrolled and even when the user moves the mouse over the page.

Two techniques for avoiding problems created by CSS expressions are 'Creating One-Time Expressions' and using 'Event Handlers' instead of CSS expressions.

54) Prefer HTML <link> Tag over @import Directive: To refer to an external style sheet file from HTML file <link> tag or @import directive can be used. @import will cause wait event until the whole page has been read and rendered before loading and applying the external reference from the @import directive.

55) Speed-up Table Layouts: Rendering HTML tables is notoriously labor-intensive on the browser. There are typically four or more levels of tag nesting that, alone, take time to parse. The browser then needs to calculate the widths of each table cell, which it attempts to do intelligently, based on the width of the contents of each cell in the table. This means the browser usually needs to read in the data for the entire table before it is able to render it correctly. On a slow page, where we see the HTML table loading progressively onto the page, we might observe that the layout of the table alters—some of the columns change width—as the data is still being downloaded, and settles on its final dimensions only when all the data has completed downloading.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

The CSS table-layout:fixed style rule reduces this constant calculation and re-evaluation work the browser must do to render our HTML tables. It's used as follows:

```
table {table-layout:fixed;        }
```

The table-layout: fixed rule "fixes" the table layout, which means that the rendering engine will calculate the widths of the table cells based solely on data found in the cells contained in the header row of the table. Since it does not need to do any more cell- width calculations, the time taken to render a long table is reduced.

56) Avoid Inefficient CSS Selectors: The CSS selector provides the mechanism for the browser to apply our style rules to our page. Typically, the browser will search for style rules to apply to a page element as it is created, comparing the tag name, id attribute, and class attribute of the element to the list of style rules. The browser then searches for style rules to apply based on possible inherited values from other CSS selectors in the document; and combine the results giving precedence to more specific style rules or rules given later in the document. The browser effectively needs to perform a series of searches to locate the styles to apply to an element, and each of these searches takes a certain length of time to complete.

Avoid specifying selectors that include tag names, as these cause the longest style rule searches to take place. Instead, filter our selectors by an appropriate id attribute located on a parent element. The search does not need to attempt to look for style rules applied to any other id attribute, thus improving the speed at which results are returned.

57) Change CSS Classes not Styles: Changing CSS class is more optimal than changing style. This boils down to another reflow issue: whenever a layout style is changed, reflow occurs. Layout styles mean anything that might affect the layout and force the browser to reflow; for instancewidth, height, font-size, float etc. CSS classes don't avoid reflow, they simply minimize it. Instead of incurring a reflow penalty every time we change a given style, we can use a CSS class to change a number of styles at once, and in turn only incur a single reflow. So it makes performance sense to use CSS classnames whenever changing more than one layout style. Additionally, it's also optimal to append a style node to the DOM if we need to define a number of classes on the fly.

Don't Repeat Work #(58-59): The concept of work avoidance is made up of two things: don't do work that isn't required, and don't repeat work that has already been completed. The first part is usually easy to identify as code is being refactored. The second part—not repeating work is usually more difficult to identify because work may be repeated in any number of places and for any number of reasons.

58) Lazy Loading: This eliminates work repetition in functions through lazy loading. Lazy loading means that no work is done until the information is necessary. For Example: There is no need to determine which way to attach or detach event handlers until someone makes a call to the function.

Calling a lazy-loading function always takes longer the first time because it must run the detection and then make a call to another function to accomplish the task. Subsequent calls to the same function, however, are much faster since they have no detection logic. Lazy loading is best used when the function won't be used immediately on the page.

Lazy Load is also capable of loading CSS files dynamically. This is typically less of an issue because CSS file downloads are always done in parallel and don't block other page activities.

59) Conditional Advance Loading: This is an alternative to Lazy Loading, which does the detection upfront, while the script is loading, instead of waiting for the function call. The detection is still done just once, but it comes earlier in the process.

