17 How to Create HTML with jQuery

jQuery is a very popular library for JavaScript that adds many functions and data structures to make it easier to write code to modify HTML pages. Although some of these additional functions are not very complicated, there are several reasons to use jQuery functions, rather than writing your own:

- Fewer bugs:
    - jQuery developers have created a large suite of tests that they run over and over to catch mistakes.
    - Thousands of programmers have used the jQuery functions, so many bugs have been found and fixed.
- Greater portability:
    - Browsers vary in how well they implement the ECMAScript standards. jQuery developers have written code to check for and deal with browser-specific issues.
- Greater familiarity:
    - Most serious JavaScript programmers have used jQuery and can understand jQuery code.

jQuery is not the only library by any means. It's a good one to know but feel free to explore others.

Here, we'll just cover a few of the most basic and critical concepts and functions in jQuery.

$()

The first thing that strikes you about jQuery code is the ubiquity of the form **$(...)**. This is not some special syntax. **$** is just the name of a function. The function has another name, **jQuery**. **jQuery(...)** and **$(...)** mean exactly the same *thing.*

$() is so ubiquitous because it is used to do several very different but important tasks. What task it does depends on what kind of argument is passed to **$()**. We'll cover the major usages below.

$(*selector*)

Given a string that looks like a CSS selector, e.g., **$(".app")** or **$("div.index ul li")**, this creates a *jQuery object*. A jQuery object is a collection of HTML DOM elements, plus functions that can be applied to every element.

```
$("div.index ul li")
```

will select -- i.e., create a jQuery object containing -- every LI element inside any DIV with the class INDEX.

> This is somewhat like **document.querySelectorAll()**. jQuery was developed before this method existed. **querySelectorAll()** returns a NodeList, which doesn't have all the useful methods that the jQuery object has.

The reason why this is useful is that once a jQuery object is created, you can apply functions to every element it contains.

Let's start with a simple case. There is a jQuery object function **fadeOut()** that will cause selected elements to fade to invisible. So

```
$("div.index ul li").fadeOut();
```

will cause every LI element inside any DIV with the class INDEX to fade away.

A very useful jQuery object function is **html()**. This function replaces the HTML in the objects collected in the jQuery object.

For the following code makes a jQuery object containing whatever element, if any, has the HTML ID "message". It then replaces the inner HTML of that element with the given HTML.

```
$("#message").html("<span class='warning'>Something broke!</span>");
```

An important property of most jQuery object functions is that they return the jQuery object. This means that you can string multiple actions together very easily.

For example, the following code would replace all the LI elements with "going away", then, 5000 milliseconds later, fade them out.

```
$("div.index ul li").html("Going away!").fadeOut(5000);
```

Attaching Event Handlers to Elements

One important jQuery object function is **on()**. This lets you attach event listeners to objects on a page.

For example, suppose we have a Search button but we haven't implemented code for it yet. We could use the following to pop up an alert box, letting users know search is not ready yet.

```
$('#search-button').on('click', function() { alert('Search is not implemented yet.'); });
```

Executing this code does not call **alert()**. It just attaches a function to the object with the ID "search-button". If that object is clicked, the function will be called and the alert will appear. Later, we can replace the **alert()** call with a call to our actual function.

The **on()** method attaches an event handler to every element the selector finds on the page. Inside the event handler function, **this** is the element that was clicked on. **$(this)** returns that element with a jQuery object wrapped around it, so that you can call jQuery functions with it.

So if you wanted to make it so that clicking on any image on a page opened that image in a new browser tab or window, you could do this

```
$('img').on('click', function() { window.open($(this).attr('src')) });
```

this will be the image clicked on, and **$(this).attr('src')** will use jQuery to get the URL for the image. **window.open()** is a standard JavaScript function to open a window with a URL.

The above examples only works with elements that already exist in the HTML. They won't work for elements that get added to the web page at runtime with JavaScript.

When you have that situation in jQuery, you can use delegated events. You place a handler on the containing element that does exist, where you will dynamically store new elements. Then, in the **on()** call, after the name of the event type, you specify another selector. When the user clicks on a dynamically added element, the click is captured by the containing element. jQuery then uses the second selector to determine what elements to actually call with the event handler function.

To make this concrete, suppose we have code that adds images to an HTML **div** with the ID **pictures**. Then to make it so that clicking an added image opens a new window with that image, we'd do this:

```
$('#pictures').on('click', 'img', function() { window.open($(this).attr('src')) });
```

$(*function*)

This is a completely different use of **$()**. This means "call the function after the page is loaded."

> This is somewhat like saying **document.addEventListener("DOMContentLoaded",** *function***)**.

The most common use of this is to include the code you want to define and run on a page. I.e., in your script file, you write

```
$(function () {
... all your code ...
});
```

This has two useful effects:

- Your code won't run until the page is ready.
- Your function and variables names will be buried inside a function. That means you won't have any name collisions with other code you load.

The downside is that in your browser's JavaScript Console, you will not normally have access to the functions and variables you define. You will have access in the debugger if you set a breakpoint to stop your code at some useful point in your code.

$ Global Functions

Besides functions on the jQuery object, there are a number of useful global jQuery functions. These are called by writing **$.***function*(). Notice that there are no parentheses after the **$**. These are just functions you can call. They are not related to modifying objects in a jQuery object.

$.getJSON(*url***)** takes a string with a URL. It sends a request for JSON to the URL. This may take a while to get. It may time out and never happen. To avoid locking up the browser waiting for an answer, **$.getJSON()** returns immediately with something called a **promise**. A promise is an object waiting for a value.

Just has jQuery objects include functions that can be called, such as **on()** and **html()**, promises include several key functions.

- *promise***.done(***function***)** says "call the function given when (if ever) the value is received."
  - This makes **done()** a bit like **on()** in jQuery objects.
  - If the function is called, it will be passed one argument, with the JSON sent by the server.
- *promise***.fail(***function***)** says "call the function given if something goes wrong."
  - This can happen if there's an error, such as a bad URL, if the server rejects the request, or if there's a timeout, because the server took too long to respond.
  - If the function is called, it will be passed three arguments:
    - a jQuery object representing the request; you can ignore this
    - a short string indicating what happened, e.g., "timeout", "error", "abort", "parsererror", or null
    - if what happened was "error", the HTTP error text from the server, e.g., "Not Found" or "Internal Server Error"

Promises have become very common in web programming. Virtually every library support calls to web services supports them now.