As web users, we know that the speed at which a page loads or refreshes is paramount to its success. This article will help you gain a better understanding of the factors that contribute to your web application's performance. Learn to identify these issues and to find the bottlenecks with your client-side content. Explore performance problems, and solutions, for JavaScript, DOM, CSS, and Dojo widgets. An example walks you through fine-tuning a Dojo widget by using YSlow and Firebug.

Overview

Rich Internet applications (RIAs) are very popular in the Web 2.0 domain. To provide new and fancy user experiences, many websites move their complex content from back-end servers to the client side using JavaScript or Flash. This provides a convenient, fashionable, and smooth user interface (UI) if the data size is fairly small. If a large amount of content must be transmitted from the server to a client and rendered in a browser, the performance will decrease noticeably. The challenge is to find the bottleneck and identify a solution.

It's harder to tune performance issues with browsers rather than in Java applications. Developers have fewer ways to debug JavaScript in every browser. With Mozilla Firefox, you can use Firebug to debug JavaScript, but you still cannot tune many performance issues, such as browser rendering consumption time. To fix these problems, it is essential to develop browser plug-ins to monitor the time response, as well as to identify other corresponding solutions like partial rendering or lazy load.

Learn to diagnose performance problems with your web applications, find the bottlenecks in your client-side content, and tune the performance.
JavaScript and HTML DOM

JavaScript is the most popular scripting language used on the Internet. Millions of web page designers use JavaScript to improve their design, validate input forms, inspect browsers, and create cookies. HTML Document Object Model (DOM) defines the standard methods to access and operate HTML documents. It represents the HTML document as a node tree, which contains elements, properties, and text content.

By using HTML DOM, JavaScript can get access to all of the nodes inside an HTML document and then operate on them. JavaScript and the HTML DOM are supported by almost all mainstream browsers and are used by most websites. Performance during their usage significantly impacts the overall performance of RIAs.

JavaScript performance and functions

With JavaScript, when you need a function, use a function. Although you can use a string instead of a function in some situations, we suggest that you use a function whenever possible. Functions are pre-compiled before they are used in JavaScript.

For example, look at the `eval` method shown in Listing 1.

### Listing 1. `eval` method using string as parameter

```javascript
function square(input) {
    var output;
    eval('output=(input * input)');//
    return output;
}
```

The `eval` method squares the input and returns the output value, but the performance won’t be very good. The example uses the string `output=(input*input)` as a parameter of the `eval` method, which can’t take advantage of the JavaScript pre-compile.

Listing 2 shows a better way to do the job.

### Listing 2. Eval method using function as parameter

```javascript
function square(input) {
    var output;
    eval(new function() { output=(input * input)});
    return output;
}
```

Using a function instead of a string as a parameter makes sure the code inside the new method is optimized by the JavaScript compiler.

Function scope

Each scope in the JavaScript function scope chain contains several variables. It's important to understand the scope chain so that you can take advantage of the benefits. Listing 3 shows an example of function scope.
Listing 3. Function scope

```javascript
function test() {
    var localVar = "test";
    test1(this. localVar);
    var pageName = document.getElementById("pageName");
}
```

**Figure 1** shows the scope chain structure.

**Figure 1. Scope chain structure**

Using local variables is much faster than using global variables because the farther the variable is in the chain, the slower the resolution will be.

If there are `with` or `try-catch` statements in the code, the scope chain is more complex. **Figure 2** shows the scope chain of the try-catch statement.

**Figure 2. Try-catch scope chain structure**

String functions

One of the least desirable functions in JavaScript is string concatenation. It typically uses a `+` sign to do this work. **Listing 4** shows an example of this.
Listing 4. String concatenation

```javascript
var txt = “hello” + “ ” + “world”;```

The statement creates several intermediate strings to contain the concatenation result. This constant creation and destruction of strings behind the scenes leads to very poor string concatenation performance. Early browsers had no optimization for such operations. We suggest that you create a `StringBuffer` class for implementation, as shown in Listing 5.

Listing 5. StringBuffer object

```javascript
function StringBuffer() {
    this.buffer = [];  
}

StringBuffer.prototype.append = function append(val) {
    this.buffer.push(val);  
    return this;
}