Conditional advance loading ensures that all calls to the function take the same amount of time. The trade-off is that the detection occurs as the script is loading rather than later. Advance loading is best to use when a function is going to be used right away and then again frequently throughout the lifetime of the page.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

Building and Deploying High-Performance JavaScript Applications #(60-67): There exists a need to make our web applications are delivered as efficiently as possible. While part of that work is done during Design and Development cycles, Build and Deployment phase is also essential and often overlooked. If care is not taken during this phase, performance of application will suffer, no matter how much effort we've put into making it faster.

60) Combining Java Script Files: Combining JavaScript files to reduce the number of HTTP requests. Simple optimization would be to group some, if not all, of this code into one external JavaScript file, thereby dramatically cutting down the number of HTTP requests necessary to render the page.

61) Pre-processing JavaScript Files:Pre-processing our JavaScript source files will not make application faster by itself, but it will allow among other things, conditionally instrument our code in order to measure how our application is performing.

62) JavaScript Minification: JavaScript Minification is the process by which a JavaScript file is stripped of everything that does not contribute to its execution. This includes comments and unnecessary whitespace. The process typically reduces the file size by half, resulting in faster downloads, and encourages programmers to write better, more extensive in-line documentation.

YUI Compressor is a tool that performs all kinds of smart operations in order to offer a higher level of compaction than other tools in a completely safe way. In addition to stripping comments and unnecessary whitespace, the YUI Compressor offers the following features:

a) Replacement of local variable names with shorter (one-, two-, or three-character) variable names, picked to optimize 'gzip' compression downstream.

b) Replacement of bracket notation with dot notation whenever possible (e.g., foo["bar"] becomes foo.bar)

c) Replacement of quoted literal property names whenever possible (e.g., {"foo":"bar"} becomes {foo:"bar"}).

d) Replacement of escaped quotes in strings (e.g., 'aaa\'bbb' becomes "aaa'bbb").

e) Constant folding (e.g., "foo"+"bar" becomes "foobar").

63) Build time vs. Runtime Build Processes: Concatenation, pre-processing and minification are steps that can take place either at build time or at runtime. Runtime build processes are very useful during development, but generally are not recommended in a production environment for scalability reasons. As a general rule for building high-performance applications, everything that can be done at build time should not be done at runtime.

64) Caching JavaScript Files: Making HTTP components cacheable will greatly enhance the experience of repeat visitors to our website. Although caching is most often used on images, it should be used on all static components, including JavaScript files.

We can also consider using client-side storage mechanisms if they are available, in which the JavaScript code must itself handle the expiration.

Another technique is the use of the HTML 5 offline application cache.

65) Caching Issues: Adequate cache control can really enhance the user experience, but it has a downside: when revving up our application, we want to make sure our users get the latest version of the static content. This is accomplished by renaming static resources whenever they change.

66) Using a Content Delivery Network: A content delivery network (CDN) is a network of computers distributed geographically across the Internet that is responsible for delivering content to end users. The primary reasons for using a CDN are reliability, scalability, and above all, performance. In fact, by serving content from the location closest to the user, CDNs are able to dramatically decrease network latency.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

Switching to a CDN is usually a fairly simple code change and has the potential to dramatically improve end-user response times. It is worth noting that the most popular JavaScript libraries are all accessible via a CDN.

67) Deploying JavaScript Resources: Deployment of JavaScript resources usually amounts to copying files to one or several remote hosts, and also sometimes to running a set of shell commands on those hosts, especially when using a CDN to distribute newly added files across delivery network.

Efficient Profiling and Debugging #(68-69): Code must be precisely profiled & debugged to produce robust code.

68) Naming Anonymous Functions for Profiling: Depending on the profiler, some data can be obscured by the use of anonymous functions or function assignments. As this is a common pattern in JavaScript, many of the functions being profiled may be anonymous, making it difficult or impossible to measure and analyse. The best way to enable profiling of anonymous functions is to name them. Using pointers to object methods rather than closures will allow the broadest possible profile coverage.

