jQuery UI Widgets

# Introduction

* 1. When building rich client-side web applications, some of the visual elements on the page will naturally take on roles, responsibilities, and state. As more of these elements are added to the page, complexity will increase, so it's important for the design to support a maintainable codebase. Maintainable solutions have at least two important characteristics: they have an intentional design that plays to the strengths of the platform, language, and key parts of the environment, and they have unit tests.
  2. The web browser represents the platform. JavaScript represents the language and various JavaScript libraries represent key parts of the solution. Among other benefits, libraries like jQuery and jQuery UI are used to:
  + Address typical challenges like browser compatibility
  + Provide consistency for AJAX interactions, animations, and events
  + Assist in creating a maintainable codebase through modularity
  1. According to the official jQuery UI project, "[it] provides abstractions for low-level interaction and animation, advanced effects and high-level, themeable widgets, built on top of the jQuery JavaScript Library, that you can use to build highly interactive web applications." A central concept in the visual parts of jQuery UI is the *widget*. Widgets are objects attached to page elements that supply services for managing lifetime, state, inheritance, theming, and communication with other widgets or JavaScript objects.
  2. It's important to know a widget *is* a jQuery plugin. This may not be obvious because they are defined differently, but they are used the same way you use official jQuery methods and most custom plugins. Sometimes a plugin is sufficient and other times a widget is more appropriate. When you need to apply behavior or state to *individual* elements and need to communicate between elements, widgets provide a number of capabilities you would otherwise have to write yourself. This chapter illustrates the use of these capabilities.

## What you will learn in this chapter

* + How to define and apply widgets
  + How to manage the lifetime of widgets
  + How to define default options that permit overrides and change notifications
  + How to use options for decoupling behavior and facilitating event subscriptions
  + How to use private methods to improve the readability of the code
  + How to define and use public methods, properties, and events
  + How to inherit from a base widget
  1. The technologies discussed in this chapter are jQuery Plugins and the jQuery UI Widget Factory.

# Widget Fundamentals

* 1. If you know how to use jQuery, you know how to use a widget. However, before you can use a widget, it has to be defined. Once it has been defined it can be applied to elements. Widgets are defined using the widget factory. When the widget factory is invoked, it creates a *widget method* on the jQuery prototype, **$.fn** – the same place plugins and other jQuery functions are located. The widget method represents the primary interface for applying the widget to elements and using the widget after it's applied.

## Defining a Widget

* 1. The dependencies for a widget can be fulfilled with script references to the [CDN locations](http://www.asp.net/ajaxlibrary/cdn.ashx) for jQuery and jQuery UI. Widgets often reside in their own .js file and are wrapped in a self-executing function. This wrapper creates a JavaScript closure which prevents new variables from being globally scoped. A single solution should prevent any more than one global object according to well-accepted JavaScript practices. The **$** and **undefined** arguments reestablish their default expectations inside the closure in case another script previously defined them as something else.
  2. JavaScript
  3. // Contained in jquery.qs.tagger.js
  4. (function($, undefined) {
  5. $.widget('qs.tagger', {
  6. // definition of the widget goes here

});

* 1. }(jQuery));
  2. The call to **$.widget** invokes the widget factory which makes the widget available to use. The first argument, **qs.tagger**, is the widget's namespace and name separated by a period. The name is used as the name of the widget method placed on the jQuery prototype. The second argument, called the *widget prototype*, is an object literal that defines the specifics of the widget. The widget prototype is stored directly on the jQuery object under the namespace provided: **$.qs.tagger**.

## Using a Widget

* 1. Once a widget has been defined, it's ready to be applied to elements. To apply the widget to the matched elements, invoke the widget method just like you would other jQuery methods. The following code shows how to apply the tagger widget to all **span** elements with a **data-tag** attribute.
  2. JavaScript
  3. // Contained in startup.widget.js
  4. $('span[data-tag]').tagger();
  5. Because the widget method is used as the primary interface to the widget, it's not only called when initially applying the widget to the element, it's also used for calling methods and reading and writing options and properties on the widget. When widgets are applied to elements, an instance of the widget prototype is created and stored inside each element. This is how the widget factory knows if a widget has already been attached to an element so it can take the correct action in subsequent calls.

