IJITKM Special Issue (ICFTEM-2014) May 2014 pp. 101-106 (ISSN 0973-4414)

Memory leaks in Web 2.0 Applications

Bhanwar Gupta[1], Dr. Puneet Goswami[2]
[1,2]GGITC Ambala
[1]bhanwargupta@gmail.com, [2] goswamipuneet@gmail.com

Abstract: Web 2.0 is a buzzword for the new definition of the Web and the Internet.Web2.0 provides the user with more user- interface, software and storage facilities, all through their browser. The client-side (Web browser) technologies used in Web 2.0 development include Ajax and JavaScript frameworks. JavaScript is a powerful scripting language used to add dynamic content to Web pages. While JavaScript is easy to learn and write, it is prone to memory leaks in certain browsers. In this paper we will explain what causes memory leaks in Web2.0 applications, demonstrate some of the common memory leak patterns to watch out for, and show how to work around them.
Keywords: Web 2.0, AJAX, Optimization, Performance, Javascript, Memory Leak, Javascript.

1. INTRODUCTION

Since the Web 2.0 era, Web Applications have changed for the better. They provide more functionality, better usability and improved performance. However this tendency led to increased complexity. It is not unusual for a user to stay on a single page for hours without leaving. The browser would just poll data from the server through Ajax requests and display it in even more interesting manners. Therefore it is no wonder that web applications have become hungrier for resources and the browsers couldn't keep up with it. The heavy utilization of JavaScript and DOM revealed the poor memory management of most of the major browsers at the time[3].

With Web2.0, the interaction model for web applications has become more robust, allowing for continuous interaction and improved usability. Adding Ajax capabilities to applications offers users a more interactive, responsive experience that can provide an organization with benefits such as fewer abandoned transactions, longer user sessions, and higher completion rates.

While these new technologies enable greater performance and code re-use, the benefits come at a cost – added complexity. With these new patterns, JavaScript memory management becomes an even more critical aspect of the development process.

A memory leak is a gradual loss of available computer memory. It occurs when a program repeatedly fails to return memory it has obtained for temporary use. JavaScript web apps can often suffer from similar memory related issues that native applications do.

Generally, browser memory leaks are not an issue for web applications. Users navigate between pages, and each page switch causes the browser to refresh. Even if there's a memory leak on one page, the leak is released after a page switch. The size of the leak is minor, and consequently often ignored.

Memory leaks became more of a problem when the Ajax technology was introduced. On a Web 2.0 style page, users don't refresh the page very often. Ajax technology is used to update the page content asynchronously. In an extreme scenario, the whole web application is constructed on one page. In this case, leaks accumulate and can't be ignored.

Although JavaScript uses garbage collection for automatic memory management, effective memory management is still important.

2. Memory Management in Web2.0 Applications

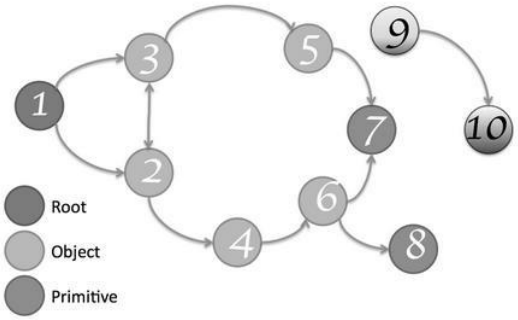
Low-level languages, like C, have low-level memory management primitives like malloc() and free() [6]. On the other hand, JavaScript values are allocated when things (objects, strings, etc.) are created and "automatically" free'd when they are not used anymore. The latter process is called garbage collection[2]. This "automatically" is a source of confusion and gives JavaScript (and high-level languages) developers the impression they can decide not to care about memory management which is a mistake.

2.1 Memory Consumption

Memory consumption stands for the amount of memory a particular program utilizes throughout its execution. This term is self-explanatory as every application relies on the underlying memory to store instances of variables [2]. The more the variables, the larger the memory consumption will be. Therefore it is a common assumption that more complicated applications require more memory.

2.2 Object Graph

All values in JavaScript are part of the object graph. The graph begins with roots, for example, the window object. Managing the lifetime of GC roots is not in our control, as they are created by the browser and destroyed when the page is unloaded. Global variables are actually properties on window.



2.3 Object Sizes

The memory graph starts with a root, which may be the window object of the browser or the Global object of a Node.js module.

An object can hold memory in two ways:

- Directly by the object itself
- Implicitly by holding references to other objects, and therefore preventing those objects from being automatically disposed by a garbage collector (GC for short).

2.4 Shallow Sizes

This is the size of memory that is held by the object itself. Typical JavaScript objects have some memory reserved for their description and for storing immediate values. Usually, only arrays and strings can have a significant shallow size. However, strings and external arrays often have their main storage in renderer memory, exposing only a small wrapper object on the JavaScript heap.

