QUICK GUIDES FOR A COMPLEX WORLD™

jQuery Plugin Development in 30 minutes™

How to build jQuery plugins that are easy to maintain, update and collaborate on.

ROBERT DUCHNIK

First Edition

you tips on how to make a good profile."

"I already had a LinkedIn account, which I use on a regular basis, but still found the book very helpful. The author gave examples and explained why it is important to detail and promote your account. **Reading this book has motivated me to return to my account** and update it to make it more thorough and attention-grabbing."

jQuery Plugin Development in 30 Minutes

How to build jQuery plugins that are easy to maintain, update and collaborate on

By Robert Duchnik

ISBN: 978-1-939924-59-9

Copyright © 2014-2016

i30 Media Corporation

An *In 30 Minutes*® Book

In30minutes.com

All Rights Reserved

Image credits

Cover design by Monica Thomas for TLC Graphics, www.TLCGraphics.com

All other photographs, images, tables, and diagrams were created by the author or publisher.

Contents

Introduction: **1**
About This Guide: **1**
Why jQuery Plugins? **1**

Section 01 – Creation: 4
Naming: **4**
Closures: **4**
Plugin Function: **5**
Summary: **5**

Section 02 – Prototyping: 6
The Main Loop: **6**
The get() Method: **6**
Plugin Class: **7**
Prototyping: **8**
Summary: **9**

Section 03 – Conventions: **10**
Generate: **10**
Destroy: **10**
Init: **10**
$var: **11**
Private Functions: **11**
This & That: **12**
$.proxy(): **12**
Summary: **13**

Section 04 – Events: **14**
Note on .hover(): **14**
e.currentTarget vs e.target: **14**
Disabling Event Bubbling: **15**
Naming Events: **15**
Naming Functions: **16**
Summary: **16**

Section 05 – Options: **18**

Default Options: **18**
Plugin Options: **18**
Additional Options: **19**
Data Options: **19**
Advanced Options Setter: **19**
Summary: **20**

Section 06 – Setters and Getters: **23**
Setters: **23**
Getters: **23**
Methods: **23**
The Routine: **23**
Auto-Creation: **25**
Summary: **25**

Section 07 – CSS and Themes: **26**
Naming: **26**
CSS: **26**
Setting Themes: **27**
Multiple Themes: **27**
Summary: **28**

Section 08: Effects: 29
States: **29**
Controller: **29**
Summary: **29**

Section 09 – Callbacks: 30
Setup: **30**
Ajax: **30**
Naming: **32**
Summary: **32**

Section 10 – Browser and Mobile Support: **33**
Browser Support: **33**
Extending $.support: **33**
Mobile Support: **34**
Summary: **35**

Section 11 – File Organization: 36

Simple: **36**

With Grunt: **36**

Git: **37**

Versioning: **37**

Summary: **38**

Section 12 – Boilerplate: 39

Section 13 – Conclusion: 44

Contact: 44

Bonus Content

Section 14 – jQuery Methods: **45**

Section 15 – jQuery Utilities: **47**

Section 16 – jQuery Selectors: **48**

Section 17 – jQuery Events: **49**

Glossary: **51**

A Request For Readers: **53**

Introduction to *LinkedIn In 30 Minutes*: **54**

Introduction To *Google Drive & Docs In 30 Minutes*: **58**

About In 30 Minutes® Guides: **61**

Introduction

Thank you for purchasing *jQuery Plugin Development In 30 Minutes*. As an experienced jQuery plugin developer and the operator of a website devoted to jQuery education, I have had many opportunities to talk with other developers and understand what works and what doesn't when it comes to learning how to build plugins. This short guide is intended to quickly get you up to speed with core concepts, which enable you to start building plugins of your own.

About This Guide

I wanted this guide to be as short and concise as possible, and provide only the information needed to start building plugins. Why give long-winded write-ups about every concept? My intention is to avoid the fluff and filler that make up 80% of most programming books, and just get straight to the point.

jQuery Plugin Development In 30 Minutes is intended for people who already have some knowledge and ability with JavaScript and jQuery. In fact, this guide is for anyone who writes JavaScript code and uses jQuery. Ultimately, the goal of this guide is to teach how to write clean and efficient jQuery plugins that are easy to maintain, update and work on with other developers.