# Managing Lifetime

## Creation

* 1. The first time the widget is applied to an element, the factory calls the widget's **\_create** function. Method names preceded with an underscore have private scope by convention, which means they only expect to be invoked from inside the widget. The following code shows the **\_create** method in the **infobox** widget.
  2. JavaScript
  3. // Contained in jquery.qs.infobox.js
  4. \_create: function () {
  5. var that = this,
  6. name = that.name;
  7. that.infoboxElement = $('<div class="qs-infobox" />');
  8. that.infoboxElement.appendTo('body')
  9. .bind('mouseenter.' + name, function () {
  10. mouseOverBox = true;
  11. })
  12. .bind('mouseleave.' + name, function () {
  13. mouseOverBox = false;
  14. that.hideTagLinks();
  15. });
  16. },

The **\_create** method is the most appropriate place to perform a number of common tasks including the following:

* + **Adding classes** to various elements in the widget is the recommended way to apply styling, layout theming and more to the widget.
  + **Storing references** can increase performance when a particular set of elements are used from a number of methods. Simply create object-level variables for them once and all other methods can use them. This is an accepted jQuery performance best practice.
  + **Creating elements** is common for widgets that have requirements like animations, effects, styling, accessibility, and cross-browser compatibility. As an example, consider the **div.qs-infobox** element created by the **infobox** widget.
  + **Applying other widgets** is recommended during creation when you need them available as soon as possible. Even if your widgets don't require each other, consider using the official jQuery UI widgets from inside yours to add useful behaviors and interactions.

## Initialization

The widget factory also supports the notion of initialization. While the **\_create** method is only called when the widget is first applied to the element, the **\_init** method is called each time the widget method is called with no arguments or with options. When the widget is applied to the element the first time, **\_init** is called after **\_create**. When the widget method is called after the widget has been attached, only **\_init** will be called. The **\_init** method is the recommended place for setting up more complex initialization and is a good way to give the widget a way to reset. Although, it's not uncommon for widgets to not implement an **\_init** method.

## Destruction

The widget's **destroy** method is used to detach a widget from an element. The goal of the **destroy** method is to leave the element exactly like it was before the widget was attached. Therefore, it's not surprising the common tasks are to remove any added classes, detach any added elements and destroy any initialized widgets. Here is the **destroy** method for the tagger widget.

* 1. JavaScript
  2. // Contained in jquery.qs.tagger.js
  3. destroy: function () {
  4. this.element.removeClass('qs-tagged');
  5. // if using jQuery UI 1.8.x
  6. $.Widget.prototype.destroy.call(this);
  7. // if using jQuery UI 1.9.x
  8. //this.\_super('destroy');
  9. }
  10. The last part calls the widget's base implementation of **destroy** and is a recommended practice. The base **destroy** will remove the instance of the widget from the element and unbind all namespaced event bindings, which are covered later in the chapter.

# Defining Options

Options are a way to give widgets state that is public, readable, writable, and callable. Options are automatically merged with the widget's default options during creation and the widget factory supports notifications when option values change. In principle, you should be able to save the options on a widget, remove the widget from memory, recreate the widget with the saved options and have the same widget you started with.

Options are defined in the **options** property of the widget prototype as shown below in the **infobox** widget.

* 1. JavaScript
  2. // Contained in jquery.qs.infobox.js
  3. $.widget('qs.infobox', {
  4. options: {
  5. dataUrl: ''
  6. maxItems: 10,
  7. },
  8. ...

To override default options during the creation of the widget, pass them in as an object literal to the widget method as shown below in the startup code.

* 1. JavaScript
  2. // Contained in startup.widget.js
  3. var infobox = $('body').infobox({
  4. dataUrl: 'http://feeds.delicious.com/v2/json/popular/'
  5. });

This can be done as many times on an element as needed. The options will always be merged with the options already in the widget.

To read the options from inside the widget, use the **options** property directly as shown on the last line below.

* 1. JavaScript
  2. // Contained in jquery.qs.infobox.js
  3. displayTagLinks: function (event, tagName) {
  4. var i,
  5. that = this,
  6. options = that.options,
  7. url = options.dataUrl + tagName + '?count=' + options.maxItems,
  8. ...