69) Using Uncompressed Version for Debugging: Always use uncompressed versions of scripts for debugging and profiling. This will ensure that our functions are easily identifiable.

70) Script Blocking: Browsers limit script requests to one at a time. This is done to manage dependencies between files. As long as a file that depends on another comes later in the source, it will be guaranteed to have its dependencies ready prior to execution. The gaps between scripts may indicate script blocking. Newer browsers have addressed this by allowing parallel downloading of scripts but blocking execution, to ensure dependencies are ready. Although this allows the assets to download more quickly, page rendering is still blocked until all scripts have executed.

Script blocking may be compounded by slow initialization in one or more files, which could be worthy of some profiling, and potentially optimizing or refactoring. The loading of scripts can slow or stop the rendering of the page, leaving the user waiting. Network analysis tools can help identify and optimize gaps in the loading of assets. Visualizing these gaps in the delivery of scripts gives an idea as to which scripts are slower to execute. Such scripts may be worth deferring until after the page has rendered, or possibly optimizing/refactoring to reduce execution time.

Tools #(71-73): We should have the right software essential for identifying bottlenecks in both the loading and running of scripts. Using profiling tools prior to beginning optimization will ensure that development time is spent focusing on right problem.

71) Page Speed Tool: Page Speed provides information about resources being loaded on a web page. But, in addition to load time and HTTP status, it shows the time spent on parsing and executing JavaScript, identifies deferrable scripts, and reports on functions that aren't being used. This is valuable information that can help identify areas for further investigation, optimization, and possible refactoring.

The Profile Deferrable JavaScript option, available on the Page Speed panel, identifies files that can be deferred or broken up in order to deliver a smaller initial payload. The Page Speed also provides a report on which functions were not called at all and which functions may be delay-able, based on the time they were parsed versus the time they were first called.

72) Fiddler: Fiddler is an HTTP debugging proxy that examines the assets coming over the wire and helps identify any loading bottlenecks.

Although 'time spent' and 'number of calls' are usually the most valuable bits of data, looking more closely at how functions are being called might yield other optimization candidates.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

73) YSlow: YSlow tool provides performance insights into the overall loading and execution of the initial page view. It scores the loading of external assets to the page, provides a report on page performance, and gives tips for improving loading speed.

74) Reducing DNS Lookups: When the client's DNS cache is empty (for both the browser and the operating system), the number of DNS lookups is equal to the number of unique hostnames in the web page. This includes the hostnames used in the page's URL, images, script files, style sheets, Flash objects, etc. Reducing the number of unique hostnames reduces the number of DNS lookups.

Reducing the number of unique hostnames has the potential to reduce the amount of parallel downloading that takes place in the page. Avoiding DNS lookups cuts response times, but reducing parallel downloads may increase response times.

In a nut shell minimize DNS lookups by using Keep-Alive and fewer domains.

Evading Redirects #(75): A redirect is used to reroute users from one URL to another, which hurts performance.

75) Alternatives to Redirects: The following are several alternatives to Redirects.

a) Missing Trailing Slash: One of the most wasteful redirects happens frequently and web developers are generally not aware of it. It occurs when a trailing slash (/) is missing from a URL that should otherwise have one.

b) Connecting Web Sites: Imagine situation where a web site backend is rewritten. As often happens, the URLs in new implementation might be different.An easy way to transition users from theold URLs to new ones is to use redirects. Redirects are a way of integrating the two code bases using a well-defined API: URLs.Although redirects reduce the complexity for developers, it degrades the user experience.

c) Tracking Internal Traffic: Redirects are often used to track the flow of user traffic. For internal traffic—i.e., traffic from web sites within the same company—it's worthwhile to avoid redirects by setting up Referrer logging to improve end user response times.

d) Tracking Outbound Traffic: When we're trying to track user traffic, we might find that links are taking users away from our web site. An alternative to redirects for outbound traffic is to use a beacon—an HTTP request that contains tracking information in the URL.The tracking information is extracted from the access logs on the beacon web server(s).

