## jQuery Review

As we have seen from the notes in WEB322, the jQuery Library is an extremely popular and valuable tool for front-end web application development. To refresh your memory:

“jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.”

Today, we will be re-examining the way that jQuery can help us manipulate the DOM to gain complete control of the elements on the page, including: “Selecting / Accessing Elements”, “Watching for Events” & “Updating Elements”.

Lastly, we will look at how jQuery can make AJAX calls simple to execute and we will try querying our Teams API for data and updating the DOM. You can think of this as a way to replace what was initially done in Handlebars.js in WEB322. However, instead of relying on the server to generate the dynamic content before serving the page, we will shift that logic to the client (in this case, a web browser). By decoupling the data-rendering tasks from the server, we gain more freedom in how we render the data. This allows us to create richer user experiences across multiple different platforms.

Before we start to re-examine creating DHTML with jQuery, we should first grab the sample code from the course [Github Repository](https://github.com/sictweb/web422). It can be found under **Code Examples/week1**.

To begin, open up the **week1/jQuery** folder and take a look at the index.html page. Here, you will see a perfectly normal HTML page with some elements on it, ie: a <table>, <form> some input elements, etc. This page also includes some important css/js files including:

#### CSS

* [The bootstrap 3.3.7 compiled/minified CSS file from the maxcdn.bootstrapcdn.com CDN](https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css)
* Our own custom css file (main.css) located in the “css” directory

#### JavaScript

* [The jQuery 3.2.1 minified JS file from the code.jquery.com CDN](https://code.jquery.com/jquery-3.2.1.min.js)
* [The bootstrap 3.3.7 minified JS file from the maxcdn.bootstrapcdn.com CDN](https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js)
* Our own custom js file (main.js) located in the “js” directory

### The ‘$(function() { });’ or ‘$(document).ready(function(){ });’ functions

It stands to reason that any JavaScript code that deals with accessing elements in the DOM **must** be executed after the DOM is built. It is for this reason that you will see most jQuery code residing in an anonymous function delcared as a parameter to either **$();** or **$(document.ready());**. When the DOM is ready, the supplied anonymous (“callback”) function will be executed, ensuring that any DOM operation within the function will be safe to use.

### Selecting / Accessing Elements

jQuery has a rich [“selector” framework](https://api.jquery.com/category/selectors/) for getting a reference to a specific element in the DOM.

As we have seen, these selectors typically follow the syntax of CSS selectors (ie: $(“#element”) in jQuery will reference the DOM element with id=”element”). The following table shows some of the most widely used selectors and how they work to access DOM elements in our sample **index.html** page within the **week1/jQuery** folder.

| **Selector** | **Description** | **Number of Elements** |
| --- | --- | --- |
| $( “\*” ); | **All Selector:** Selects all elements | 96 |
| $( “#animal-table” ); | **id Selector:** Selects a single element with the given id attribute. | 1 |
| $( “.table-heading” ); | **class Selector:** Selects all elements with the given class. | 3 |
| $( “:input” ); | **input Selector:** Selects all input, textarea, select and button elements. | 9 |
| $( “:radio” ); | **radio Selector:** Selects all elements of type radio. | 2 |
| $( “:checkbox” ); | **checkbox Selector:** Selects all elements of type checkbox. | 2 |
| $( “:visible” ); | **visible Selector:** Selects all elements that are visible. | 80 |
| $( “:hidden” ); | **hidden Selector:** Selects all elements that are hidden (the opposite of :visible). | 16 |
| $( “tr:odd” ); | **odd Selector:** Selects odd elements, zero-indexed. See also [even](https://api.jquery.com/even-selector/) | 3 |
| $( “.row:has(#animal-table)” ); | **has() Selector:** Selects elements which contain at least one element that matches the specified selector. | 1 |

As you can see from the above table, it is often the case that a selector will select **more than one** element. This is fine if we want to apply the same operations to a group of elements. However, if we wish to access each selected element individually, we can leverage jQuery’s [each() function](http://api.jquery.com/jquery.each/) to iterate over the elements in the collection. For Example:

$( "tr:odd" ).each(**function**(index){

$(**this**).css({"background-color":"#eff6f7"}); *// color the odd rows gray*

});

The .each() function will iterate over every selected element returned by the selector and invoke the provided “callback” function. You can optionally pass a parameter to the callback function if you wish to know the current index (0, 1, 2, 3, etc).