Reading the values directly off of **options** is acceptable when reading values from inside the widget, but you should not use this approach when changing the value of options. Instead, use the **option** method (without an 's').

* 1. JavaScript
  2. // Code illustration: not in QuickStart
  3. var max = this.option('maxItems');
  4. this.option('maxItems', max + 4);

The **option** method is called with one argument when reading the option's value, two arguments when setting a value and a single object hash when setting more than one option. The **option** method should always be used to change the value of options so change notifications will work as expected. Changing the option directly on the **options** property bypasses the notification mechanism.

## When Options Change

If the widget needs to react to an option's value being changed, it should use the **\_setOption** method. This method is called by the widget factory just after the value has been set on the **options** property. The Widget QuickStart doesn't have a need for **\_setOption**, but if the number of links in the **infobox** widget were configurable by the user, as an example, the widget might need to adjust the size of the box when **maxItems** changes.

* 1. JavaScript
  2. // Code illustration: not in QuickStart
  3. \_setOption: function (name, value) {
  4. name === 'maxItems' && this.\_resizeBoxForMaxItemsOf(value);

$.Widget.prototype.\_setOption.apply(this, arguments);

* 1. },

If **maxItems** is the name of the option being provided, the **\_resizeBoxForMaxItemsOf** method will be called. The last line is calling the base widget's **\_setOption** method. This will set the value of the option and will aid in supporting a **disabled** state.

* 1. **Note:** All widgets support the notion of being disabled whether they choose to implement it or not. The Boolean value is stored at **this.options.disabled** or **$(selector).widget('option', 'disabled')** if you're asking from the outside. In return for honoring this option (whatever that would mean for the UI and behavior of your widget) the widget factory will default it to false and manage some CSS classes related to theming and accessibility.

The **\_setOption** method is not called when the options are merged on creation. When a widget has changed some options, inside **\_create** for example, and wants to call **\_setOption** on each option, a convenient approach is to use the **\_setOptions** (with an 's') method as in the following example.

* 1. JavaScript
  2. // calls this.\_setOption on all options
  3. this.\_setOptions(this.options);
  4. If options are passed to the widget method after it has been created, **\_setOption** will be called on each passed option just before **\_init** is called.

## Functions as Options

Defining functions as options is a powerful way to decouple the widget from functionality better located elsewhere; like in other widgets.

* 1. **Note:** The widgets in Mileage Stats use this approach for publishing and subscribing to global events using their **publish** and **subscribe** options and getting data from the **dataManager** using their **getData** option. To learn more about the pub/sub engine, see the Communication chapter and the Data chapter for more on the **dataManager**.

For example, rather than forcing the tagger widget to know how to invoke the public methods on the **infobox** widget, they can be kept free of any knowledge of each other by passing in the functions from the startup script since it already knows about both widgets. To set this up, the tagger widget defines **activated** and **deactivated** options.

* 1. JavaScript
  2. // Contained in jquery.qs.tagger.js
  3. $.widget('qs.tagger', {
  4. options: {
  5. activated: null,
  6. deactivated: null
  7. },
  8. Just like normal options, these can either define defaults or not. The startup script will provide these options when it applies the tagger widget to the **span** elements.
  9. JavaScript
  10. // Contained in jquery.qs.tagger.js
  11. $('span[data-tag]').tagger({
  12. activate: function (event, data) {
  13. // call displayTagLinks() on infobox here
  14. },
  15. deactivate: function () {
  16. // call hideTagLinks() on infobox here
  17. }
  18. });

In the above code examples, the options are being set and read from inside the widget's implementation or passed in during creation or initialization. These options can also be read and written to from outside the widget through a public interface. Later in the chapter you'll see how function-based options are used as callbacks for events.

# The Widget Method

* 1. Well-designed objects have public interfaces that are intentional, intuitive, and focused. Widgets go one step further and provide a single method that represents the entire public interface of the widget. The action the widget performs when you call this method depends on the number and type of arguments provided in the call. In addition to creating and initializing the widget as shown earlier, the widget method is also used to do the following:
  + Invoke public methods
  + Read and write public properties
  + Read and write options

## Public Methods

Public methods are defined on the widget prototype as you can see here in the **infobox** widget.

