ARCADA

Developing jQuery Plugins: Best Practices

Rickard Lindgren

Degree Thesis
Information and Media technology
2013

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informations- och medieteknik
Identifikationsnummer:	4165
Författare:	Rickard Lindgren
Arbetets namn:	Utveckling av jQuery insticksmoduler - bästa praxis
Handledare (Arcada):	Hanne Karlsson
Uppdragsgivare:	Milk+Chocolate

Sammandrag:

Detta examensarbete behandlar bästa praxis i utvecklingen av jQuery insticksmoduler. jQuery är ett populärt JavaScript ramverk som ger utvecklare möjlighet att utvidga den med insticksmoduler. Dessa instiksmoduler har en specifik struktur som kan förbättras med de bästa praxis och mönster som presenteras i detta arbete. I detta arbete presenteras bästa praxis och mönster från utvecklings och lanserings skedet av utvecklingsprocessen. Kodexempel visas för de olika praxis som presenteras med och utan det mönster som beskrivs för att visa nyttan av praxisen. Varför dessa bästa praxis är nyttiga och ger bättre prestanda åt en utvecklares kod är ämnen som behandlas. Detta arbete presenterar bästa praxis som är allmänna och användbara för utvecklare som arbetar med insticksmoduler. Målsättningen med arbetet är att ge uppdragsgivaren ett dokument som ger en grund för nybörjare i jQuery insticksmodul utveckling. De bästa praxis som tas fram ger en stadig grund för utvecklare.

Nyckelord:	JavaScript, praxis, jQuery, plugins, utveckling, milk & chocolate, lansering
Sidantal:	39
Språk:	Engelska
Datum för godkännande:	7.5.2013

DEGREE THESIS	
Arcada	
Degree Programme:	Information- and media technology
Identification number:	4165
Author:	Rickard Lindgren
Title:	Developing jQuery Plugins: Best Practices
Supervisor (Arcada):	Hanne Karlsson
Commissioned by:	Milk+Chocolate

Abstract:

This thesis cover best practices in jQuery plugin development. jQuery is a popular JavaScript framework that enables developers to expand it by writing so called plugins. These plugins have a specific structure that can be augmented by the use of best practices presented in this thesis. In this thesis best practices and patterns are presented from the development and deployment parts of a plugins development process. Code examples are presented with and without the best practices mention to illustrate the advantages of each pattern. Why best practices are beneficial is discussed as is the performance advantages of particular patterns and practices. The thesis has a particular set of best practices that have been chosen as the most general and useful for a developer working with plugins. The goal of the work is to give developers are baseline to work from when building jQuery plugins. The thesis has best practices that have been extensively researched and are favored.

Keywords:	JavaScript, best practice, jQuery, plugins, development, deployment
Number of pages:	39
Language:	English
Date of acceptance:	7.5.2013

Contents

Acronyms and concepts 1

1. Introduction 2

1.1 Background 2

1.2 Methods 2

1.3 Scope and Goals 2

1.4 Best practice definition and philosophy 3

1.4.1 A best practice example in JavaScript 4

1.4.2 Thinking ahead 5

2. Javascript: Then and Now 5

2.1 ECMAScript, JavaScript and Standards 5

2.2 Applications and popularity 6

2.3 Frameworks and jQuery 7

jQuery Plugin Development Best Practices 7

3.1 Basic jQuery plugin best practices 8

3.1.1 Extending jQuery with plugins 8

3.1.2 A foundation to build on 8

3.1.3 Maintaining chainability 9

3.2 Avoiding conflicts and handling dependencies 10

3.2.1 Namespacing jQuery plugins 11

3.3 Adaptable code 12

3.3.1 Overridable options 13

3.4 Understandable and styled code 15

3.4.1 Spacing 15

3.4.2 Assignments 16

3.4.3 Comments and Quotes 16

3.4.4 A word on consistency 17

4. jQuery Plugin Deployment Best Practices 17

4.1 Code processing 18

4.1.1 Optimizing for minification 19

4.1.2 Concatenation 21

4.2 Testing 21

4.2.1 The most volatile of environments 21

4.2.2 Code linting with JSLint 22

4.2.3 Unit testing with QUnit 23

4.3 Documentation 25

4.3.1 Internal & External 25

4.3.2 Plugin documentation 26

5. Conclusions 26

References 28

Appendices 33

Appendix 1 – Addy Osmani namespaced jQuery plugin pattern 33

Appendix 2 – jQuery Bigger 34

Acronyms and concepts

API – Application Programming Interface

CSS – Cascading Style Sheets

DOM – Document Object Model

ECMA – European Computer Manufacturers Association

Framework – A collection of methods for developers

HTML – HyperText Markup Language

HTML5 – The newest version of HTML

jQuery – A JavaScript framework

JS – JavaScript

Linting – A process for finding errors in code

Unit testing – A type of testing that tests units of the source code of a program

Plugin – An extension to a framework

Pattern (Software) – A reusable solution to common problems

Source code – An applications code

1. Introduction

1.1 Background

This thesis has been commissioned by Milk+Chocolate, a digital creative agency with customers ranging from large multinationals to smaller brands and companies in fields as varied as fashion and politics. Milk+Chocolate take pride in the quality of their products and by extension the quality of the code produced.

1.2 Methods

This thesis includes code taken from well-known patterns and from small code snippets developed by the author. The code will be examined and presented to the reader with a description of the best practice that has been used. These examples have been implemented with the best practice that is described. To better illustrate to the reader what the code does function and variable names have been simplified for better clarity in the code.

The best practices have been chosen through study of literature from leading JavaScript and jQuery developers in book and blog form. The choice of best practices has been highly influenced by a project the thesis commissioner has built.

