GYMNASIUM

JQUERY: BUILDING BLOCKS

Lesson 6 Handout

External Dynamism & Epilogue

ABOUT THIS HANDOUT

This handout includes the following:

- A list of the core concepts covered in this lesson.
- The assignment(s) for this lesson.
- A list of readings and resources for this lesson including books, articles and websites mentioned in the videos by the instructor, plus bonus readings and resources hand-picked by the instructor.
- A transcript of the lecture videos for this lesson

CORE CONCEPTS

In this lesson, you will learn how to work with dynamic page content.

1. “AJAX” is the outdated buzzword for what is now the universal process of web pages reaching out to servers after load, and fetching and using additional data. Facebook and Google Maps are canonical examples of web pages which make very heavy use of AJAX to load news and chats, or map tiles and search results. Even mostly static pages like Yahoo! or the New York Times have a few dynamic widgets. You will never in your life need to know what “AJAX” stands for.
2. Data exchange works best when it’s structured, and the state of the art data structure format is called JSON, which stands for JavaScript Object Notation, and looks very much like the JavaScript objects you’re familiar with. This makes it very easy to use in JavaScript, and has largely replaced the more verbose XML as the Internet’s data format of choice.
3. JSON is made up of objects and arrays full of values. Those values may be other objects and arrays, strings, numbers, true, false and null. Object keys must be in quotation marks, like this: `{ "firstKey": "value", "secondKey": 42 }`
4. Once we have our distilled data in hand, we have to use it to dynamically generate HTML for the user to see, via a process called “templating”. Templating is the name for any system that takes partial HTML strings (called templates) and adds data into them.
5. There are many templating systems, including some jQuery plugins. If you have complex needs, I recommend checking out Mustaches. If your needs are simple, you can roll your own rudimentary system very easily using string replacement:

   ```
   var myTemplate = '<div>%{greeting}!</div>',
         myHtml = myTemplate.replace('%{greeting}', 'Hello');
   ```

   Note however that JavaScript’s “replace” method only replaces the first instance it finds, so beware if you’re trying to drop a piece of data into multiple places in a string.
6. JSON is what our data looks like, and templating is how we get it into the HTML so the user can see it; now we need to reach out and fetch it from the server. AJAX is powered by “XHR”, which stands for “XML-HTTP Request”, even though you’ll rarely use it with XML. WIth XHR objects, you can request

data from a server, and use it when it returns. Working with XHR directly is fairly complex, and is a notorious hive of cross-browser compatibility, but jQuery takes care of all of that for us with the `$.ajax() method.`

7. The `$.ajax()` method takes as an argument a single hash of options. See the Further Reading sections for a link to the full documentation. Here are the most important ones.
   a. Data on the web lives at specific addresses, just like web pages, images and stylesheets.You specify the data's address with the "url" option.
   b. Requesting data is no use if you don't do something with it when it comes back; you can specify a callback function for this via the "success" option. (This only gets called if the server successfully responds to your request; you can also specify an "error" callback for when the server fails to respond, and a "complete" callback which is called whether the request succeeds or fails. For advanced needs, note for your Googling pleasure that the `$.ajax()` method returns a "jqXHR" object, which conforms to the "Promise" interface.)
   c. You may optionally specify a "dataType" option, describing what format of data you expect to receive. By default, jQuery does an excellent job guessing; you can speed things up slightly by specifying "json" or (less often) "xml". Another option is "jsonp", which we will cover below.
8. So to make a dynamic web page, you use `$.ajax(options)` to request data from a server at a particular URL; you specify a callback to be run when the data returns; your day will likely come down in the JSON format; and you will use some form of templating to turn the raw data into visible HTML.
9. A "web API" is any set of URLs which systematically expose data. For example, in the process described above, your server is exposing your data via an API. There also exist public web APIs for many different kinds of data. For example, openweathermap.org and forecast.io are two weather APIs which are free for a small number of requests. (You must sign up with them for a free "API key", which allows your site to identify itself.) The World Bank exposes a comprehensive set of global demographic data via its API (see "The World Bank API Call" below), which does not require an API key, and is great for experimenting with even if you don't care about percentage access to clean water by country.
10. To access data from other servers like these, your `$.ajax()` call must specify a dataType of "jsonp". This safely bypasses "same-origin" restrictions (a frustrating but absolutely essential internet security precaution). APIs which do not support JSONP are not generally accessible from other websites.

ASSIGNMENTS

1. Quiz
2. Remove our last piece of hard-coding: get the category to fetch and display from the URL.

RESOURCES

- Full `$.ajax()` documentation: http://api.jquery.com/jQuery.ajax/
- Experiment with dynamic data via the World Bank API: http://data.worldbank.org/node/9
- The World Bank API Call:

```
$.ajax({
url: "http://api.worldbank.org/countries/indicators/1.1_ACCESS.ELECTRICITY.
TOT",
data: { per_page: 100, date: "2000:2013" },
success: function(data) { console.log(data); /* Play here! */ },
dataType: 'jsonp',
jsonp: 'prefix' // (WB API uses a non-standard callback name)
});
```

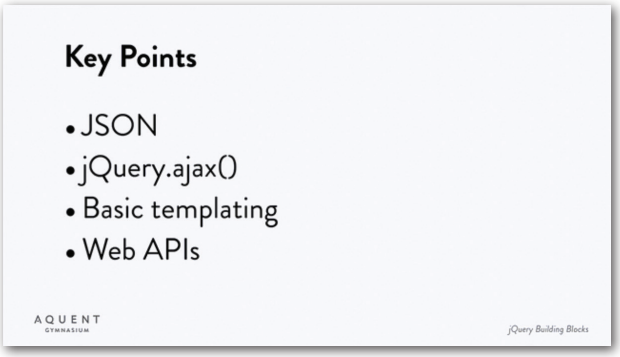
INTRODUCTION

(Note: This is an edited transcript of the jQuery: Building Blocks lecture videos. Some students work better with written material than by watching videos alone, so we're offering this to you as an optional, helpful resource. Some elements of the instruction, like live coding, can't be recreated in a document like this one.)

Hello. And welcome to an Aquent Gymnasium production of jQuery Building Blocks, 5 Ways to Cut Your Web Development Time in Half. I'm Dave Porter.

This is Lesson 6, External Dynamism, also known as the last lesson. We're going to finish this class out by extending the dynamic tool kit we built last lesson. But instead of reacting locally to the user's actions, we're going to reach out to the network, whether it's to our own servers with our own data, or to a public server with information like the weather or stocks or demographic statistics about access to clean drinking water, and update the page based on that.

Today's lesson begins with Chapter 1, Ajax. In Chapter 2, I will teach you the three techniques that you need to make the changes we're going to make today. In Chapter 3, I'll show you some things that are available out on the Internet in order to make your page more interesting, with data from farther afield. And then in Chapter 4, we're going to take a look back at this entire course, where we've come from and how far we've come.



