Leonardo Bighi

Better jQuery

You already know how to use jQuery. Now it's time to get better.

Content

3 -------- Beginner Tips --------

4 Tip #1: Use jQuery from Google's Servers

5 Tip #2: How to always load jQuery's latest version

6 Tip #3: Use console.log() to see what's wrong

8 Tip #4: Use #id selector to make code faster

9 Tip #5: Don't repeat the same selector over and over

10 Tip #6: Use chained methods to shorten your code

12 Tip #7: Wait for the whole page to load

13 Tip #8: How to check if an element exists

14 Tip #9: Use no-conflict mode to avoid... guess what... conflict!

15 Tip #10: Use a cheat sheet!

16 Intermediate Tips

17 Tip #11: Create elements on the fly with ease

19 Tip #12: How to preload images

20 Tip #13: Do something when an image has been loaded

21 Tip #14: Use animate() to change attributes with style

23 Tip #15: Use delay to create sequences of animations

25 Tip #16: How to prevent the default behavior in events

27 Tip #17: Don't let ajax requests capture your this

28 Tip #18: Set ajax defaults to standardize ajax requests

30 Tip #19: Live events can bind to elements in the future

33 Tip #20: Use the right event binding to fit your needs

Chapter 1

-------- Beginner Tips --------

The following 10 tips are very basic ones. They're available here just in case you have missed them when learning jQuery.

If you are more experienced, feel free to jump ahead to the Intermediate section and have fun.

Tip #1: Use jQuery from Google's Servers

You all know how to include the jQuery script in your page. But did you know you can get a little weight off your server by including jquery.js from Google's servers?

All you have to do is use this script tag:

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
```

This is the path to include version 1.7.2. Change to number to point to whatever version you want to load.

This little trick will make it load faster (Google has better servers than you do) and also save you some bandwidth.

You can also include jQuery-UI by using this script tag:

```
<script src="https://ajax.googleapis.com/ajax/libs/jqueryui/1.8.18/jquery-ui.min.js"></script>
```

Just remember you can change version number to whatever version you want to load.

In case you're interested in finding out what other libraries you can use from Google's servers, you can check the Google Libraries API website: https://developers.google.com/speed/libraries/devguide.

Tip #2: How to always load jQuery's latest version

This is a quick tip expanding on the last one. if you want to always load the latest version of jQuery, there are some ways to do this.

Method 1: Load it from the jQuery website

Just use this script tag:

```
<script src="http://code.jquery.com/jquery-latest.min.js"></script>
```

Method 2: Load it from Google

Do you remember what we did in chapter 1? Google allows you to be less specific when picking a version. If you want to load the latest version from the 1.7.x family (anything starting with 1.7), you do:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7/jquery.min.js"></script>
```

If you want to load the latest version from the 1.x.x family (any version starting with 1), you do:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js"></script>
```

Tip #3: Use console.log() to see what's wrong

During the development of JavaScript code there are surely those moments where you need to check the value of something. Maybe you need to check if a function is being called, or what the hell is being saved on that variable.

For those moments, console.log() is a gift from heavens, brought to you by Zeus himself.

Every modern browser has a JavaScript console, where errors and warnings go to die. In Chrome, for example, you can open it by clicking on the wench icon, then going to the Tools submenu and then clicking on JavaScript Console.

Every message you log using the console.log() command is shown on your browser's JavaScript Console. It's not a jQuery tip per se, but this a great tool when you're doing something complex and error-prone.

You use it like this:

```
console.log("Hello World");
```

But error messages are not all. The console is so powerful it can help you inspect HTML elements and JavaScript functions. You can inspect a jQuery selector to see if it's capturing the correct element (or elements).

Example:

```
element = $('.complex-selector:not(.simple)');
console.log(element);
```

Then you can see what HTML the variable contains (if any), check the element's properties, etc. If you pass it an object, you can see everything about the object.

So, remember console.log() whenever something is not working as intended. It may save your life! (If your life is the price for not finishing your work in time, I mean)

Tip #4: Use #id selector to make code faster

jQuery is a fantastic tool that lets you use any CSS3 selector to find HTML elements and some more. But not everyone knows that every selector takes a different time to run.

Let's say you have this HTML element:

```
<input type="text" id="myid" class="myclass">
```

You can find it using many different selectors, like:

```
$('#myid');
$('.myclass')
```

Everytime you tell jQuery to find an element it will scan the HTML DOM to find what you want. When you are doing it many times in a row, the time it takes to find your element matter a lot.

I did a test in my computer to calculate the difference. I ran both selectors 1000 times in a row and logged the time it took my browser to run it.

When I used the .myclass selector, it took 5.06 seconds. When I used the #myid selector, it took my browser 0.061 seconds to find it one thousand times. That's a HUGE difference.

What lesson can we take from this? While you shouldn't go out of your way to make your code faster, but you should use ID to find elements whenever possible.

Tip #5: Don't repeat the same selector over and over

If you read the previous chapter you know that every selector has a cost, and this cost is paid in time. Every time you use a selector, it takes some time for the jQuery engine to find the element in the page.

When you need to work on the same element many times, it's not very effective to make the engine search for your element again and again. Look at the following code:

```
$('.element:first').click(function(){...});
$('.element:first').addClass("myclass");
$('.element:first').fadeIn('slow');
```

Look at the code. It's not very good, right? Why are we making the jQuery engine search for the same element again? We are losing time here! There's a smarter way to do that.

```
first_element = $('.element:first');
first_element.click(function(){...});
first_element.addClass("myclass");
first_element.fadeIn('slow');
```

One more line of code, but it's worth it. On the first line we find the element and store it in a variable. Then all we have to do is refer to the variable whenever we want to manipulate that same element again.

In cases where you only want to call a few methods on the same element, there's another way that's even better. Want to know what it is? Come with me to the next chapter!

Tip #6: Use chained methods to shorten your code

On the previous chapter we saw it's a good think to avoid repeating the same selector many times. We learned how to store the element in a variable. But when you want to call many short methods on the same element, there's an even better way to do it. A way that will spare you some lines of code.

Let's recall the code from the previous example:

```
first_element = $('.element:first');
first_element.click(function(){...});
first_element.addClass("myclass");
first_element.fadeIn('slow');
```

Most jQuery methods return the element (or collection of elements) we gave it. This means we can simply call a new method right after the previous one. This is what the professional ninja experts call "chaining methods".

If you chain methods on your first selector, you don't even need to store the element in a variable. Let's try.

```
$('element:first').click(function(){...}).addClass('myClass').fadeIn('slow');
```

Just one single line of code! That's like magic! Why don't you use it every time? The problem of working like this is that you may end up with very very long lines of code. It makes your code ugly, and ugly code is not good.

If you still want to chain methods without storing the element and don't want a huge code, there's another option. You can break the line right before the dot. Let's try chaining methods with creating a huge line of code.

```
$('element:first')
  .click(function(){...})
  .addClass('myClass')
  .fadeIn('slow');
```

Between this and the method from the previous chapter you can use both, depending on the situation. Sometimes you will need a variable. Sometimes you can just chain them like you're getting a dollar for every chained method. Pick whatever makes your code easier to read and understand.

Tip #7: Wait for the whole page to load

When you're using jQuery, most of the time you'll be manipulating DOM elements (those things on the page). Sometimes your JavaScript may load so fast that your code is going to be executed before the page is ready.

If you try to make that third paragraph blink and spin and zoom before your browser even knows a third paragraph is suposed to be there, you're going to run into trouble.

In these cases, there's a method to make you code wait for the entire page to load before it runs.

```
$(document).ready(function() {
  // INSERT YOUR CODE HERE
});
```

Doing it like this, you can be 100% sure all your HTML and CSS content is loaded before your JavaScript does its magic.

There's also a shorter version for the code above. It's not my personal favorite, but here it is in case you like it better:

```
$(function() {
  // INSERT YOUR CODE HERE
});
```

Tip #8: How to check if an element exists

Sometimes you want to do something but you don't know if an element exists. Since you can't manipulate the nothing, there are times when it's needed to check for the existence of an element.

Everytime you use a jQuery selector it sets a length property equal to the number of elements found. So if you want to know if an element exists you have to check if your selector found at least one.

```
if ($('.myclass').length > 0) {
    // DO SOMETHING HERE
}
```

In JavaScript, every number different from zero validates as true. This means you can leave out the `> 0` part.

```
if ($('.myclass').length) {
    // DO SOMETHING HERE
}
```

Tip #9: Use no-conflict mode to avoid... guess what... conflict!

When you load jQuery, it creates a super mega powerful $ variable. It's short to type, it's visually different from the rest of the code, it's great.

The problem is that other some JavaScript frameworks are jealous and fat and decided to use the same $ variable. If you load jQuery and one of the other drunk and bald frameworks on the same page, both are claiming the $ variable as their own and then you've got a conflict.

As jQuery is a gentleman framework, it has a "no-conflict mode". When you load it in this mode it will not try to claim $, so other frameworks can do their jobs uninterrupted.

To activate no-conflict mode, you have to use the following code as soon as possible:

```
jQuery.noConflict();
```

It will make jQuery let go of the $ variable. Also, if some other framework had claimed $ before jQuery was loaded, jQuery will give back $ to that other framework as it were before jQuery was loaded. So remember, after you activate no-conflict mode, the $ variable will no more point to jQuery.

In these cases you can use the jQuery variable instead of $. If you go back to any example from previous chapters, you can safely replace every instance of `$` for `jQuery` and it will work just the same.

An example of code using jQuery instead of $:

```
jQuery('div#myid').addClass('myClass');
jQuery.ajax({...});
```

Now you can go ahead and use any other framework you want along jQuery and there'll be no conflict between them.

Tip #10: Use a cheat sheet!

This is my last basic tip, but it's a very important one. jQuery is a big framework with lots and lots of options. Truste me, you will not memorize it all.

You will forget something important when you most need it. And it will happen more than once.

Do what every good professional do: cheat!

A cheat sheet is a compendium of the most important stuff, condensed in a few pages (sometimes even one). If you forget something, take a look at the cheat sheet and BAM you remember! You don't have to spend time googling for it.

The guys at Future Colors did an online cheat sheet and you can see it here.

If you want a PDF cheat sheet so you can print, frame and pin it to your wall, download it here.

Some other good cheat sheets:

This one from Color Charge is an image

INCLUDE ONE MORE

Pick the one you like best and cheat!

Chapter 2

Intermediate Tips

There are not for your first day in the jQuery world.

Tip #11: Create elements on the fly with ease

There are moments when you need to create a new element and insert it into your page. You can do it the old JavaScript way (that's the way old people do, and it's ugly and wordy) or you can do it the beautiful way. If you prefer the latter, come with to the next paragraphs.

The first thing you should know is that you're going to use the same old $() function. Surprised, right? No? OK.

Let's pretend you have a need to insert a new link in your page. How do you create a new `<a>` element?

```
$('<a>');
```

Voilá! That small code is enough to tell jQuery you want to create an `<a>` tag. If you're a perceptive person you'll see this element is being created and discarded, because it's not being inserted anywhere in the page. Let's fix this.

```
$('<a>').appendTo('p:first');
```

Ah, now we're on to something. We are creating a new empty `<a>` tag and inserting it at the end (`appendTo()`) of the first paragraph. You could use `prependTo()` to insert it at the beginning.

But our link is empty and boring! How do I put stuff in my link? you may ask. Stick with me and you'll see. To be continued...

...right now. We want to add at least 2 things to our link: a text and an address. Let's first do it the basic way.

```
$('<a href="http://disney.com">Here be Mickey</a>').appendTo('p:first');
```

See what I did there? I just declared an HTML tag inside $() just like I would do in HTML code. That's the most basic way of telling jQuery to create a new element. But there are two issues. First, when you want to create an element with a lot of attributes, this is going to become a very long line of code. Second, if you want to define programatically the value of these attributes, this is going to become a mess of string concatenation.

Being jQuery the beautiful framework it is, it gives us a better way to do this. Let's do again the previous `<a>` tag, this time with an id and a class, using the second method:

```
$('<a>', {
  href: 'http://disney.com',
  text: 'Here be Mickey',
  class: 'linkClass',
  id: 'mickey'
}).appendTo('p:first');
```

That is nicer, right? And it's easier to use a variable as the content of any attribute. To create your elements like this you just pass a second parameter, an anonymous object (also called a hash) containing the attributes you want it to be created with.

Tip #12: How to preload images

In dynamuc you want to dinamically show images to the user, it's a better practice to load the image before adding the element to your page. This way the image will show up instantly, without the loading time.

This is called preloading and is very easy to do.

Whenever a new element is created with a `src` attribute, your browser downloads the image. What's important is that it happens even if the element isn't anywhere in your page. This means we can create an (read tip #12), store it in a variable (read tip #5) and the image will be already download when we finally insert it into our page.

Let's say we want to preload the image of a funny cat and append it to a paragraph when the user clicks on any link. Because every website gets better with cats.

We preload it by doing:

```
var image = $('<img>', {src: '/funny-cat.jpg'});
```

And then you can refer to the variable whenever you want to use it. It will already be preloaded.