Enhancing End User Response Time #(76-89): Measures should be taken to minimize End User Wait time.

76) Use Separate Domain Names for External Assets: We should consider setting up a second domain name for our web application, if we intend to use HTTP cookies within our site.

By creating two separate domain names for a site, we can choose to host all images, style sheets, JavaScript files, and other external assets from one domain name, and all HTML from the other. By having the cookie data saved only to the domain that hosts the HTML files, we ensure that the cookie data is not sent to the server when requesting any other file type. This makes the data sent in the HTTP request message smaller, allowing it to reach the web server faster.

This technique is recommended only where we rely on cookie data being stored in the user's browser. In other circumstances, the extra DNS lookup, to locate the IP address of the second domain name, would actually result in worse performance for the end user, so this technique should be used carefully.

77) Send HTML to the Browser in Chunks: Most web servers aren't serving flat HTML files to their end users. Typically, the user requests a file, which causes the server to perform some action. The result of that action is the piecing together of a final HTML file, which is then sent to the user. Those actions are typically written in server- side scripting languages such as PHP, ASP.NET, or Java Server Pages (JSP).

After making a request to the server to view a page, the browser is left waiting while the server pieces together the HTML code to send back. If we could get access to some part of this resulting HTML as soon as possible, the browser could begin its work of loading in external assets, such as style sheets, while waiting for the server to send the

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

remainder of the HTML page. This means the whole page will be ready for the end user to interact with sooner than if the browser had to wait for the entire HTML page to be sent in a single, large chunk.

78) Compress the Output from the Server: The biggest web server performance improvement can be achieved by compressing our text-based content-HTML, style sheets, and JavaScript files-before it is sent to the browser, letting the browser decompress the data before displaying it. Compression involves encoding data in a different, more efficient way. Compression can reduce file sizes by upto 70% of original. The two popular compression algorithms are known as gzip and deflate.

The great thing about this technique is that it is simple to enable on server. It is supported by every modern web browser. Browsers that don't support compression technique will be sent the data uncompressed from the server instead, so data is never lost. We should enable this feature in our web server, if it is not set by default, at our earliest available opportunity. We will immediately notice its impact, and so will our end users.

79) Don't Load Every Asset from the Home Page: Our home page is most often the first page our visitors will see when they visit our web site. We need to make a good impression with this page, both with its layout and its performance.

We may make the decision that, in order to speed up the rest of our web site, we are going to load all of the CSS and JavaScript for the entire site from the home page. These files will then be stored in the browser cache, so subsequent page requests will be faster. Unfortunately, by weighing down the site- entry page in this way, we cause a lot more data than is necessary to be downloaded, slowing down the rendering of what is the most important page on the site. Only reference what we absolutely need to on our home page, as it is the one that, above all others, needs to be perfect from a performance perspective.

80) Split Components Across Domains: The HTTP 1.1 specification includes a recommendation relating to the number of simultaneous files the browser should be able to download in parallel from the same domain name: it suggests a limit of two concurrent files. Some browsers enforce this limit to the letter; some choose to allow more than two simultaneous downloads from the same domain name.

The browser is limited in the number of simultaneous requests per domain name. So we have to set up another domain, point its DNS record to the same IP address as the first domain—since the limitation is only by domain name, not by IP address—so they are both referencing the same code on the same web server. By dividing our assets among the available domain names in this way, we are able to introduce a potentially large performance boost to our pages.

81) Reduce the Number of HTML Elements: The number of HTML elements on our page not only affects how long our HTML takes to download to the browser, but also the performance of our JavaScript code and the speed with which the browser's rendering engine is able to apply our CSS to our page.