Today's key points include JSON, which is a data exchange format. We're going to cover jQuery.ajax, which is the method that's actually used to go and get data. We're going to cover basic templating, which you need in order to turn data into HTML. And at the end, we're going to take a look at some web APIs. So we're going to look at what they are and what's available out there.

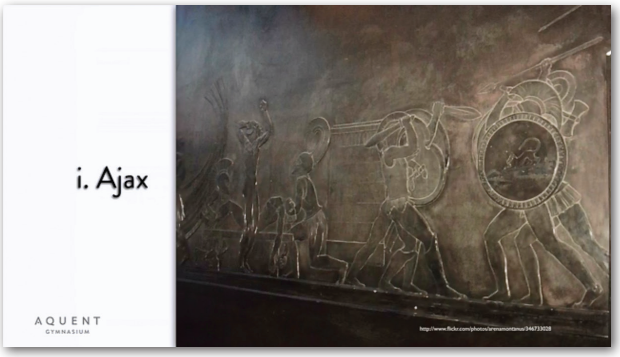
Don't forget, I am not live. I am a video. So please feel free to use the Pause button, jump around, catch up, go get a drink, whatever you need to do in order to stay current with the code.

One other brief note. Before we get started, because of some peculiarities with how the Chrome browser enforces cross-domain security—which is a term we'll come back to later in the lesson—with how it enforces cross-domain security on local files on your hard drive, we're actually going to be developing in Safari today. You can also use Firefox.

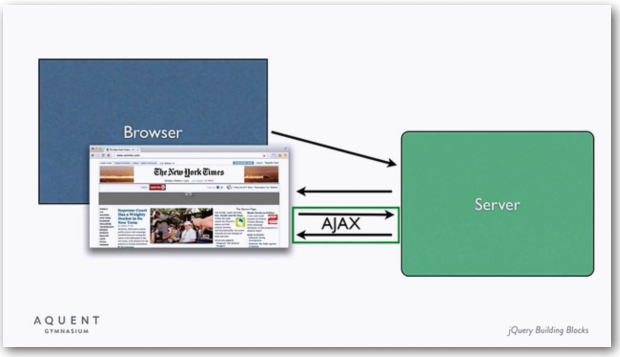
If you are new to developing in Safari, here is how to get access to the developer tools we'll be using. From Safari, pop open the Preferences. If you're on a Mac, that's command, comma. Then switch over to the Advanced tab. Make sure the Show Developer menu in the Toolbar option is checked. Once it's checked, you can click Command-I on the keyboard, or Control-I if you're in Windows, and open up the Dev tools down here. Click the little speech bubble over on the left here to get to the console itself. And now we are up to speed on using this in Safari.

AJAX

Here's some "Iliad" imagery for folks that know their Homeric epics. Ajax stands for Asynchronous JavaScript and XML. And you do not ever need to know that. Ajax is sort of last decade's buzz-word for something that's now very common. It's a website requesting, and then processing and using, information from the web in JavaScript after the page has initially loaded.

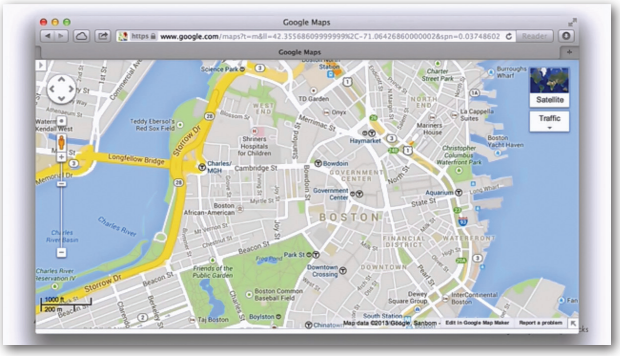


So with the standard website, and with the pages we've built so far, the user sort of enters a web address in the browser, which makes a request to the server. The server then takes this request, fetches—or creates—whatever the HTML page is going to look like, and returns it to the browser, which then displays it. But what if the web page itself wants to update? It wants to pull down a little bit of extra information, maybe on the fly, maybe just so it doesn't have to sort of do that at the beginning, and change some little part of itself.



Well that's where Ajax comes in. The web page can make the same kind of request to the server, but instead of requesting a full web page, it requests and receives just a little bit of data, and then updates itself. So this is Ajax. And that's what we're tackling today.

Now on a simple, static, hyperlink-filled web page, you might have a little dynamic stuff going on. You've got a slide show or something. The page may tweak itself a little bit as the user does stuff, per our last lesson. But the data that's on the page when it loads, is basically all the data that the page has to work with. Right? That's basically all that it's got.

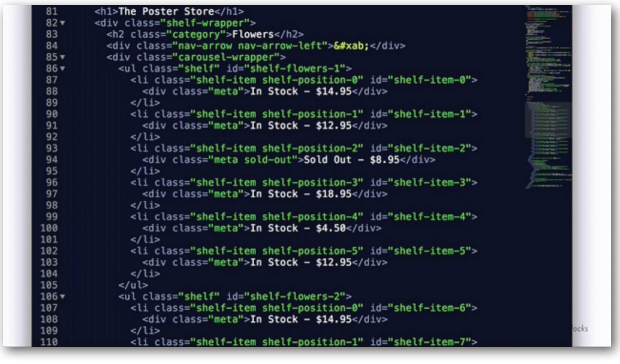


Now on the other end of this sort of Ajax-y spectrum, is a fully dynamic web page like Google Maps, which is a really fantastic and groundbreaking example of this process. It launched back in 2005 as one of the original dynamic web pages. And it remains an excellent example of Ajax in action. You know, as you drag the map around, the page loads tiles dynamically from Google's server. As you type into the search bar, it uses Ajax to reach out to the server and get a list of things that you might mean. And then when you hit Enter, instead of reloading the whole page, it reaches out to the server for results and displays them by itself.

Another great example that you're probably familiar with is Facebook. Its website is very, very heavily into Ajax, from the live updating of your news feed to chats. Every dynamic thing here is powered by Ajax or some more modern version. Now, Facebook looks and feels like a web page. Right? But it's probably one of the biggest dynamic web apps out there.

You don't even have to be behemoth to use a little Ajax. If you've got a news site that takes a decent amount of server power to put together, you're not going to want to have to reload the whole thing if the weather changes. Right? Or if the user enters a new location for weather. And even our initial, fairly static, web page is likely to have a few little islands of Ajax- powered externally facing dynamism built in. Right? Ajax is just everywhere today.