```
image.appendTo('p:first');
```

Now go and make your dynamic images super fast!

Tip #13: Do something when an image has been loaded

In tip #12 you learned how to dynamically preload images. But there is more to it. There are cases when you want to run a piece of code when the image is loaded. I once needed it myself when trying to make an image gallery and the solution is not very easy to find. So here I am to help you.

Images can fire a `load` event when they have been loaded, but you have to bind a callback to the event BEFORE you set the image's src. It's not technically necessary to do this, but some internet connections these days are so fast they may download the image before you are able to hook your function to that callback. So follow my advice and set your callback first.

You can do this by using the `on` method like you would with any other method binding. If you learned jQuery in the old days, you may be using `bind` to do the same. If you do, please stop using `bind` and start to use `on`. It's the new default. Now to the code:

```
// We create an empty image element and store it in a variable
var image = $('<img>');

// We bind a function to the load event
image.on('load', function() {
    console.log("It's alive!!"); // Just a log to make sure it's working
});

// We use jQuery's attr() method to change the src attribute.
image.attr('src', './funny-cat.jpg');
```

When we set the `src` attribute the image will be loaded and our function will be called.

Tip #14: Use animate() to change attributes with style

Not many people know this, but you can easily create an animation whenever you change visual CSS attributes of an element. Any numeric attribute change that can be visually perceived by the user can be animated.

Let's say you want to expand a div from 200px wide to 400px wide. Instead of just changing it, you can create a progressive animation where the width grows from 200 to 400.

Just look at the code below:

```
$('#myDiv').animate({"width": 400}, "slow");
```

That's it. A simple code that animates my width growth slowly. You can use "normal" or "fast" instead of "slow". You can also use a number to define a duration in milliseconds.

```
$('#myDiv').animate({"width": 400}, "fast");
// OR
$('#myDiv').animate({"width": 400}, 3000); // The animation will take 3 seconds to finish
```

If you want to animate multiple attributes at the same time, just declare more attributes inside the hash. Like this:

```
$('#myDiv').animate({"width": 400, "height": 500, "opacity": 0.8}, "slow");
```

Any CSS numeric attribute can be animated. Width, height, opacity, font size, etc. The list goes on and on.

It's important to note again that this only works on numeric attributes. Width is numeric, font-family is not. Background-color and font-color also cannot be animated unless you use a plugin.

Have fun!

Tip #15: Use delay to create sequences of animations

In our previous tip we saw how to create an animation composed of one or more attribute change. But these changes happen at the same time. If you want to do two or more animations in sequence, you can't just call `animate()` after `animate()`.

Everytime you call `animate()`, a new animation starts. Take a look of the following code:

```
$('#myDiv').animate({"height": 400},"slow").animate({width: 700}, "fast");
```

The code above is similar to what many people do when trying to chain animations for the first time. But the results are not the expected. You are telling jQuery to begin the first animation and right after you tell it to begin the second one. Both animations will be happening at the same time and it's possible the second one may finish first (because it's `"fast"`).

This is not the result you want.

If you want the second animation to start only after the first one ends, there are two ways.

First Method: Function as a parameter of animate()

The `animate()` method accepts a function. If you do pass one, it will call the function as soon as the animation stops. Look at the example below, where we get the effect we want:

```
$('#myDiv').animate({"height": 400, "slow", function() {
  animate({"width": 700}, "fast");
}});
```

Second Method: Use delay()

There are moments when you want to delay the second animation but you don't want it to begin exactly when the first one stops. Maybe you want it to begin a little sooner or later. The `delay()` method comes in handy.

This method takes a number as a parameter to define how many milliseconds it should delay execution of the next animation.

Let's say we want the first animation to have a duration of 3 seconds, and we want the second animation to begin half a second BEFORE the other one stops. Here's the code.

```
// Method chaining in several lines as seen on tip #6
$('#myDiv').animate({"height": 400}, 3000) // First animation starts with 3s length
  .delay(2500) // We wait 2.5s
  .animate({"width": 700}, "fast"); // Second animation starts
```

You can chain several `delay()` and `animate()` as you wish to create very complex multi-step animations.

Tip #16: How to prevent the default behavior in events

When the user submits a form, he's redirect. If another user clicks on a link she goes to a new page. That's default behavior, right?

But what if you don't want it to happen? You don't want the form submission to redirect the user. You don't want the link to take her to another page (or even reload the same page).

What if you want to prevent the default behavior of the event? Again jQuery comes to the rescue.

First you should know this: when you create a function to handle an event, this function can get the event itself as its argument. You just have to declare it takes an argument, like this:

```
$('a#myLink').on('click', function(event) {
  // Do something
});
```

Did you see the `event` parameter over there? It will point to the instance of the event that triggered the function. So if a user clicks on that link, the `event` variable will refer to that exact click.

Now that you know this, you just have to know that every event in jQuery has a method called `preventDefault()`. If you call this method anywhere in your function, the default behavior will not happen.

Example:

```
$('a#myLink').on('click', function(event) {
  event.preventDefault(); // Stops default behavior
  alert('You clicked me!');
});
```

By the code above, whenever a user clicks the link he will see the alert and that's it. No page redirect. If you call it on a form's submit event, it will not be submitted. On a link's click event, it will not redirect the user. And so on.

This is the basis of building dynamic user interfaces on your website. You can make links and buttons that do something else instead of redirecting the user somewhere else.

Tip #17: Don't let ajax requests capture your this

Tip #18: Set ajax defaults to standardize ajax requests

I was once in a project where there was many different asynchronous interfaces on the same page. Any time the user clicked on something an ajax request was done and it would show a spinning wheel. I was not very experienced in jQuery at that time and I used many times the same code to show the wheel while waiting and then hide it again when the process was completed.

There's a smarter way. There are 3 very handy methods to deal with ajax requests: `ajaxSetup()`, `ajaxStart()` and `ajaxComplete()`.

Use ajaxSetup() to set defaults

You can use the `ajaxSetup()` method to set defaults that any ajax method will follow. You can pass it any parameter a normal ajax request would take, and then these parameters will be passed to any ajax method you call.

Look at the example:

```
$.ajaxSetup({
    url: 'http://disney.com',  // Mickey likes ajax requests
    dataType: 'json'  // I want JSON unless I tell it otherwise
});
```

Now every ajax request I do on this page will have these two parameters as default. If I want a different dataType in one request, for example, I give it its own dataType parameter.

ajaxStart() and ajaxComplete()

These other two methods let you do something at the beginning and the end of ajax requests, respectively. Let's pretend we have a method `showIndicator()` to show a spinning wheel and another method called `disableButtons` that... well... disable buttons.

I want to disable buttons and show a spinning wheel while the request is pending. I want to revert it (hide wheel, enable buttons) when it's complete. Instead of repeating the code inside every request, I can do the following:

```
$.ajaxStart(function() {
    showIndicator();
    disableButtons();
});

$.ajaxComplete(function() {
    enableButtons();
    hideIndicator();
});
```

That's nice and easy, right? You should be using these methods to save you some time when doing a lot of similar requests.

Some other similar methods you might find useful: `ajaxStop()`, `ajaxError()`, `ajaxSuccess`, `ajaxSend`.

Tip #19: Live events can bind to elements in the future

The regular way to bind functions to events is nice, but it only affects elements that are already on the page when the binding happens. If you're working on a dynamic website where stuff gets added after the page is loaded, you need some way to make sure you're binding events of elements that will appear some time in the future.

Let's say you have a Task List website where tasks are added by javascript and you want to capture not only the events of existing tasks, but of every task yet to be added. Let's explore the `on()` method to know how to do that.

Since jQuery 1.7, the `on()` method is the default recommended method of binding events to functions. A regular binding of the click event works like this:

```
$('a.task').on('click', function() {
    alert('I was clicked!');
});
```

As we saw earlier, this is not enough to capture events of element in the future. To do what we want, it's important to first learn about event bubbling.

Event Bubbling

Inside browsers, any uncaptured event bubbles up. Imagine you have the following HTML code:

```
<div>
    <p>
        <a>Go to Disney!</a>
```

```
    </p>
</div>
```

When the user clicks on the link, a `click` event is fired. If there's nothing set up to capture click events in the `<a>` element, it will bubble up to our `<p>` element. If nothing captures the event, it will again bubble up to the `<div>` element. And it will keep going up the element tree, up to `<body>`, then `<html>` and then, if it's still not captured, it will fade away.

Why is this important? Because of event delegation!

Delegating events

How do we set up code to capture events on any task in our task list, including future ones? We tell its parent element to watch out for events! That's event delegation.

And how do we do this in jQuery? Look at the code below:

This is our HTML:

```
<ul>
    <li><a>Task One</a></li>
    <li><a>Task Two</a></li>
    <li><a>Task Three</a></li>
</ul>
```

And our javascript:

```
$('ul li').on('click', 'a', function() {
    alert('a task was clicked!');
});
```

If you look closely at the `on()` method, this time I used 3 parameters instead of just 2. Let me break it down and explain it bit by bit.

First of all I'm using the `on()` method on the `<li>` element. He will be the one watching for click events.

As the second parameter I used `'a'` to tell the `<li>` element it will be watching for click events on `<a>` elements. In the background, the following will happen: anytime an event bubbles up to our `<li>`, it will check where it originate from. If it came from any `<a>`, the function will be called.

This means it does not matter how many tasks I add to my list and when I add them, because the `<li>` will watch out for events in any `<a>` inside it.

Tip #20: Use the right event binding to fit your needs

Better jQuery
Leonardo Bighi