1.3 Scope and Goals

This thesis will discuss best practices specific to the jQuery framework. Many of the best practices discussed can be applied to JavaScript as well but will not be elaborated on in the context of plain JavaScript. Best practices in software development can not only be applied to code but also to how the code is presented, documented, tested and deployed. All of these parts will be discussed and analyzed.

What do best practices provide for program developers and what do they contribute to the workflow of a developer? Does a common structure for elements enable developers to work better on code that is changed by multiple persons.

The objective of this thesis is to improve code modularity, style and performance in projects that include JavaScript and jQuery. The objective is to provide guidelines for program developers so that they may write code that is all of the above.

1.4 Best practice definition and philosophy

A best practice is defined as the recognized methods of correctly running businesses or providing services (Collins 2012). This is a definition that can be applied to many different industries and fields within industries and is the broadest definition of the term.

Within software programming a best practice can in many ways be equated with a software design pattern and often is *exactly* the same. Not all best practices are software patterns and best practices within the software industry also include how all of the auxiliary parts of the software development process are executed. This can include how documentation is written, how code is maintained and updated and in what ways the program or code may be used by others. These are points that will be discussed throughout this thesis.

What is the philosophy behind a best practice? Who benefits and in what way? These questions can be answered in a more simple way for fields other than software programming. A best practice in preventative medicine is to clean a minor wound and apply a topical antiseptic ointment. Within aviation eliminating distractions within the operational area (FAA 2012) is a common best practice that is very logical to follow even for a person who is not versed at all in the art of flying. These are things that seem obvious and are when thought

about. The same cannot be said about best practices within software. An example of this is in order.

1.4.1 A best practice example in JavaScript

```
/* Example 1 */
var foo = function() {
      for (var i = 0; i < array.length; i++) {
            var element = array[i];
            //Do something with element
      }
}

/* Example 2 */
var foo = function() {
      var element = null;
      var i = null;
      for (i = 0; i < array.length; i++) {
            element = array[i];
            //Do something with element
      }
}
```

Both examples illustrated above work for iterating through a list of items in a JavaScript array. They do exactly the same thing. How is the second example better? Such a deceptively simple example requires a deep knowledge in software development and JavaScript to truly understand why the seemingly more complex version of the same thing is actually better. In the first example the variable `element` is declared inside the loop. In a language with a so called *block scope* this would mean that the variable is initialized as many times as the loop is run and thus used up more processing than is needed for just assigning a new value to the variable. Rapidly explained *block scope* means that a variable is only accessible within the block of code it is declared in (a function loop for instance). JavaScript does not have *block scope* and thus there is not a performance gain by declaring the variable before it is used. The best practice is

justified by the fact that JavaScript not having a block scope is unusual and thus programmers coming from other languages, that have block scope, could be confused by the variable declaration within the loop. (Stack Overflow 1 2010)

1.4.2 Thinking ahead

Here the best practice of declaring variables before they are used come from what is logical when thinking ahead to when another programmer needs to modify the code. A programmer that has spent a large amount of time building programs with JavaScript knows that declaring variables inside a loop is not a problem or performance detriment. Another programmer with most of their experience coming from C or another language with block scope would notice the variable declaration inside the loop and possibly be confused of the pattern used. (Crockford 2009)

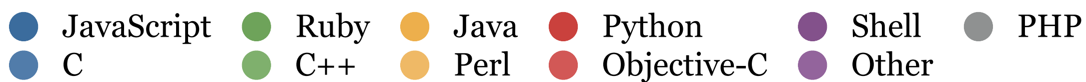
2. Javascript: Then and Now

2.1 ECMAScript, JavaScript and Standards

JavaScript is a scripting language developed in 1995 by Brendan Eich, a computer programmer who at the time worked for Netscape Communications. Eich developed the language mainly as a way to validate forms on webpages so that a user would not need to wait for a server to respond with a message if the input was incorrect (Zakas 2012:1). When Microsoft released Internet Explorer 3 and with it their own implementation of JavaScript named JScript a need for a common standard was realized and JavaScript was submitted to the European Computer Manufacturers Association (ECMA5 2011) for standardization (W3C 2012). The standardization of the JavaScript core as ECMAScript paved the way for the languages current status as one of the most used programming languages in the world. (Tiobe 2012)

2.2 Applications and popularity

JavaScript has had a renaissance in recent years that has been propelled by virtue of being the only true programming language available in a wide variety of web browsers. This has lead to JavaScript being used for many tasks it was not designed for and thus many of the languages limitations are being hit. Rapid development in the web browser vendor community has lifted many of the obstacles that previously stood in the way (Google 2013). This has enabled the development of applications on the web that have many of the same capabilities as an equivalent native program on an operating system. As more and more applications are being moved to the web, the abilities of JavaScript are being fully realized by developers. The current popularity of JavaScript in the web community can best be illustrated by the top languages in use on the social coding site GitHub (figure 1).



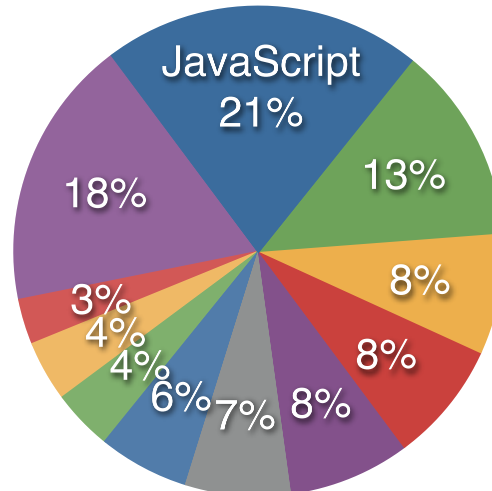


Figure 1. Top Languages on GitHub (GitHub 2013)

Even though JavaScript has a specification that can be followed web browsers vendors have built interpreters that behave differently. This has lead to incompatibilities across browsers that are still a cause of frustration for many developers. This is where frameworks such as jQuery come in to play.

