# Chapter 6

# Scalable, Maintainable Ajax

In This Chapter


- 6.1 General Practices 188
- 6.2 A Multitude of Simple Interfaces 194
- 6.3 Dense, Rich Interfaces 201

While performance optimization should wait until after the development of primary functionality, scalability and maintainability need to happen starting with the design of the application. The implemented architecture has a direct impact on scalability and needs to have enough consideration driving it to keep the application solid under any circumstance.

At the same time that the application developers create a scalable architecture, they can also use the same techniques for maintainability. The development team can separate each aspect of the code into logical, easy-to-load objects and libraries that the application then can load or pre-load as necessary. This isolation encourages abstraction between each object of the application, making it easier to track down bugs and to add functionality later in development.

## 6.1 General Practices

While an application's architecture can dictate much of its scalability, some general coding practices can help keep smaller pieces of the application from growing sluggish under more demanding circumstances. If developers do not make an effort at the coding level to make the application scalable, unscalable functionality will mar the architectural scalability of the application. The users care only about the overall experience of the application, not at which point it fails.

Though many factors can affect an application's scalability, over-usage of the processor and memory plague web applications in particular. PHP has a `memory_limit` setting in `php.ini`, which generally defaults to 8MB. This may not seem like much, but if a single hit uses more than 8MB, then a constant stream of multiple hits each second will pin memory usage. If performance starts dropping in that stream, the application will run itself into the ground.

### 6.1.1 Processor Usage

As the profiling output in Chapter 5, "Performance Optimization," showed, particularly with the Xdebug examples, the amount of time spent in a function does not necessarily correlate with the amount of memory used in that function. Several other factors can cause slow-downs in a function, including disk access, database lag, and other external

references. Sometimes, however, the function uses just too many processor cycles at once.

When this processor drain occurs in the JavaScript of the application, it can seize up the browser because most browsers run JavaScript in a single thread. For this reason, using DOM methods to retrieve a reference to a single node and then drilling down the DOM tree from there scales much better than custom methods to find elements by attributes such as a certain class or `nodeValue`.

As an example, an application could have a table with twenty columns and one thousand rows, with each table cell containing a number. Because this display gives the users quite a lot of information in a generic presentation, the application may offer a way of highlighting the cells containing values above a given threshold. In this example, the functions will have access to this minimum, held in a variable named `threshold`. This cell highlighting can come about in several ways.

The first of these methods, shown below, gets a `NodeSet` of `td` elements and then iterates through the entire list at once. For each cell, the function gets the text node value and compares it to the threshold. If the value exceeds the threshold, the cell gets a one-pixel border to highlight it:

```
function bruteForce() {
var table = document.getElementById("data");
    var tds = table.getElementsByTagName("td");
    for (var i = 0; i < tds.length; i++) {
        var td = tds.item(i);
        var data = td.firstChild.nodeValue;
        if (parseInt(data) > threshold) {
            td.style.border = "solid 1px #fff";
        }
    }
}
```

While this function does work (running through 20,000 `td` elements and applying highlighting where required in just over a second), the browser stops responding entirely for the duration of the function. During that second, the processor usage of Firefox jumps to approximately 74 percent.

To prevent the browser from locking up, the script can simulate threading by splitting the work up into sections and iterating through each section after a minimal timeout. This method takes almost ten times the length of time that the `bruteForce()` function took to complete, but this next function runs in parallel to any actions the user may want to take while applying the highlighting:

```
function fakeThread() {
    var table = document.getElementById("data");
    var tds = table.getElementsByTagName("td");
    var i = 0;
    var section = 200;
    var doSection = function() {
        var last = i + section;
        for (; i < last && i < tds.length; i++) {
            var td = tds.item(i);
            var data = td.firstChild.nodeValue;
            if (parseInt(data) > threshold) {
                td.style.border = "solid 1px #fff";
            }
        }
        if (i < tds.length) {
            setTimeout(doSection, 10);
        }
    }
    doSection();
}
```

The fastest method comes in revisiting the functionality required, namely that the user can enable highlighting of `td` elements when the value contained exceeds a threshold. If the server flags the `td` elements with a class when the value exceeds this threshold, it can cache these results, and the script then has to apply a style rule only for the given class. The example below assumes that the function needs to create a new style element and write the rule into that, though it could simply edit an existing rule if the stylesheet had one in place:

```
function useClass() {
    var head = document.getElementsByTagName("head")[0];
    var style = head.appendChild(
        document.createElement("style")
    );
    style.type = "text/css";
    style.appendChild(
        document.createTextNode(
            ".high { border: solid 1px #fff; }"
        )
    );
}
```

By rethinking functionality that takes large amounts of processor cycles to work, developers can enable the application to handle data and interfaces of enormous size without impacting performance.