Most of the concepts in this guide will not require any advanced knowledge. However, I will provide some explanation concerning more advanced parts, along with further reading for those who are interested in learning more.

jQuery is by far the most widely used library for JavaScript. It is used on more than 50% of websites. Many frameworks, such as Backbone and Twitter's Bootstrap, are built on top of jQuery. Being able to extend and write plugins for jQuery can not only save lots of time, but also makes code much cleaner and easier to maintain.

I believe the power of jQuery is highly underutilized. Most developers will take advantage of its shortcuts and CSS selectors, but most of the time they fail to take advantage of much else. Being able to extend jQuery, whether by adding your own functions, CSS selectors or full-blown plugins, makes you a much stronger and smarter developer.

Why jQuery Plugins?

Why should developers write plugins in the first place? I like to write as many plugins and libraries as I can. Reusability is key in reducing bugs and coding quickly. The more I use a piece of code, the more confident and familiar I become with it, which in turn significantly speeds up my development time.

When I write plugins, it may be for anything from larger-sized plugins for tooltips or select boxes to small jQuery extensions for removing a class by using a regular expression. When I solve a problem that might be reusable, I toss it right away into a utilities file (if it's small enough) or build out a standalone plugin.

In my experience, requirements change quite often, or new situations will arise that weren't anticipated at the start of the project. If the situation can be addressed with a plugin, I just whip open the standalone plugin page, make the updates and pop the new plugin back in. Because the plugin is self-contained, it's easy to recreate the problem, fix it, and get it back into the codebase. This is a far better approach than trying to find some random piece of code in the main codebase, and hoping that a change there won't affect something else.

I write plugins using a standard format. It's easy for me to go back months or even years later and understand exactly what is going on inside that plugin. It's also easy for other developers to jump right in and make modifications. They can test it in the standalone environment, and, if it works, there is some confidence that it will also work once we insert the updated plugin into our main codebase again.

Ultimately, I try to think of my application's main codebase as just stringing together various components and code from many sources. It just controls logic and flow. The real nitty-gritty is handled behind the scenes. This is why frameworks like Backbone are so important — they hide a lot of the details in the background and allow you to just focus on the flow and control of your application.

It's also possible to open source our plugins on GitHub and similar sites. This can generate bug reports and contributions from the community while empowering other developers. In this sense, you will often be far better off investing your time searching for a plugin that is already available in the community and potentially updating that plugin rather than building out a completely new one of your own. A great place to look for plugins is the official jQuery plugins site. Sometimes, purchasing a plugin for a few dollars is the best bet. A purchased plugin will likely come with better support and be much more stable if there is money going into its development.

Finally, a good standalone plugin can also make you a fair amount of money. Many developers make a decent living by simply maintaining and updating one or two crucial plugins that are far better than anything available for free.

In the time we have left, we’ll cover the nuts and bolts of building a jQuery plugin. Let’s get started!

Section 01: Creation

This section covers the most basic, bare-bone pieces required to get your plugin up and running.

Naming

You should start by coming up with a name for your plugin. Here are a few points to keep in mind:

- Avoid random names like `blipper` that wouldn't give anyone an inkling about what the plugin's function might be. Fancy names are reserved for things such as frameworks.
- The name should represent the plugin's function in some way, while avoiding vagueness. For example, if we are building a `tooltip` plugin, it's probably better to call it `tooltip` rather than `tip`.
- Employ unique names to avoid collisions. For instance, rather than `tooltip`, use a name like `wTooltip` or `wTip`.

Closures

Make sure your code is contained within a closure. This will take care of any `namespace` issues so any functions you declare will only exist for the plugin. This is particularly important once we start prototyping our plugin class.

```
(function ($) {
  'use strict';

  // Code here...

})(jQuery);
```

(Online: http://jsfiddle.net/3EDQ4/)

Note that the `strict` declaration is not really required. It's more useful for testing and outputting a stricter set of errors. However, I would recommend using `strict` to help decrease the number of bugs in your plugin.

Plugin Function

We can go ahead and create our plugin function by simply extending the jQuery `$.fn` object. This will make our plugin available to any jQuery elements.

```
(function ($) {
  'use strict';

  // Plugin class and prototype will go here.

  $.fn.wTooltip = function () {

    return this;
  };
})(jQuery);
```

(Online: http://jsfiddle.net/8nsBa/)

We can now initialize our plugin on any element using our plugin name.

```
$('#elem').wTooltip().hide();
```

Note the importance of returning the `this` object. This returns all of the elements you called the plugin method on. This is important in maintaining jQuery's method chaining, which allows you to make additional calls after the plugin call.