82) Define Arrays for HTML Collection Objects:JavaScript uses a number of HTML collection objects such as document.forms.document.images, etc. Additionally these are called by methods such as getElementsByTagName and getElementsByClassName. As with any DOM selection, HTML collection objects are pretty slow, but also come with additional problems. As the DOM Level 1 spec says, "collections in the HTML DOM are assumed to be live, meaning that they are automatically updated when the underlying document is changed". While collection objects look like arrays, they are something quite different: the results of a specific query. Anytime this object is accessed for reading or writing, this query has to be rerun, which includes updating all the peripheral aspects of the object such as its length.

HTML collection objects are extremely slow. Additionally, these collection objects can lead to infinite loops where we might not expect. For instance:

```
vardivs = document.getElementsByTagName('div');
for (var i=0l i <divs.length; i++ ) {  var div = document.createElement("div");      document.appendChild(div); }
```

This causes an infinite loop because divs represents a live HTML collection, rather than an array like we might expect. This live collection is updated every time we append a new to the DOM, so i <divs.length never terminates.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

The way around this is to define these items in an array, which is a little more complex than just setting vardivs = document.getElementsByTagName('div');. Here is a script that uses to force an array:

```
function array(items) {      try {      return Array.prototype.concat.call(items);      }
catch (ex) {      var i = 0,   len = items.length,   result = Array(len);
while (i <len) {      result[i] = items[i];      i++;      }
return result;      }      }
vardivs =          array( document.getElementsByTagName('div'));
for (var i=0; i<divs.length; i++ ) {var div = document.createElement("div");document.appendChild(div);      }
```

83) Creating Reference Variables: When working with a specific node recurrently, it is optimal to define a variable with that particular note, instead of switching to it repeatedly. This is not a significant enhancement but it can have a bigger impact on a large scale.

84) Compress Files with GZip: GZip can reduce a JavaScript file considerably, saving bandwidth, and accelerate the response time. JavaScript files can be very large, and without compression, it can bog down any website. Smaller files provide a faster and more satisfying web experience.

85) Using JQuery as Framework:JQuery is an easy to use JavaScript library that can help to speed up any website. JQuery provides a large number of plug-ins that can quickly be used.

86) Don't Link to Non-existent Files: As we've already seen, HTTP requests that don't return an intended response are wasteful; they keep the web server and the browser occupied for little benefit.

If a JavaScript file is being requested by the page that does not exist on the server, and an error page is returned by the server instead of the requested file, the browser will pause its parsing of the page while waiting for the contents of the error page to download. This exacerbates the effect of linking to non-existent files.

87) Reduce Size of HTTP Cookies: Each HTTP request message that is sent from the browser to server contains the HTTP cookie data associated with that domain. This is so that both the client, through JavaScript, and the server, through any back-end processing, have access to this information in order to customize the site for that particular user. Obviously, this data takes up space in the request message. Since it is communicated for each file being requested, it can add a lot of extra information that is rarely used except, typically, for the page that returns the HTML document.

By ensuring that the data stored in the cookie is small, the data being sent in each HTTP request message is reduced, and the server receives the message sooner.Consider including nothing more than a simple, unique user identifier within the HTTP cookie and storing other information within a database on the server, using the unique identifier as a key to look up this extra information.

88) Consider PNG Images: The three major image file formats used on web pages are Graphics Interchange Format (GIF), Joint Photographic Experts Group format (JPEG), and Portable Network Graphics format (PNG). PNG is a lossless image format apart from being smaller in size.

89) Configuring ETags: Entity tags (ETags) are a mechanism that web servers and browsers use to validate cached components. Reducing the number of HTTP requests necessary to render our page is the best way to accelerate the user experience. We can achieve this by maximizing the browser's ability to cache the components, but the ETag header thwarts caching when a web site is hosted on more than one server. Most of the components in the page have ETags that follow the default format for IIS. The same images downloaded from different servers have different ETags, meaning they will be downloaded more frequently than needed. Extra effort can be made to modify the ETag syntax to improve their cacheability.Simply Configure ETags efficiently or remove them.