2.3 Frameworks and jQuery

A software framework is a collection of reusable tools and functions that are built to help a developer build applications better and faster (Maxxess 2012). Frameworks have a long history in software development since they give developers the ability to much more rapidly develop functioning applications by simplifying common tasks such as building a login system for a web application or building a database based on variable names. These are tasks that can be done manually but are often tedious and susceptible to bugs.

jQuery is a framework built in JavaScript. The jQuery foundation (jQuery 2013) describe jQuery as "*a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development.*"

An example of simplified HTML document traversing is illustrated below:

```
/* Regular JavaScript */
element = document.getElementById("id");

/* jQuery */
element = jQuery("#id");
```

The function above retrieves an element by its ID. The two line of code above are not strictly equivalent since the jQuery version can do much more than just retrieve elements by their ID, but is enough to give an example of what a framework can do to simplify everyday tasks for developers.

3. jQuery Plugin Development Best Practices

Plugins have made jQuery such a popular library and enables developers to add functions to jQuery in a manner that is identical to adding functions to the

jQuery source code itself. The best practices in this chapter cover how this is done in a manner that is correct and efficient.

3.1 Basic jQuery plugin best practices

3.1.1 Extending jQuery with plugins

Plugins are one of the main reasons for jQuery's popularity. The ease with which a developer can extend jQuery by extending the jQuery `$.fn` object and the user of the plugin can invoke the plugin are large factors in this. In essence a plugin is a regular function that is added to the jQuery object (jQuery Docs 2010). The simplicity of this approach gives a developer very quick start in building plugins. Building a plugin to use with jQuery instead of modifying the `$.fn` object directly is a best practice that is considered to be obligatory. Updating frameworks that have been modified is inconvenient and often not possible without a considerable amount of work.

3.1.2 A foundation to build on

Adding a function to the jQuery `$.fn` object is simple.

```
jQuery.fn.plugin = function() {};
```

It is common practice in the jQuery community to use the dollar sign (`$`) as a shorthand property for referencing jQuery. To make sure that there are no conflicts with regards to the use of the dollar sign the plugin function should be wrapped inside a immediately invoked function expression to which the jQuery object is passed (jQuery Docs 2010). This means simply put that the function is run immediately when it has been loaded.

```
/* Dollar sign is mapped to the jQuery object */
;(function( $ ) {
  $.fn.plugin = function() {};
})( jQuery );
```

This foundation safeguards against two things. The semicolon before the function expression makes sure that any other libraries or scripts that have not been closed correctly do not interfere with the functionality of the plugin. Passing the dollar sign shorthand into the function enables the safe use of the sign within the function without having to worry about conflict with other libraries.

jQuery plugins are generally used to manipulate the DOM (Document Object Model) and this means that plugins need to access the global window object through which developers can access the document object. The `window` object can be passed to the plugin through the function expression used to pass the jQuery object (Irish 2010).

```
;(function ( $, window, document, undefined ) {
    $.fn.plugin = function () {};
})( jQuery, window, document );
```

Here the jQuery (`$`) is passed in with `window`, `document` and `undefined`. Passing the global window object with document to the function as local variables improves the object resolution process (Osmani 2011). Passing undefined into the function is a way to safeguard against accidental or malicious overwriting of `undefined` in ECMAScript 3 (Irish 2010). ECMAScript 3's `undefined` is overwritable (ECMA3 1999) and can be tampered with by any JavaScript code. The ECMAScript 5 language specification has changed this so that `undefined` is non-writable (ECMA5 2011). All browser have not yet adopted ECMAScript 5 and thus the passing of undefined into the function and not passing it out guarantees that `undefined` behaves in the intended way.

3.1.3 Maintaining chainability

The ability to chain jQuery functions is one of the most useful features of the framework. Maintaining chainability is very important when building a plugin as it is an expectation from users that the plugin is chainable as it is such a heavily used feature of the jQuery framework. Chainability has been popularized

by jQuery but is not exclusive to it. The chainability seen in jQuery is a type of prototypal inheritance that can be done with normal JavaScript also (Padolsey 2009).

```
$("h1").bigger().addClass("big").css({ color : "red" });
```

The model that jQuery follows, allows the developer to run many functions on a selected element in a sequence. In the example the heading 1 element on the page is made bigger, gets a class called "big", and has its color changed to red through calls to three functions.

The best practice for maintaining chainability in a jQuery plugin is achieved by returning what has been passed to the function with the $.each() method.

```
;(function ( $ ) {
    $.fn.plugin = function () {
        /* return the elements passed in */
        return this.each(function() {
            var $this = $(this);
        });
    };
})( jQuery );
```

Since the elements passed in to a jQuery function can be one element or a selection of elements the $.each() method is used to return all of the passed in elements in order. This allows for the chainability that is expected.

3.2 Avoiding conflicts and handling dependencies

JavaScript uses the global object `window` to reference functions and variables. This means in practice that declaring a function attaches it to the window variable. This can be best illustrated with the example below.

```
globalTest = "Testing 123";

var alertTest = function() {
    alert(window.globalTest);
    // alerts "Testing 123"
}
```

In the example a variable is initialized outside of a function. As JavaScript has *function scope* this means that the variable is accessible globally by any function. In practice this means that the variable is added to the `window` object. This presents a problem as any variable attached to the window object can be overwritten any time by any code. To combat this problem a developer should implement the best practice of namespacing their code. Namespacing in JavaScript is not natively supported as it is in many other high level languages such as Java or Go. In a language such as Java a program is automatically namespaced (Flanagan 1999) and as such cannot cause conflicts in the same way a wayward function in JavaScript can.