Summary

At the most basic level, we now have a plugin function that extends jQuery. We can call this function on any selected elements allowing us to manipulate the `this` object containing all of the elements. In some simple cases, this is even as far as we may need to go.

For example, say we wanted to have a method called `.opacity50()` that simply sets the `opacity` of our elements to `50%`.

```
(function ($) {
  'use strict';

  $.fn.opacity50 = function () {
    return $(this).css('opacity', 0.5);
  };
})(jQuery);
```

(Online: http://jsfiddle.net/w827q/)

Section 02: Prototyping

In this section, we will take things a step further and begin looking at more of an advanced layout for our plugin code.

The Main Loop

Before we get into prototyping, we'll want to setup a main loop for our function. This will be used to iterate through each element that the plugin method was called on.

Note this can be done *many* ways. Here, I'm simply presenting a tried-and-tested method in laying out plugin code.

We'll start by calling the `.each()` method on our `this` object containing all of our elements. We do this because our plugins will typically be a little more complicated than just setting some CSS properties.

```
$.fn.opacity50 = function () {
  return this.each(function () {
    // Code here...

    $(this).css('opacity', 50);
  });
};
```

(Online: http://jsfiddle.net/5tCQL/)

The get() Method

Now that we are iterating through our elements, it allows us to start performing some more advanced operations on each element. However, keeping a function inside of our `.each()` method is not very efficient, as it will instantiate a copy of that function in each iteration.

It's good practice in JavaScript to separate this functionality so the `.each()` method only points to one copy of the function. We'll do that by creating a `get()` function that will house the creation of our plugin object.

```
$.fn.opacity50 = function () {

  function get() {
    $(this).css('opacity', 50);
  }

  return this.each(get);
};
```

(Online: http://jsfiddle.net/8Bzm2/)

Plugin Class

Next, we'll create our plugin class, which is really just a regular function that we'll loosely treat as a class.

The key takeaway is that this will allow us to create objects where we can store properties for each plugin instance. Typically, this will be something like options or anything that keeps track of the state of each individual plugin object.

```
(function ($) {
  'use strict';

  function Tooltip(el) {
    this.$el = $(el);
  }

  $.fn.wTooltip = function () {

    function get() {
      var wTooltip = $.data(this, 'wTooltip');

      if (!wTooltip) {
        wTooltip = new Tooltip(this);
        $.data(this, 'wTooltip', wTooltip);
      }

      return wTooltip;
    }

    return this.each(get);
  };
})(jQuery);
```

(Online: http://jsfiddle.net/7cuaq/)

You can see we have added our `Tooltip` function class, which is instantiated in the `get()` function. This is where our closure becomes important, as it hides the `Tooltip` function class from any other code and makes it available only to our `$.fn.wTooltip()` function.

In the `get()` function, the first thing we do is check to see if the object already exists on the element. This is in case we iterate over an element that has already been instantiated with our plugin. In this case, we can simply return the object. Otherwise, we can go ahead and run our plugin code.

For now, our object just stores a reference to the element, `$el`, which is something our plugin will commonly reference.

Prototyping

Finally, we'll want to set up a prototype function that will hold the majority of our code. There are two main reasons why we want to set up a prototype function:

1. It saves lots of memory by not having to recreate all applicable methods for every instance of the object we create.
2. Referencing an existing function is faster than creating a new one, regardless of memory usage.

A prototype basically extends an object by providing it methods without having to instantiate a copy in each object. This is similar to the concept of keeping functions outside of the `.each()` loop. It also serves as a good way to organize code more efficiently.

```
(function ($) {
  'use strict';

  function Tooltip(el) {
    this.$el = $(el);

    this.generate();
  }

  Tooltip.prototype = {
    generate: function () {
      // Code here...
    }
  };

  $.fn.wTooltip = function () {

    // Code here...
  };
})(jQuery);
```

(Online: http://jsfiddle.net/5x59q/)

The prototype functions have absolutely no state, as this will all be kept in our object. This object is what is stored in our element using `.data()` as we see in the previous example.

Note we can create as many function classes and prototypes as we like. For most plugins this is rare, but for something more advanced and requiring more complexity, we can go ahead and create many of them.

Summary

We now have upped the complexity of our plugin. We have added a loop using `.each()` to iterate over each element. We are also keeping track of our plugin with an object that allows us to reference the plugin code later on.