StringBuffer.prototype.toString = function toString () {
    return this.buffer.join(“”);
}
```

All properties and methods are defined on the string object (rather than the value). When you reference a property or method of a string value, the ECMAScript engine must implicitly create a new string object with the same value as your string before running the method. This object is only used for that specific request, and will be re-created next time you attempt to use a method of the string value.

In this situation, use a new statement for those strings whose methods will be called several times.

An example of a new string object is shown in Listing 6.

Listing 6. An example of creating new string object

```javascript
var str = new String(“hello”);```

`StringObject.indexOf` is faster than `StringObject.match`. When searching for simple string matches, use `indexOf` rather than regular expression matching wherever possible.

Avoid matching against very long strings (10KB or greater) unless you absolutely must. If you are certain that the match will only occur in a specific portion of the string, take a substring and compare against that instead of against the entire string.

DOM performance

This section outlines some of the things you can tweak to improve DOM performance.

Repaint

DOM will be repainted whenever something is made visible if it was not previously visible, or vice versa. Repaint is also known as redraw. The behavior will not alter the document layout. Anything
that doesn’t change the size, shape, or position of the element but does change its appearance will trigger a repaint.

For example, adding an outline or changing the background color of an element will prompt a repaint. This repaint is expensive in terms of performance; it requires the engine to search through all elements to determine what is now visible, and what must be displayed.

**Reflow**

A reflow is a more significant change than a repaint. With a reflow:

- The DOM tree is manipulated.
- A style is changed that affects the layout.
- The className property of an element is changed.
- The browser window size is changed.

The engine will do a reflow to the relevant element to identify where the various parts must now be displayed. Children elements will also be reflowed to reflect the new layout of their parent element. Elements that appear after the element in the DOM will also be reflowed to calculate their new layout, as they might have been moved by the initial reflows. Ancestor elements will also reflow to account for the changes in the size of their children. Finally, everything will be repainted.

Every time you add an element to the document, the browser has to reflow the page to work out how everything must be positioned and rendered. The more things you add, the more it has to reflow. If you can reduce the number of separate elements you add, the browser will have fewer flows to do, and will therefore perform faster.

**CSS performance**

Put Cascading Style Sheets (CSS) at the top. If the stylesheets are at the bottom, they’ll be loaded last. Until then, the page is completely blank for several seconds while the browser waits for the stylesheet to be loaded before rendering anything else on the page—even the static text.

With Internet Explorer, `@import` behaves the same as using `<link>` at the bottom. We recommend that you don’t use it.

**Shorthand properties**

Use shorthand properties to set several properties at once, in a single declaration, rather than using a separate declaration for each individual property. With shorthand properties, you can reduce file size and lessen maintenance.

For example, you can set background, border, border color, border style, border sides (border-top, border-right, border-bottom, and border-left), border width, font, list style, margin, outline, and padding properties.

**CSS selectors**

CSS selectors are matched by moving from *right to left*. As shown in Listing 7, the browser must iterate over every anchor element in the page and determine whether its parent’s ID is `aElement`.
Listing 7. Selector is matched from right to left

```css
#aElement > a{
  font-size: 18px;
}
```

If you remove `>` from the code, as shown in Listing 8, performance worsens. The browser has to check every anchor in the entire document. And, instead of just checking each anchor’s parent, the browser has to climb the document tree and look for an ancestor with the ID `aElement`. If the anchor being evaluated is not a descendant of `aElement`, the browser has to walk the tree of ancestors until it reaches the document root.

Listing 8. Performance worsens if there is no `>`

```css
#aElement a{
  font-size: 18px;
}
```

Best practices

This section outlines some best practices that will help you refine and improve the performance of your web applications.

