GYMNASIUM

JAVASCRIPT FOUNDATIONS

Lesson 5 Transcript

Server Communication

ABOUT THIS DOCUMENT

This handout is an edited transcript of the JavaScript Foundations lecture videos. There's nothing in this handout that isn't also in the videos, and vice versa. Some students work better with written material than by watching videos alone, so we're offering this handout 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. We encourage you to use this handout alongside the videos, rather than as a replacement of them.

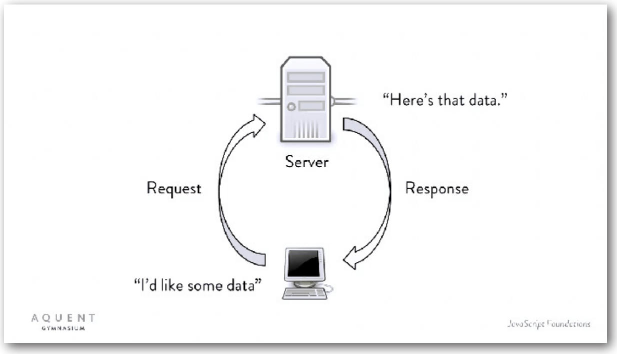
INTRODUCTION

Up to now, all of our projects in this course have been in their own little sandbox. Each site, or application has consisted of an HTML file, some JavaScript, and CSS files, delivered in a single package upfront.

LESSON 5:
SERVER
COMMUNICATION

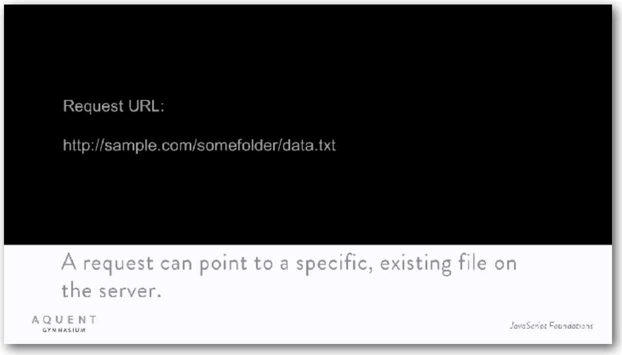
AQUENT GYMNASIUM
JavaScript Foundations

In this lesson, we'll reach outside of this sandbox, and start communicating with the world at large. We'll do this by sending data to, and receiving data from, servers.



So, what is a server? At its most basic, it's simply a computer connected to the Internet, running some special software that allows other computers to connect to it. You use servers every day. Every time you type in a URL to a web page, or an image, or a video, or even a simple text document, you're contacting a server, possibly sending some information to that server, and getting some data back.

We call the act of contacting a server, a server request. And, when we get data back, that's a server response.

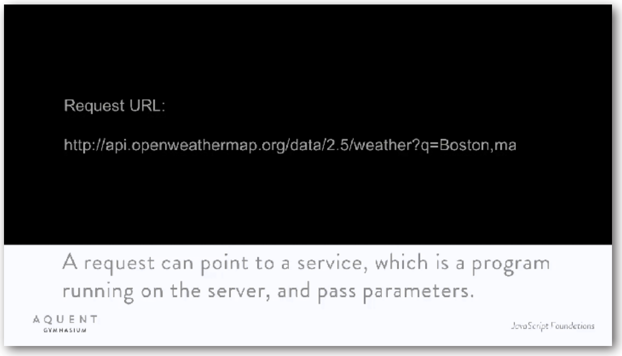

Request URL:

http://sample.com/somefolder/data.txt

A request can point to a specific, existing file on the server.

AQUENT GYMNASIUM
JavaScript Foundations

The simplest type of server request is to have a URL point directly to a file that exists somewhere in a server, like so. Here, the request is simply the URL itself, containing the path to the actual file, and the response the server sends back to you is the file that you requested.


Request URL:

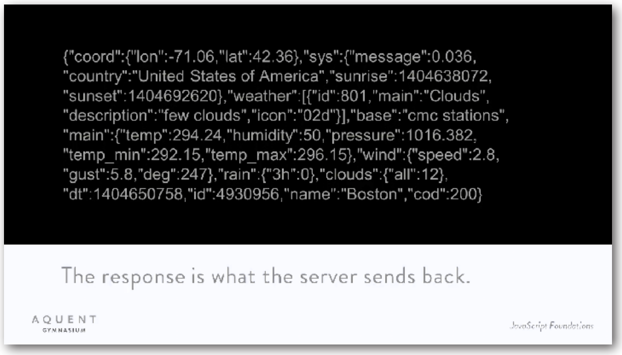
http://api.openweathermap.org/data/2.5/weather?q=Boston,ma

A request can point to a service, which is a program running on the server, and pass parameters.

AQUENT GYMNASIUM
JavaScript Foundations

But, from the viewpoint of building applications, much of the server communication you will be doing will be a lot more complex than this. Servers can run other software that allows them to do a lot more than simply return files. These programs can accept parameters along with requests, do all kinds of calculations on those parameters, reach into databases connected with the server, and return any type of dynamic data.

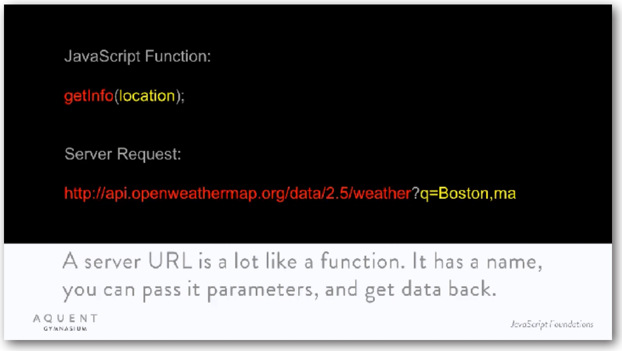
Now, here's a server request to a weather service requesting the current weather for Boston, Massachusetts. The part before the question mark is the address to the particular service we're going to contact. The part after the question mark contains the data that we're sending to the server, telling it the location for which, we'd like to know the weather. We'll cover more about that later.



```
{"coord":{"lon":-71.06,"lat":42.36},"sys":{"message":0.036,
"country":"United States of America","sunrise":1404638072,
"sunset":1404692620},"weather":[{"id":801,"main":"Clouds",
"description":"few clouds","icon":"02d"}],"base":"cmc stations",
"main":{"temp":294.24,"humidity":50,"pressure":1016.382,
"temp_min":292.15,"temp_max":296.15},"wind":{"speed":2.8,
"gust":5.8,"deg":247},"rain":{"3h":0},"clouds":{"all":12},
"dt":1404650758,"id":4930956,"name":"Boston","cod":200}
```

The response is what the server sends back.

And, here's the response we get back from the request. This might not make much sense to you now, but you can at least see little bits of data there, that look meaningful.



JavaScript Function:

```
getInfo(location);
```

Server Request:

```
http://api.openweathermap.org/data/2.5/weather?q=Boston,ma
```

A server URL is a lot like a function. It has a name, you can pass it parameters, and get data back.