Section 03: Conventions

We'll take a moment here to cover some conventions used for naming functions and variables.

Generate

Earlier, we introduced a function in the prototype named `_generate()`. This function is used as an area to create the actual plugin elements. So for example, with a tooltip, we might simply create a `div` here.

```
_generate () {
  this.$tooltip = $('<div></div>');

  $('body').append(this.$tooltip);
}
```

We are creating a reference to our tooltip object here since we'll need to show/hide it later on when we write our `.hover()` code.

Note this function could be called anything, e.g., `_create()` or `_make()` or whatever makes sense to you. The point here is it's an area used discretely to create the visual components of your plugin.

Finally, for our tooltip example, we'll `.append()` it to the body of our page, thereby making it visible and active.

Destroy

It's also a good idea to provide a `destroy()` method that can be called to quickly send your plugin into oblivion.

```
destroy: function () {
  this.$tooltip.remove();
  $.removeData(this.$el, 'wTooltip');
}
```

We'll cover exactly how to call this method later in the guide.

Init

Although quite rare, sometimes we may want to keep track of the initialization of the plugin on each element. Such a situation may occur if your plugin is interacting with a back-end server.

Let's say we have a `setColor()` method that updates the plugin's color somewhere, but then also sends a call to the server to update some property.

```
_generate: function () {
  this.$tooltip = $('<div></div>');

  this.setColor('red');

  this.init = true;

  $('body').append(this.$tooltip);
},

setColor: function (color) {
  this.$tooltip.css('color', color);

  if (this.init) {
    // Update on back end.
  }
}
```

The problem occurs when we are preloading our plugin on load. At this point, we probably don't want to call the server because the action was not initiated by the user.

$var

You will notice we create some variables preceded with a `$` such as `$el` or `$tooltip`. This is not so much related to plugin development, but rather good practice when working with jQuery.

Whenever we are referring to a jQuery element, we will precede it with `$` just to let us know this is a jQuery object we can operate on.

```
this.$el = $(el);
this.init = false;
```

Private Functions

Our plugin will contain many functions both private and public. In JavaScript, there is no real concept of private functions since they can all be accessed one way or another.

To give the idea that a function is private and should not be publicly available, we will just precede it with an underscore character (_). You will see we have already done this with our `_generate()` function above. Otherwise, functions such as `setColor()` in the example above are public, as we can call them to alter the plugin's state at any time.

This & That

Often in our plugins, we will need to pass a reference to our plugin object `this` to another function scope with a different, local `this`. You probably will have seen a declaration like this before, which creates a reference to the `this` object under a different name, thereby making it accessible in other function scopes.

Some more common styles are below:

```
var _this = this;
var $this = this;
var that = this;
var _self = this;
```

It really doesn't matter which one you go with. However, I prefer to use `_this` although using `_self` is also quite popular.

The example below will illustrate the point:

```
_generate: function () {
  var _this = this;

  function elMousemove (e) {
    _this.$tooltip.css({left: e.pageX, top: e.pageY});
  }

  this.$tooltip = $('<div></div>');

  this.$el.mousemove(elMousemove);

  $('body').append(this.$tooltip);
}
```

If we were to use the `this` object in the `elMousemove()` function, we would actually be referring to the `$el` element instead of the actual plugin `this` object.

$.proxy()

If you dislike having to declare `_this = this` all over the place, there is a workaround using jQuery's `$.proxy()` function. Essentially, it does the same thing — passing any object you like as the `this` for your function.

```
_generate: function () {
  function elMousemove (e) {
    this.$tooltip.css({left: e.pageX, top: e.pageY});
  }

  this.$tooltip = $('<div></div>');

  this.$el.mousemove($.proxy(elMousemove, this));

  $('body').append(this.$tooltip);
}
```

We could even pass in `this.$tooltip` all together if we liked.

```
this.$el.mousemove($.proxy(elMousemove, this.$tooltip));
```

Whichever approach you decide to take, make sure you are consistent. Otherwise, it can lead to a lot of confusion.

I personally recommend setting `var _this = this`. It's not expensive, as it's just a reference to the object and probably will result in less code than using multiple `$.proxy` statements everywhere. However, it is available should you need it.

Summary

We almost have a working plugin and have covered how to keep your plugin code consistent. This sets the groundwork for starting to fully build out our plugins.