3.2.1 Namespacing jQuery plugins

jQuery is regular JavaScript and follows all of the concepts mentioned previously. jQuery itself is an extension of the `window` object and is globally accessible once it is loaded.

```
// Writes out the jQuery object
> console.log(window.jQuery);
  function (a,b){return new p.fn.init(a,b,c)}
```

A plugin for jQuery is an extension of the `$.fn` object (jQuery Docs 2010). What a developer does when building a plugin for jQuery is simply adding a function to the jQuery framework which can then be accessed as any other jQuery function.

JavaScript's lack of built-in support for namespacing can be worked around by creating a limited number of global objects that serve as wrappers for the rest of

a developers code. This is accomplished in practice by creating an anonymous function and within it creating an object that serves as a namespace. The plugin code is then attached to the namespace and then the $.fn object.

```
;(function ( $ ) {
  // Check if namespace has already been initialized
  if (!$.namespace) {
    $.namespace = {};
  };
  // Create our plugin object
  $.namespace.plugin = function ( elements, options ) {};
  // Extend the $.fn object
  $.fn.namespace_plugin = function ( options ) {
    return this.each(function () {
      (new $.namespace.plugin(this, options));
    });
  };
})( jQuery );
```

Here the namespace is added to the jQuery object with `$.namespace = {};` and is used as a wrapper for the rest of the plugins code. The plugin is then added to the $.fn object in the form of `namespace_plugin` so it can be called. This way of namespacing plugins is the preferred way going forward as it avoids conflicts but also enables developers of jQuery plugins to use more generic names for their functions such as slider() for an image gallery or date picker() for a function that enables a user to choose a date from a calendar, two very common plugin functions in the world of jQuery. Another benefit is that if the developers namespace remains consistent the code can be recognized easily by the developers namespace i.e.. `$.mc` for Milk+Chocolate.

This example is based on Addy Osmani's namespaced jQuery pattern that can be found in appendix 1.

3.3 Adaptable code

One of the strengths of jQuery is the amount of adaptability that can be built into functions, plugins and objects. jQuery plugins and functions are generally invoked on a DOM element that is then manipulated in some way by the

function. A basic scenario could be that a developer would like to increase the size of all headings on a page after the user has done specific action.

```
// Make all heading 1 elements bigger
$("h1").bigger();
```

Here the `h1` element is passed into the `.bigger()` function where it is manipulated. The plugins default option is to set the font-size of the element that is passed in to `50px`. If the uses wishes to use another value it has to be added to the function call.

3.3.1 Overridable options

When developing a jQuery plugin that should be easily adapted by the user a developer needs to handle the incoming options in a manner that avoids conflicts and enables the user to change any aspect of the functions the plugin uses. The best practice is to build code that uses overridable options for all settings. In practice this means much more than giving the user a way to change given settings. The developer of the plugin needs to use the options throughout the code and avoid using variables that are not declared in the options of the plugin.

A pattern for accepting options that uses jQuery's build in `$.extend` function is the correct way to override the default options that are set by the developer of a plugin.

```
;(function ( $, window, document, undefined ) {
    // Options are received by the plugin
    $.fn.bigger = function ( options ) {
        // Merge the passed in options with the default options
        options = $.extend( {}, $.fn.bigger.options, options );
        return this.each(function () {
            var elem = $(this);
            // The options is used to determine the font-size
            elem.css({ "font-size" : options.fontSize });
        });
    };
    // Default options specified by the developer
    $.fn.bigger.options = {
        fontSize: "50px"
    };
})( jQuery, window, document );
```

This pattern allows the developer to override options every time the plugin function is called and also set options globally (Alman 2010). What this means is that instead of having to pass options to the function on each call the developer may set the options for the plugin once and have it use those options each time it is called.

The `$.bigger()` plugin can be used as an example. If the user of the plugin wants to always increase the font-size to 100px he could pass the options to the function each time the function is called.

```
/* Options are passed on each call */
$("h1").bigger({ fontSize : "100px" });
$("h2").bigger({ fontSize : "100px" });
$("h3").bigger({ fontSize : "200px" });
```

As the `$.bigger()` plugin allows for globally overridable options the preferred way of accomplishing the increase of the font-size for the different elements is to set a the fontSize option before invoking the function.

```
/* Options are set globally once */
$.fn.bigger.options = { fontSize : "100px" };
$("h1").bigger();
$("h2").bigger();
$("h3").bigger({ fontSize : "200px" });
```

Here we may set the h1 and h2 elements to our new default font size but the third call to the `$.bigger()` function accepts a new option for the font size and proceeds by overwriting the globally set option as specified in the overridable options pattern.

The complete source for the `$.bigger()` plugin can be found in Appendix 2.

3.4 Understandable and styled code

One of the easiest best practices to adopt as a developer is structuring code in a consistent manner. In the world of jQuery this means following the jQuery JavaScript style guide which gives a comprehensive but concise overview of how to write JavaScript in the same way as the developers of jQuery.

3.4.1 Spacing

The jQuery JavaScript style guide states that developers should use tabs for indentation, no unnecessary whitespace at the end of lines and use spacing liberally. All examples are taken from the jQuery JavaScript style guide. (jQuery Contribute 2013)

```
// Bad
if(condition) doSomething("something");
// Good
if ( condition ) {
    doSomething( "something" );
}
```

Spacing liberally improves readability dramatically and avoids confusion. The use of brackets in the good example makes it clear that the function doSomething() depends on the if condition. In a program with many hundreds of lines of code this improves the chances that misunderstandings regarding the function of the if statement do not occur. Adding spaces to the function calls attributes adds clarity to what is passed in to the function.