AQUENT GYMNASIUM
JavaScript Foundations

Now, you might see an analogy between server requests, and calling functions within your own programs. In calling a function, you use the name of the function, along with some parameters in parentheses. The function does some work, and it returns you a value.

For a server request, the name of the service is in the URL. You send parameters, and the server does some work, and returns a value.

Making a server request is a lot like, calling a function, where that function exists on another computer, possibly hundreds, or thousands of miles away. But, that distance introduces some important differences.

First of all, it may take a few seconds for your request to go out across the net, the server to do its work, and the response to get back to you. In a local function call the result is nearly instantaneous. So, your program will make a function call, and wait for the result, before continuing on to the next line.

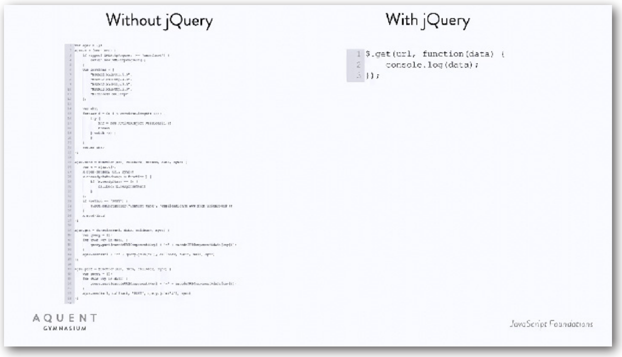
But, for a server call it wouldn't be feasible to have your program just stop working, until it gets a result from the server. Instead, we make the call, and set up a function that will be called, when the result finally gets back to us.

Calling functions locally, where we get the results almost instantaneously, is called a synchronous process. On a server request, where the response is not instantaneous, and we don't know how long it will take, we call this, an asynchronous process.

There's also the fact that the server call may fail altogether, through no fault of yours, or the server's. You may just have a bad, or slow connection, or no connection. So, in addition to providing a way to get an asynchronous response, we need a way of detecting a failure to make the request, and get a response.

Now, throughout this course there's been a pattern, where I admit that you can accomplish a certain task with jQuery, but then show you that it's really not so hard to do the same thing, without jQuery. And, here again, jQuery's a really good solution for making server requests, and handling responses.

But, this time I have to tell you that doing a server communication without jQuery isn't really that simple. I could easily spend an entire lesson just explaining how to set up requests, deal with the differences between platforms, and browsers, listening for various types of responses that can occur, and parsing out the data you get back from a response.


Without jQuery

```
[illegible]
```

With jQuery

```
$.get(url, function(data) {
    console.log(data);
});
```

JQuery has done a great job of hiding all that complexity, and making server communication as simple as humanly possible. Including jQuery in a project just for the purpose of making server calls is something I would consider justified.

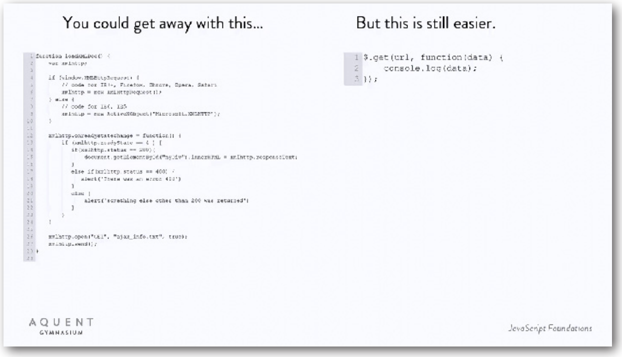
However, if you're determined to break your jQuery chains, there are some other useful libraries that deal with server communication, and nothing else.

One such library is microajax, available from this GitHub site. But, there are others if you search around. Now, AJAX stands for Asynchronous JavaScript and XML. We've just covered the asynchronous part, and you know what JavaScript is. XML is a data format that's used behind the scene, to send the request and responses.

https://code.google.com/p/microajax/

AJAX was a huge buzzword a few years ago. But, while all the principles of AJAX are in use now, more than ever, you don't see the term thrown about so much anymore. However, it did make it into a lot of programming libraries, where you'll still see it. In fact, we still call this kind of asynchronous request and response an AJAX request.

Now, for the viewpoint of creating an application, it's really not that big of a deal to write your own server communication code, but it would be a big deal to try to explain it all to you. So, for the rest of this course we'll be using jQuery for server communication. That's also what you'll see most often in the wild.


You could get away with this...

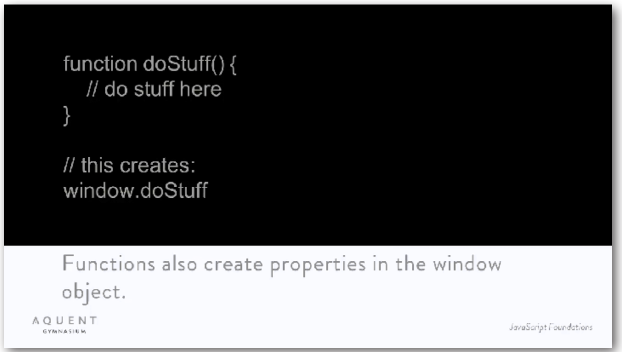
```
function loadXMLDoc() {
    var xmlhttp;

    if (window.XMLHttpRequest) {
        // code for IE7+, Firefox, Chrome, Opera, Safari
        xmlhttp = new XMLHttpRequest();
    } else {
        // code for IE6, IE5
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }

    xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState == 4 ) {
           if(xmlhttp.status == 200){
               document.getElementById("myDiv").innerHTML = xmlhttp.responseText;
           }
           else if(xmlhttp.status == 400) {
              alert('There was an error 400')
           }
           else {
               alert('something else other than 200 was returned')
           }
        }
    }

    xmlhttp.open("GET", "ajax_info.txt", true);
    xmlhttp.send();
}
```

But this is still easier.

```
$.get(url, function(data) {
    console.log(data);
});
```

Now, there are a couple of more things I need to mention before we actually get coding. One is that, we really do need a server to test the next few examples with. Up to now, we've just been running applications from our own local file system, which has worked out, very well. But, these AJAX calls will not work with a file system. They will only work if you connect to an actual server on the web.

The next problem this brings up is security. Many web services require a paid account with the company that provides the service. Others are free, but require that you apply for, and receive a special code. You use this code when you send requests to identify you.



```
function doStuff() {
    // do stuff here
}

// this creates:
window.doStuff
```

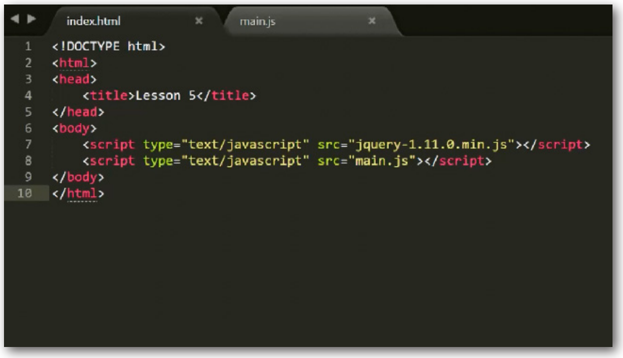
Functions also create properties in the window object.