Now, I'm going to show you in this lesson how to use this very powerful process of web pages updating themselves from the server. As you've seen, even simple web pages are using it to improve their user experience. Specifically in our case, we're going to completely rewire our poster carousel. It's going to end up looking the same, but it's going to become much, much easier to update.



```
<h1>The Poster Store</h1>
<div class="shelf-wrapper">
  <h2 class="category">Flowers</h2>
  <div class="nav-arrow nav-arrow-left">&#xab;</div>
  <div class="carousel-wrapper">
    <ul class="shelf" id="shelf-flowers-1">
      <li class="shelf-item shelf-position-0" id="shelf-item-0">
        <div class="meta">In Stock - $14.95</div>
      </li>
      <li class="shelf-item shelf-position-1" id="shelf-item-1">
        <div class="meta">In Stock - $12.95</div>
      </li>
      <li class="shelf-item shelf-position-2" id="shelf-item-2">
        <div class="meta sold-out">Sold Out - $8.95</div>
      </li>
      <li class="shelf-item shelf-position-3" id="shelf-item-3">
        <div class="meta">In Stock - $18.95</div>
      </li>
      <li class="shelf-item shelf-position-4" id="shelf-item-4">
        <div class="meta">In Stock - $4.50</div>
      </li>
      <li class="shelf-item shelf-position-5" id="shelf-item-5">
        <div class="meta">In Stock - $12.95</div>
      </li>
    </ul>
    <ul class="shelf" id="shelf-flowers-2">
      <li class="shelf-item shelf-position-0" id="shelf-item-6">
        <div class="meta">In Stock - $14.95</div>
      </li>
      <li class="shelf-item shelf-position-1" id="shelf-item-7">
```

As you may recall, the contents of the poster carousel are currently hard-coded into the HTML here. And worse, the CSS. Now that's really bad news. Right? Because making changes to the list of available products—presumably that's a regular thing that happens—is going to require going into the page and editing it in the HTML and CSS. You know, group the six li's into each ul, making some very tedious work for somebody, or for some very fragile automation.

Now sometimes this kind of stuff is done with some jenky PHP on the server. So the page arrives in the client looking like this already. And that's fine. But you know, that's going to make for a worse user experience, a slower user experience, if there's any chance whatsoever

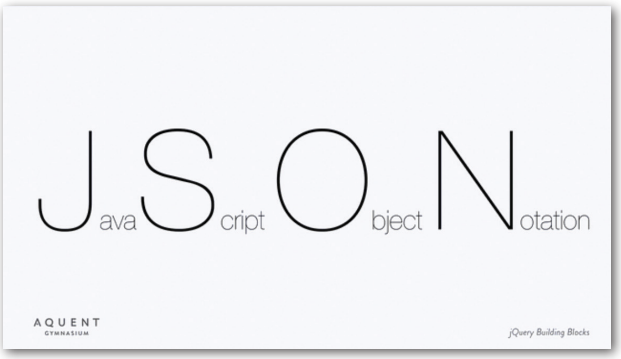
that the user might do something that would be handled faster and more smoothly through Ajax. Plus, this is a jQuery class. So we're going to move that processing from the static file and from the server into the client.

This will allow us to update and swap image lists very easily. So easily, in fact, that switching between two lists of pictures like this is going to require a change to only one line of code. That's the promise of today's lesson. And if you do your homework, so to speak, it will require only a change to the URL itself.

THREE TECHNIQUES

There are three techniques that we need to learn in order to get this done. Here's the first new idea that we need, in order to make this transition from hard-coded to dynamic loading.

This acronym (JSON) is pronounced "Jaysonn," or "Jason." Your pick. And it's just a data format. It's pretty much replaced the older, more verbose format XML, which you may have heard of, which looks a lot like HTML. You're still going to run into lots of XML. It's all over the Internet. It's all over all kinds of legacy systems. And that's fine. But JSON is definitely the more popular, more modern, replacement.



And JSON stands for JavaScript Object Notation. And just like it sounds, it's a way to represent data in JavaScript's native format, which makes it extremely easy to translate from bytes over the wire into data that you can actually use. For example, here's some data about me in JSON. You've got my name. It's separated into a first and last name, just so that you can do different things with it as needed. You've got my age, status, all the stuff about me is all encoded in a way that makes it very easy to use.

Now, JSON is a little more restrictive than JavaScript's actual native objects. For example, note that all property keys are in double quotes here. Values are restricted to strings, numbers, objects, arrays, true, false, and null. Now, things like functions aren't allowed. Fancy data like dates have to be serialized, which is a fancy word for turning it into a string or a number, or an object, basically turning complex data into one of these formats, one or more of these formats, in order to go over the wire.

Our poster data, meanwhile, currently looks like this. This isn't the absolute worst that I've ever seen, in terms of mixing concerns. Right? In terms of data structure and presentational structure colliding. But it's not great. It's got CSS classes in here. It's pre-split into groups of six, which, that's a user interface decision. Right? That shouldn't be made in the data. And the stock and price data is kind of munged together.

And worst of all, the poster images themselves are hard-coded in the CSS. Now we've been running with this terrible setup since Lesson 3. But it's time to go back and settle old scores. So we're going to extract the pure data from the HTML and CSS. And we're going to encode it in JSON.

Now things in code are generally represented as objects with the curly brackets. So we're going to represent a single poster's data like this. Everything needs an ID, so we're going to give it an ID. The image URL, we're going to specify. Presumably in the real world, they wouldn't be named 1.jpeg and 2.jpeg. They'd be named some random things, because they'd be in random places.

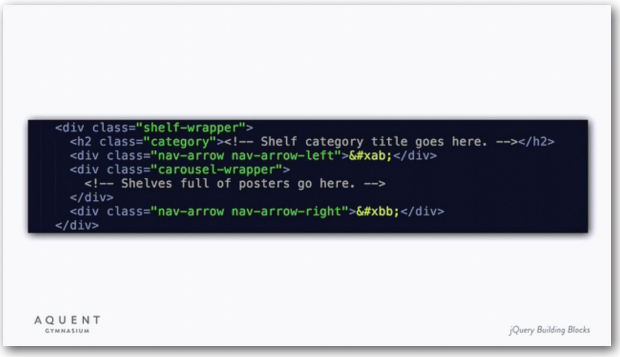
We're going to have our price in dollars. Notice that there may also be price in Canadian dollars, maybe price in yen. Maybe your website is a very international concern, and you need to do that, need to have several different prices in there. We're going to have a true/false value for whether it's currently in stock. The decision to call it "in stock" or "out of stock" or "available" or "not available," those are user interface decisions. The data is, do we have it in stock, true or false. We've got a category value there.

And then, just because we're good consumers of public domain stuff, we've got an attribution in there as well. Again, jump on Flickr, go tell these people how lovely their pictures are.

Now we're going to be pulling down a list of data. So it's going to be an array of those poster objects, just a simple, ordered list. And so it's going to end up looking like this. Now again, we're changing it from this, where we've got the position sort of hard-coded into the li classes. We've got the image URL, is actually over in the CSS. We've got the in-stock information, sold out, we've got our price. It's all in here. It's bad.

And again, here are the images themselves, way over in the CSS. Don't forget this terribleness. But now we've purified it down to this. It's much cleaner, which means it's going to be much easier to update without breaking things. And everyone's going to be thrilled about that.