3.4.2 Assignments

Creating and assigning variables should be done in a clear way. When creating variables for later use without a value they can be put on the same line while a new line is required if a value is assigned on declaration (jQuery Contribute 2013).

```
// Bad
var foo = true;
var bar = false;
var a;
var b;
var c;
var object = {};
var array = [];

// Good
var a, b, c,
    foo = true,
    bar = false,
    object = {},
    array = [];
```

By following this style repetition is avoided and clarity is upheld. Here the use of spacing is also demonstrated again as the equals sign between the variable and its value has a space on each side and the empty object and array do not have any spaces within their braces and brackets respectively.

This style of assigning variables is widely supported and can also be found in the Idiomatic.js (Idiomatic.js 2013) and Google JavaScript style guide (Google Style).

3.4.3 Comments and Quotes

Comments that span one line should be denoted with two slashes and multiple line comments should use the slash and star syntax (jQuery Contribute 2013).

```
// Single line comment
/*
Multi-line
comment
*/
// Use double quotes with jQuery
$("h1").bigger();
```

This commenting system signals to the reader if the content is a short description or a more comprehensive text such as a feature explanation. The use of double quotes is preferred to single quotes when using jQuery but also enables the use of single quotes within the double quotes when needed.

3.4.4 A word on consistency

The purpose of style guides is to provide a vocabulary for developers so that another developer who uses the code or needs to change it can concentrate on what is being said and not how it is being said (Google Style).

None of the rules presented in any kind of style guide matter if they are not followed in a consistent manner. Even if there isn't a style guide available for the type of code a developer is currently writing there should always be consistency. When a developer continues development on a for example a jQuery plugin can notice patterns in how the existing code is structured. If all function calls have comments before them describing what the function does then the new code that is written should also have a comment before it. What is most important is that *a* style is followed so that extensive refactoring is not required when a new feature has to be added.

4. jQuery Plugin Deployment Best Practices

Developing a jQuery plugin can be a fast task that takes a few minutes and solves a specific problem in a project. Developing a jQuery plugin can also be a project of many *years* and could mean that many thousands of developers use

the code. For the latter case a set of deployment best practices and patterns can be followed to improve code quality and also to make it easier for others develop the code further.

4.1 Code processing

The current popularity of JavaScript has brought with it many different tools for optimization. The most used are different kinds of code minifiers such as the YUI Compressor developed by Yahoo (YUI).

JavaScript is an asset that has to be downloaded by the end user before it is executed in the browser environment. This means that any savings in file size speeds up and improves the user experience.

Minification is a system for optimizing code in a way that removes unnecessary characters and symbols from the source to lessen the amount of data that has to be stored. As minified code is not meant to be changed or edited in the minified state. No comments or clear and understandable function names are needed and thus a very large amount of characters, symbols and whitespace can be removed. To illustrate a very basic minification the example pattern for a jQuery plugin can be used.

```
// Before
;(function ( $ ) {
    $.fn.plugin = function () {
        /* return the elements passed in */
        return this.each(function() {
            var $this = $(this);
        });
    };
})( jQuery );

// After
;(function(a){a.fn.plugin=function(){return this.each(function(){var
b=a(this)})}})(jQuery);
```

In this example 209 characters were input and the minified code had 91. This means that a 56% reduction was achieved with no effort at all on the developers part.

4.1.1 Optimizing for minification

While minification works on any JavaScript code that does not have syntax errors there are ways to improve the minification ratio. There are two main ways to help a minification algorithm which will be discussed.

To be able to effectively minimize anything the minification process relies on repetition of constants.

```
function toggle( element ) {
  if ( $(element).hasClass("selected") ) {
    $(element).removeClass("selected");
  } else {
    $(element).addClass("selected");
  }
}
```

The minification process will not create variables for repeated strings. The minified version of this code is not as small as it could be.

```
function toggle(a){if($(a).hasClass("selected")){$
(a).removeClass("selected")}else{$(a).addClass("selected")}};
```

An improved version would be to store the string as a variable to improve the compression ratio.

```
function toggle( element ) {
  var selected = "selected";
  if ( $(element).hasClass(selected) ) {
    $(element).removeClass(selected);
  } else {
    $(element).addClass(selected);
  }
}
```

```
function toggle(a){var b="selected";if($(a).hasClass(b)){$
(a).removeClass(b)}else{$(a).addClass(b)}};
```

This approach enables the compressor to recognize the variable and replace the occurrences of it with a very short variable name. This is noticeable in files with variables that are repeated hundreds of times such as the jQuery source code. For version 1.9.1 of jQuery this means that there are 277977 characters in the source code before minification and 104761 after it. A saving of 62% in file size is sizable and once the amount of work needed to accomplish the savings are taken into account the choice to minify is apparent.

The example pattern for building a jQuery plugin has `window` and `document` passed to it in the function declaration. This is not only for the improved resolution process but also for the improved minification ratio.

```
;(function ( $, window, document, undefined ) {
    window.onload = function() {
        console.log("load");
    }
    $(document).ready(function(){
        console.log("ready");
    });
})( jQuery, window, document );
```

```
;(function(c,b,a,d){b.onload=function()
{console.log("load")};c(a).ready(function(){console.log("ready")})})
(jQuery,window,document);
```

The two keys to improved minification is declaring variables for often used strings and and storing local references to objects and values (Zakas 2008).

4.1.2 Concatenation

In a real-world situation many different JavaScript plugins and libraries can be needed for an application or page. Adding them into one http request helps speed up an application or page. (Zakas 2012) This can be accomplished in a few different ways but the method is not as important as the outcome.