AQUENT GYMNASIUM
JavaScript Foundations

There are other security issues that can arise as well, but I've identified at least a couple services that you should be able to use without any issues, for the purposes of the examples we do here. I can't guarantee that these services will be free and open forever, but at the time I'm creating this course they are.

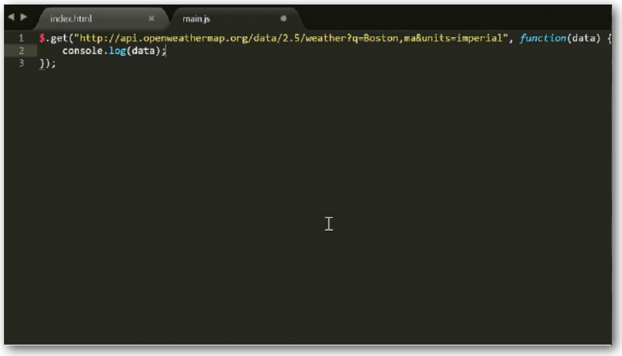
So, before we end off with this video, let's do a little bit of coding, and see if we can't make a single server request. We'll use the open weather map service I referenced a bit earlier in this video, and you can download the project files from this course's website.

Here, I've set up an HTML file called index.html. This just loads the jQuery library, and the main JS file. The project files include the latest version of jQuery available at the time of this course. You might want to grab the latest version from the jQuery site, but I doubt anything will change in the foreseeable future that would affect anything in this lesson. If you do use a different version, just make sure you change the name in the script tag.



```
<!DOCTYPE html>
<html>
<head>
    <title>Lesson 5</title>
</head>
<body>
    <script type="text/javascript" src="jquery-1.11.0.min.js"></script>
    <script type="text/javascript" src="main.js"></script>
</body>
</html>
```

Now, in the script file, we'll call this jQuery get method. You can spell jQuery out, like this, or you can use the abbreviated version, which is just the dollar sign. This method accepts various parameters, but minimally we'll want to give it a URL to the service, and a function to call when we get a response from the service. I'll just put the function in line, there.

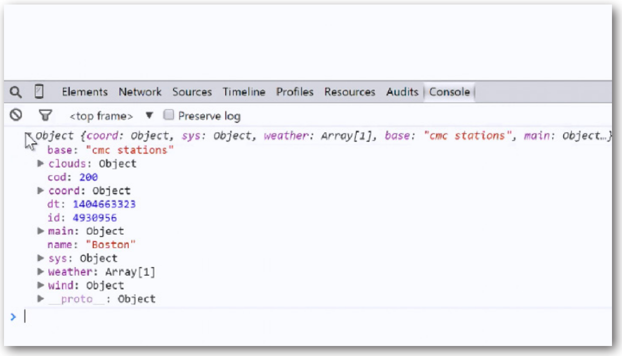


```
$.get("http://api.openweathermap.org/data/2.5/weather?q=Boston,ma&units=imperial", function(data) {
    console.log(data);
});
```

The response function will get a single parameter, which we'll call data. And, we'll just log that data variable to the console. See? Simple.

You might notice that I've included two parameters in the call. One is obviously the location, (feel free to change that to your own location), and the other tells the server that I want the results in imperial units; Fahrenheit for temperature, miles per hour for wind speed, et cetera. We'll cover request parameters in more detail, later in this lesson.

Well, let's run this in the browser, and open up the console. And, sure enough, we've gotten some kind of object back. Let's expand this, and see what we've got. Here, in the main section, we can see the current humidity, barometric pressure, and temperature. Down here, we can see that the sky is clear, and we can check in on speed, and direction of the wind. Not bad. Dig around there, and see what else you can find.



```
Object {coord: Object, sys: Object, weather: Array[1], base: "cmc stations", main: Object…}
  base: "cmc stations"
▶ clouds: Object
  cod: 200
▶ coord: Object
  dt: 1404663323
  id: 4930956
▶ main: Object
  name: "Boston"
▶ sys: Object
▶ weather: Array[1]
▶ wind: Object
▶ __proto__: Object
>
```

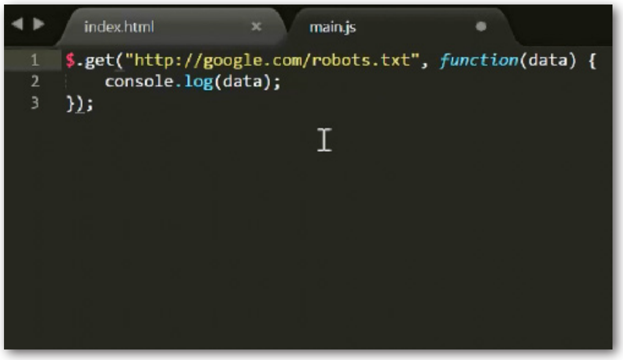
Okay, we've covered a lot of theory and managed to contact a server, and get back a meaningful, dynamic result. We'll build on all of this in the rest of the videos for this lesson.

REQUESTS AND RESPONSES

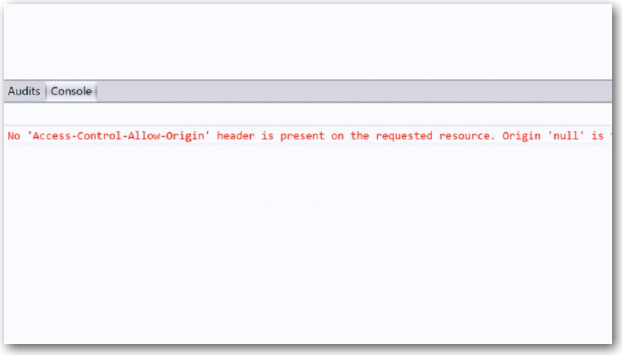
At the end of the previous video in this lesson, we managed to use jQuery to load some data from a web service. I want to show you a few more examples of this, here, and go a bit more into the various mechanics of server requests and responses. Earlier, I said that, one of the simplest server requests you can make is to request the URL of a specific, existing file. Oddly, this winds up being a bit harder to demonstrate than a request to a dynamic server program, due to security concerns.

Let's try to load a simple text file called robots.txt from google.com. This is a file used for determining what parts of a site, a search engine can access. We can plainly see it, here in the browser, so it is accessible in Open. Now, let's go back to the project file we used in the last video, and replace the URL to point to this robot's text file.

Now, when we run that, we get an error. XML HTTP request cannot load google.com robot's text. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'null' is therefore not allowed access. So, what all this is saying, is that the file cannot be loaded via JavaScript because it's not on the same domain as the original site.