### 6.1.2 Memory Usage

Similar to processor usage, memory usage rapidly increases in problem areas, but can have certain measures taken to prevent it. Some types of functions, especially those that load the entire data set into a returned value, will max out memory usage unless developers put thought and planning behind their usage.

For instance, many PHP database extensions offer methods of retrieving entire record sets into an array or even just a column of data into an array. These methods, though useful and easy to use, can drive up memory usage to the breaking point when not used carefully. The following code fetches a list of user IDs and names into an array using the PDO extension:

```
// First, run the query and get the list
$query = 'SELECT 'id', 'name' FROM 'users' ORDER BY 'name'';
$stmt = $database->prepare($query);
$stmt->execute();
$users = $stmt->fetchAll(PDO::FETCH_ASSOC);

<!-- Later in the application, output the list -->
<ol>
<?php foreach ($users as $user) { ?>
    <li><a href="?id=<?php echo (int)$user['id']; ?>">
        <?php
        echo Utilities::escapeXMLEntities($user['name']);
        ?>
    </a></li>
<?php } ?>
</ol>
```

This example works perfectly well for a few dozen users, or even a hundred. However, once the list of users grows to hundreds, thousands, and especially millions, the `$users = $stmt->fetchAll(PDO::FETCH_ASSOC);` line will trigger an out of memory error, and the page will fail to render at all. To get around this issue without putting the database query and method calls directly into the template, the code can instead use a simple layer of abstraction and the implementation of the standard PHP library `Iterator` interface:

```
class PDOIterator implements Iterator {
    /**
     * The PDO connection object
     */
    protected $database;
    protected $statement;
    /**
     * The query to run on the first iteration
     */
    protected $query;
    /**
     * Optional parameters to use for prepared statements
     */
    protected $parameters;
    /**
     * The current record in the results
     */
    protected $current;
    /**
     * The row number of the current record
     */
    protected $key;
    /**
     * A Boolean as to whether the object has more results
     */
    protected $valid;

    /**
     * Forward-only cursor assumed and enforced
     */
    public function rewind() {
        return false;
    }

    public function current() {
        if ($this->key === -1) {
            if (!$this->runQuery()) {
                $this->valid = false;
                return false;
            } else {
                $this->next();
            }
        }
        return $this->current;
```

```
    }

    public function key() {
        return $this->key;
    }

    public function next() {
        $this->current = $this->statement->fetch(PDO::FETCH_ASSOC);
        if ($this->current) {
            $this->key++;
            if (!$this->valid) {
                $this->valid = true;
            }
            return true;
        } else {
            $this->statement = null;
            $this->valid = false;
            return false;
        }
    }

    protected function runQuery() {
        $this->statement = $this->database->prepare($this->query);
        $this->statement->execute($this->parameters);
    }

    public function valid() {
        return $this->valid;
    }

    public function setParameters($params) {
        $this->parameters = $params;
    }

    public function __construct($database, $query) {
        $this->database = $database;
        $this->query = $query;
        $this->parameters = null;
        $this->current = null;
        $this->key = -1;
        $this->valid = true;
    }
}
```

This class may seem like a large amount of work when compared to the previous example, but it doesn't replace that example just yet. The `PDOIterator` class merely gives the application the ability to replace the earlier example easily and cleanly, by using it as shown in this next example:

```
// First, run the query and get the list
$query = 'SELECT 'id', 'name' FROM 'users' ORDER BY 'name'';
$users = new PDOIterator($database, $query);

<!-- Later in the application, output the list -->
<ol>
<?php foreach ($users as $user) { ?>
    <li><a href="?id=<?php echo (int)$user['id']; ?>">
        <?php
        echo Utilities::escapeXMLEntities($user['name']);
        ?>
    </a></li>
<?php } ?>
</ol>
```

Because the `PDOIterator` class implements `Iterator`, the usage in the template does not change at all from the array of results originally assigned to the `$users` variable. In this example, though, `$users` contains a reference to the `PDOIterator` instance, and the query does not actually run until the first iteration, keeping the database connection clean and using very little memory. Once the code starts iterating through the results, it immediately renders that entry in the markup, keeping none of the results in memory afterward.