* 1. JavaScript
  2. // Contained in jquery.qs.infobox.js
  3. $.widget('qs.infobox', {
  4. hideTagLinks: function() {
  5. ...
  6. },
  7. displayTagLinks: function(event, tagName) {
  8. ...
  9. }

Widgets must be created before their methods can be called. So the following calls to the **infobox** widget assume the widget method has already been called once to apply the widget to the **body** element. To call **hideTagLinks** from outside the widget, use a jQuery selector to match the element and pass the name of the method to the widget method as its only argument.

* 1. JavaScript
  2. // Code illustration: not in QuickStart
  3. $('body').infobox('hideTagLinks');

When you have to pass arguments into the call, like **displayTagLinks**, simply add the arguments after the method name.

* 1. JavaScript
  2. // Code illustration: not in QuickStart
  3. $('body').infobox('displayTagLinks', event, data.name);
  4. The **option** method covered earlier in Defining Options (not to be confused with the **options** property) is an example of a public method. When one argument is passed to it, the method will return the value of that option. When two arguments are passed, it will set the option specified in the first argument to the value of the second argument. When calling this method from outside the widget, slide the arguments over by one and pass the method name, **option**, as the first argument as shown here.
  5. JavaScript
  6. // Code illustration: not in QuickStart
  7. var max = $('body').infobox('option', 'maxItems', 12);

As you can see above, public methods can also return values by putting the statement on the right hand side of a variable declaration. Returning a value from methods on **infobox** is reasonable because it is only attached to a single element. But be aware if you call a method on a wrapped set that contains more than one element, the method will be called on all elements, but the return value will come from the first element.

## Public Properties

* 1. Public properties are defined as methods and share their same calling convention. A common pattern for defining properties is to return the value of the property if no arguments are passed and set the value of the property if an argument is provided. The Widget QuickStart doesn't use public properties, but if the tagger widget needed to store the tag name and expose it as a property, it might look similar to this.
  2. JavaScript
  3. // Code illustration: not in QuickStart
  4. $.widget('qs.infobox', {
  5. tagName: function(value) {
  6. if(value) {
  7. this.element.data('tagName', value);
  8. } else {
  9. return this.element.data('tagName');
  10. }
  11. },
  12. Use this property the same way you would use a method.
  13. JavaScript
  14. // Code illustration: not in QuickStart
  15. var tag = $('body').infobox('tagName'); // reading tagName's value
  16. $('body').infobox('tagName', $.trim(tag)); // setting tagName's value
  17. In the examples so far, each time the widget method is invoked, it is being called on the instance returned by the jQuery function, **$(selector)**, which requires accessing the DOM. The next section recommends a couple of alternatives.

## Reusing an Instance

Each time the jQuery function uses a selector to invoke the widget method it must search the DOM. This has a negative impact on performance and is unnecessary because widget methods return a jQuery object, which includes the wrapped set of matched elements.

* 1. JavaScript
  2. // Code illustration: not in QuickStart
  3. var ib = $('body').infobox(); // accesses the DOM
  4. ib.infobox('displayTagLinks'); // does not access the DOM
  5. Rather than use a selector with the jQuery method each time you need to call a method on a widget, create a variable when the widget is initially attached to the elements. This will access the DOM, but it should be the only time you need to. In subsequent calls, like the second line in the snippet above, you can call the widget method on the variable you created and it won’t access the DOM.

### Pseudo Assistance

* 1. In a situation where neither the selector nor the instance is available, there is still a way to obtain all instances of a particular widget. As long as you know the name of the widget you can use a pseudo selector to get all instances that have been applied to elements.
  2. JavaScript
  3. // contained in an older, more tightly coupled version of startup.js
  4. $('body').infobox();
  5. // contained in an older, more tightly coupled version of jquery.qs.tagger.js
  6. var ibInstance = $(':qs-infobox');
  7. ibInstance.infobox('displayTagLinks', // method name
  8. $(this).text(), // tag
  9. event.pageY + offsetY, // top
  10. event.pageX + offsetX); // left
  11. The pseudo selector begins with a colon, like all pseudo selectors and is then made up of the widget's namespace and name separated by a hyphen. As indicated in the code above, this selector has the potential to increase coupling between widgets so be aware of this if you intend to use it.