So all of that data is out of our HTML. So here's what it looks like now. It's like a blank canvas waiting to be painted on. Now I mentioned before that we wanted to separate out the data structure from the presentational structure. And we've got our data, but where'd the presentation go? To get that back, we have to turn to our second technology, which isn't actually an acronym, but it does start with a letter. And it's called templating.



Now templating is the fancy name for any process that takes reusable bits of HTML with the data left out, and combines it with the data to spit out actual HTML elements for us to use. Right here is the HTML from one of our posters. The CSS for the background image style, I moved it into the style tag now, just to keep everything in one place. It's long. So let's just collapse that down and pretend that we see it.

Now again, this is mixing data with presentation. The data is here. We've got the poster's position on the shelf. We've got the poster image itself in the style tag. And then we've got the in stock information and the price in the little meta-element.

Now that's the stuff that's going to be different for each poster. So let's pull it out and replace it with some placeholders. This is now a reusable template. And we're going to store it as a string. And we can reuse it as many times as we need. So we've got our data in JSON. And we've got our display skeleton in a string template with some almost-HTML, some placeholders.

We put them together to get our final product, which is some functional HTML. Or, more to the point, a poster that the user can actually see and interact with. And we will walk through this process in the demonstration portion.

Let's take a simpler look, and we'll spit out a couple of results. Here is our greeting template. Let's say greeting, and then to whom the greeting is specified. And let's wrap it in a div. Let's say our greeting HTML is the template. Our greeting HTML, we're going to replace the greeting with Hello, let's say. And we're going to replace the subject with World.

Now we spit this out and we get a nice little div with Hello World in it. And we're going to take that same template, and we're going to replace our greeting with, Never let go. And we're going to say that to Jack. And we get a lovely little '90s retrospective greeting and subject, out of the same template.

Now the way we're loading our data into the template, it's extremely rudimentary. It's literally just like one-at-a-time string replacement. Right? Now there exist a number of very nice templating engines, including several that are jQuery plug-ins. So if you end up doing a lot of this stuff and it starts to get a little complicated,

I highly recommend going with one of the fully functional prefab options. They do a lot of extra stuff for you. They're great. I recommend taking a look at Mustaches, for example. But we're going to be doing it the easy way today.

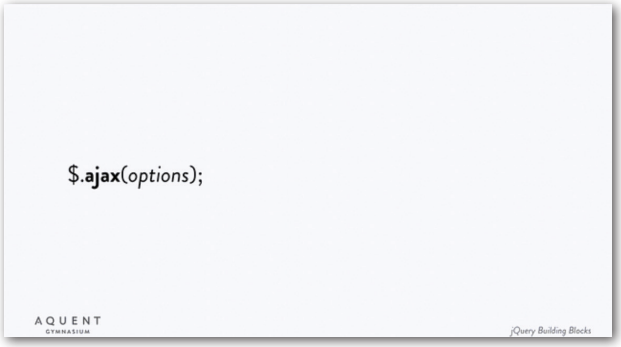
```
var greetingTemplate = '<div>%{greeting}, %{subject}!</div>',
    greetingHtml = greetingTemplate;
greetingHtml = greetingHtml.replace('%{greeting}', 'Hello');
greetingHtml = greetingHtml.replace('%{subject}', 'World');
> <div>Hello, World!</div>
greetingHtml = greetingTemplate;
greetingHtml = greetingHtml.replace('%{greeting}', 'Never let go');
greetingHtml = greetingHtml.replace('%{subject}', 'Jack');
> <div>Never let go, Jack!</div>
```

Now that we've got our data in JSON and our HTML in templates, how do we get the data? For that, we need our third and final technology of the day. And this is the one that powers all of Ajax, and it's called XHR, which stands for XML HTTP Request. This technology was inadvertently invented by Microsoft back in 1999 for use with the original web outlook. And it enabled the entire dynamic web revolution of the mid '00s. And by the way, even though XML features prominently in all of the technology's acronyms, you're probably not going to be using it very much. Again, JSON has pretty much taken over as the web's favorite data format.

Now, doing XHR is complicated and annoying, and exposes you to some pretty sticky cross-browser issues. So we will, of course, be doing it the jQuery way, which makes things easy and compatible all in one swoop. In most of the jQuery methods that we've been using, we've called like this, with a selector that's used to get some elements. And then we're going to do something to those elements.

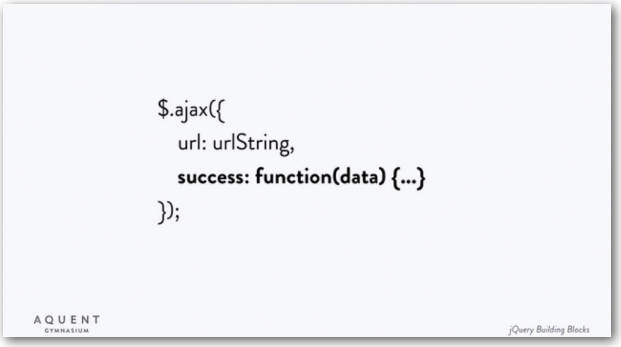
In the case of Ajax, though, we're not doing something to a list of elements. Right? We're just kind of doing it. So this method, along with several others, lived directly on the dollar sign, like this. The method that we're going to use to grab our data from the server is appropriately named Ajax. Now this is one of the last places around that you're going to still find this buzzword. Most other places, it's just called a normal thing that your website does.

Now the Ajax method takes one argument. It's a hash, just a regular object full of options. And there's a ton of options. The first one, and the only one that's sort of really required, is the URL of the data that you want to get.



Now on the web, data lives at unique addresses, no different than pages, images, CSS files, script files, whatever. You already know all the rules to use when specifying a whole or a partial URL. You can start with HTTP to specify a specific website. You can start it with a slash to specify the address relative to the root of the current website. And you start with no slash, right, in order to specify an address relative to that of the page that's requesting it. Think about it like, relative to a sub-folder, in that last case.

These are all the same rules that you're using when linking CSS files into your document, or setting up href-style hyperlinks. This is all the same stuff. Now you don't want to just request the data. You also presumably want to do something with it once it arrives. So the next option is a callback function, which jQuery is going to call whenever the data returns back from the server.