Any function that pulls a full list, a file's contents, or any other resource of unknown size and then returns it should fall under heavy scrutiny. In some cases, these convenience functions does make sense. For instance, if a configuration file will never have more than five or ten lines in it, using `file_get_contents` makes the task of pulling in the contents of the file much simpler. However, if the application currently has only a dozen user preferences, it still cannot know that it will always have a reasonable list for retrieving in full.

## 6.2 A Multitude of Simple Interfaces

When an application calls for many different interfaces, each offering a different set of functionality, the interfaces should load only as much of the available function

library as necessary. This practice keeps each page load light and fast, even as the application and its data grow. Once loaded, each page can load additional data and functionality as needed; these requests will return as lightly and as quickly as the initial page load itself.

### 6.2.1 Modularity

To keep a library flexible enough to support many different interfaces that offer a variety of functionality, each piece of that functionality needs to exist independent of the rest. The pieces can extend an object from a core set to make it easier to offer a consistent object interface without duplicating code.

Throughout this book, classes continually extend other base classes. This not only brings the advantages of object-oriented programming to the applications and code samples, but also makes it much easier to load each set of functionality as necessary. For example, the following JavaScript classes exist in `effects.lib.js`, but extend the `EventDispatcher` class:

```
/**
 * The base class of all Effects in the library
 */
function Effect() { }
Effect.prototype = new EventDispatcher;

/**
 * Kind of useless by itself, this class exists to get extended for
 * use with text, backgrounds, borders, etc.
 */
function ColorFade() { }
ColorFade.prototype = new Effect;
// Triggers at the start and end of the effect
ColorFade.prototype.events = {
    start : new Array(),
    end : new Array()
};
// Default to changing from a white background to a red one
ColorFade.prototype.start = 'ffffff';
ColorFade.prototype.end = 'ff0000';
// Default to a second, in milliseconds
ColorFade.prototype.duration = 1000;
ColorFade.prototype.step_length = 20;
// The current step (used when running)
```

```
ColorFade.prototype.step = 0;
// Reference to the interval (used when running)
ColorFade.prototype.interval = null;
// Calculated values used in color transformation
ColorFade.prototype.steps = 0;
ColorFade.prototype.from = null;
ColorFade.prototype.to = null;
ColorFade.prototype.differences = null;
ColorFade.prototype.current = null;
/**
 * Parse a three- or six-character hex string into an
 * array of hex values
 */
ColorFade.prototype.explodeColor = function(color) {
    // Three or six, nothing else
    if (color.length != 6 && color.length != 3) {
        throw 'Unexpected color string length of ' + color.length;
    }

    var colors = new Array();
    if (color.length == 3) {
        colors[0] = parseInt('0x' + color.charAt(0) + color.charAt(0));
        colors[1] = parseInt('0x' + color.charAt(1) + color.charAt(1));
        colors[2] = parseInt('0x' + color.charAt(2) + color.charAt(2));
    } else {
        colors[0] = parseInt('0x' + color.charAt(0) + color.charAt(1));
        colors[1] = parseInt('0x' + color.charAt(2) + color.charAt(3));
        colors[2] = parseInt('0x' + color.charAt(3) + color.charAt(5));
    }
    return colors;
}
/**
 * Executes the fade from the start to end color
 */
ColorFade.prototype.run = function() {
    this.from = this.explodeColor(this.start);
    this.to = this.explodeColor(this.end);
    this.differences = new Array(
         this.from[0] - this.to[0],
         this.from[1] - this.to[1],
         this.from[2] - this.to[2]
     );

    // Steps in portions
```

```
    this.steps = Math.round(this.duration / this.step_length);
    // Reset the step so that we can run it several times
    this.step = 0;

    clearInterval(this.interval);
    this.interval = setInterval(this.runStep, this.step_length, this);

    // Success!
    return true;
}
/**
 * Called from an Interval, runStep takes what this should resolve
 * to and references it
 */
ColorFade.prototype.runStep = function(dit) {
    dit.step++;
    var state = (dit.step / dit.steps);
    dit.current = new Array(
         dit.from[0] - Math.round(state * dit.differences[0]),
         dit.from[1] - Math.round(state * dit.differences[1]),
         dit.from[2] - Math.round(state * dit.differences[2])
     );
    if (dit.step == dit.steps) {
        clearInterval(dit.interval);
    }
}
```

The `ColorFade` class serves as the base of all color-fading classes, offering a consistent interface. All of these effects exist in the same library, so that the application can load them as necessary, rather than at load time. In this way, simple interfaces that do not require the functionality can spare the user the loading time for unnecessary libraries.