Reduce the number of HTTP requests

Every HTTP request has overhead, including searching the DNS, creating a connection, and waiting for a response, so trimming the number of requests can reduce unnecessary overhead. To reduce the number of requests:

- Combine files. Combining scripts that are always required at the same time into one file won’t reduce the overall size, but will reduce the number of requests. You can also combine CSS files and images in a similar way. You can implement file combining automatically:
  - In the build step. With the `<concat>` tag, combine files by running Ant.
  - In the runtime step. Enable the `mod_concat` module. If the httpServer is Apache, use `pack:Tag` as the JSP taglib to combine the JavaScript and stylesheet files. (`pack:Tag` is a JSP-Taglib that minifies, compresses, and combines resources, such as JavaScript and CSS, and caches them in memory or in a generated file.)
- Use CSS Sprites. Combine your background images into a single image, and use the CSS `background-image` and `background-position` properties to display the desired image segment. You can also use inline image to reduce the number of requests.

Post-load the components

Render only the components that are absolutely required; the rest of them can wait. It’s best to avoid rendering too many elements at one time.

In some situations, you can use post-load. Since the components that are out of the browser’s visible area can be post-loaded, the initial rendering can be fired the first time these components enter the visible area and no sooner.
Parts of JavaScript can be post-loaded after onload events, such as dragging an element in JavaScript after the initial rendering.

**Pre-load the components**

By pre-loading components, you take advantage of the idle time of the browser and request components (such as images, styles, and scripts) that you'll need in the future. When the user visits the next page, your page will load much faster if most of the components are already loaded into the cache.

There are two types of pre-loading:

- **Unconditional**: As soon as onload fires, you fetch some extra components.
- **Conditional**: Based on a user's action, you make an educated guess as to where the user is headed next and pre-load accordingly.

**Put scripts at the bottom**

Scripts can be problematic because they block parallel downloads. While a script is downloading, the browser won't start any other downloads—even those on different host names. Put scripts, like the stylesheets, at the bottom so that they download after other loadings are finished.

You can also use deferred scripts, which are supported only on Internet Explorer. The `defer` attribute indicates that the script does not contain `document.write()`. It's a clue to browsers that they can continue rendering.

**Use cookie-free domain components**

When the browser makes a request for a static image and sends cookies together with the request, the server doesn't use any of those cookies. Since the cookies only create unnecessary network traffic, make sure that static components are requested with cookie-free requests. Then use a subdomain and host all your static components there.

**Make JavaScript and CSS external**

Using external files in the real world generally produces faster pages because the JavaScript and CSS files are cached by the browser. JavaScript and CSS that are inlined in HTML documents get downloaded every time the HTML document is requested. This reduces the number of HTTP requests that are needed, but increases the size of the HTML document. On the other hand, if the JavaScript and CSS are in external files cached by the browser, the size of the HTML document is reduced without increasing the number of requests.

**RIA widget performance**

Mainstream RIA Ajax frameworks, such as ExtJS, YUI, Dojo, and others, all provide libraries of fancy widgets to enhance user experience. Compared to other frameworks, Dojo is stronger in the enterprise development arena due to:

- Object-oriented programming (OOP) coding
• Cross-platform
• Dojo offline API support of local data storage
• DataGrid, 2D, and 3D graphics (chart components provide an easier way to present reports in a browser)

Dojo has been widely used in many websites. We'll use Dojo as an example to analyze the performance of RIA widgets. Many tools for Dojo widget tuning are at your disposal, such as Page Speed, Rock Star Optimizer, and Jiffy. We strongly recommend YSlow and Firebug.

**YSlow**

YSlow analyzes web page performance by examining all of the components on the page, including those created with JavaScript, based on a set of rules for high-performance web pages. YSlow is a Firefox add-on integrated with the Firebug web development tool; it offers suggestions for improving the page's performance, summarizes its components, displays statistics about the page, and provides tools for performance analysis.

*Figure 3* shows an example of information on the YSlow Grade tab.

**Figure 3. YSlow Grade tab**

YSlow's web page analysis is based on 22 testable rules, which are listed below in rough order of importance and effectiveness. Studies show that web page response time can be improved by 25 to 50 percent by following these rules:

- Minimize HTTP requests.
- Use a content delivery network.
- Add an Expires or a Cache-Control header.
- Gzip components.
- Put stylesheets at the top.
- Put scripts at the bottom.
- Avoid CSS expressions.
- Make JavaScript and CSS external.
- Reduce DNS lookups.
- Minify JavaScript and CSS.
- Avoid redirects.
- Remove duplicate scripts.
- Configure ETags.
- Make Ajax cacheable.
- Use `GET` for Ajax requests.
- Reduce the number of DOM elements.
- Eliminate 404s.
- Reduce cookie size.
- Use cookie-free domains for components.
- Avoid filters.
- Do not scale images in HTML.
- Make favicon.ico small and cacheable.