A basic page that utilizes JavaScript and jQuery often has several scripts that are needed for the webpage to function. A restaurant website can be taken as an example. They have a slideshow on the front page and a date picker for reservations on the reservations page. It is presumed that the webpage uses jQuery. For this functionality the webpage uses the jQuery library, a plugin for the slideshow and the date picker, and a script file that runs the required functions once the page has loaded. This adds up to four files that need to be requested by the browser for the site to function.

Basic concatenation for a site with the setup mentioned is to include jQuery and a script file which contains the rest of the required files. jQuery should be included separately for easier upgrading and caching (Coyier 2010) and the rest of the scripts into one file. This is something that is done when pushing the code into a production state.

4.2 Testing

4.2.1 The most volatile of environments

JavaScript typically executes in one of the most volatile run-time environments that have been in wide use. This is the web browser, where many different companies build completely different engines to run JavaScript. These different engines are often touted as the reason one web browser is better than another.

The different platforms and implementations of JavaScript engines has resulted in a plethora of different environments that have to be taken into account by a

developer. On top of this different browsers support different versions of JavaScript (ECMAScript 3 or 5) (Kangax 2012) and the implementations of the languages are not always exactly to specification.

Different implementations and environments lead inevitably to bugs. No JavaScript engine is perfect and many have specific bugs that are known but have not and will not be fixed due to the browser where it is found not being updated anymore, as in the case of a few Internet Explorer 8 bugs (Stack Overflow 2).

One of the goals of the jQuery project was to give developers a toolbox of functions that could be used with confidence without having to worry about different browser implementations of a function which could behave in a different manner than what was intended by the developer.

Even though jQuery enables the developer to write JavaScript in a consistent manner for different platforms there is of course still the change for bugs in the developers own code.

4.2.2 Code linting with JSLint

When the C programming language was in its infancy there were errors which the compilers of that day were not able to detect. To combat this problem a small code checking program called lint was developed by Bell Labs (Johnson 1979). As the C compilers were developed further and the language specification for C was stabilized the need for lint was no longer needed (Crockford 2008).

JavaScript has not had the same chance to mature as C as it was not at its stage of invention intended for use in many of the ways it is now. This has led to a need for a syntax checker and verifier for JavaScript, a lint for JavaScript. Fortunately Douglas Crockford, a JavaScript developer, saw the need for this and developed JSLint.

JSLint is a code quality tool which uses a strict subset of JavaScript to check your code against and rejects code that is accepted by browsers (Crockford 2008). The most common things that JSLint captures are undeclared variables and inconsistent whitespace. The errors are presented with an explanation and a approximate location within the code.

```
Line 132 [col. 5]
test = 0;
'test' is not defined.
```

This is a valuable tool for spotting errors that could otherwise pass undetected because it is valid JavaScript but could lead to performance or other problems down the line. Code that does not pass through JSLint without errors is not acceptable. Thus the recommended best practice is to always check any JavaScript code that will be used in a production environment with JSLint.

4.2.3 Unit testing with QUnit

The testing of code is one of the most important parts of any software project. The subject of testing is something that can and has been explored extensively throughout the history of programming. Ways of programming, such as Test Driven Development, have emerged which require a systematic approach to the development process which requires extensive testing.

Testing JavaScript is a challenge as the code is often not stand alone and can in many instances be intermixed with HTML as inline elements on a web page. This presents challenges for developers who wish to implement unit testing. A unit is in its most simple form a function that gives a output based on an input. This is not very often the case when testing JavaScript where the output of a specific function can depend on what is present in the DOM. QUnit is a JavaScript unit testing framework build by the jQuery foundation which aims to simplify the process of testing for JavaScript developers.

One area where unit testing shines is as a tool for refactoring. Refactoring is the process of rewriting or restructuring code without modifying its behavior.

(QUnit 2013). As a section of code is changed the chances for bugs are increased and developers need a way to easily ensure that the functionality of a program stays the same as it is improved from a code structure standpoint.

Including unit tests alongside the code you have developed is considered a best practice, particularly if the code is to be changed or improved by other developers. The tests may also help other developers understand how a more complex program is supposed to behave. The inclusion of tests in a program that is developed by many individuals helps in the process of programming. This is because a developer not very familiar with the code can get into improving small pieces of it without having to worry about breaking something down the line. This is of course only possible if the tests for the program are well structured.

For better understanding of how QUnit works an example is needed. In its most basic form a unit test is setting in an expected result an comparing it to what the function returns.

```
QUnit.test( "square test", function( assert ) {
      function square( x ) {
            return x * x;
      }
      var result = square( 2 );
      assert.equal( result, 4, "square(2) equals 4" );
});
```

The QUnit test function test for the expected result of 4 when 2 is passed to the square() function. The result is then presented as passed.

Figure 2. QUnit result example.

Figure 2 hardly illustrates the power of the QUnit testing library but gives a feeling for the power of unit testing.

A complete testing battery included with a plugin gives developers confidence to change even functions that are critical to a plugins functioning. This makes the chance that other developers will improve your code greater.

4.3 Documentation

Documentation is a vital part of any software project and can be critical if the aim of a developer is for their plugin to become popular. If users of a plugin and other developers need to read the code for a plugin to understand how to use it then the threshold for usage is raised substantially.

Vital parts of any documentation are a description of the functionality, how to use the functionality and an example (Watters 2010). With these a plugin developer has a good start and what is needed for a user to get up and running. This is adequate documentation and taking the work to another level requires a substantial amount of work.

4.3.1 Internal & External

The documentation of a plugin should be internal, in the code, and external, as a separate document describing the functionality on a higher level. The internal

code documentation can describe in detail what a function does and what a variable is supposed to contain. The internal should be concise and describe only necessary parts. Any longer descriptions or philosophical choices in how a function has been structured should be relegated to the external documentation. (Watters 2010)

4.3.2 Plugin documentation

A decision has to be made for how the documentation for a plugin should be structured. A popular way is the tutorial style of documentation that is a how-to of the most basic usage of the plugin (Watters 2010). This enables users to get a quick start and understand the basics. Another style is the deep dive reference style which many higher level programming languages use in their API documentation (Kaplan-Moss 2009). This way of documenting is not as suited for plugins since the scope of a plugin is nothing near the scope of a high-level programming languages API.