In fact, the class definitions would also allow for a particularly intensive effect to exist in a separate, optionally loaded library. This would mean that the interface would have even more of an ability to load only the required functionality for the interface in question, and the load time could stay even smaller.

This modularity has just as great an impact on the server-side application as it has on the client. From the server application, the more power the application has over the choice of resources to load, the more precise the libraries loaded and the faster the response returns from the server. Even if the server-side application has thousands of paths that it can take for properly handling a response, it still can use efficient lookup

tables for loading only the necessary modules and returning them quickly, no matter how much available functionality the application has to offer.

On the client side, this technique keeps the memory usage low even for complex applications, as the application moves from interface to interface, removing the unused libraries from memory and loading in the classes and functions it needs for the next step in the application. By loading in these libraries after the initial page load, the client-side application can keep even full page loads fast and responsive. It need only manage the timing of the additional library loads properly so that the user does not need to wait an unreasonable amount of time for the next piece of the interface to load.

### 6.2.2 Late Loading

Once an interface initially loads, it can offer the most commonly used functionality from the already available resources. This functionality should cover the most typical interaction scenarios, but should not restrict the users to only that small set of functionality. By loading additional functionality as needed, the interface can support applications as light and simple or complex as the user needs for that particular instance, without bogging down the loading time of the page itself.

The following code enables the application to late load code in two ways. It allows the inclusion of known classes by calling `Utilities.include("ClassNameHere", callbackWhenIncluded)`, which uses a simple lookup to load the appropriate file for a given class. It also allows the inclusion of arbitrary files by calling `Utilities.loadJavaScript("filename.js", callbackWhenLoaded);`, which can load internal or external JavaScript files. Both of these methods take an optional callback argument, which receives a single Boolean parameter of true when successfully loaded, or false when the load failed due to a timeout or some other issue:

```
/**
 * A list of available classes, as keys to their
 * corresponding source files. A script should
 * pre-generate this list rather than having it
 * hard-coded, so that it could always have the
 * latest classes and files.
 */
Utilities.classFiles = {
    "AjaxEvent" : "includes/ajax.lib.js",
    "AjaxRequest" : "includes/ajax.lib.js",
    "AjaxRequestManager" : "includes/ajax.lib.js",
    "BackgroundFade" : "includes/effects.lib.js",
```

```
    "ColorFade" : "includes/effects.lib.js",
    "Controller" : "includes/main.lib.js",
    "CustomEvent" : "includes/ajax.lib.js",
    "Effect" : "includes/effects.lib.js",
    "ElementEffectEvent" : "includes/effects.lib.js",
    "EventDispatcher" : "includes/ajax.lib.js",
    "FadeEvent" : "includes/effects.lib.js",
    "Field" : "includes/main.lib.js",
    "ForegroundFade" : "includes/effects.lib.js",
    "Messenger" : "includes/main.lib.js",
    "Model" : "includes/main.lib.js",
    "Throbber" : "includes/main.lib.js",
    "Utilities" : "includes/main.lib.js",
    "View" : "includes/main.lib.js"
}
/**
 * Late-loading of JavaScript files based on object to
 * file lookups. Once the file loads (or times out), it
 * triggers the callback (if specified), passing a Boolean
 * indicating whether it successfully loaded.
 */
Utilities.include = function(class, callback) {
    // First, if already loaded, just call the callback
    if (window[class]) {
        if (callback) {
            setTimeout(callback, 10, true);
        }
        return true;
    } else if (Utilities.classFiles[class]) {
        return Utilities.loadJavaScript(
            Utilities.classFiles[class],
            callback
        );
    } else {
        // Class not found, just return false
        return false;
    }
}
/**
 * Keep track of files already loaded
 */
Utilities.loadedJavaScript = { };
/**
 * Load the specified JavaScript file, optionally
```

```
 * calling a callback function and passing a Boolean
 * as to whether the file loaded
 */
Utilities.loadJavaScript = function(file, callback) {
    if (Utilities.loadedJavaScript[file]) {
        if (callback) {
            setTimeout(callback, 10, true);
        }
        return true;
    } else {
        var head = document.getElementsByTagName("head")[0];
        var script = head.appendChild(
            document.createElement("script")
        );
        // Set timeout of a very liberal 10 seconds
        var timeout = setTimeout(
            head.removeChild(script);
            function() {
                callback(false);
            },
            10000
        );
        script.addEventListener(
            "load",
            function() {
                clearTimeout(timeout);
                Utilities.loadedJavaScript[file] = true;
                callback(true);
            },
            false
        );
        script.type = "text/javascript";
        script.src = file;
        return true;
    }
}
```