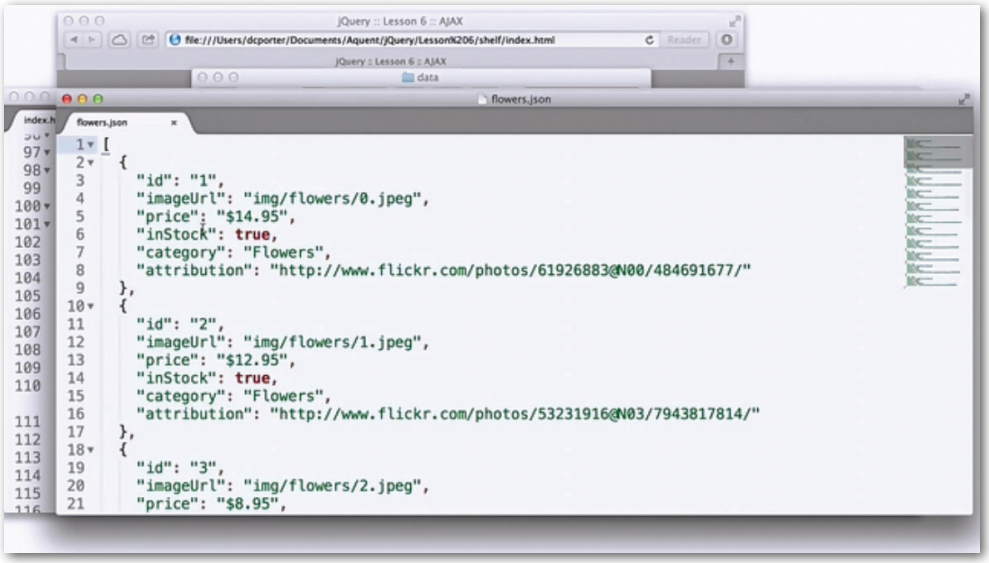


Now the last argument that we're going to use is data type. If we specify what kind of data we're expecting to be retrieved from this call, jQuery's actually going to have a go at parsing the response for us. This kind of data, and most data, comes down over the wire, basically, as text. So by telling jQuery that we expect JSON, it'll take that text, and it's actually going to parse it into a JavaScript object or array or whatever, for us to deal with directly. It's super-convenient.

Other options for this value include XML, which you're going to run into sometimes. But again, for your own convenience, you should generally avoid it if you have any control over the situation. HTML, which can be useful in some advanced line-blurring client server configurations, but you won't usually use. And JSONP, which we will get to later in the lesson.

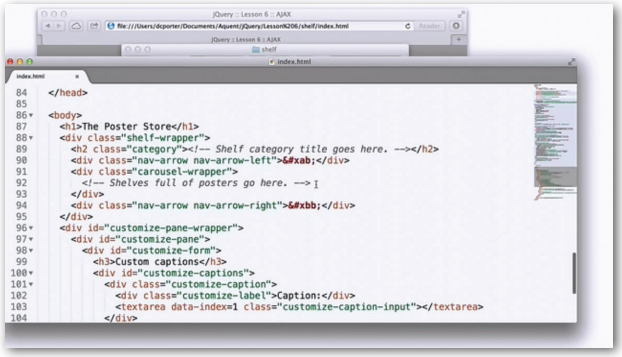
DATA FROM FARTHER AFIELD

So please pop open your lesson materials and go to the Shelf folder. Now inside this folder, we've got a new folder called Data. Let's take a look in there. We've got these two files, Flowers.json and Space.json. Pop open Flowers, and this is where the JSON data that we discussed is living.

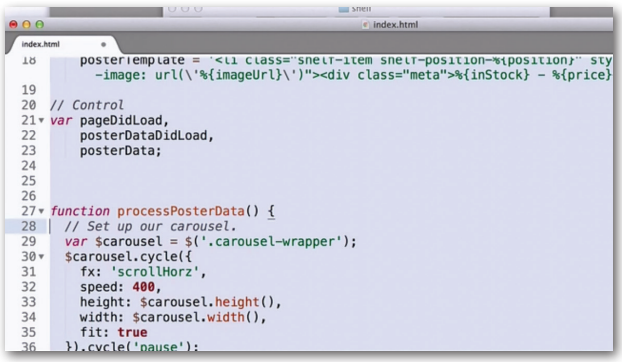


In the real world, these files would probably be little server-kind of scripts. They'd be little PHP or something, server-side scripts, which would connect to a database. They'd get the poster data. They kind of munge it together into JSON data and then they'd send it out over the wire. But doing anything at all on the server side is way out of scope for this lesson.

So we're going to fake it. We're going to have our server be some static data files that are just hanging out here. Now pop open Index in your IDE of choice. And let's take a look through our code. Here we've got the spot where the shelves are going to go. And we've got the spot where the name of the category is going to go, so the title of the shelf.



Now let's scroll back up to where our code is going to go. Here we go. Now, I've already cleared that stuff out, and I've turned it into templates. So these are going to be our templates. You'll notice they've got a bunch of placeholders in there for all the different poster information that we're going to add.



Now we've got an interesting challenge here, because currently we are set to do all of our processing when the page is loaded. But the page loading is no longer the only important thing. We've now got a page loading. And then, at some point, which may or may not be related to when the page loads, we've got some data loading as well. And we need to put off processing everything until both events have happened, until the data loads and the page loads.

This is an example of a problem called “control flow management,” and it’s something you’re going to run into as you work with more and more complicated things. For now, we’re going to take a very simple approach to control flow management. We’re just going to create two flags, one to note when the page has loaded, and one to note when the data has loaded. And we’re also going to have to store our poster data.

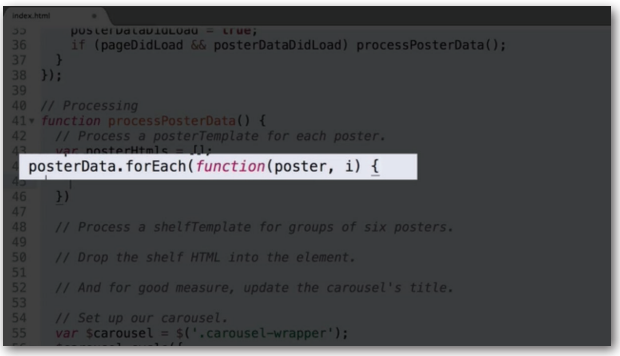
Let’s go ahead and rename this function processPosterData. That’s going to fire once both the page has loaded and the poster data has loaded. In our Document Ready method, we’re going to make this a very simple thing. Now all it’s going to do is, it’s going to say, OK, now the page has loaded. So we’re going to go ahead and mark pageDidLoad as true. Then we’re going to check and see if, at this point, both things have happened. So we want to see if pageDidLoad and posterDataDidLoad, if those have both already happened, now we can go ahead and process that poster data. Great.

And, of course, we’re also going to need to actually load our data. So let’s trigger that with our friend, the Ajax call. Pass in some options here, we’re going to pass in the URL. Now, that’s just going to be the static file in the data folder. Let’s specify a data type here so that jQuery automatically parses that for us. And let’s go ahead and create our success callback.

Now, the success callback is also going to be very simple. All it’s going to do is to save the results. And it’s going to mark the poster data as loaded. It’s going to say, yes, this has already happened. Then in our control flow process, we are going to use the exact same check. We’re going to see if both things have happened. If they have, we’re going to go ahead and process that poster data.

So what you’re going to end up seeing is one of these things is going to happen first, the page loading and the poster data loading. It doesn’t matter which one. One of them is going to happen first. It’s going to notice that the other one hasn’t happened. And it’s going to then just continue waiting. Then the other one will happen. Again, it doesn’t matter which one. And it’ll go ahead and, at that point, it will know that both things have happened. And it will go ahead and move on with its day.