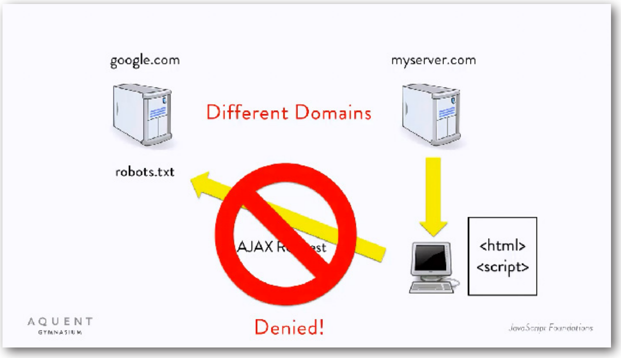
```
$.get("http://google.com/robots.txt", function(data) {
    console.log(data);
});
```



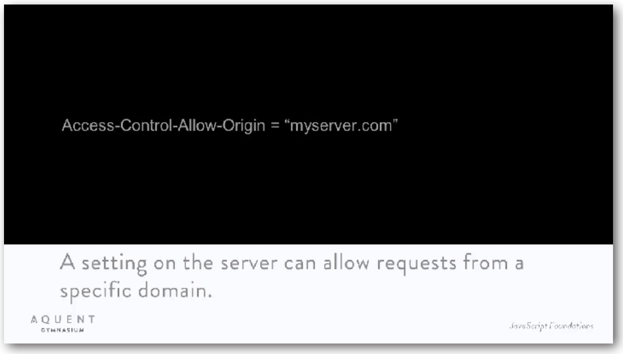
```
No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'null' is
```

If you try to load a file from the same domain, you should be fine. But, there is a restriction against loading files from another domain. The Access-Control-Allow-Origin is a setting that can be made on the server to allow domains to load files from it. Specific domains can be allowed, or it can be said with a wild card, to allow all domains to load that data. But this setting is off by default. And, because it's on the server that you're trying to load the data from, there's nothing you can really do about it.

So, now you know that if you ever see that Access-Control-Allow-Origin error, it has to do with loading files from other domains. There are various solutions, but none that we're going to go into right here. We'll just deal with web servers that have opened up their services for anyone to use.



But to give you an idea of what it would look like to load a single file with an AJAX request, I've set up a local Apache server on my computer. You could also set up a simple Python-based server, or Node.JS server. A local server is very useful for testing, particularly if you're doing a lot of AJAX work. Now, although it's beyond the scope of this course, I will include some links in this lesson's materials to some resources that will help you set up your own local web server.



Access-Control-Allow-Origin = "myserver.com"

A setting on the server can allow requests from a specific domain.

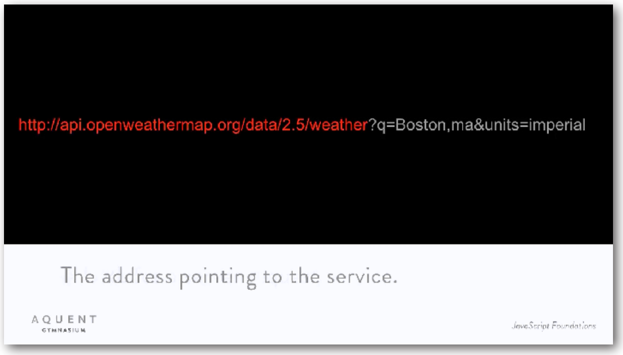
AQUENT GYMNASIUM
JavaScript Foundations

Ultimately, if you have access to an online web server, via our hosting company, which is likely, you can upload the files to that server, and test the example I'm about to do. If not, don't worry about it. It's a quick and simple example. So, you can just sit back and watch. So, here, I have a simple text file called, data.txt. It just contains the text, "hello world." In the JavaScript file, I have the same code as before, a call to jQuery get passing in the name of the file I want to get, and a handler function that logs the received data.

I'll run that on the browser. And here, you can see that it loaded the text file, and logged its results. Now, notice here that the address is localhost. This means that I'm running this from my local Apache web server. Now, let me try to run this from the file system. This is the exact same HTML JavaScript end data file. But, note that the URL now starts with "file", and now, we're back to the Access-Control-Allow-Origin error. So, this error also means you're trying to make AJAX requests from the file system.

Now, that's all I'm going to say about loading individual files. But, remember in lesson four, when we were discussing templates, I said that there was a way to create templates in external files, and load them in with jQuery. This is the exact mechanism that you'd use to do that.

Now, let's go back to the Weather Map Service, and learn some more about requested responses. Let's discuss the parameters that we're sending to the request. Here's the URL we're using. You'll see that in the middle of all that is a question mark. Everything to the left of the question mark is the address of the service that we're going to send the request to. Everything to the right of the question mark is called the query string. Query strings can have different formats. But, the for-



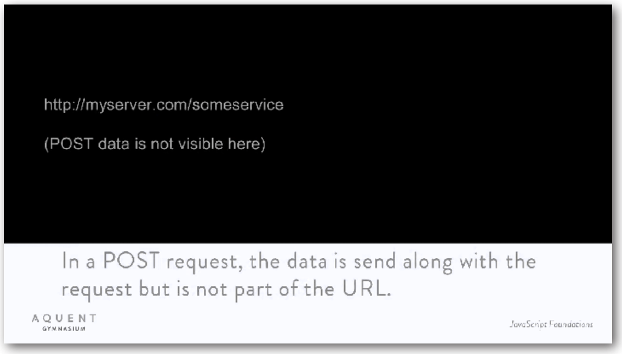
http://api.openweathermap.org/data/2.5/weather?q=Boston,ma&units=imperial

The address pointing to the service.

AQUENT GYMNASIUM
JavaScript Foundations

mat you'll see most often is what we have, here. This is often called a key value list.

Here, we have q=Boston.ma and units=imperial So, we have two key value pairs. The key is the name of the parameter; in this case, the letter "q" and "units." And, the value is the data that you're providing for the parameters; in this case, "Boston Mass" and "Imperial." Each key value pair is separated by an ampersand. We could add additional key value pairs here, as well. Just put in another ampersand, and another key and value, separated by an equal sign.



```
http://myserver.com/someservice

(POST data is not visible here)
```

In a POST request, the data is send along with the request but is not part of the URL.

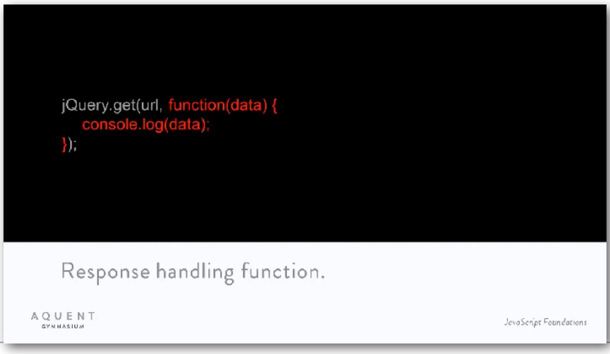
AQUENT GYMNASIUM
JavaScript Foundations

Now, if you're wondering how I know what key value pairs to use for parameters, this is all documented at the service's site at this page. Most online services will have some kind of page like this, documenting the service address, or addresses, the different parameters that can be sent, and what you'll get in response. This is all collectively referred to as the API, or application programming interface of the service. One thing worth noting here, is that there are two main ways to send data to a service center request. These are called get, and post. We've been using a get type of request, where the data is tacked on to the URL as a query string. Obviously, this is not always feasible.