The script loads the additional JavaScript files by appending additional `script` elements to the `head` of the document. Before setting the `src` attribute of the element, it adds an event listener to clear the `timeout` and dispatch the `load` event. This ensures that any script using it to load additional files will know if and when it has loaded.

Because the loading of each additional JavaScript file returns asynchronously, late loading of resources (which can easily extend to loading images and stylesheets) needs to happen early enough to prevent the user from having to wait before proceeding. The script then can use the optional callback functionality of the class/file loader to tell whether the required resource has successfully loaded by the time it needs to use the file.

By keeping a balance between the initially loaded scripts and the scripts that are loaded as required, the application can stay light and fast to load; this practice will expand its functionality without interrupting the user.

## 6.3 Dense, Rich Interfaces

Interfaces requiring large amounts of functionality cannot scale using modular loading, especially when they are used with a client-side application. Even in high-bandwidth, low-latency environments, the time required for each additional request for functionality will turn the application sluggish. Each user action hitting a yet-unloaded piece of the function library will effectively require a synchronous call to the server in order to respond directly to the new action.

### 6.3.1 Monolithic Applications

Having a monolithic application does not forbid the application from having an object-oriented design, but it does mean that the application loads all at once, preferably in only a couple of files. Because browsers will load only a couple of resources at a time from the same domain, an interface requiring dozens of externally loaded JavaScript files (not even taking stylesheets and images into account) will take a ludicrous amount of time to load even over a fast connection.

While modular applications can take advantage of the caching of multiple files over multiple requests, monolithic Ajax-based applications tend not to have more than one or two initial page loads. They resort instead to having most (if not all) of the application loaded into the browser's memory at once. By doing this, even incredibly complex applications can respond quickly to the user and support a wide range of functionality on demand.

To keep a monolithic application scalable, developers need to have naming conventions in place to reduce the risk of collisions, which can cause problems in JavaScript without making it obvious that the problems stemmed from a collision in the first place. In the following example, two different pieces of the same application need to define their own `Player` class. The first class defines a `Player` as a class that runs a slide show, while the second class defines it as the user of the application:

```
function Player(slides) {
    this.slides = slides;
}
Player.prototype = {
    slides : [],
    current : -1,
    next : function() {
        if (this.slides[this.current + 1]) {
            if (this.current > -1) {
                this.slides[this.current].style.display = "none";
            }
            this.slides[++this.current].style.display = "block";
            return true;
        } else {
            return false;
        }
    }
};

function Player() { }
Player.prototype = {
    alias : "",
    level : 1,
    login : function(login, password) {
        var req = request_manager.createAjaxRequest();
        req.post.login = login;
        req.post.password = password;
        req.addEventListener(
            "load",
            [Player.prototype.loggedIn, this]
        );
        req.open("POST", "login.php");
        req.send();
    },
    loggedIn : function(e) {
        var response = e.request.responseXML;
        if (user = response.getElementsByTagName("user")[0]) {
            var alias_node = user.getElementsByTagName("alias");
            this.alias = alias_node.firstChild.nodeValue;
               var level_node = user.getElementsByTagName("level");
            this.level = level_node.firstChild.nodeValue;
        }
    }
};
```