The YSlow Statistics tab, shown in Figure 4, gives you a comparison between your page size for visitors coming with an empty cache and those that have previously visited the page.

**Figure 4. YSlow Statistics tab**

![YSlow Statistics tab](image)

The Components tab displays every component, along with relevant information regarding performance. For example, you can see if the component was gzipped, or what the ETag is (if anything). Component size and expiration date are also displayed in the Components tab, which is shown in Figure 5.

**Figure 5. YSlow Components tab**

![YSlow Components tab](image)
Firebug

Firebug integrates with Mozilla Firefox to put a wealth of development tools at your fingertips when you browse. You can edit, debug, and monitor CSS, HTML, and JavaScript live in any web page.

You use the Firebug Net panel, shown in Figure 6, to monitor HTTP traffic initiated by a web page. It presents all collected and computed information to the user. Each entry represents one request/response roundtrip made by the page.

Figure 6. Firebug Net

The Firebug Console panel, shown in Figure 7, provides two methods for monitoring your code performance.

Figure 7. Firebug Console panel

Profile

For a particular function, use Profiler. JavaScript Profiler is useful Firebug feature that measures the execution time of each JavaScript code. Use JavaScript Profiler to improve the performance of your code or to investigate why a particular function takes so long. It is similar to console.time(), but JavaScript Profiler can provide you with more detailed information about what is happening with your code.

code.time()

For particular code pieces, use `console.time()`. The console displays the results of the commands that you entered into the command line. You can use the `console.time(timeName)` function to measure how long a particular code or function takes. This feature is very helpful if you're trying to improve the performance of your JavaScript code. Listing 9 shows an example.
Listing 9. console.time() example

```javascript
var timeName = 'measuringTime';
console.time(timeName); //start of the timer
for(var i=0;i<1000;i++){
    //do something
}
console.timeEnd(timeName); //end of the timer
```

`measuringTime:xxms` will be printed in the console.

**Dojo widget performance**

This section explores ways to analyze and improve the performance of your Dojo widgets.

**Loading costs**

As pointed out in *Improving performance of Dojo-based web applications* (E. Lazutkin, Feb 2007), most Dojo users' first impression is that it is huge. For example, the dojo-release-1.5.0-src.zip is 19M, and even the compressed package dojo-release-1.5.0.zip is still 4.5M. The majority of files in the minimal build are redundant and will never be served. All Dojo builds come with a full copy of all Dojo files and a customized dojo.js file that combines some of the more frequently used files. The best way to minify your loading costs is to use the proper Dojo build.

Dojo.js activates Dojo object and loads the rest of the modules dynamically unless they've already been loaded by an optional part of dojo.js. When the browser loads the dojo.js file for the first time, it will upload and set up the following files:

- Dojo bootstrap code
- Dojo loader
- Frequently used modules

To reduce the loading time, consider which build is a good match with your applications. Otherwise, you have to do a custom Dojo build. For more information about Dojo documentation, see the *Related topics* section.

**Parsing costs**

To minimize your Dojo widget parsing costs, optimize initialization using one of two methods:

**Tag instantiating**

You can create a Dojo widget with an HTML tag by adding the attribute `dojoType`, as shown in *Listing 10*. This method works on the premise that dojo.parser is included in `dojo.require("dojo.parser");` and the `djConfig="parseOnLoad:true"`. This is an easy way to announce a component with lightweight code. Every tag that has the `dojoType` attribute in the page will be parsed automatically after the document is loaded. This method is very convenient for small applications, but it can significantly increase the start-up time for web applications with a lot of HTML. The parser will visit every element to check whether it must be parsed. Use a profile, such as the one provided with Firebug.