During each iteration, the current element can be accessed using “$(this)”. It’s a little strange to see “this” wrapped in a jQuery function, however we need to wrap “this” (ie: the current element) into a jQuery object so that we can perform jQuery specific operations on it (ie, using the [.css()](http://api.jquery.com/css/) method). If we remove the code surrounding “this”, we will run into the error:

TypeError: this.css is not a function

### Watching for Events

Now that we have access to the element (or elements) we wish to modify in the DOM, we can apply a number of operations to modify their appearance and/or behaviour using jQuery. One extremely important aspect in creating a reactive and dynamic view (DHTML) is watching a given element for a user / system initiated event and responding to it with custom code.

jQuery provides this functionality via it’s [on()](http://api.jquery.com/on/) method:

.on( events [, selector ] [, data ], handler )

We can apply this method to any selected element in order to wire up (“respond to”) an “event” by passing a callback function to the “handler” parameter, ie:

$(".row").on("change", ":input", **function**(){ *// watch existing "row" div elemennts for when input elements change*

console.log("id: " **+** $(**this**).attr("id") **+** "changed");

});

What makes this function so powerful is that if we invoke the “.on” method on an element that isn’t likely to change (an outer container or “document”, for example), we can use the “selector” parameter to apply the callback function to **any** child element that matches that selector (even if it’s dynamically created in the future).

This means that even if we use some of jQuery’s DOM modification methods (see below) to dynamically create a new input element (contained within an existing element with class=”row”) **after** we define the event, it will **still be applied** to the new input element. This can sometimes lead to event handlers that take the form:

$("document").on("click", "tr", **function**(){ *// watch the whole document for when existing (or new) tr elements are clicked*

console.log("table row clicked!");

});

By defining the “click” element on the entire “document” object and filtering it by only affecting “tr” elememnts, we can guarantee that all “tr” elements will invoke the supplied function when clicked, regardless of when they are created in the future. This saves us the hassle of wiring up new events every time we create a new <tr> element.

### Updating Elements

The final piece in creating dynamic html (DHTML) is actually modifying the DOM by creating, destroying or modifying elements (nodes) within the DOM tree. JavaScript itself provides a number of methods for dealing with the DOM, however jQuery extends this functionality and provides some new methods as well. Some of the more common methods are as follows:

| **Method** | **Description** | **Result** |
| --- | --- | --- |
| let newDiv = $('<div>'); | **$('', {})** Create a new element by specifying a string defining a single, standalone, HTML element (e.g. <div> or <div></div>), followed by an optional object consisting of attributes, events, and methods to call on the newly-created element. | Creates a new "div" element and stores it in the variable: "newDiv" |
| newDiv.css({"border":"1px solid lightgray", "padding":"10px"}); | **.css()** Get the value of a computed style property for the first element in the set of matched elements or set one or more CSS properties for every matched element. | Sets the "border" and "padding" CSS properties on the "newDiv" element via it's "style" attribute |
| newDiv.html("<span>New Div!</span>"); | **.html()** Get the HTML contents of the first element in the set of matched elements or set the HTML contents of every matched element. | Sets the inner HTML of the "newDiv" element to contain a <span> element with the text "New Div!" |
| $("#new-content").append(newDiv); | **.append()** Insert content, specified by the parameter, to the end of each element in the set of matched elements. | Adds the element "newDiv" as a child of the element with id="new-content" |
| let newDiv2 = newDiv.clone(); | **.clone()** Create a deep copy of the set of matched elements. | Creates an exact copy of the "newDiv" element and saves it as "newDiv2" (does not automatically add it to the DOM) |
| newDiv2.attr("id", "clonedDiv1"); | **.attr()** Get the value of an attribute for the first element in the set of matched elements or set one or more attributes for every matched element. | Set the "id" attribute of the "newDiv2" element as "clonedDiv1" |
| newDiv2.addClass("bg-color-lightgray"); | **.addClass()** Adds the specified class(es) to each element in the set of matched elements. Also see [.removeClass()](https://api.jquery.com/removeclass/) | Add the class "bg-color-lightgray" to the "class" attribute (since there is no class attribute, it creates one) |
| newDiv2.wrap("<div class='outer'></div>"); | **.wrap()** Wrap an HTML structure around each element in the set of matched elements. | Wrap the "newDiv2" element in a <div class="outer">...</div> element |
| newDiv2.text("Cloned Div!"); | **.text()** Get the combined text contents of each element in the set of matched elements, including their descendants, or set the text contents of the matched elements. | Set the text content inside the "newDiv2" element to read "Cloned Div!" (effectively replacing the text "New Div!") |
| $("#to-be-replaced p").replaceWith(newDiv2); | **.replaceWith()** Replace each element in the set of matched elements with the provided new content and return the set of elements that was removed. | replace the <p> element inside the element with id="to-be-replaced" with the "newDiv2" element |
| $("#exampleInputEmail1").val("from jQuery!"); | **.val()** Get the current value of the first element in the set of matched elements or set the value of every matched element. | Sets the value of the input element with id="exampleInputEmail1" to "from jQuery!" |

Once again, this is only a small sampling of some of the more commonly used DOM manipulation methods - for a full list of the 40+ methods available, refer to the [Official Documentation here](https://api.jquery.com/category/manipulation/).

### AJAX with jQuery

One of the most important technologies used in this course will, of course be AJAX (Asychronous Javascript and XML). As we have seen from previous courses, AJAX allows us to make requests for data using JavaScript without having to reload the page. This will form the basis for our Single Page Application (SPA) architecture: we will be sending / recieving data from our Back-End REST API on demand using AJAX.