From the tutorial stage of documentation a developer can choose to dive deeper but for the purposes of many plugins this is superfluous. If further documentation is required the next step would be to describe specific functions in a deeper manner.

5. Conclusions

This thesis presents simple but key practices and patterns that are valuable to any developer who works with jQuery in general and its plugins in particular. The best practices presented have been used and recommended by people on the forefront of the JavaScript world.

The use of the best practices presented ensures that the groundwork for a reliable plugin is set for the development and deployment processes. They also give a developer not accustomed to jQuery a head start in writing jQuery flavored JavaScript well. A more abstract advantage of using best practices is

the confidence it gives to developers. The feeling of confidence in their code is something that every developer worth their salt should strive for. This thesis also gives a baseline for developers so that they have a guide to follow which is preferred and by virtue speeds up development.

There are many things that have been left out of the thesis as discussing all best practices would not be possible. I have chosen the ones that can be used on any project that includes the use of jQuery plugins and not just esoteric situations. The included best practices were chosen based on what I have found that was not clear to me with the JavaScript language but also discussions with other developers. This has worked adequately but I feel now that the work would benefit immensely by going even deeper into the topics that are included and by removing parts of the the work such as the style of how the code is written.

Milk+Chocolate will able to use this document for internal training as jQuery development is a large part of the development workflow at the company. This has been done already while the thesis was still in the editing phase and the results were encouraging but improvement suggestions were also presented. One suggestion was to include an even deeper dive into the world of JavaScript best practices and not just jQuery plugins. One reader also suggested that the work should include even more topics to cover a wider breath of the jQuery development field. These are all improvements that I agree with and can see that will be added to the work as it continues it’s life as a living document on the Milk+Chocolate wiki.

Best practices are recommendations. It is inherent in their name. Who defines what is a best practice? Sometimes the best practice is logical and clear even to the uninitiated as in the example of keeping the area around an aircraft clear of extraneous material. As stated before the same cannot be said of best practices in the world of software development and jQuery plugins in particular.
What can be accomplished by following the best practices described in this thesis is above all an efficiency for accomplishing simple tasks in a manner that is proven to be reliable.

References

Alman, Ben. 2010. *jQuery Pluginization [www]*. Slide 10. available: http://benalman.com/talks/jquery-pluginization.html#10. accessed 9.3.2013

Collins. *English Dictionary - Complete & Unabridged 10th Edition*. available: http://dictionary.reference.com/browse/best practice. accessed 25.2.2013

Coyier, Chris. 2010. *Google CDN Naming Conventions [www]*. available: http://css-tricks.com/google-cdn-naming/, accessed: 04.04.2013

Crockford, Douglas. 2008. *JavaScript: The Good Parts*. 1st Edition. Sebastopol: O'Reilly Media. Inc.. 155 pages

Crockford, Douglas. 2009. *Code Conventions for the JavaScript Programming Language [www]*. available: http://javascript.crockford.com/code.html#variable%20declarations. accessed 4.5.2013

Croll, Angus. 2010. *Namespacing in JavaScript [www]*. available: http://javascriptweblog.wordpress.com/2010/12/07/namespacing-in-javascript/. accessed 9.3.2012

ECMA3. 1999. *ECMAScript Language Specification (Standard ECMA-262)*. Third Edition. 170 pages. available: http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf. accessed 10.3.2013.

ECMA5. 2011. *ECMAScript Language Specification*. Version 5.1. available: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf. accessed 25.2.2013

FAA. *Known Best Practices for Airfield Safety*. available: http://wwwstage.faa.gov/airports/runway_safety/bestpractices.cfm?print=go. accessed 5.3.2013

Flanagan, David. 1999. *Java in a Nutshell*. Third Edition. Sebastopol. CA: O'Reilly Media. 1256 pages. available: http://docstore.mik.ua/orelly/java-ent/jnut/index.htm. accessed: 9.3.2013

GitHub. 2013. *Top Languages [www]*. available: https://github.com/languages. accessed 10.3.2013

Google Style. *Google JavaScript Style Guide [www]*. Revision 2.64. available: http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml. accessed 10.3.2013

Google. 2012. *V8 JavaScript engine. Subversion repository change log*. available: http://v8.googlecode.com/svn/trunk/ChangeLog. accessed 22.5.2013

Idiomatic.js. 2013. *Idomatic.js JavaScript Style Guide [www]*. available: https://github.com/rwldrn/idiomatic.js/#readme. accessed 10.3.2013

Irish, Paul. 2010. *10 Things I Learned from the jQuery source [www] [video]*. available: http://www.youtube.com/watch?v=i_qE1iAmjFg&feature=player_detailpage#t=252s. accessed: 9.3.2013

Johnson, S. C. 1979. *Lint. a Program Checker. in Unix Programmer's Manual*. Seventh Edition. Vol. 2B. M. D. McIlroy and B. W. Kernighan. AT&T Bell Laboratories: Murray Hill. NJ.

jQuery. *What is jQuery? [www]*. available: http://jquery.com/. accessed: 10.04.2013

jQuery API. *jQuery API Documentation [www]*. available: http://api.jquery.com/. accessed: 25.2.2013

jQuery Contribute. 2013. *JavaScript Style Guide [www]*. available: http://contribute.jquery.org/style-guide/js/. accessed: 10.3.2012

jQuery Docs. 2010. *Plugins/Authoring [www]*. available : http://docs.jquery.com/Plugins/Authoring. accessed 9.3.2013

jQuery Learn. *jQuery Learning Center [www]*. available: http://learn.jquery.com/. accessed 25.2.2013

Kangax. 2012. *ECMAScript 5 compatibility table [www]*. available: http://kangax.github.com/es5-compat-table/. accessed 23.3.2013

Kaplan-Moss, Jacob. 2009. *Writing great documentation: what to write [www]*. available: http://jacobian.org/writing/great-documentation/what-to-write/. accessed 29.3.2013.