2.5 Retained Sizes

This is the size of memory that is freed once the object itself is deleted along with its dependent objects that were made unreachable. A distinguished set of objects are assumed to be reachable: these are known as the roots. Typically, these include all the objects referenced from anywhere in the call stack (that is, all local variables and parameters in the functions currently being invoked), and any global variables.

2.6 Garbage Collection

A garbage collector needs to be able to locate objects in the application which are live, as well as, those which are considered dead (garbage) and are unreachable.

If garbage collection (GC) misses any dead objects due to logical errors in your JavaScript then the memory consumed by these objects cannot be reclaimed. Situations like this can end up slowing down your application over time.

This often happens when code is written in such a way that variables and event listeners you don't require are still referenced by some code. While these references are maintained, the objects cannot be correctly cleaned up by GC.We need to check and nullify variables that contain references to DOM elements which may be getting updated/destroyed during the lifecycle of your app. Also, check object properties which may reference other objects (or other DOM elements).

3. LEAK PATTERNS

Due to the nature of JavaScript and browser memory management for JavaScript and DOM objects, carelessly coded JavaScript causes browser memory leaks. Two well- known patterns cause these leaks [3]

3.1 Circular References

Circular references are the root cause of nearly every leak. Generally, IE can handle the circular references

and dispose of them correctly in the JavaScript world. The exception occurs when DOM objects are introduced. Circular references occur and cause the DOM node to leak when the JavaScript object holds the reference to the DOM element, and the DOM element's property holds the JavaScript object. Listing 1 shows a code sample that's often used to demonstrate this problem in articles about memory leaks.
A circular reference is formed when two objects reference each other, giving each object a reference count of 1. In a purely garbage collected system, a circular reference is not a problem: If neither of the objects involved is referenced by any other object, then both are garbage collected. In a reference counting system, however, neither of the objects can be destroyed, because the reference count never reaches zero. In a hybrid system, where both garbage collection and reference counting are being used, leaks occur because the system fails to identify a circular reference. In this case, neither the DOM object nor the JavaScript object is destroyed. Listing 1 shows a circular reference between a JavaScript object and a DOM object.

```
<scripttype="text/javascript">
        document.write("The circular
references!");
        var object;
        window.onload = function(){

        obj=document.getElementById(" MyDivElement");

        document.getElementById("MyDi vElement").expandoProperty=obj;
                obj.bigString=new Array(2000).join(new Array(39393).join("JSJWUW"));
                                        };
        </script>
```

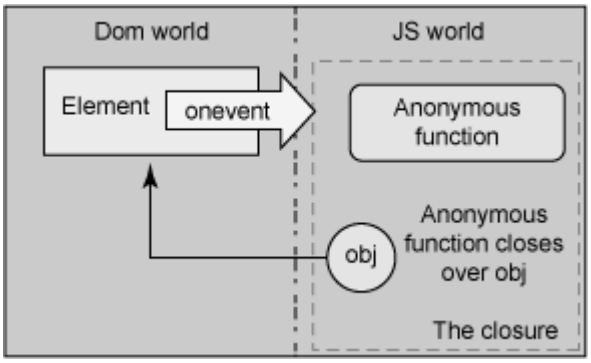
In the above listing, the JavaScript object obj has a reference to the DOM object represented by DivElement. The DOM object, in turn, has a reference to the JavaScript object through the expandoProperty. A circular reference exists between the JavaScript object and the DOM object. Because DOM objects are managed through reference counting, neither object will ever be destroyed.

3.2 Closures in Web2.0

One of JavaScript's strengths is that it allows functions to be nested within other functions. A nested, or inner, function can inherit the arguments and variables of its outer function, and is private to that function. JavaScript developers use inner functions to integrate small utility functions within other functions [3].
Closures cause memory leaks because they create circular references without being aware of it. The parent function's variable will be held as long as the closure is alive. Its life cycle goes beyond the function scope, which will cause a leak if not handled carefully.

```
<script type="text/javascript">
        document.write("The
closures");
        window.onload=
        function        parent(A)
        {
                var a = A;
                return function
innerFunct (paramB)
                                        {
        alert( a +" "+ B);
                                        };
        };
        var x = parent("the outer value");
        x("inner value");
        </script>
```

In the above listing innerFunct is the inner function defined within the parent function parent. When a call is made to parent with a parameter of outer x, the outer function variable a is assigned the value outer x. The function returns with a pointer to the inner function innerFunct, which is contained in the variable x.



It is important to note that the local variable a of the outer function parent will exist even after the outer function has returned. This is different from programming languages such as C/C++, where local variables no longer exist once a function has returned. In JavaScript, the moment parent is called, a scope object with property a is created. This property contains the value of paramA, also known as "outer x". Similarly, when the parent returns, it will return the inner function innerFunct, which is contained in the variable x. Because the inner function holds a reference to the outer function's variables, the scope object with property a will not be garbage collected. When a call is made on x with a parameter value of inner x -- that is, x("inner x") -- an alert showing "outer x innerx" will pop up.
Closures are powerful because they enable inner functions to retain access to an outer function's variables even after the outer function has returned. Unfortunately, closures are excellent at hiding circular references between JavaScript objects and DOM objects.