Evaluating the Time Complexity #(90): Simple code snippet can compute the amount of time the code is under execution.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

90) Tracking the Running Time of Code: Tracking how long a piece of code has been running by using the native Date object. This is the way most JavaScript profiling works.

IV. EXPERIMENTAL SETUP

In this section the methodologies which correspond to section 3 are explained via pragmatic programs and code snippets which could serve as exemplars.

```
1. <html><head><title>Script Tuning</title><link rel="stylesheet" type="text/css" href="styles.css">
</head><body><p>Hello world!</p>
<-- Example of recommended script positioning -->
<script type="text/javascript" src="file1.js"></script><script type="text/javascript" src="file2.js"></script>
</body></html>
```

Instead of

```
<html><head><title>Script Tuning</title>
<script type="text/javascript" src="file1.js"></script><script type="text/javascript" src="file2.js"></script>
<link rel="stylesheet" type="text/css" href="styles.css">
</head><body><p>Hello world!</p></body></html>
```

Since each <script> tag blocks the page from continuing to render until it has fully downloaded and executed the Java- Script code, the perceived performance of this page will suffer.

```
2. <html><head><title>Script Example</title>
<link rel="stylesheet" type="text/css" href="styles.css"></head><body>
<p>Hello world!</p>
<-- Example of recommended script positioning -->
<script type="text/javascript" src="
  http://yui.yahooapis.com/combo?2.7.0/build/yahoo/yahoo-min.js&2.7.0/build/event/event-min.js ">
</script></body></html>
```

This code has a single <script> tag at the bottom loading multiple Java-Script files; the best practice for including external JavaScript on an HTML page.

```
4. <script type="text/javascript" src="loader.js"></script>
<script type="text/javascript">
loadScript("the-rest.js", function(){
Application.init();
  });
</script>
```

Place this loading code just before closing </body> tag. Now execution won't prevent the rest of the page from being displayed. Second, when the second JavaScript file has finished downloading, all of the DOM necessary for the application has been created and is ready to be interacted with, avoiding the need to check for another event (such as window.onload) to know when the page is ready for initialization.

```
11. if (obj.a===undefined){   a = obj.b === undefined ? b : onj.b;     }
else  {obj.a = obj.b === undefined ? b : obj.b;     }
```

Instead of

```
with(obj ) {          a=b;      }
```

```
24. functionsaveDocument(id){
var tasks = [openDocument, writeText, closeDocument, updateUI];
setTimeout(function(){
```

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

```
    //execute the next task
var task = tasks.shift();
task(id);
    //determine if there's more
    if (tasks.length> 0)        {            setTimeout(arguments.callee, 25);        }     }, 25);     }
                Instead of
functionsaveDocument(id){
   //save the document
openDocument(id);writeText(id);    closeDocument(id);
   //update the UI to indicate success
updateUI(id);
   }
```

If function is taking too long, it must be split up into a series of smaller steps by breaking out the individual methods into separate timers. We can accomplish this by adding each function into an array and then using a pattern similar to the array processing pattern from the previous section.

29. Example of an XHR wrapper that first checks to see whether a URL has been fetched before:

```
varlocalCache = {};
functionxhrRequest(url, callback) {     // Check the local cache for this URL.
if (localCache[url]) {     callback.success(localCache[url]);     return;     }
    // If this URL wasn't found in cache, make the request.
var req = createXhrObject();
req.onerror = function() {     callback.error();     };
req.onreadystatechange = function() {
if (req.readyState == 4) {
if (req.responseText === '' || req.status == '404') {
callback.error();
return;
    }
    // Store the response on the local cache.
localCache[url] = req.responseText;
callback.success(req.responseText);
    }
    };
req.open("GET", url, true);
req.send(null);
    }
```

Setting an Expires header is a better solution. It's easier to do and it caches responses across page loads and sessions. But a manual cache can be useful in situations where we programmatically want to expirea cache and fetch fresh data.

