10
Testing jQuery Plugins

The jQuery library is very powerful by itself, but its true power comes with its extensibility through the development of plugins.

A jQuery plugin is an extension to the jQuery object to add extra functionality. A common example of a plugin is the disable input plugin.

Normally to disable inputs on jQuery, we use the `attr` function to add an attribute named `disabled` with value `disabled` to the `input` element:

```
$('input').attr('disabled', 'disabled');
```

But we could instead create a plugin that works with a simple call to a new `disableInput` function:

```
$('input').disableInput()
```

To develop this plugin, and to understand how a jQuery plugin works, let's start by defining the acceptance criteria:

- `jquery.disableInput` should be chainable
- `jquery.disableInput` should disable an input

Which can be translated into these specs:

```
describe("jquery.disableInput", function() {
  var $input;

  beforeEach(function() {
    setFixtures('<input type="text" name="add" value="Add">')
    $input = $('input[type=text]');
  });

  it("should be chainable", function() {
```

```
    expect($input.disableInput()).toBe($input);
  });

  it("should disable an input", function() {
    $input.disableInput();
    expect($input).toBeDisabled();
  });
});
```

We will need an input to test the plugin, so we use the `setFixtures` function to add an input to the document:

```
setFixtures('<input type="text" name="add" value="Add">')
```

Then to test if the plugin can be chained, it is just a matter of verifying that after its execution, it returns the jQuery object again:

```
it("should be chainable", function() {
  expect($input.disableInput()).toBe($input);
});
```

Chainability is useful in scenarios where more than one action is performed to a particular element, (such as disabling it and hiding it). By allowing the function to be chained, we can write a code to perform two tasks:

```
$input.disableInput().hide();
```

Next, we want to test if the plugin works, and actually disables the input. To write this expectation we use the Jasmine jQuery `toBeDisabled` matcher:

```
it("should disable an input", function() {
  $input.disableInput();
  expect($input).toBeDisabled();
});
```

After the spec is coded, implementing the plugin couldn't be simpler:

```
(function ($) {
  $.fn.disableInput = function () {
    this.attr('disabled', 'disabled');
    return this;
  };
})(jQuery);
```

A jQuery plugin is a function set as a property of the `$.fn` object.

So if you want to add a plugin named `disableInputs`, all you have to do is add another property with this name to the `$.fn` object:

```
$.fn.disableInput = function () {};
```

This function will then be available to be called on any jQuery object:

```
$('input').disableInput()
```

After the function gets executed, its receiver (`this`) will be the jQuery object in which you ran the plugin:

```
$('input');
```

So the plugin code:

```
$.fn.disableInput = function () {
  this.attr('disabled', 'disabled');
  return this;
};
```

Is basically doing the same as:

```
$('input').attr('disabled', 'disabled');
```

And this is how you create and test a jQuery plugin.

Although this implementation works, if you want to publish your plugin to the community, it is a good idea to follow the jQuery coding standards. A good place to start is the **jQuery Boilerplate** project that you can find at `http://jqueryboilerplate.com/`.

Summary

In this chapter you learned about jQuery plugins, and how you can create one driven by tests.

You have seen the basics of a plugin, and the guidelines on how to start publishing your own plugins to the community.