Now that we’re actually sure that both of our things have happened, let’s go ahead and process that data. First thing we’re going to do is we want to process one poster template for each of the poster items that came down. After we’ve got those, we need to process a shelf template for each group of six posters. And that’s going to require a little bit of annoying math. There’s our poster template. There’s our shelf template. Then after we’ve got that, by then we’ll have all of the HTML that we need. So we’ll just drop it into the shelf element that already exists. And then for good measure, we’re going to label the shelf correctly.



So we’re going to need a place to store each generated poster item as we go. So we create an array for that. Now we’re going to do a forEach loop through each of the poster data items. So it’s going to be an array of objects. So

we're going to get the poster data down. And I'm going to take that second argument there as well, which we haven't dealt with before. ForEach gives you both the item, and it gives you the number, the which item you're currently looking at. And we're going to need that to correctly position our elements. Don't forget your semicolons.

Now in here, we are going to generate poster HTML. Again, one for each poster. So let's grab the template. Now that we've got the template, we want to start filling it out. So let's look through the template. Here's the first thing. It's the position. Now, the position, if you recall from last time, is going to be 0 through 5. Right? We've got the 5 spots in the poster. And I'll show you some fun math to get at 0 through 5, based on just an arbitrary number.

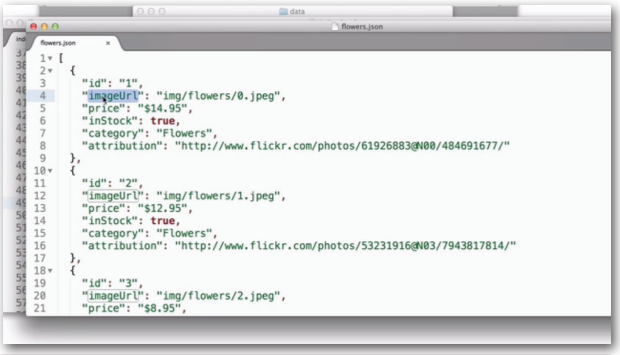
So per templating, we're going to replace this placeholder with the number. And then we're going to do something called modulo. That's i, and that's the percent sign. So that's i modulo 6. And what that does, is it divides the number by 6 and then gives you the remainder. So that's position taken care of. Now what's the next one we've got?

index.html

```
30 $.ajax({
31   url: 'data/flowers.json',
32   dataType: 'json',
33   success: function(result) {
34     posterData = result;
35     posterDataDidLoad = true;
36     if (pageDidLoad && posterDataDidLoad) processPosterData();
37   }
38 });
39
40 // Processing
41 function processPosterData() {
42   // Process a posterTemplate for each poster.
43   var posterHtmls = [];
44   posterData.forEach(function(poster, i) {
45     var posterHtml = posterTemplate;
46     // Position (0 - 5)
47     posterHtml = posterHtml.replace('%{position}', i % 6);
48
49   });
```

Looks like our style is the background image. Now note here that if we want to have single quotes within our single quoted string, we have to escape them with a backslash. So let's grab that, head on down here, and we will put the image URL in.

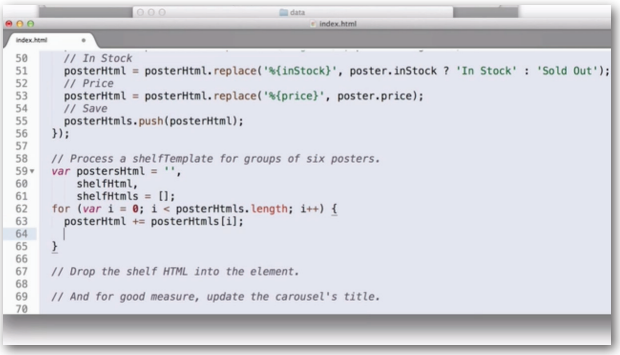
And we'll replace that placeholder with posters.imageUrl, I happened to remember from the data. But let's go ahead and take a look, you may want to keep this open in the background. So here's all of the data that we're going to be hitting. This is going to be, so poster.imageUrl. It's going to be poster.price, poster.instock, et cetera.



Now let's take a look at In Stock next. That's the next one. There it is. Now, In Stock is going to be interesting, because In Stock comes down as a true/false value. But we don't want it to say True and False. So we're going to do a nifty little bit of JavaScript, which is called In-Line Conditionals. And what you do is, is this first thing true. Yes? Give me In

Stock. No? Give me Sold Out. That's just the thing, question mark, if true, colon, if false.

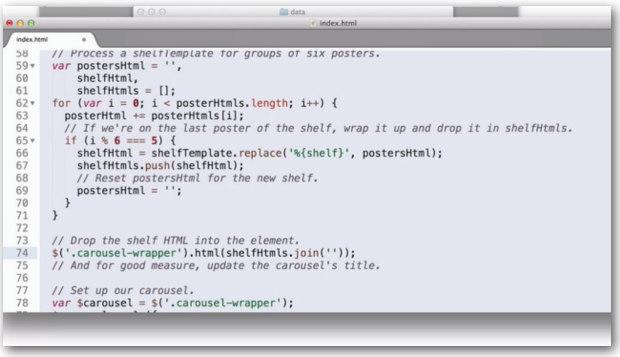
Now let's do the price, that's coming up next. That's the, I believe, the last one in our template that we need to replace. And we'll replace it with poster.price. Great. Now that we've got our poster template fully filled out, let's go ahead and push it onto the poster HTMLs array, for future use.



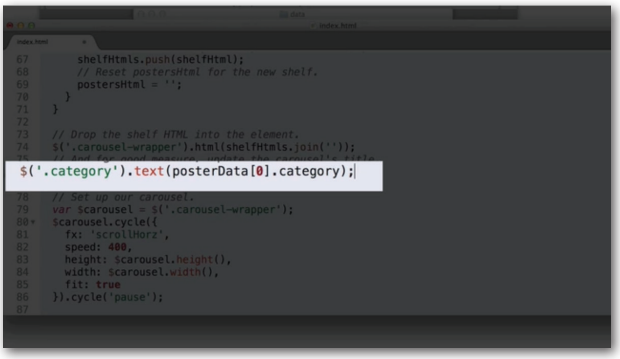
And now that we've got that together, we've got a list of completed templates, one for each poster. Now we need to actually cycle through, and for each group of six posters, we're going to create one shelf. So let's get a posters string. We're going to do this iteration slightly differently, for reasons that I found convenient while running. So we're going to do a traditional JavaScript loop here, presuming that you're familiar with them.

So for each item, each poster, filled-out poster template, that's poster HTMLs, we are going to tack them together one at a time. That's very simple. And that's all we're doing most of the time. But if we're on the last poster of the shelf, which means if i modulo 6 equals 5, which is the last number before you kind of loop back around to a sixth item, then we're going to do some extra processing.