4. AVOIDING MEMORY LEAKS

Memory leaks can be very hard to indentify. Most of the time
this is because the amount of memory that leaks is so small we won't see it all at once on the process task manager. There are things we can do that will prevent memory leaks or at least allow the page to be fully collected when the page reloads. It seems in my research that it is very hard to get 100% leak free in all browsers. But we will be able to reduce the amount of leaks to a minimal level. With that in mind it will help us to know the best practices to avoid the major risk of memory leaks.
Leaks require prevention in general, not reactive repair on a running system. If a developer's JavaScript code somehow misses a leak, it is not possible to automatically fix it [on the user's system after the fact]. If a developer did have a big table that carelessly stored references to all objects ever created, of course, you might treat it as a cache and purge old object references from it, but that's not really a leak, just a cache that needs careful tuning and reclamation policy implemented by your JavaScript.
Memory tools can also help. A memory profiler can show which objects are taking up the most memory. Then, a garbage-collection tracing tool can tell which program values are causing those objects to be kept.

4.1 Breaking the circular references

This is to make the JavaScript object obj null, thus explicitly
breaking the circular reference

```
document.write("Breaking the circular reference");
window.onload=function outer(){
var obj = document.getElementById("element");

obj.onclick=functioninner()
{

alert("Avoid the leak");
obj.bigString=new Array(2828).join(new Array(1000).join("SJSJS"));
obj = null; //This
breaks the circular reference
};
```

4.2 Avoiding the closures

Closures are functions that refer to independent (free)
variables.

```
document.write("Avoiding the closures!"); window.onload=function()
{
var obj =
document.getElementById("element");
obj.onclick = noLeak;
}
function noLeak()
{
alert("Avoid the leak");
};
}
```

4.3 Nullifying the javascript reference to a DOM node.

Assume another repository that you don't know about is available and holding the references to your widgets, and that, for some reason, it can't be cleaned. Removing the widget will cause the DOM node referenced by it to be orphaned.

```
widgetArray = [];
widgetRepo = {};

function createWidget() {
        var container =
dojo.byId("widgetContainer");
        var widget = new
leak.sample.MyWidget(container); widgetArray.push(widget); widgetRepo[widget.ID] =
widget;
}
```

4.4 Nullify the DOM references

The solution is to nullify the DOM node references during clean-up. It is considered a good practice to add these nullify statements when possible since it won't harm your original function.

```
## the destroy method of MyWidget class
destroy: function() {
        dojo.destroy(dojo.byId(this.I D));
        this.domNode = null;
this.container = null;}
```

4.5 Setting innerHTML

Memory leaks can be caused if we aren't careful with how we set innerHTML with JavaScript.

```
var elem =
document.createElement(“DIV”);

// now the node is not orphan anymore document.body.appendChild(elem);

elem.innerHTML = “<a onclick=’alert(1)’>no leak</a>”;
```

5. CONCLUSION

In this paper we have presented patterns that lead to memory leaks in JavaScript applications, approaches to finding and fixing them, and recommended practices for avoiding them in the first place.

If we are developing client-side re-usable scripting objects, sooner or later we will find ourselves spotting out memory leaks. Chances are that our browser will suck memory like a sponge and we will hardly be able to find a reason why our Web 2.0 responsiveness decreases severely after visiting a couple of pages. This

paper demonstrates some of the common memory leak patterns to look for, and show how to work around them.

6. REFERENCES

[1] Schlueter. Memory leaks in Microsoft Internet Explorer, 2007. URL - http://foohack.com/2007/06/msie-memory- leaks/

[2] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. IEEE Softw., 25 (5):22–29, Sept. 2008.

[3] A. Bhattacharya and K. S. Sund. Memory leak patterns inJavaScript. http://www.ibm.com/developerworks/web/ library/wa-memleak/, 2007.

[4] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. POPL '07, pages 31–38, New York, NY, USA, 2007. ACM.

[5] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. SIGPLAN Not., 42(6):480–491, June 2007.

[6] Dr. Puneet Goswami paper titled "Security in Cloud Reference Models and Secure Identity Management Mechanism" 1st International IBM Cloud Academy Conference ICA CON 2012 on Thursday and Friday, April 19-20, 2012 9:00 AM - 5:30 PM at the IBM Employee Activity and Fitness Center Building 400, Cornwallis Drive Research Triangle Park, North Carolina Organized by : The IBM Cloud Academy and IBM Centers for Advanced Studies (RTP, Chicago, Heritage Corridor, Tucson, Florida).

[7] Puneet Goswami "Intelligent Web Crawler: software technology for the new millenium". Published at the proceedings of national conference Israna Panipat on 16th and 17th of march2006