6.170 Tutorial 6 - jQuery

Prerequisites

1. Basic understanding of JavaScript

Goals of this tutorial

1. Understand the distinction between jQuery and JavaScript
2. Understand the core features of jQuery (AJAX, DOM manipulation, events, and effects)

Resources

- jQuery documentation: http://api.jquery.com/
- Convenient JS/HTML/CSS playground: http://jsfiddle.net/

Topic 1: JavaScript libraries - why jQuery?

JavaScript has been notorious for its inconsistent implementation across browsers, its verbosity, and in general its clunky API. In recent years, browsers like Chrome and Firefox have really accelerated the development of web technologies, and JavaScript APIs have been cleaned up and standardized. Messy parts still remain, though, and compatibility with older browsers will always be an issue.

jQuery is an extremely popular JS library in use across a huge range of sites including Wikipedia, Amazon, Twitter, and more. It's main offering is a concise, cross-browser API that handles many common JS tasks for you. The best way to get a better idea of what that means is to look at jQuery's core strengths: AJAX, DOM manipulation, and DOM events.

How to use jQuery

More concretely, you include jQuery as another JS file that gives you a variable named **$** - jQuery's API is exposed through calls to **$**. In the example below, we'll use the CDN-hosted jquery (provided by MediaTemple and officially distributed by jQuery):

```
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script>
$(document).ready(function() {
       $('img').click(function() {
              $('table li').slideUp();
       });
});
</script>
```

For Rails specifically, though, you'll have jQuery automatically included in your JS files. You should have a main application.js file that "includes" jquery already - as part of the Rails asset pipeline jQuery is "compiled" into the final JS files.

This tutorial will cover a sampling of the helper functions jQuery provides. You're encouraged to peruse through http://api.jquery.com for a thorough (but easy-to-read!) listing of methods.

If you're wondering what $(document).ready() does, it's to ensure that the entire document has finished loading before we bind code to elements. Check out the ***Event Handling*** *topic for more details.*

Topic 2: AJAX

In project 1, you got a brief taste of asynchronous requests:

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
  if (xhr.readyState == 1) { // call to open succeeded }
  if (xhr.readyState == 2) { // call to send succeeded }
  if (xhr.readyState == 3) { // response is coming in }
  if (xhr.readyState == 4) { // response completed }
}
xhr.open('GET', '/url/to/something', true);
xhr.send();
```

jQuery makes this much easier with `$.ajax` (many more settings are listed at http://api.jquery.com/jQuery.ajax/):

```
$.ajax("/url/to/something", {
       type: 'PUT',
       data: {
              param1: 'param1Value',
              param2: 'param2Value'
       },
       dataType: 'JSON' // Usually unnecessary, jQuery can guess the return type
       // etc - many more settings are available
}).done(function(data) {
       console.log("Request successful: " + data);
}).fail(function(data) {
       console.log("Server gave an error: " + data);
}).always(function() {
       console.log("Regardless, this request has finished.");
});
```

You can leave out the fail and always handlers if you're only interested in the successful calls.

(NB: You'll see this "chaining" style of coding a lot in jQuery. Many object methods will return the object again so you can call more methods on it: `done()` is called on the return value of `$.ajax()`, this same return value is returned by `done()` to be used for `fail()`, and so on.)

There's also simplified `$.post` and `$.get` methods that are wrappers around `$.ajax`:

```
$.post("/url/to/something", {
       param1: 'param1Value',
       param2: 'param2Value'
}, function(data) {
       console.log("Success! data: " + data);
});
```

Using $.ajax and its variants have a few key benefits:

- More concise syntax, cleaner error handling

- Cross-browser compatible (XHR behaves badly in certain browsers like IE)
- Automatic data-parsing (JSON responses get detected and transformed into JS objects)

In general, if you're already using jQuery on your website (for more reasons we'll see below) there's very few reasons to *not* use it's AJAX functionality. Check out https://github.com/jquery/jquery/blob/master/src/ajax.js to get some appreciation for the messy work handled by jQuery - try searching for "IE" for example.

Topic 3: DOM traversal and manipulation

jQuery also makes working with HTML elements (also referred to as “DOM nodes”) very easy.

Manipulating

API docs: http://api.jquery.com/category/manipulation/

We’ll refer to moving, inserting, and deleting DOM nodes as “DOM manipulations” - you’ll find DOM manipulation to be useful for adding new elements, showing new data, showing AJAX results, and responding to user interaction.

Creating new elements is easy:

```
var newList = $(‘<ul>’).addClass(‘my-class’);
var el1 = $(‘<li>this is literally HTML</li>’);
var el2 = $(‘<li class=”inline-classes”></li>’);
var el3 = $(‘<li>’).html(‘It’s cleaner to chain it though’).addClass(‘anotherClass’);
var el4 = el1.clone().html(‘Cloned but edited!’);
newList.append(el1).append(el2).append(el3).append(el4);
$(‘#container’).append(newList);
```

- **parent.append(child)** inserts child as the last child inside parent
- **parent.prepend(child)** inserts child as the first child inside parent
- **sibling1.after(sibling2)** inserts sibling2 after every sibling1
- **sibling1.before(sibling2)** inserts sibling2 before every sibling1
- and many more: insertAfter, insertBefore, appendTo, replaceWith, etc

Deleting elements is simple too:

```
$(‘li’).remove() // remove all list elements
$(‘ul’).empty() // keeps the ul, but removes all children inside them
var detached = $(‘ul’).children().detach() // like remove, but doesn’t destroy them
detached.appendTo(‘ul.aggregate’); // attach them all to one ul
```

Selectors

API docs: http://api.jquery.com/category/selectors/

jQuery uses CSS-style selectors (NB: they only resemble CSS, and aren’t dependent on any underlying CSS implementation) to make finding elements very easy. With vanilla JS, you might see:

```
var els = document.getElementById("foo").getElementsByClassName("bar");
```

With jQuery, this becomes:

```
var els = $(“#foo .bar”);
```

The benefit becomes clearer when your queries become more complex:

```
// Password fields
$("input[type=password]")
// Anchor tags that directly follow an image with class="foo bar"
$("img.foo.bar + a")
// Even rows in a table
$("table tr:nth-child(2n)")
```

The return value of jQuery selectors is a **result set** that can behave differently with different functions (more below). For example, $('img').html() will return the html of the **first** image, while $('img').parent() will return the set of DOM nodes that have an image as a child. Most of the modifiers and selectors operate on all elements, but trial-and-error will reveal quickly which does what.

Modifying DOM nodes

Once you can select the elements you'd like to modify, you can actually manipulate them.

.attr() is the most generic method, and can modify or access any HTML attribute:

```
// Set the alt text property of every profile image to 'profile'
$('img.profile').attr('alt', 'profile')
// Get the src attribute of the (first) image
$('img').attr('src')
// Set multiple attributes on ALL images
$('img').attr({
       alt: 'alt text',
       title: 'title',
       src: '/image.png'
});
```

.addClass() / .hasClass() / .removeClass() are handy functions for modifying classes:

```
$('img').addClass('oneClass').addClass('twoClass').addClass('three four');
$('img').removeClass('oneClass twoClass');
```

.css() is similar to .attr() but works with css properties:

```
$('img').css('display', 'none');
```

.html() is like the .innerHTML property (from the DOM API) and returns everything inside the node:

```
// <p><a href="/">Go home</a> or check out this image: <img src="/img.png" /></p>
$('p').html() // <a href="/">Go home</a> or check out this image: <img src="/img.png" />
```

.val() gets the you value of form elements:

```
$('input[type=password]').val() // What password got typed in?
```

And many more! Check out .width()/.height(), and .toggleClass() in the docs.

Traversing

API docs: http://api.jquery.com/category/traversing/

jQuery makes it easy to iterate, to search through parents, and to search through siblings.

.each(f(index, value)) takes a callback function and passes in the index and value of each element:

```
$('img').each(function(i, el) {
        var src = $(el).attr('src');
        console.log("Image #" + i + " has src=" + src);
});
```

Note that the **this** variable is bound to the iterated element inside the function, so we can instead do this:

```
$('img').each(function(i) {
        var src = $(this).attr('src');
        console.log("Image #" + i + " has src=" + src);
});
```

.parent() finds the direct parent, **.parents()** finds all parents,, **.children()** finds all direct children, **.find()** finds all children, direct or not. They all take optional selectors to filter by, i.e. **$('img).parents('table')** will find tables that contain images.

.closest(selector) find the first ancestor that matches the css selector.

.filter(selector) lets you narrow down elements selected, and **.not(selector)** will take the opposite.

.eq(i) will take the i-th element, so $('img').eq(1) selects the 2nd image

Topic 4: Event handling

API docs: http://api.jquery.com/category/events/

DOM “events” are a general concept that allows JS to respond to user input and other asynchronous changes on a webpage. jQuery makes it easy to handle them. The functions below all take callback functions as parameters - calling them binds the callbacks to be called when the corresponding events take place.

.click() calls a function on click:

```
$('img').click(function(event) {
	$(this).remove(); // let’s remove any image when it’s clicked.
	console.log(‘image was clicked!’);
	console.log(event) // check out the event information
});
```

Note the **event** parameter passed to the callback function. All callback functions will generally take in an event object - this contains useful information that contains different information depending on the context. A click event will have information about the click location, for example, while keydown events have information about which keys were pressed.

jQuery binds **this** to the DOM element that triggered the event. In the above example, **this** refers to the image that was clicked.

- **.dblclick()** binds to double clicks
- **.focus()** binds to “focus” for inputs - when you click inside a text box, when you expand a dropdown, for example
- **.blur()** is the opposite of focus - when you leave a text box, for example
- **.change()** is useful for when inputs change value
- **.scroll()** binds to scroll events inside the element
- **.submit()** binds to when a form gets submited
- **.keyup() and .keydown()** bind to when keys are pressed - use the event’s keyCode property to figure out what was pressed. (Printable characters will correspond to their character codes, while other keys like arrow keys have codes as well.)

  ```
  $(document).keypress(function(e) {
      // e.keyCode == 38 for UP
      // e.keyCode == 65 for A
      // e.keyCode == 16 for SHIFT
      // e.keyCode == 13 for ENTER
      // etc
      console.log(e.keyCode);  // DIY
  });
  ```

Some handlers take two functions:

```
$('img').hover(function() {
	console.log('mouse over');
}, function() {
	console.log('mouse gone');
});
```

It's important to realize that when you call these binding functions, they get attached to elements that exist at the time of the function call. Thus, if you call $('img').click(..) and then add new images, the new images won't have the click handlers attached.

parent.on(eventName, target, handler) is useful to get around this, and also works on elements created in the future:

```
$('.container').on("click", "img", function(event) {
	// Works on an image on the page, now or future.
});
```

.ready() will be especially useful; the snippet you'll use the most will probably be:

```
$(document).ready(function() {
	// Do DOM-related work here.
});
```

When the document finishes loading everything, the ready() handler is called. This is important if you want to bind to DOM elements that may not have loaded at the time the script is called. The below code, for example, does **not** work:

```
<script>
// Binds to 0 img elements.
$('img').click(function() { alert('clicked!'); });
</script>
<img src="image1.png" />
<img src="image2.png" />
```

The browser will evaluate the script tag before building the img nodes, so no click handlers get attached. Especially since script tags should be included in the head and not body of the document, you'll find this true for most if not all of your event binding. Instead, you want:

```
<script>
$(document).ready(function() {
	// Binds to 2 img elements.
	$('img').click(function() { alert('clicked!'); });
});
</script>
<img src="image1.png" />
<img src="image2.png" />
```

A final note on event propagation: Event propagation and event listeners are a relatively complicated subject in JavaScript, but it's useful to know some basics:

- **event.preventDefault()** stops the current element's default browser handler from being called. This is useful for overriding anchor tags, for example, and preventing the user from leaving the page when a link is clicked:

```
$(‘a’).click(function(event) {
    console.log(‘the link with href=’ + $(this).attr(‘href’) + ‘ was called!’)
    // User stays on page.
    event.preventDefault();
});
```

- **event.stopPropagation()** prevents events from propagating to parent nodes. If you have an image inside a div and both have click handlers, returning false from the image’s click handlers will prevent the div’s handler from being reached. This is useful, for example, for stopping a submit() action from reaching the parent form if you want to abort a submit or handle it via AJAX instead.
- **return false** from an event handler has the effect of doing both. This is considered bad practice - use preventDefault() or stopPropagation() instead.

Topic 5: Effects and animations

API docs: http://api.jquery.com/category/effects/

Less important than the previous topics, jQuery's effects and animations can still save you a good amount of work. Some examples (most take callback functions and length parameters):

```
$('.container').hide(); // equivalent to setting display:none
$('.container').show(); // also .toggle()
$('.container').fadeIn(2000); // fade in over 2s, also .fadeOut() and .fadeToggle()
$('.nav li).slideDown(); // also .slideUp() and .slideToggle()
// Animate CSS properties:
$('a').animate({
       'opacity': 0,
       'color': 'red'
});
// Stop any current animations and start a new one:
$('a').stop().animate({
       'background-color': 'blue'
});
```

Topic 6: JavaScript and Rails

If you've worked with JavaScript before, you'll be familiar with using script tags in the beginning of your document:

```
<!DOCTYPE html>
<html>
<head>
  <script src="/path/to/script.js"></script>
…
```

You'll notice, however, that Rails doesn't do this. Instead, you'll see:

```
<!DOCTYPE html>
<html>
<head>
  <%= javascript_include_tag "application" %>
...
```

The method javascript_include_tag takes a list of JS files - each string can include or omit the "js" or "js.coffee" extension (it'll try to guess if necessary) and can be relative or absolute:

```
<%= javascript_include_tag "application", "visits" %>
  Generates:
  <script type="text/javascript" src="/javascripts/application.js"></script>
  <script type="text/javascript" src="/javascripts/visits.js"></script>
<%= javascript_include_tag "visits.js", "/absolute/path/tofile" %>
  Generates:
  <script type="text/javascript" src="/javascripts/visits.js"></script>
  <script type="text/javascript" src="/absolute/path/tofile"></script>
```

A quick note on CoffeeScript (http://coffeescript.org/): CoffeeScript is a "ruby-fied" version of JavaScript that attempts to clean up a lot of syntatical issues in JS. The .js.coffee files get pre-processed at deploy time into actual JavaScript. Some people like it; some people don't. We don't require or suggest using it in this course, as it'll make it harder for TAs to consistently grade code. When you see .js.coffee files, though, which rails will generate for you, now you know it's looking for CoffeeScript. If you remove the files and use plain .js files you should be fine.

You'll notice your **application.js** (which is automatically included into your templates) looks like:

```
//
//= require jquery
//= require jquery_ujs
//= require_tree .
```

What's happening here is Rails is using a pre-processor to clump all of your JS files together.

The idea is to allow you, the developer, to develop in separate JS files for reusability and logical organization. **require** directives indicate which dependencies your JS has, and **require_tree .** tells Rails to blindly include all the JS files in app/assets/javascript/ and its subfolders recursively.

This sounds bad, but in most cases this will actually be just fine. You might think that including exactly the JS files you need will help avoid unnecessary loading, but you're usually better off loading them all at once. At the cost of one slow load, the subsequent pages will all have their JS cached in the browser and loading will actually be faster. Furthermore, when Rails automatically compresses the files into a single file for you (see below) only one HTTP request is made and the download is faster.

When you run in development mode, the JS files don't get modified and you'll see a long list of included JS tags:

```
<script src="/assets/jquery.js?body=1" type="text/javascript"></script>
<script src="/assets/jquery_ujs.js?body=1" type="text/javascript"></script>
<script src="/assets/posts.js?body=1" type="text/javascript"></script>
<script src="/assets/application.js?body=1" type="text/javascript"></script>
```

In production, though, you'll see much less:

```
<script src="/assets/application-6cb7fbef54b7407f120c1360a22a4bfd.js"
type="text/javascript"></script>
```

In a process called the Rails Asset Pipeline (http://guides.rubyonrails.org/asset_pipeline.html), Rails combs through your scripts, concatenates them together, minifies them (it shortens variable names, removes newlines, and more to produce a mess like in http://code.jquery.com/jquery-1.9.1.min.js), and outputs them into a single file like application-**6cb7fbef54b7407f120c1360a22a4bfd**.js.

You'll notice a long hash at the end of the filename - this is meant to guarantee that the filename produced is different. As a result, your browser won't mistake the file for an older one and a fresh, non-cached version of the JS will be downloaded.

Long story short, *to work with JS in rails rename all your controller-specific JS files (e.g. visits.js.coffee, shopper.js.coffee inside app/assets/javascripts) to their pure JS counterparts (e.g. visits.js). Put JS in those files, and your application.js file will pick them up for you and you'll get them on every page.*

Topic 7: Beyond jQuery, and into “web apps”

You’ve seen how jQuery can be very useful for certain elements of web-development (AJAX, DOM manipulations, event handling, and so on), but there are other elements remaining that other libraries are handy for:

jQuery UI (http://jqueryui.com/) is an accompanying library that you’ll see a lot as well. There are nice widgets like accordions, date pickers, tabs, sliders, spinners, and more. (As a word of caution, though, these widgets tend to come with a very distinct default style that’ll scream “jQuery UI” - you might notice yourself that you’ve seen them in quite a few places before.)

handlebars (http://handlebarsjs.com/) makes templating very easy. When you have more sophisticated web sites, you’ll find yourself making AJAX calls to retrieve data and then formatting that data into HTML snippets to be inserted into the DOM. Handlebars uses “templates” to let you move from

```
// Assuming you have some response, resp = {author: “Josh”, body: “A blog post!”}
var snippet = ‘<div class=”post”><span class=”author”>’ + resp.author + ‘</span><div
class=”content”>’ + resp.content + ‘</div></div>’
```

to something cleaner:

```
<script id="entry-template" type="text/x-handlebars-template">
    <div class=”post”>
        <span class=”author”>{{ author }}</span>
        <div class=”content”>{{ content }}</div>
    </div>
</script>
…
var source   = $("#entry-template").html(),
    template = Handlebars.compile(source);
// Assuming you have some response, resp = {author: “Josh”, body: “A blog post!”}
var html = template(resp);
```

Into Web Applications, or “web apps”

jQuery is very close to the DOM. You won’t get utilities for anything at higher levels of programming, and you’ll find there’s more to JS than just the DOM!

When your application gets more complicated, you’ll find it developing more and more “state” per session. (Think Google Docs, Facebook, Gmail for example.) Instead of rendering everything server-side, you’ll start asynchronously sending updates, handling local interactions, and evolving into less of a “site” and more of an “application” or “web app.”

This is where libraries like **backbone.js** (http://backbonejs.org/#introduction) and **ember.js** (http://emberjs.com/about/) come in. The big ideas they introduce are Models, Views, and

Controllers, which you'll find familiar from working with Rails. They essentially offer utilities to connect your data with your DOM in a more explicit, thought-out manner. It has some overlap with jQuery, especially when it comes to events, but it also offers *models* (not dissimilar to Rails), a way to bind these models to events, and some convenient library methods to work with your data.