We're going to grab a shelf template. We're going to replace the shelf placeholder with our list of posters. And then we're going to push that shelf item onto our array of shelves. And then we need, of course, to reset posters HTML to sort of loop back around to the new shelf. Each shelf needs to start from scratch, of course. It looks good.

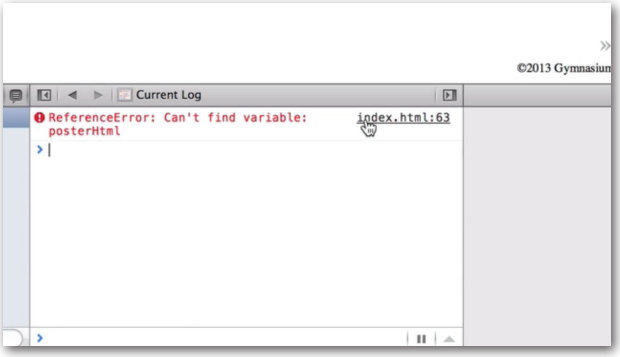


Now let's go ahead and, by this point, we've got the finished product. Right? We've got an array of shelves. And we can jam those together very easily. So we're going to drop that into our carousel. So let's grab that class, the carousel wrapper class. This code assumes that we've only got the one element in the page. If you're going to go ahead and get fancier later on, you'll have to tweak this a little bit. So let's take our shelves, and join them with an empty string. Otherwise, it joins them with a comma, and that's ugly and annoying. And finally, let's go ahead and put the category title into place. And this is just text. So I just used text instead of that HTML.



Now this is a little bit of cheating here. I'm assuming—correctly, but it's still an assumption—I'm assuming that every poster in my list of poster data has the same category, which means I can assume that the first poster is representative of all the other ones. So I'm just going to grab that first poster and pull its category off.

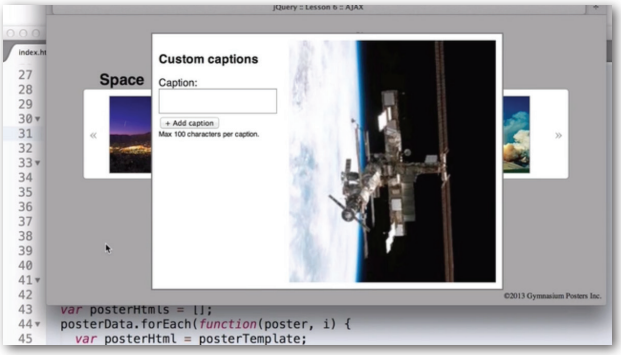
Now that we've got that, we take a look at the rest of our code. This is all from last time. Right? This is from earlier. Here's our cycle plug-in from the plug-ins lesson. We animate everything in. We hook up our navigation buttons. And here's all the stuff from last lesson, with the new caption, the pop-up. That stuff all still works. And now we need to get rid of this close parenthesis that's left over from earlier code. We failed at that earlier. My apologies.



And once we've got all that, we give it a Save, take one last look, and give it a Refresh. Now this should—ah, this didn't work. Now, using command Alt-I, we can pop open the Inspector. And then if we click on this little bubble over here, that'll take us over to the console.

And it's got an error for us. It says, Reference Error, can't find variable, poster HTML. And it gives us a great place to just click and look at that code. So let's do that. Poster HTML, plus, equal, poster—All right. So it looks like what I did there is, that's just a simple typo. This poster HTML doesn't exist, which is a good thing, because it threw an error for us. Posters HTML is what we wanted to do. Posters HTML is the name of the variable that we're using to keep all of our posters together.

And now we give it a Refresh. And it works perfectly. Wonderful. Everything still works here. Pop open the pane from last lesson. That guy still works. That's great. Now here comes the prestige. The promise of this lesson was that we would get to a point where switching between the list of flowers and some other pictures—the space pictures that I put together—would be as simple as editing one line of code. As you were going through, you probably noticed which line of code that was. And that's what data we load. The only difference is what data we load.

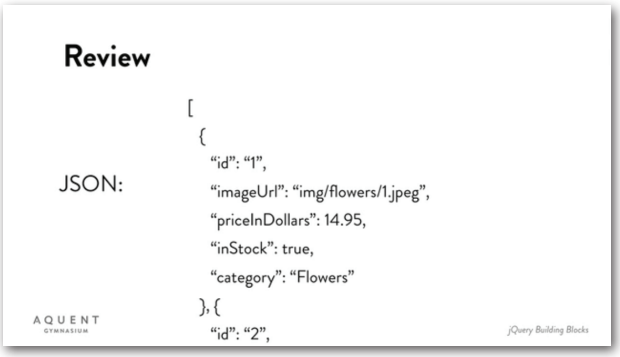


So let's go ahead and swap that out for space. And give it a Reload. And there's the prestige. Pop it open and we've got these lovely shots of the space station, of all these wonderful things. But the upshot is that we've got a different set of data just by changing which data is loaded.

And that is as simple as that.

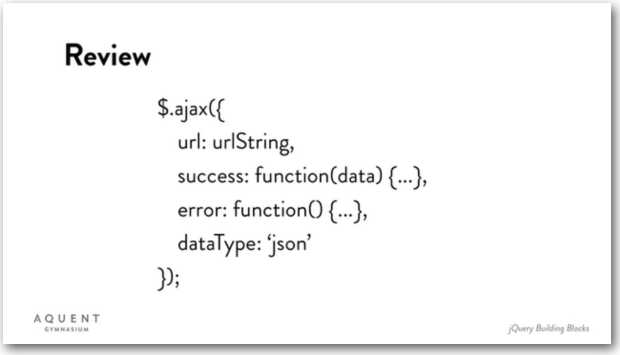
REVIEW

So, in review, we used three new technologies today. We learned about the JSON data format, which has largely replaced XML as the web's data language of choice. JSON stands for JavaScript Object Notation, and it looks like this. Note that property keys like ID and image URL are in quotes. That's important. Values are restricted to strings, numbers, objects, arrays, Boolean values like true and false, and null to represent emptiness.



Now you can't send functions, obviously. And any fancy data like dates, you need to serialize into one of these data formats in order to transmit it over the wire.

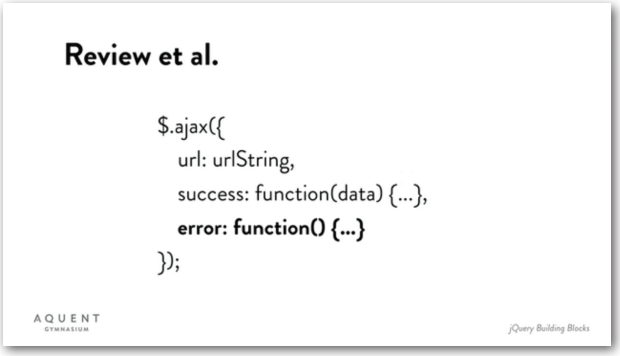
We did a real rudimentary pass at templating. We used this technology to combine HTML templates with data from our JSON to spit out HTML that the user will actually see.