While PHP throws a fatal error when you attempt to define an existing class, JavaScript will quietly let it happen, overwriting any existing variables and methods, and altering the behavior of existing instances when modifying the prototype. Luckily, JavaScript's prototype-based object model makes it easy to implement something close to namespacing. By encapsulating each class definition in another object, one that can hold the class definitions for everything within a set of functionality, the classes can exist almost untouched from the previous definition:

```
var Slideshow = {
Player : function(slides) {
        this.slides = slides;
    }
}
Slideshow.Player.prototype = {
    slides : [],
    current : -1,
    next : function() {
        if (this.slides[this.current + 1]) {
            if (this.current > -1) {
                this.slides[this.current].style.display = "none";
            }
            this.slides[++this.current].style.display = "block";
            return true;
        } else {
            return false;
        }
    }
};

var Game = {
    Player : function() { }
}
Game.Player.prototype = {
    alias : "",
    level : 1,
    login : function(login, password) {
        var req = request_manager.createAjaxRequest();
        req.post.login = login;
        req.post.password = password;
        req.addEventListener(
            "load",
            [Player.prototype.loggedIn, this]
```

```
        );
        req.open("POST", "login.php");
        req.send();
    },
    loggedIn : function(e) {
        var response = e.request.responseXML;
        if (user = response.getElementsByTagName("user")[0]) {
            var alias_node = user.getElementsByTagName("alias");
            this.alias = alias_node.firstChild.nodeValue;
               var level_node = user.getElementsByTagName("level");
            this.level = level_node.firstChild.nodeValue;
        }
    }
};
```

Now, code using each of the classes can reference either one (without any doubt of which class it is using) by calling `new Slideshow.Player()` to instantiate a new `Player` that will display a slideshow. To instantiate a new `Player` representing the user, the code can call `new Game.Player()`. By using techniques like this to emulate namespaces, multiple developers can work on large, monolithic applications without fear of class or function name collisions; this practice makes such applications much easier to maintain (see Appendix B, "OpenAjax").

### 6.3.2 Preloading

An interface loading just six JavaScript files totaling 40k will load in an average of 150 milliseconds on a LAN connection. This instance does not take long, but the setup will not scale. The loading time grows linearly as the application loads more files, taking double the time for double the files. However, random network fluctuations can cause a higher incidence of bandwidth and latency issues tripping up the loading process, causing it to sporadically take a second or two for a single file.

Even though an application may have its functionality existing in separate JavaScript files, it still can take advantage of the faster load time of a smaller number of files by using the server-side application to consolidate files. This keeps the client-side application maintainable without affecting how the browser will load the scripts necessary for a given interface; this practice supports the monolithic application-loading scenarios with modular application development.

In order to get around this problem, the application can consolidate the files into a single file, requiring only one request to get the functionality of several files. The

following example takes two arguments and implements a consolidation of the list of files specified in the first argument, saving the result to the path specified in the second. This static method exists in a generic, globally accessible `Utilities` class of the application:

```
/**
 * Consolidates files into a single file as a cache
 */
public static function consolidate($files, $cache) {
    $lastupdated = file_exists($cache) ? filemtime($cache) : 0;
    $generate = false;
    foreach ($files as $file) {
        // Just stop of missing a source file
        if (!file_exists($file)) {
            return false;
        }
        if ($u = filemtime($file) > $lastupdated) {
            $generate = true;
            break;
        }
    }
    // Files changed since the last cache modification
    if ($generate) {
        $temp = tempnam('/tmp', 'cache');
        $temph = fopen($temp, 'w');
        // Now write each of the files to it
        foreach ($files as $file) {
            $fileh = fopen($file);
            while (!feof($fileh)) {
                fwrite($temph, fgets($fileh));
            }
            fclose($fileh);
        }
        fclose($temph);
        rename($temp, $cache);
    }
    return true;
}
```

When using this script on the first load with the same six files, the full script loads in 45 milliseconds on the first hit and in an average of about 35 milliseconds from then onward. When using this script on the first load with the full twelve files, the full script loads in 80 milliseconds on the first hit and in an average of about 50 milliseconds from then onward.

This method can have other functionality built into it in order to make the page load even faster, especially for rather large applications. Once it consolidates the files into a single, cached file, it then can create a secondary, gzipped file by running `copy('compress.zlib://' . $temp, $cache . '.gz')` so that the browser can load an even smaller file. It even can run the script through a tool to condense the script itself by removing comments and white space and by shrinking the contents of the script prior to gzipping.

By using these methods, even megabytes of script necessary for a rich, full interface can load quickly. The expanses of functionality will add more to the application without dragging down its performance and becoming unwieldy.