For this next example, open the “jQuery-AJAX” folder within the “week1” examples folder. Here, you will find that the file structure is much the same, ie:

* a single index.html page referencing our own custom JavaScript / CSS (main.js / main.css) as well as the jQuery and Bootstrap libraries (JS & CSS).
* a js folder for our main.js file
* a css folder for our main.css file

However, the content of the files is very different: Instead of a complete page in the index.html file, we only have a skeleton of a Bootstrap table:

<table class="table" id="employees-table">

<thead>

<tr>

<th class="table-heading">First Name</th>

<th class="table-heading">Last Name</th>

<th class="table-heading">Position</th>

</tr>

</thead>

<tbody></tbody>

</table>

Instead of filling the table with static data in the HTML file, we will instead use AJAX and our Teams API to fetch the data.

Recall: to perform an AJAX “Get” request with jQuery, we can use the following code (**Note:** this assumes that the Teams API is running locally. If you have pushed the Teams API to your Heroku account and wish to use that instead, simply replace “http://localhost:8081” with your Heroku URL, ie: “https://some-crazy-heroku-name.herokuapp.com”.

$.ajax({

url: "http://localhost:8081/employees", *// This only works if the Teams API is running*

*// locally - change this url to your Heroku API*

*// (/employees) to use your API on Heroku*

type: "GET",

*//data: JSON.stringify({ some: "data" }), // we can also send data using the "data" option with PUT or POST*

contentType: "application/json"

})

.done(**function** (employees) {

console.log(employees); *// output all employees to the console*

})

.fail(**function** (err) {

console.log("error: " **+** err.statusText);

});

If the Teams API is up and running properly, this will show an array in the console that consists of the complete list of all 300 employees. This data is no good to us in the console however, so let’s make use of some of those DOM modification techniques that we learned above to populate our #employees-table?

To begin, why don’t we restrict the results to the first 5 employees (for now)? Each employee should be in it’s own <tr> element with each piece of data (First Name, Last Name & Position) in their own nested <td> element. For each employee that we’re going to add to the DOM (specifically, the <tbody> element inside the #employees-table), the end result should look like:

<tr>

<td>First Name</td>

<td>Last Name</td>

<td>Position</td>

</tr>

To achieve this, we can use the following code within the “.done()” callback (explained below)

**let** tbody **=** $("#employees-table tbody");

**for**(**let** i **=** 0; i **<** 5; i**++**){ *// only show the first 5 employees*

*// Create each table cell for the current employee row and add the text*

**let** fNameTD **=** $('<td>').text(employees[i].FirstName);

**let** lNameTD **=** $('<td>').text(employees[i].LastName);

**let** positionTD **=** $('<td>').text(employees[i].Position.PositionName);

*// append all table cells to a new row*

**let** row **=** $('<tr>').append(fNameTD).append(lNameTD).append(positionTD); *// create the row and append*

*// all of the TD elements.*

*// append the row to the table body*

tbody.append(row);

}

Here, we save a reference to the target <tbody> element as “tbody” so that we do not have to query the DOM over and over again unnecessairly when we want to append our data (<tr> elements) to it. Additionally, we will restrict the loop to only 5 iterations (ie, the first 5 employees in the collection) and append our data to the “tbody” at the end of every iteration.

To construct the table row & cells as efficiently as possible, we can leverage [“chaining”](https://www.w3schools.com/jquery/jquery_chaining.asp) in jQuery. Essentially, since most jQuery methods return the newly modified element, we can keep invoking DOM modification methods one at a time on the same element in the pattern:

element.doSomething().doSomethingElse().doSomethingElse().etcEtcEtc();

In the code above, we create new <td> elements for each of the properties that we want to show in the table - in this case: First Name, Last Name and Position Name (PositionName being a property of the “Position” object within the “employee” object). We can immediately invoke the **.text()** method because $(‘ ‘) returns a reference to the new object!

This notion of chaining really starts to show it’s value when we append all of the newly creaed <td> elements to a brand new <tr> element.

Lastly, we append the new row onto the tbody element and move onto the next employee.

Using this strategy of creating DOM elements in response to the acquisition of new data is the first step in creating rich, data-driven web applications. It may seem easier to simply serve up the rendered page using Handlebars, but we’re only getting started with client-side development. The JavaScript community has been busy and there are lots of frameworks to help us to deal with dynamic, on-demand data on the client side, and we will see some of these in the coming weeks. In the mean time…

## More jQuery Next Week

* Continuing coverage of jQuery, including topics:
  + [jQuery Utility Methods](http://api.jquery.com/category/utilities/)
  + jQuery Plugin Community: <https://plugins.jquery.com/> & [NPM](https://www.npmjs.com/browse/keyword/jquery-plugin)
  + Advanced data rendering (Bootstrap, flot, populating forms)
  + Using jQuery to invoke Bootstrap Controls
  + Using jQuery to implement “data binding”

© 2017 - Seneca School of ICT