# Private Members

## Methods

Private methods are methods that start with an underscore. They are expected to be accessed directly using **this**. Private methods are common and recommended. When used appropriately, they produce maintainable, well-written code.

Private methods are only private by convention. This means if a widget isn’t called according to the convention for calling public methods, described later, then its private methods can still be accessed. The convention is easy and consistent, and the underscore makes it easy to distinguish between the public and private interface.

## Properties

* 1. Unlike methods, properties on the widget prototype are not made private by prepending an underscore – they are private by default. Only methods are made private with underscores. Properties don't need underscores because they cannot be accessed through the widget method.
  2. JavaScript
  3. // Code illustration: not in QuickStart
  4. $.widget('qs.infobox', {
  5. dataUrl: '', // should only be accessed using this.dataUrl
  6. \_maxItems: 10 // unnecessary, properties are already private
  7. });

Because each element contains its own instance of the widget, the **dataUrl** property can be different for each element.

Clearly **dataUrl** is best exposed as an option, but if this was not a configurable option, you would likely want to define it so only one copy of the value was available to all instances of the widget. Let's call these static members.

# Static Members

To define a variable that's available to all instances of the widget but nowhere else, place them inside the self-executing function wrapper, but above the call to the widget factory as shown in the tagger widget.

* 1. JavaScript
  2. // Contained in jquery.qs.tagger.js
  3. (function ($) {
  4. var timer,
  5. hideAfter = 1000; // ms
  6. $.widget('qs.tagger', {
  7. ...

Because the **timer** variable is defined outside of the widget prototype, only a single timer will be created and shared across all instances of the tagger widget. Functions that don't rely on the instance of the widget can also be defined here.

If you need access to static members from outside the widget, they can be added to the widget after the widget's definition. A fictitious change can be made to the infobox widget to illustrate this. Inside the **displayTagLinks** method in the infobox widget, a function variable called **displayResult** is defined.

* 1. JavaScript
  2. // Contained in jquery.qs.infobox.js
  3. displayResult = function () {
  4. // don't need to pass in elem, html, top, or left
  5. // since they are in scope
  6. elem
  7. .html(html);
  8. .css({top: top, left: left});
  9. .show();
  10. };

It is defined in **displayTagLinks** because it's the only method that uses it. If the infobox widget needs to make AJAX calls from other methods, the **displayResult** function might need to be moved so it is available to all methods that need it. Defining it outside the scope of the widget is a way to make this happen.

* 1. JavaScript
  2. // Code illustration: not in QuickStart
  3. $.widget('qs.infobox', {
  4. ...
  5. });
  6. $.extend($.qs.infobox, {
  7. displayResult: function(elem, html, top, left) {
  8. elem
  9. .html(html);
  10. .css({top: top, left: left})
  11. .show();
  12. }
  13. });
  14. With **displayResult** defined here, the infobox widget can use it from anywhere as shown here.
  15. JavaScript
  16. // Code illustration: not in QuickStart
  17. // assume elem, html, top, and left variables were already defined
  18. $.qs.infobox.displayResult(elem, html, top, left);

# Events

* 1. Events are an effective way to communicate between widgets without forcing them to be tightly coupled.

## Binding Handlers

* 1. Event handlers bind to widget events the same way they bind to other events.
  2. JavaScript
  3. // Code illustration: not in QuickStart
  4. $('span[data-tag]').bind('tagactivated', function(event, data) {
  5. // handle the event
  6. });
  7. The **event** argument should be the original event if this event is being raised from a different event handler. The **data** argument is data specifically intended for the event handler.
  8. Notice how the name of the event being bound to has had the name of the widget prepended. This is the default behavior for event names. If you would prefer a different name so your code is more readable, this behavior can be changed.

## Event Naming

* 1. The **widgetEventPrefix** property is on the base widget and defines what will be prepended to the event names. If you are not happy with the default event names, an appropriate place to set this value is in the **\_create** method.
  2. JavaScript
  3. // Contained in jquery.qs.tagger.js
  4. $.widget('qs.tagger', {
  5. widgetEventPrefix: 'tag',
  6. options: {
  7. activated: null,
  8. deactivated: null
  9. },
  10. When **widgetEventPrefix** has a value, it will be used instead of the widget name.