For example, say you want to submit some kind of long-form text to a server. You certainly wouldn't want to include several paragraphs of text in your URL. A more serious example is any kind of login, or authentication service with a user name and password, or any other sensitive information. This is the last kind of thing you'd ever want to put right in a URL as a query string. Thus, there's an alternate type of request called post.

With a post request, the data is sent along with the request, but it's not visible as with a get request. Other actions can be done to encrypt or obfuscate any sensitive information. I'm not going to cover post-type requests here, but you should know that they exist. JQuery has a post method, as well, that makes sending these requests very easy.

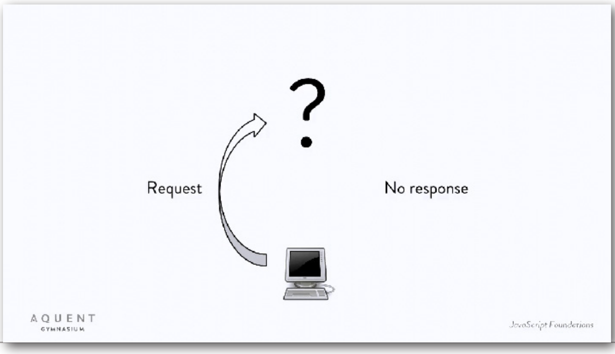
Finally, I just want to go a little bit more into responses. We've been passing a function right into the get method to handle the response. This function will be called when the response is complete and successful, and returns a valid response. But, what if that request fails? Maybe the URL is bad, or the server's down, or you lose your internet connection for a bit. All kinds of things could happen. Because of the nature of asynchronous calls, we send out the request, and the application just does whatever else it needs to. Or, just does nothing until the request returns successfully. So, if that request never returns successfully, we'll just be sitting there, forever.



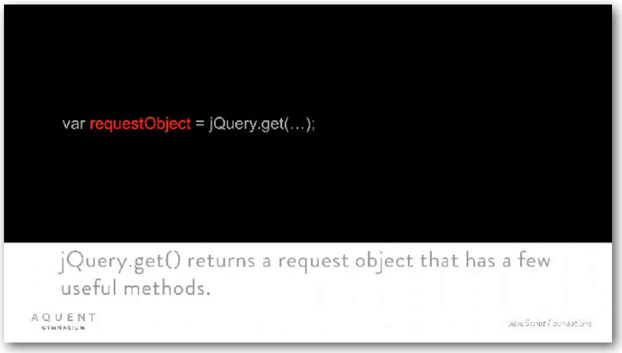
```
jQuery.get(url, function(data) {
    console.log(data);
});
```

Response handling function.

AQUENT GYMNASIUM
JavaScript Foundations



It would be good to be able to put some logic in there. Maybe to attempt the request again, or at the very least, alert the user to the fact of the failure, and recommend that they try it again. So, we have a function here, that will be called on success. Is there any way of being informed of a failure? Well, you bet there is. We've been passing a function as one of the parameters to the jQuery get call, which is one way to handle a successful call. But, it doesn't allow us to be informed of when there's a failure. There is another method, though.



```
var requestObject = jQuery.get(...);
```

jQuery.get() returns a request object that has a few useful methods.

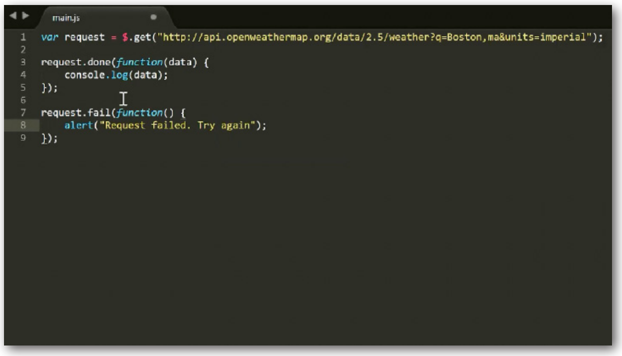
AQUENT GYMNASIUM
JavaScript Foundations

It turns out that when you call this jQuery get method, it returns an object, immediately. This is a special request object that has various properties and methods on it. I've reverted back to that Weather Service call. Now, let's alter a call, so that we save this request object. And, we'll remove the whole success-handling function from the call.

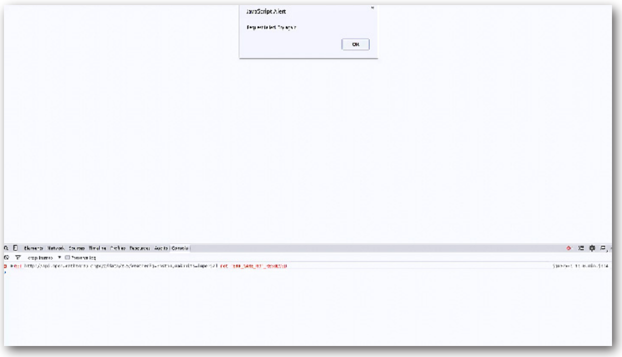
This request object has a method called, "done". You can call this "done" method passing in a function. And, that function will be called when the request successfully returns. So, let's pass that function that we used originally. Now, this code will function in exactly the same way as the original version. We make the call, and when it's done, the function passed at "done" will be called, logging a result.

But here's the good part. The request object also has a fail method. This method works the same way. You pass it a function. And, if the request fails for whatever reason, that function gets called. Let's try it.

Here, I'll throw up an alert, telling the user that the request failed. In a more full-featured application, you might want to add some code to try the request again, maybe two or three times, before giving up, in case it was just a momentary network issue. Now, to create an error, I'm just going to mess up the URL, a bit. You could also turn off your wireless, or unplug your ethernet cable, or do something else, to prevent the call from reaching the server. Sometimes, leaving off required parameters, or sending incorrect parameters may cause an error, too.



```
var request = $.get("http://api.openweathermap.org/data/2.5/weather?q=Boston,ma&units=imperial");

request.done(function(data) {
    console.log(data);
});

request.fail(function() {
    alert("Request failed. Try again");
});
```



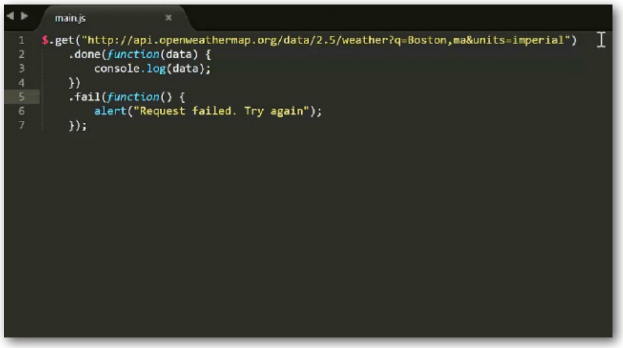
JavaScript Alert

Request failed. Try again

Now, I'll try this with a bad URL, and bam, we get an alert. Now, there's a shortcut to all that. It's called "method chaining." It makes for more concise code, but may seem a little off at first. You should get used to it, though, because you'll see it a lot, especially in jQuery.

The original jQuery get method returns that request object, we know that. So, we can call “done”, directly on that function call, like so. That whole first line with the get call returns an object. So, that line itself can be used as an object. Removing the semicolon, and tacking on a dot, plus the call to “done.”