If you find that you spend a lot of time on things like `dj_load_init()`, `modulesLoaded()`, or anything else resembling initial loading, consider widget initialization.
Listing 10. Creating a Dojo widget with dojoType

```html
id="errorDialog" dojoType="dijit.Dialog" title='dialog1' class="pop-window"/>
```

Code instantiating

Dojo is an OOP framework, so you can use `new` to create a widget. To create a widget, you must input two parameters: a JSON object with attributes, and the element to be parsed. Every widget needs at least two sentences. An example is shown in Listing 11.

Listing 11. Creating a Dojo widget with JavaScript

```javascript
new dijit.Dialog({"title":"dialog1 "},dojo.byId("dialog1"));
```

Promote parsing performance

To optimize your code structure and performance, consider promoting parsing when creating a widget. Disable the auto-parse by setting `parseWidgets` to false, and then creating an array to set the element's ID, as shown in Listing 12. You can also push the new element's ID dynamically during the runtime. Parse all of the elements in the array with `dojo.forEach()` when the document is loaded.

Listing 12. Parse widgets with iterating searchIds

```html
<head>
    ....
    <script>
        djConfig = { parseWidgets: false, searchIds: [..] }; </script>
    ....
</head>
<body onload='dojo.forEach(djConfig.searchIds,
    function(id){dojo.parser.parse(dojo.byId(id))});'>
    ....
</body>
```

Solutions

Performance problems with the Dojo grid mainly concern input/output operations, mass data access, and data rendering by the browser. You can improve the performance of a Dojo grid widget by using several mechanisms in combination with the features of the grid.

Review your use of the cache mechanism. When the data is loaded from the database to local, keep the data in memory for a long time. This is a good way to reduce the time response for requesting data from the server side. You won't send a request to the server side until users update or modify the data in the grid. The cache mechanism is roughly implemented by the Dojo grid itself, but problems arise when users do certain operations on the grid. The following scenarios illustrate such problems:

Sort the grid

In most cases, you can sort the grid correctly because the grid itself implements sort functions at the data store level. The data store reflects the cache, though. If the type of grid column is Chinese characters, for example, the sort activity will produce incorrect results and the performance of the grid will be poor due to undefined factors.
The solution is to redefine the sort functions yourself at the data store level. Listing 13 and Listing 14 below show how to do this: rewrite the sort logic based on the onHeaderCellMouseDown function, render the data, and update the header view of the grid.

**Listing 13. Redefine the sort logic**

```javascript
grid.onHeaderCellMouseDown = function(event){
    var items = DataStore.data.items;
    //Sort the "Name" column which might contain characters like Chinese and so on
    if (event.cellIndex == 1) {
        sortAscending = !sortAscending;
        //Change the string to localestring before comparison with localeCompare method
        if (sortAscending) {
            items.sort(function(m, n){
                return m["name"].toString().localeCompare(n["name"].toString());
            });
        } else {
            items.sort(function(m, n){
                return n["name"].toString().localeCompare(m["name"].toString());
            });
        }
    } else {
        //Sort the "Date" column
        if (event.cellIndex == 2) {
            sortAscending = !sortAscending;
            //Compare the date with milliseconds computed from 1970/07/01
            if (sortAscending) {
                items.sort(function(m, n){
                    return Date.parse(m["date"].toString()) - Date.parse(n["date"].toString());
                });
            } else {
                items.sort(function(m, n){
                    return Date.parse(n["date"].toString()) - Date.parse(m["date"].toString());
                });
            }
        }
    }
}
```

**Listing 14. Render data, update header view of the grid**

```javascript
updateGridAfterSort : function(sortColIdx){
    //Render the data of the grid
    var store = new dojo.data.ItemFileWriteStore(DataStore.data);
    grid.setStore(store, null, null);
    grid.update();
    //Update the header view of the grid
    var headerNodeObjs = document.getElementsByTagName("th");
    for (var i = 0; i < 2; ++i) {
        //gridLayout" is a global array defining the layout of the grid
        var headerNodeObjName = gridLayout[0].cells[0][i].name;
        var headerNodeHtml = ['<div class="dojoxGridSortNode'];
        if (i == sortColIdx) {
        }
        headerNodeHtml = headerNodeHtml.concat([headerNodeObjName, '</div>']);
        headerNodeObjs[i].innerHTML = headerNodeHtml.join(" ");
    }
}
```
Search the grid

When the grid has mass data, the search function will lead to poor performance, especially when the grid might support real-time search and fuzzy match features. The solution is to use the extra memory space to cache the data before they are converted to JSON objects used in data store; this will avoid lots of function calls, such as `getItem` of data store. **Listing 15** shows an example.