## Raising the Event

* 1. The widget naming convention described above is only applicable to the event handler. Inside the widget, the original event name is used to raise the event. The following code sample shows one way the tagger widget might raise the **activated** event when the mouse enters the element.
  2. JavaScript
  3. // Code illustration: not in QuickStart
  4. \_create: function () {
  5. var that = this,
  6. tag = that.infoboxElement.text();
  7. that.infoboxElement
  8. .bind('mouseenter', function (event) {
  9. that.\_trigger('activated', {name: tag});
  10. });
  11. },
  12. When **trigger** is called, the event will be raised and any bindings will be invoked. The problem with binding directly is it creates more coupling than is needed for event handlers. If the widget is following well-accepted widget design practices, the widget will have callbacks defined in its options.

## Relationship to Options

* 1. When options are defined as functions and their names correspond to an event name without the prefix, they are referred to as callbacks. The **\_trigger** method on the base widget will automatically invoke the callback whose name matches the event being raised.
  2. JavaScript
  3. // Contained in jquery.qs.tagger.js
  4. \_create: function () {
  5. var that = this,
  6. name = that.name(),
  7. tag = that.element.text();
  8. that.element
  9. .bind('mouseenter.' + name, function (event) {
  10. that.\_trigger('activated', event, {name: tag});
  11. });
  12. },

# Inheritance

* 1. Sometimes when building a widget, another widget already has a lot of what the new widget needs. The widget factory's inheritance support is designed for this case. For illustration purposes, consider the following widget.
  2. JavaScript
  3. // Code illustration: not in QuickStart
  4. (function ($) {
  5. $.widget('a.container', {
  6. ...

resize: function() {

// resize width and height

},

...

});

}(jQuery));

* 1. If this widget was built elsewhere and you want to change its resizing behavior to animate, a reasonable approach would be to inherit from **a.container** and override its resize method. Inheritance is accomplished by passing three arguments into the widget factory. The first argument is the namespace and name of the widget, the second is the base widget being inherited from, and the third argument is the object prototype of the derived widget.
  2. JavaScript
  3. // Code illustration: not in QuickStart
  4. (function ($) {
  5. $.widget('an.animatedContainer', $.a.container.prototype, {
  6. ...

resize: function() {

// override with animations

},

});

}(jQuery));

1. The only difference between the signature above and the signature usually used for defining widgets is addition of the second parameter.
2. Inheritance is a useful tool when you are using a widget that almost does what you want it to do. In version 1.9 of jQuery UI, widgets will be able to inherit from themselves, which makes it even easier to extend the functionality of widgets.

# Summary

* 1. Using the jQuery UI widget factory is a great way to add modularity to client-side web applications. Their lifetimes can be managed with **\_create**, **\_init**, and **destroy**. Options should have intelligent defaults but can be overridden at any time by passing them to the widget method or calling the **option** property directly. Functions are a powerful way to decouple functionality and using them for callbacks makes raising and handling events straight-forward. Widgets can have public methods and properties and uses a prepended underscore for private methods. Define functions and variables outside of the widget prototype but inside the self-executing function wrapper when it's appropriate for all instances of the widget to use a single function or variable. Widgets can also be inherited when base functionality can be shared across different widgets. See the next chapter for more places to learn about jQuery plugins and jQuery UI.

# Further Reading

* 1. Appendix B: Widget QuickStart
  2. Widget Factory documentation on the jQuery UI wiki:   
     <http://wiki.jqueryui.com/w/page/12138135/Widget-factory>
  3. jQuery Documentation for Plugins/Authoring:   
     <http://docs.jquery.com/Plugins/Authoring>
  4. jQuery UI Developer Guidelines:   
     <http://jqueryui.com/docs/Developer_Guide>
  5. jQuery UI source code:   
     <https://github.com/jquery/jquery-ui>

# Resources

* 1. Content Delivery Network (CDN) Addresses:
  2. <http://www.asp.net/ajaxlibrary/cdn.ashx>