It may seem odd to have the dot, and the method name on the next line indented like that, but it’s perfectly legitimate. And, you’ll see it like that very often. Now, what about this fail call, since we no longer have the request object saved?



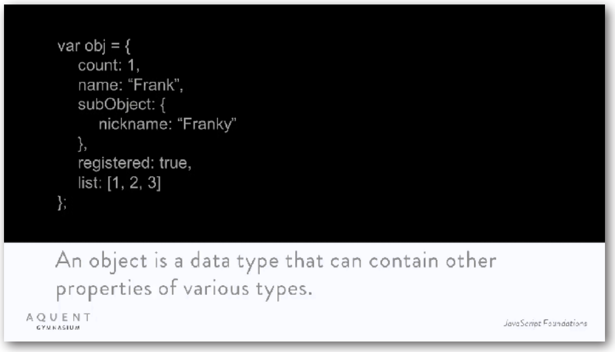
```
$.get("http://api.openweathermap.org/data/2.5/weather?q=Boston,ma&units=imperial")
    .done(function(data) {
        console.log(data);
    })
    .fail(function() {
        alert("Request failed. Try again");
    });
```

Well, it turns out that calling “done” on that request returns that request object, again. So, we can just tack on our fail call in the same way. The fail method also returns a request, so, we could switch these around and it would still work. Again, you’ll see this a lot in jQuery. Sometimes chaining a half dozen or more commands on a single HTML element, or group of elements.

You might notice that I’ve also corrected the URL. So that, if we try that again in the browser, we are good. So, now, you know a bit more about server requests, and how to make them, and how to handle both successful and failed requests. In the next video, we’ll take a look at what you might get back from a request.

WHAT IS A RESPONSE?

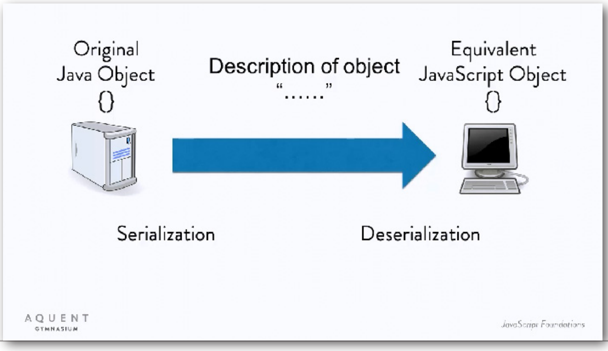
In the last video, we learned more about making requests, and receiving responses. In this video, we’ll cover what will be in those responses, and how to handle the data that you get back. Let’s first review what we’ve been doing in the example so far. We have a response handler function that we either passed directly into the get method, or to the done method of the request object. We have defined that function, so that it has a single parameter called data, and we’ve just been logging that data.



```
var obj = {
    count: 1,
    name: “Frank”,
    subObject: {
        nickname: “Franky”
    },
    registered: true,
    list: [1, 2, 3]
};
```

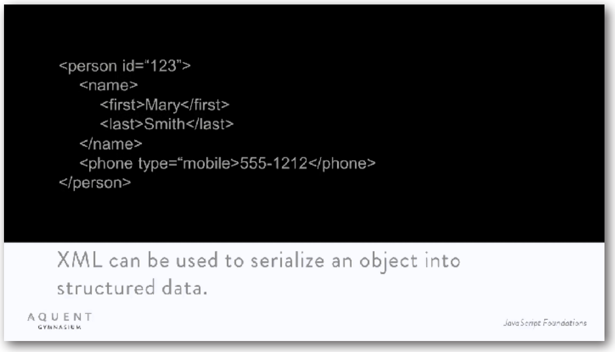
An object is a data type that can contain other properties of various types.

AQUENT GYMNASIUM
JavaScript Foundations



When I did the local host example, I loaded a simple text file. And, when I logged that in the console, we saw that the data was simply a string containing the text that was in that file. But, in the open weather map example, we saw that the data was an object, with several nested objects containing some strings, and numbers about the weather. To quickly review, an object is a special JavaScript data type that can have properties attached to it. These properties can be strings, numbers, Booleans, arrays, or even other objects, or functions. So, does the open weather map service create a JavaScript object, and send it across the net into your browser? Well, not really. You can’t actually send objects around the web, and it just doesn’t work like that.

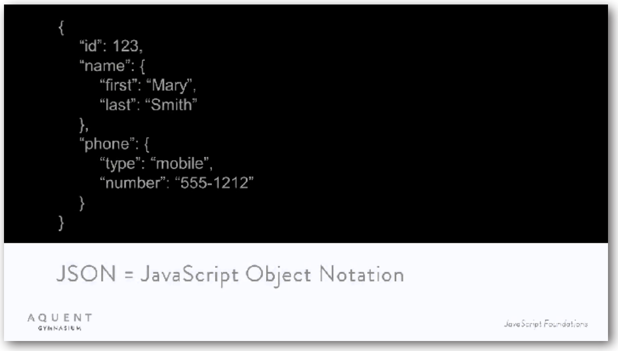
What you can do is take an object on the server, transform it into some information that describes that object, and send that information across. In JavaScript, we can take that information, and use it to create an object that's equivalent to the object that existed on the server. We call this process serialization, and deserialization. The object is serialized into a format that can be sent across the web, and deserialized back into an object. So, what format is that serialized data? Most often, it's simply a string that describes the object, and all of its properties.



```
<person id="123">
    <name>
        <first>Mary</first>
        <last>Smith</last>
    </name>
    <phone type="mobile>555-1212</phone>
</person>
```

XML can be used to serialize an object into structured data.

There are two widely used formats for serializing objects into strings. XML and JSON. Up until relatively recently, XML was the king. You've most likely seen, or even worked with XML in the past. This format is very much like HTML, though its rules are a bit more strict. The data is structured in tags, created with angle brackets. Tags can have attributes and trial tags. Unlike traditional HTML, you can define tags with whatever names you want. So, it's a great way to store structured data.



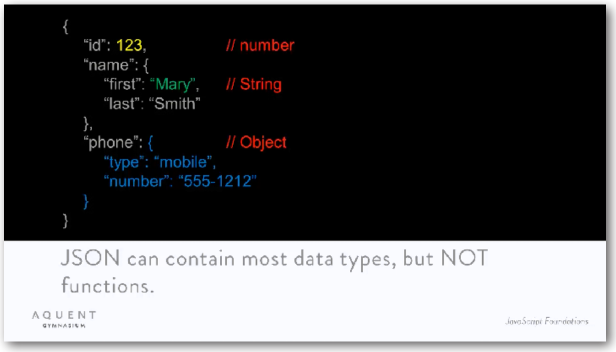
```
{
    "id": 123,
    "name": {
        "first": "Mary",
        "last": "Smith"
    },
    "phone": {
        "type": "mobile",
        "number": "555-1212"
    }
}
```

JSON = JavaScript Object Notation

However, XML can get a bit verbose. All those opening tags, and closing tags surrounding every piece of data; the complex structure can start to get very tough to read, much less write. And, even when it's never read by a human, all those tags add a lot of overhead to the size of the file. Thus, in recent years, another format has been getting a lot of popularity, JSON. JSON stands for JavaScript Object Notation. And, although it was designed for use by JavaScript, JSON is now supported in nearly all commonly used languages, on every popular platform. But, as you're already very used to creating objects in JavaScript, JSON should come very easy to you. Although it's created as simple text, and received as a simple string, it's almost the exact same syntax we've been using to create objects in JavaScript.