### Listing 15. Cache data fetched from database in an array and search it

```javascript
//Fetch data from database
getData : function() {
    function callback(ResultSet) {
        //ResultSet is an array of records fetched from database and make variable
        //rawData refer to it to cache it in memory
        GlobalVariables.rawData = ResultSet;
        //Convert the raw data ResultSet to JSON for data store use
        GlobalVariables.dataJSON = JSONUtil.convert2JSON(ResultSet);
    }
    DBUtil.fetchAll(callback);
}
//Search functionality
search: function(col, value){
    if (value == null || value == "") {
        return;
    }
    //Used here
    var rawData = GlobalVariables.rawData;
    var newResults = [];
    for (var i = 0; i < rawData.length; i++) {
        var result = rawData[i];
        //Fuzzy match
        if(result[col].toLowerCase().indexOf(value.toLowerCase()) != -1){
            newResults[newResults.length] = result;
        }
    }
    //Render the new results
    GridManager.renderNewResults(newResults);
}
```

**Lazy load mechanism**

The Dojo grid is designed to support a lazy loading mechanism, which improves performance and provides a better user experience. Lazy loading in the Dojo grid means rendering some data in data store, but not all of it. The grid will not display the remaining data until you drag the scroll bar.

By default, the grid doesn't open the lazy loading mechanism; you must explicitly start it. **Listing 16** shows how to start it in two different ways. The `rowsPerPage` and `keepRows` attributes are the key components.
Listing 16. Start lazy loading mechanism

```javascript
//The programmatic way
var grid = new dojox.grid.DataGrid(
    store: store, //data store
    structure: gridLayout,
    rowsPerPage: 10, //Render 10 rows every time
    keepRows: 50, //Keep 50 rows in rendering cache
}, "grid");
//The declarative way using HTML label
<table
    dojoType="dojox.grid.DataGrid"
    id="grid" store="store" structure="gridLayout"
    query={ id: '*' }
    rowsPerPage="10" keepRows="50">
    <!--  other definitions  -->
</table>
```

Pre-load mechanism

The pre-load mechanism loads the remaining data in advance, even though users might only need them temporarily. For the Dojo grid, there might be plenty of data in the data store; users can drag the scrollbar to view the remaining data. If there's a lot of data in one page, it's not convenient to see only one particular row. Use the pre-load mechanism and paging techniques for more convenient views (similar to Google’s paging bar) and better performance.

Listing 17 shows the basic implementation with paging techniques. First, construct several JSON objects used by data store for the initial paging bar, and then add some new JSON objects dynamically when users click on the last page in the paging bar.

Listing 17. Construct several JSON objects initially and switch them on demand

```javascript
//Fetch data from database and convert them to JSON objects
dataStore = new dojo.data.ItemFileWriteStore({data:GlobalVaribles.dataJSONs[0]});
var grid = new dojox.grid.DataGrid({
    store: dataStore ,
    structure: gridLayout,
}, "grid");
grid.startup();

//Assuming that the user clicks the i-th item in the paging bar, we just update the data
//store for the grid simply and the performance is still very good.
dataStore = new dojo.data.ItemFileWriteStore({data:GlobalVaribles.dataJSONs[i-1]});
grid.setStore(dataStore, null, null);
grid.update();
```
Summary

In this article you learned how to identify several problems, or bottlenecks, in your web applications. You now have several tools, tricks, and tips to use to tweak and improve performance for your users.
Related topics

- View the slideshow **Speed up your JavaScript**, by N. Zakas (2009), to learn more about JavaScript performance issues.
- Learn more about the **Dojo toolkit**. Look at examples, documents, blogs, and more.
- Look at all the **Official Dojo Documentation**.
- Read the translated version of "**Dojo widget performance best practices**."
- Download **IBM product evaluation versions** or explore the online trials in the **IBM SOA Sandbox** and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

© Copyright IBM Corporation 2010
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)