Spark

The (fairly) complete guide

By Oliver Caldwell

→Contents

→Introduction

So you want to learn how to use Spark? Well you have downloaded and opened the right document! This ‘not quite a book’ was written by the developer of Spark and is aimed at taking someone that knows nothing about it and turning them into some sort of JavaScript god.

So what is Spark? Well you may already know but Spark is a all-in-one JavaScript library that handles AJAX, animation, JSON manipulation among many other smaller features. It was built to help you make more engaging sites with less effort and repeated code.

I won’t waffle on here because I know you want to get learning, I just hope you enjoy using it as much as I have enjoyed writing it!

→Prerequisites

Spark does require a small amount of knowledge in JavaScript and HTML. After all it extends JavaScript's functionality and does not replace it so the underlying language must be learnt first.

You must have access to a browser, a text editor and at least three cups of coffee. I recommend using either Firefox with the Firebug extension or Google Chrome with its built in developer tools. These are a must have when it comes to debugging your scripts.

Other than that, a brain should just about cover what you need.

→Getting started

↳Downloading the latest version

Now this step is easy, just head on over to the downloads page (http://sparkjs.co.uk/download), right click the latest version and click ‘Save as’. You will then be prompted as to where you wish to save it, I recommend keeping the file name as it is. This is so you know what version you currently have without having to open it up and check what it says in the header comment block.

So now you should have a folder for all of your tests containing the latest version of Spark. *At the time of writing it will have been Spark-v1.4.6.js*.

→Getting started

↳Setting up your page

We are going to need a basic page to test all of our code in. This is fairly simple but you can copy the document below for convenience.

```
<!DOCTYPE html>
<html lang='en'>
     <head>
          <title>Spark</title>
          <meta http-equiv='Content-Type' content='text/html;
charset=utf-8'>
          <style type='text/css'>

          </style>
     </head>

     <body>
          <script type='text/javascript' src='Spark-v2.2.2.js'></
script>
          <script type='text/javascript'>

          </script>
     </body>
</html>
```

So all we have here is a basic HTML5 document with a style tag and two script tags. The first script tag is including Spark, you will have to amend the src to what ever your file is called. The second script tag will be where your code goes. You should normally make all of your CSS and JavaScript external but just for learning this is easier.

You will add elements to manipulate just above the two script tags within the body. Add a p tag for us to create a ‘Hello, World!’ with.

So now you have your basic document with an additional p tag. Before we go adding content to it with Spark we need to use Spark's ready function to make sure our code is only run when all of the DOM elements have loaded. Place this code within your second script tag.

```
Spark.ready(function() {

});
```

Remember not to just copy, no matter how trivial or boring you may think it is, you have to write it out if you want to remember it.

So what is this doing? Well it is running the ready function and we are passing a function to be run when the DOM is finished loading. So all of our code (other than other functions and things that don't need to be run immediately) will be placed in here.

Now lets use the html function to put some content in our p tag. Because we have not given our tag a class or id we can just reference it by the selector 'p'. Spark uses CSS3 selectors so you can select elements just as you would in CSS. You can also provide an element or an object of elements and bypass the selector engine (Sizzle).

```
Spark('p').html('Hello, World!');
```

So now if you run it you will see that your p tag contains the words 'Hello, World!'. This html function actually has a second argument. If you pass 'true' as the second argument then the content will be appended rather than replaced. There is also the text function which converts any html to text ie < to <. The html function is slightly faster though.

→Calling Spark

There are two ways of calling Spark, via the Spark object or the $ object. Commonly known as dollar notation. I will be using the Spark object for all of the examples because its what I prefer but you can use what you like.

If you want to use Spark alongside another library such as jQuery then they will conflict due to them both trying to use the $ object. You can put Spark in no conflict mode wherein it does not use the $ object by running the no conflict function.

```
Spark.noConflict();
```

Now $ has been reverted to what it was before Spark got there, if you provide an argument of ‘true’ then the same will be done to the ‘Spark’ variable so you can run different versions at the same time. This function returns the object it reverted so if you wanted to remove $ but still use it under a different name such as $S you could use this line.

```
$S = Spark.noConflict();
```

→Chaining functions

Almost all of the functions, other than functions like AJAX and JSON which do not involve elements, can be chained. Chaining is the process in which you pass the elements returned by the selector engine to the next function. It makes cleaner, faster and more compact code. Lets try the ‘Hello, World!’ example again but this time using the second appending argument and function chaining.

```
Spark('p').html('Hello').html(', World!', true);
```

This can be laid out better like so.

```
Spark('p')
     .html('Hello')
     .html(', World!', true);
```

I feel the latter format is easier to read and just generally looks better.

In that example we have chained the content function and also utilised its append argument.

→Animation

So now our example code is adding content to an element, lets do something with it.

First we are going to need to set its overflow to hidden because we are going to shrink it and the only way you will see that is with the overflow hidden. So you can either chain this function onto the end of where you where setting the content or run the selector engine again with 'Spark('p')' it makes no difference. I have run the selector engine again because it will be easier for you to read.

```
Spark('p').css({overflow: 'hidden'});
```

So this CSS function has set overflow to hidden. This function takes an object so you can set as many properties as you want at one time. If you do not provide an object then it returns an object containing all of its current CSS properties. The same goes for the attribute and computed functions.