And finally, we learned how to use jQuery's Ajax method to actually fetch the JSON from our little fake server. We covered the Ajax method's options argument, including options like URL and success. I also mentioned error, which allows you to handle it if something goes wrong, and the data type as well.

Another option for the Ajax call that I want to mention quickly is data. We faked our dynamic data servers today so this wouldn't apply to our current situation. But usually, your request is going to have some sort of parameters in it. Right? Like this one, requesting the current weather from the lovely city of Boston. You can certainly do that in the URL, which requires you to either hard-code it or dynamically build the URL string yourself.

String building, by the way, sounds really easy, but it's actually a notoriously fragile process. So instead, you can pass the data in as a data object, with the data right in it, like, city, colon, Boston, and let jQuery do all the string building for you, which you want. It's definitely a good idea if your URL has any arguments on it like this.

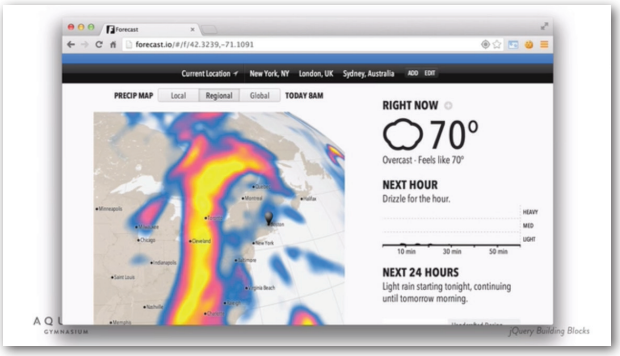


And just one more option that I want to mention, along with the success option: you can pass in a function to handle an error as well, if, for example, the file isn't found, or the user has turned off their Wi-Fi, or whatever. There are arguments you can use in there to check the request status and react to it, depending on what went wrong. So there's a lot to learn in there. There's also like a dozen other arguments you can pass into this very flexible method into Ajax, and we are certainly not going to cover all of them. But I've include a link to the full documentation in the reading materials.

This dynamic network interaction concept is super powerful, and nearly every website these days uses it in some way. Most often it is to fetch some of the site's own data in a dynamic way. But you can use it to get data from farther afield, too. Lots of web companies, from weather to stock to demographics companies, trade in that data. And they've opened that data for you to use. Now, getting and using that information is beyond the scope of this course. But I want to sort of send you off into the future with a real vibrant sense of all the different things that are possible.

So this chapter's acronym is API, which stands for Application Programming Interface. Again, not important thing for you to know. You can think of a web API as an interface for requesting data from a server. In fact, you can think of it as an Ajax-y way of getting JSON from a foreign server, i.e., what we've been doing this whole lesson.

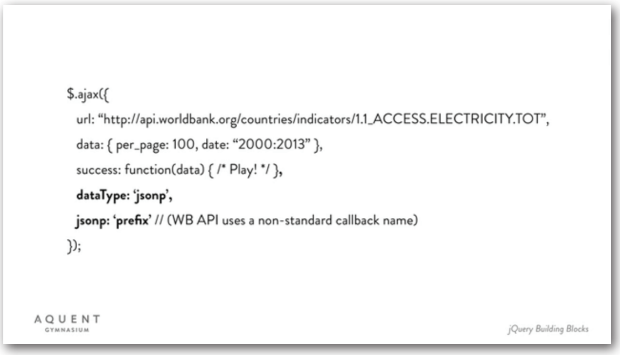
Now our little fake data files are actually fake little web APIs. It's not too different from how you'll handle information fetched from the great beyond. If a service has a web API, then you can get its data and use it. For example, if you want to create a weather widget like the one I highlighted on yahoo.com, you might check out openweathermap.org or forcast.io. Those are two weather data companies that have free tiers to their APIs.



Now these two services do require your web page to identify itself. You need to register with them for a unique API key, it's called, so that they can know who you are, and know that you're not taking advantage of the free tier. But you know, they're easy to sign up for. They're free to play around with. If you get to the point where your great idea is making more than 1,000 requests per day, hopefully you've got a plan for monetizing it as well. But while you're developing, while you're playing around, it's completely free.

Another great resource to play with is the World Bank API. And this doesn't even require an API key. They want you to have their data. So if your website happens to be focused on percentage of population with access to electricity by country, then the World Bank has got everything you need. But even if that's not what your website is about, it's still a great service to learn from. So if you're curious and would like to play around, I've put some code in the lesson materials that you can copy and paste into your app, or into the console and play around with.

Again, this is useful for learning about Ajax out on the web, whether or not you care about per capita electricity access, or whatever.



Now note the use of JSONP, rather than JSON, as the data type. Sufficient to say for now that JSONP lets you cooperate with the server to bypass those same origin security measures that we talked about at the beginning, which are the reasons that we're using Safari instead of Chrome today. And so on.

There are loads of APIs available on the Internet. And if you've got an idea for a way that your website can use it, you can probably find a way to get at it. Again, a full review of those technologies is beyond the scope of this course. But we have a forum. And so I hope you'll jump on the forum and share whatever awesome stuff you come up with.

RETROSPECTIVE

This is our last chapter together. So let's take a look back.

In Lesson 1, one we took a look at all the things that jQuery can do for you, summarized by the immoral aphorism, "Write less, do more."

In Lesson 2, we discovered the life light of the web, animation, which when used with appropriate restraint brings delight to user interface and can afford manipulable elements with a sense of manipulability—manipulability, that's very difficult to say.

In Lesson 3, we took a look at jQuery's enormous ecosystem of plug-ins—some really amazing, some really silly—in the hopes that someone had already done most of our work for us. And we were in luck. We discovered a great carousel plug-in which got us what we needed with minimal effort.

In Lesson 4, we built tabs on a deadline. And then, still on the deadline, we wrapped our code up in a plug-in to help and impress our co-workers. And then the last lesson in this lesson, we filled out your jQuery toolbox by focusing on ways to react dynamically to a user's actions, and even to reach out to the Internet and react to dynamically loaded information.

And that's all she wrote. For your first assignment, turn to the lesson materials, where we have linked a short quiz for you. And for your second assignment, we're going to turn the one last thing that we're still hard-coding on our web page, on our poster page, which is the location of the data file. But I'd like you to move that into the URL. So however you decide you want to structure your URL is up to you.

Please modify the page so that instead of hard-coding it, it gets what type of data is being loaded from the URL. You should include some error handling. If the nonexistent data file is requested, it should say something to the user, whether it's something subtle or something. Depends on the brand of your poster store, which is up to you. So be creative.

If you run into any issues, or just want to say hi, please do hop on the forum. This has been Lesson 6, External Dynamism. I'm Dave Porter Aquent Gymnasium.