30. Example for 4 methods

```
var num1 = 5,     num2 = 6,
    //eval() evaluating a string of code
result = eval("num1 + num2"),
    //Function() evaluating strings of code
sum = new Function("arg1","arg2","return arg1+arg2");
    //setTimeout() evaluating a string of code
setTimeout("sum = num1 + num2", 100);
    //setInterval() evaluating a string of code
```

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

```
setInterval("sum = num1 + num2", 100);

        var item = array[0];
                Instead of
        var item = eval("array[0]");
```

31.
```
//creating an object
varmyObject = {
name: "Vamsi", count: 50, flag: true, pointer: null };      //create an array
varmyArray = ["Vamsi", 50, true, null];
        Instead of
// creating an object
varmyObject = new Object();
myObject.name = "Nicholas";      myObject.count = 50;      myObject.flag = true;      myObject.pointer = null;
        Or
// creating an array
varmyArray = new Array();
myArray[0] = "Nicholas";      myArray[1] = 50;      myArray[2] = true;      myArray[3] = null;
```

53. One-Time Expression: If the CSS expression has to be evaluated only once, it can overwrite itself as part of its execution. The background style defined at the beginning of this chapter is a good candidate for this approach:

```
<style>
P {      background-color: expression(altBgcolor( this) );      }
</style>
<script type="text/javascript">
functionaltBgcolor(elem) {
elem.style.backgroundColor = (new Date()).getHours()%2 ? "#F08A00" : "#B8D4FF";
  }      </script>
```

CSS expression calls altBgcolor( ) function, which sets style's background-color property to an explicit value, and this replaces CSS expression. This style is associated with 10 paragraphs in page. Even after resizing, scrolling, and moving mouse around the page, CSS expression is evaluated only10 times.

Event Handlers: Below setMinWidth( ) function to resize all paragraph elements when browser is resized:

```
functionsetMinWidth( ) {
setCntr( );
varaElements = document.getElementsByTagName("p");
for ( var i = 0; i <aElements.length; i++ ) {
aElements[i].runtimeStyle.width = (  document.body.clientWidth<600 ? "600px" : "auto" );
 }
}
if ( -1 != navigator.userAgent.indexOf("MSIE") ) {          window.onresize = setMinWidth;      }
```

56.
```
#header .title { ... }
    #results a { ... }
    #field .label { ... }
        Instead of
tabletbodytr td { ... }
     #body #content #results a { ... }
formdiv#field label { ... }
```

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

58. Lazy Loaded Version

```
functionaddHandler(target, eventType, handler){
if (target.addEventListener){ //DOM2 Events
addHandler = function(target, eventType, handler){
target.addEventListener(eventType, handler, false);
    };
    } else {    //IE
addHandler = function(target, eventType, handler){    target.attachEvent("on" + eventType, handler);    };    }
    //call the new function
addHandler(target, eventType, handler);    }
functionremoveHandler(target, eventType, handler){
if (target.removeEventListener){ //DOM2 Events
removeHandler = function(target, eventType, handler){
target.addEventListener(eventType, handler, false);
    };
    } else {    //IE
removeHandler = function(target, eventType, handler){  target.detachEvent("on" + eventType,
                                                                                    handler);    };    }
    //call the new function
removeHandler(target, eventType, handler);    }
        Instead of
functionaddHandler(target, eventType, handler){
if (target.addEventListener){ //DOM2 Events
target.addEventListener(eventType, handler, false);
    } else {    //IE
target.attachEvent("on" + eventType, handler);    }    }
functionremoveHandler(target, eventType, handler){
if (target.removeEventListener){ //DOM2 Events
target.removeEventListener(eventType, handler, false);
    } else {    //IE
target.detachEvent("on" + eventType, handler);    }    }
```