Now the overflow is set to hidden we can get animating, the line below uses the animate function which also takes an object containing your properties. If you provide a number then it presumes it to be a measurement in pixels although you can provide a string such as '20%' and that will work too.

```
Spark('p').animate({height: 0});
```

Now run that and gasp in wonder as the element shrinks before your very eyes. It is by default set to take place over 600 milliseconds. You can change that by providing an argument like so.

```
Spark('p').animate({height: 0}, 1000);
```

You can set the easing you wish the function to use by passing a string as the next argument, if you provide false then it will default to 'outQuad' easing. For a full list of available easing's head over to the documentation. (sparkjs.co.uk/documentation/animate)

You can also provide a callback function to be run when the animation completes like so.

```
Spark('p').animate({height: 0}, 1000, false, function() {
```

```
     console.log('Animation complete!');
});
```

Now lets chain another animation, lets fade the elements opacity to 0. Animation chaining works in such away that the next animation in the chain will be run when the previous animation completes, so if you used this line.

```
Spark('p').animate({height: 0, opacity: 0});
```

Both would be animated at the same time. Instead we will use this line to animate them one after the other.

```
Spark('p')
     .animate({opacity: .1})
     .animate({height: 0});
```

If you run this you will see the p tag fade almost completely out and then slide up.

If you are looking to show and hide elements I recommend using the transition function. (http://sparkjs.co.uk/documentation/transition)

→Events

This is a fairly small section but still. You can bind functions to a specific event on a set of elements, for example make it so whenever you click any p tag on a page it fades out.

Lets bind the click event to our p tag so that when it is clicked an alert pops up with the elements content.

```
Spark('p')
     .html('Hello')
     .html(', World!', true)
     .event('click', function(e) {
          alert(Spark(e.target).html());
     });
```

→AJAX

The AJAX function is used to get or post data to your server, it can not go cross domain because of browser security restrictions. Look into JSONP (http://sparkjs.co.uk/documentation/jsonp) if this is a must.

To perform an AJAX call we are going to need a file to load, so make a file next to your example document called test.txt, fill it with some text, it does not matter what it is.

Now we are going to load the contents of that file into our p tag. This is accomplished like so. But beware, sometimes your AJAX call can fail, due to the server not responding or the file not existing etc. So you have to check if the AJAX function returned false to make sure you don't have nasty errors occurring in your script. You can see these checks in the examples.

```
var text = Spark.ajax('get', 'text.txt');
if(text) {
     Spark('p').html(text);
}
else {
     // Error, let the user know
}
```

This is what is known as a synchronous request, this is where the JavaScript pauses until it has finished the AJAX call. To use an asynchronous call (recommended) you just need to say false (no parameters) and then provide a callback function.

```
Spark.ajax('get', 'text.txt', false, function(data) {
     if(data) {
          Spark('p').html(data);
     }
     else {
          // Error, let the user know
     }
});
```

This will start the call and then pass the data to your call back when it is done allowing the JavaScript to continue running while it loads.

If you wanted to add get or post variables to the call then you would replace false with a string containing something like this ‘foo=bar&x=y’.

→JSON

JSON is the way in which we can turn a JavaScript object into a string for easy transportation and storage. This is the method that most API's use. It can also turn that string back into an object.

So lets encode something. Create an object like so containing what ever you want. I have gone for my name and age.

```
var myObject = {
     name: 'Oliver Caldwell',
     age: 17
};
```

And then we can encode it into a string and place it into our p tag like so.

```
var encoded = Spark.json('encode', myObject);
Spark('p').html(encoded);
```

As you can see it looks a lot like JavaScript syntax. Now try changing encode to decode and changing it back into an object.

→Using a plugin

Plugins add extra functionality to Spark, one of the plugins that I wrote allows you to place a label within an input. When the user goes to type it disappears, it even works for passwords. If then then click back out without typing anything it will go back to its original value.

So to get this plugin head over to the downloads page (http://sparkjs.co.uk/download) and save the replace plugin next to your test document. Now include that JavaScript file like so, just remember to place it just under where you included Spark.

```
<script type='text/javascript' src='spark.replace.js'></script>
```

Now add an input with a type of text and a value of what ever you want. All you need to do now is run the plugin on your element.

```
Spark('input').replace();
```

When you run it now you can see what it does. This is very useful for thinks like login forms or search boxes.

The other plugin on that page retrieves the specified users latest tweet. Why not try it out.

→Writing a plugin

So now you know how to use plugins, lets write a very simple one. First off we need the structure for our plugin.

```
SparkFn.read = function() {

};
```

What this is doing is assigning a function called read to the SparkFn object.

Now we need to add some structure to loop through all of the elements returned by the selector engine and allow chaining of our function.

```
SparkFn.read = function() {
     // Set up the el varaible
     var el = null;

     // Loop through all of the returned elements
     for(var e in this.elements) {
          // Grab the current element
          el = this.elements[e];
     }

     // Return the Spark object to allow chaining
     return this;
};
```

As you can see all we are doing here is setting up the ‘el’ variable which will contain the current element. We are then looping through all of the elements returned by the selector engine and assigning the current one in the loop to ‘el’.

The last line where we return ‘this’. We are returning what is returned by ‘Spark(element)’. This allows chaining. So now we can stick our code just underneath where we assign ‘el’.

Try putting this there and running it on a p tag that has some content.

```
console.log(el.innerHTML);
```

Now if you look inside your developer console (firebug or chromes native one) then you will see the contents of every element returned by the selector engine.

→A little extra

If this whole document has not been enough to get you started then please visit the forum, email me or lurk in the IRC channel until I show up. All the information and links you need should be available on the site.

Until then, have fun and I hope Spark can help you create amazing websites.