Maxxess. Systems. 2012. *What is a Software Framework [www]*. available: http://www.maxxess-systems.com/whitepapers/what_is_a_software_framework.pdf. accessed 25.2.2013

MDN. Mozilla Developer Network. *JavaScript [www]*. available: https://developer.mozilla.org/en-US/docs/JavaScript. accessed 25.2.2013

Node. Nodejs.org. 2013. *Node's goal is to provide an easy way to build scalable network programs [www]*. available: http://nodejs.org/about/. accessed 25.2.2013

Osmani, Addy. 2011. *Essential jQuery Plugin Patterns*. available: http://coding.smashingmagazine.com/2011/10/11/essential-jquery-plugin-patterns/. accessed 22.2.2013

Osmani, Addy. 2012. *Learning Javascript Design Patterns*. Version 1.5.2. available: http://addyosmani.com/resources/essentialjsdesignpatterns/book/. accessed 25.2.2013

Padolsey, James. 2009. *Prototypal Chainability [www]*. available: http://james.padolsey.com/javascript/prototypal-chainability/. accessed 10.3.2013

QUnit. 2013. *Introduction to Unit Testing [www]*. available: http://qunitjs.com/intro/. accessed 23.03.2013

Rauschmayer, Axel. 2011. *Patterns for modules and namespaces in JavaScript [www]*. available: http://www.2ality.com/2011/04/modules-and-namespaces-in-javascript.html. accessed 25.2.2012

Stack Overflow 1. 10.9.2010. *Question: JavaScript variables declare outside or inside loop?*. available: http://stackoverflow.com/questions/3684923/javascript-variables-declare-outside-or-inside-loop/3685090#3685090. accessed: 6.3.2013

Stack Overflow 2. 14.9.2010. *Question: IE8 bug in for-in JavaScript statement?*. available: http://stackoverflow.com/questions/3705383/ie8-bug-in-for-in-javascript-statement. accessed: 23.32013

Tiobe Software. 2012. *TIOBE Programming Community Index for February 2013 [www]*. available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. accessed 24.2.2013

W3C. 2012. *A Short History of JavaScript [www]*. available: http://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript. accessed: 24.2.2013

Watters, Audrey. 2010. *Tips for Writing Good Documentation [www]*. available: http://readwrite.com/2010/08/14/tips-for-writing-good-document. accessed 29.3.2013.

WebPlatform. 2012. *Purpose of JavaScript [www]*. available: http://docs.webplatform.org/wiki/concepts/programming/the_purpose_of_javascript. accessed 25.2.2012

YUI. *YUI Compressor Introduction [www]*. available: http://yui.github.com/yuicompressor/. accessed 10.3.2013

Zakas, Nicholas C. 11.2.2008. *Helping the YUI compressor [www]*. available http://yuiblog.com/blog/2008/02/11/helping-the-yui-compressor. accessed 17.3.2013

Zakas, Nicholas C. 2012. *Maintainable JavaScript*. Sebastopol: O'Reilly Media. Inc.. 238 pages

Zakas, Nicholas C. 2012. *Professional JavaScript for Web Developers*. 3rd Edition. Indianapolis: John Wiley & Sons. Inc.. 964 pages.

Appendices

Appendix 1 – Addy Osmani namespaced jQuery plugin pattern

```
/*!
 * jQuery namespaced 'Starter' plugin boilerplate
 * Author: @dougneiner
 * Further changes: @addyosmani
 * Licensed under the MIT license
 */

;(function ( $ ) {
    if (!$.myNamespace) {
        $.myNamespace = {};
    };

    $.myNamespace.myPluginName = function ( el, myFunctionParam,
options ) {
        // To avoid scope issues, use 'base' instead of 'this'
        // to reference this class from internal events and functions.
        var base = this;

        // Access to jQuery and DOM versions of element
        base.$el = $(el);
        base.el = el;

        // Add a reverse reference to the DOM object
        base.$el.data( "myNamespace.myPluginName" , base );

        base.init = function () {
            base.myFunctionParam = myFunctionParam;

            base.options = $.extend({},
            $.myNamespace.myPluginName.defaultOptions, options);

            // Put your initialization code here
        };

        // Sample Function, Uncomment to use
        // base.functionName = function( parameters ){
        //
        // };
```

```
        // Run initializer
        base.init();
    };

    $.myNamespace.myPluginName.defaultOptions = {
        myDefaultValue: ""
    };

    $.fn.mynamespace_myPluginName = function
        ( myFunctionParam, options ) {
        return this.each(function () {
            (new $.myNamespace.myPluginName(this,
            myFunctionParam, options));
        });
    };

})( jQuery );
```

Appendix 2 – jQuery Bigger

```
;(function ( $, window, document, undefined ) {
    /*Options are received by the plugin */
    $.fn.bigger = function ( options ) {
        /* Merge the passed in options with the default options */
        options = $.extend( {}, $.fn.bigger.options, options );
        return this.each(function () {
            var elem = $(this);
            /*The options is used within the code to determine the
font-size*/
            elem.css({ "font-size" : options.fontSize });
        });
    };
    /* Default options specified by the developer*/
    $.fn.bigger.options = {
        fontSize: "50px"
    };
})( jQuery, window, document );
```