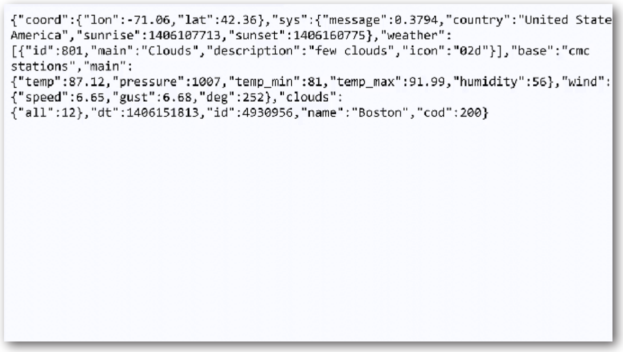
Arrays are denoted with square brackets, with array elements listed and separated by commas. Objects are denoted with curly brackets, with property names, and values separated by colons, and each property listed, separated by commas. There are two main differences between JSON and the JavaScript syntax you use to create an object in a JavaScript program. First is that, in JSON, property names need to be strings enclosed in double quotes. Second is that, you cannot define functions in JSON. Numbers, strings, objects, arrays, and Booleans are all fine, but not functions. Remember that, JSON can be used across languages, and platforms so a JavaScript function would not be valid in Java, or Objective-C, for example.

The distinction between JSON and JavaScript can get confusing, and I've even seen veteran programmers get tripped up by it. The bottom line is, if you're writing it as code in a JavaScript file, code then will be executed, then it's JavaScript. If you're saving, or loading an object description in a file, or getting it from a server request, that's JSON, and should follow the rules of JSON. So, let's see some JSON in action. Take that URL that we used to contact the Weather Map Service, and paste it right into a browser, and see what we get. What do you know? It's JSON. You see that the whole thing is surrounded by a pair of curly brackets, indicating that it is a single object.



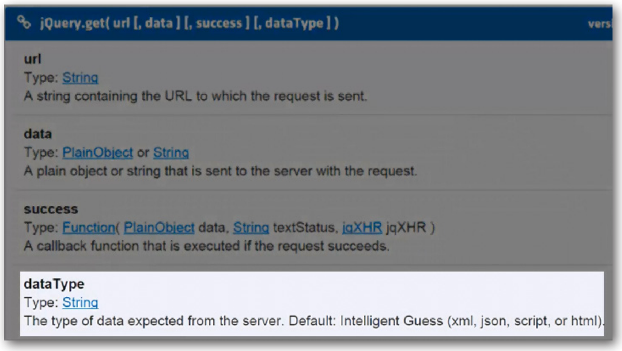
```
{
    "id": 123,              // number
    "name": {
        "first": "Mary",    // String
        "last": "Smith"
    },
    "phone": {              // Object
        "type": "mobile",
        "number": "555-1212"
    }
}
```

JSON can contain most data types, but NOT functions.



```
{"coord":{"lon":-71.06,"lat":42.36},"sys":{"message":0.3794,"country":"United State
America","sunrise":1406107713,"sunset":1406160775},"weather":
[{"id":801,"main":"Clouds","description":"few clouds","icon":"02d"}],"base":"cmc
stations","main":
{"temp":87.12,"pressure":1007,"temp_min":81,"temp_max":91.99,"humidity":56},"wind":
{"speed":6.65,"gust":6.68,"deg":252},"clouds":
{"all":12},"dt":1406151813,"id":4930956,"name":"Boston","cod":200}
```

First, we have the coord property. Note, the coord is enclosed in double quotes. Coord is an object containing two other properties, lon and lat, for longitude and latitude. These are both numbers, and so on. It's not all nicely formatted, like you might do by hand. But this isn't really meant to be read by humans, only by a computer programmer. Now, let's go back to that sample app we used in the last video, and run that again. Here we are in the console, after we logged the data property. Note well, that the data is not a string containing JSON, but it's an actual object. So, if the service set does say JSON string, how did we wind up with an object?



jQuery.get(url [, data] [, success] [, dataType])

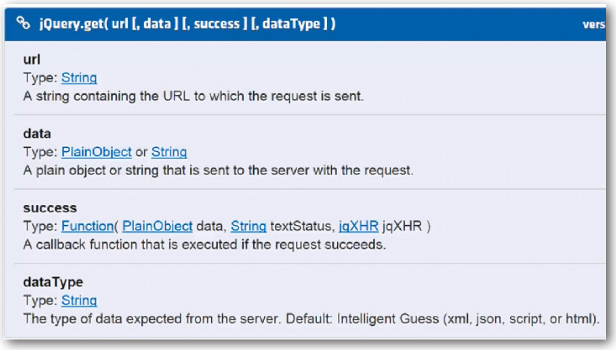
url
Type: String
A string containing the URL to which the request is sent.

data
Type: PlainObject or String
A plain object or string that is sent to the server with the request.

success
Type: Function(PlainObject data, String textStatus, jqXHR jqXHR)
A callback function that is executed if the request succeeds.

dataType
Type: String
The type of data expected from the server. Default: Intelligent Guess (xml, json, script, or html).

Well, jQuery does its best to determine what type of data it has received as a request. And, if it determines that the string contains JSON, it will try to deserialize it into an object. In this case, it's succeeded. So, all we ever see is an object, just as if the server had sent us that object directly. But, you can tell a jQuery query exactly what type of data you're expecting, and it will respect that, if it can. For example, let's tell jQuery that we want a response back as simple text. Now, the get method actually has four parameters: the URL, any data you want to send to the server with the request, the success handler function, and the data type.



jQuery.get(url [, data] [, success] [, dataType])

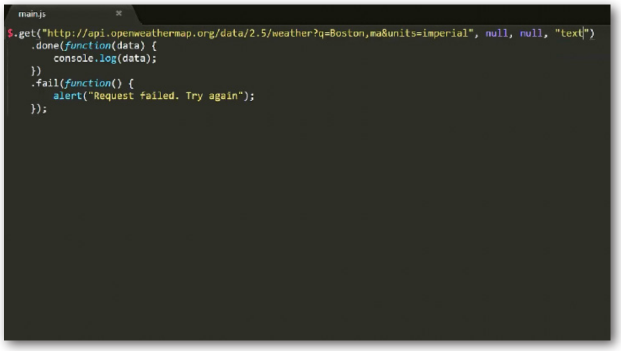
url
Type: String
A string containing the URL to which the request is sent.

data
Type: PlainObject or String
A plain object or string that is sent to the server with the request.