59. varaddHandler = document.body.addEventListener ?

```
function(target, eventType, handler){
target.addEventListener(eventType, handler, false);
    }:
function(target, eventType, handler){
target.attachEvent("on" + eventType, handler);
    };
varremoveHandler =  document.body.removeEventListener ?
function(target, eventType, handler){
target.removeEventListener(eventType, handler, false);
    }:
function(target, eventType, handler){
target.detachEvent("on" + eventType, handler);    };
```

This example checks to see whether addEventListener() and removeEventListener() are present and then uses that information to assign the most appropriate function. The ternary operator returns the DOM Level 2 function if these methods are present and otherwise returns the IE-specific function. The result is that all calls to addHandler() and removeHandler() are equally fast, as the detection cost occurs upfront.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

68. Inline function: myNode.onclick = function() { myApp.loadData(); };
Method call:myApp._onClick = function() { myApp.loadData(); };
myNode.onclick = myApp._onClick;
Naming above corresponding Inline Function
myNode.onclick = function myNodeClickHandler() { myApp.loadData(); };
Naming above corresponding Method
varonClick = function myNodeClickHandler() { myApp.loadData(); };

90. The beneath code tells the time, the code snippet is consuming to get executed.

```
var start = +new Date(),stop;
    …
    // Code to be profiled goes here
    …
stop = +new Date();
alert("Time taken to Execute Code " + stop-start);
if(stop-start < 50){ alert("Just about right."); } else { alert("Taking too long."); }
```

V. CONCLUSION

JavaScript performance really begins with getting the code onto a page in the most efficient way possible. Code interpretation is inherently slower than compilation since there's a translation process between the code and the computer instructions that must be run. No matter how smart and optimized interpreters get, they always incur a performance penalty.

In order to know what to improve, we need to know where the user spends her time waiting. Page weight and response time are strongly correlated.

The ultimate goal of writing usable JavaScript code is to have a web page that will work for the users, no matter what browser they are using or what platform they are on. To accomplish this, we set a goal of the features that we want to use, and exclude any browsers that do not support them. For the unsupported browsers, we then give them a functional, albeit less interactive, version of the site. The benefits to writing JavaScript and HTML interactions in this manner included cleaner code, more accessible web pages, and better user interactions.

VI. REFERENCES

[1] Vamsi Krishna Myalapalli andSunithaGeloth, "High Performance JAVA Programming", IEEE International Conference on Pervasive Computing, Pune, January 2015.
[2] Vamsi Krishna Myalapalli, Jaya Krishna Myalapalli and Pradeep Raj Savarapu, "High Performance C Programming", IEEE International Conference on Pervasive Computing, Pune, January 2015.
[3] Nicholas C. Zakas , "High Performance JavaScript", Yahoo Press, First Edition, 2010.
[4] Den Odell, "Pro JavaScript RIA Techniques", Apress Publishers, 2009.
[5] Steve Sounders, "High Performance Web Sites", O Reilly Publishers, First Edition, 2007.
[6] Douglas Crockford, "JavaScript: The Good Parts", O'Reilly Publishers, 2008.
[7] 'http://en.wikipedia.org/wiki/JavaScript' referred on 27th December, 2014.

BIOGRAPHY

Vamsi Krishna completed his Bachelor's in Computer Science from JNTUK University College of Engineering, Vizianagaram. He is Microsoft Certified Professional (Specialist in HTML5 with JAVA Script & CSS3), Oracle Certified JAVA SE7 Programmer, DBA Professional 11g, Performance Tuning Expert, SQL Expert and PL/SQL Developer. He has fervor towards SQL & Database Tuning.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

Srinivas Karri completed his Bachelor's in Computer Science from Gayathri Vidya Parishad College of Engineering, Visakhapatnam. He is Microsoft Certified Professional (Specialist in HTML5 with JAVA Script & CSS3), Oracle Certified JAVA SE7 Programmer.