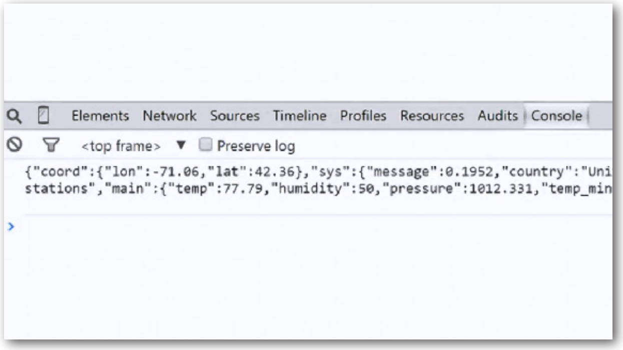
success
Type: Function(PlainObject data, String textStatus, jqXHR jqXHR)
A callback function that is executed if the request succeeds.

dataType
Type: String
The type of data expected from the server. Default: Intelligent Guess (xml, json, script, or html).

Sometimes with calls like this, you can skip parameters and jQuery can actually figure out what you meant. For example, earlier we just passed a URL and a success handler function. Skipping the data parameter, jQuery was able to figure out that the second parameter was a function and it used that as a success handler. But here we'll need to specify all the parameters to eliminate any confusion. We'll just make the data and success parameters null, and specify text as the data type. Now, when we run this, it sees that we're expecting text. So, it does not try to do any special JSON to object deserialization. It just gives us raw JSON text, just like we saw in the browser.

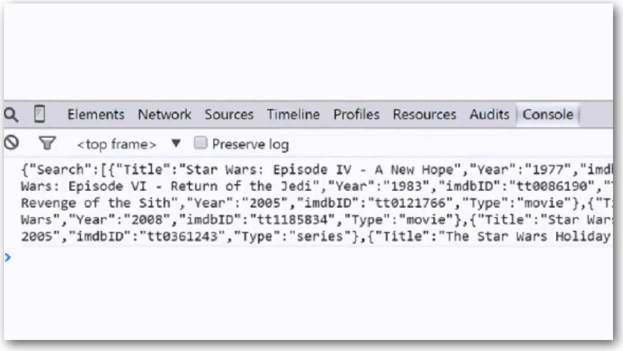


```
$.get("http://api.openweathermap.org/data/2.5/weather?q=Boston,ma&units=imperial", null, null, "text")
    .done(function(data) {
        console.log(data);
    })
    .fail(function() {
        alert("Request failed. Try again");
    });
```


Elements Network Sources Timeline Profiles Resources Audits Console
<top frame> Preserve log

```
{"coord":{"lon":-71.06,"lat":42.36},"sys":{"message":0.1952,"country":"Uni
stations","main":{"temp":77.79,"humidity":50,"pressure":1012.331,"temp_min
```

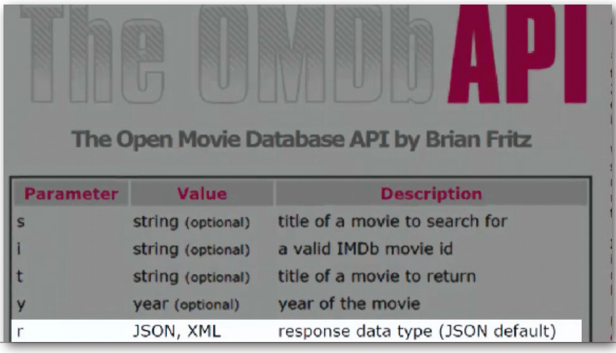
Of course, if we don't want to rely on jQuery's automatic data conversion, we could explicitly specify JSON as the data type. And, now you see we're back to an object. Now, let's try another service. This is the Open Movie Database service. This allows you to search for movies by title, or year, and return a list of results. Or, you can look up a specific movie by title or IMDB ID number, and get information about that movie. Here's the URL for the service. To search for any movies with a particular word, or words in their title, you pass the key value pair of s= whatever. I'll try "Star Wars." I've left off any other parameters. So, let's see what we get.


Elements Network Sources Timeline Profiles Resources Audits Console
<top frame> Preserve log

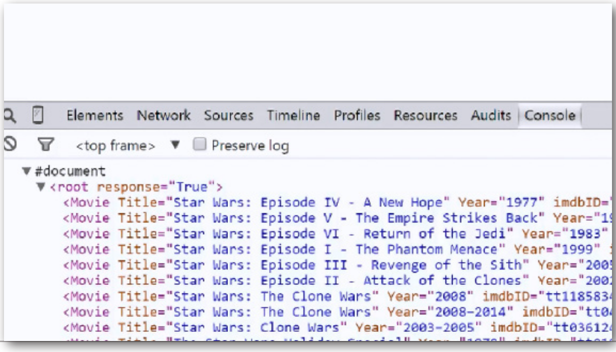
```
{"Search":[{"Title":"Star Wars: Episode IV - A New Hope","Year":"1977","imd
Wars: Episode VI - Return of the Jedi","Year":"1983","imdbID":"tt0086190","
Revenge of the Sith","Year":"2005","imdbID":"tt0121766","Type":"movie"},{"T
Wars","Year":"2008","imdbID":"tt1185834","Type":"movie"},{"Title":"Star War
2005","imdbID":"tt0361243","Type":"series"},{"Title":"The Star Wars Holiday
```

Well, we had a JSON string, and we can see that there's some stuff about Star Wars in there. Now, for whatever reason, jQuery did not detect that this was JSON. So, it didn't automatically convert it to an object. But, we know how to fix this. Just add the data type parameter, again and now, we get an object. This object has a property called, search, which is an array. If we expand that array, we can see that there are quite a few movies with the words, "star wars" in the titles. And, there's a bit of information about each movie.

Now, back over here, on the OMDB API website, you see that we can add another parameter, r, to specify the response type as JSON or XML, and we see that JSON is the default. Well, let's see if we can get some XML back. Back in the request, we can just add &r=XML to the URL, and we'll change the data type to text for now. We'll run that and look at the console and you can see that we received a string of XML that represents the movie data. And, if we go back and change the data type to XML, now we have an XML document that has all the same data, instead of just a string.



Parameter	Value	Description
s	string (optional)	title of a movie to search for
i	string (optional)	a valid IMDb movie id
t	string (optional)	title of a movie to return
y	year (optional)	year of the movie
r	JSON, XML	response data type (JSON default)


Elements Network Sources Timeline Profiles Resources Audits Console
<top frame> Preserve log

```
▼#document
  ▼<root response="True">
      <Movie Title="Star Wars: Episode IV - A New Hope" Year="1977" imdbID="
      <Movie Title="Star Wars: Episode V - The Empire Strikes Back" Year="19
      <Movie Title="Star Wars: Episode VI - Return of the Jedi" Year="1983"
      <Movie Title="Star Wars: Episode I - The Phantom Menace" Year="1999"
      <Movie Title="Star Wars: Episode III - Revenge of the Sith" Year="2005
      <Movie Title="Star Wars: Episode II - Attack of the Clones" Year="200
      <Movie Title="Star Wars: The Clone Wars" Year="2008" imdbID="tt118583
      <Movie Title="Star Wars: The Clone Wars" Year="2008-2014" imdbID="tt0
      <Movie Title="Star Wars: Clone Wars" Year="2003-2005" imdbID="tt03612
```

I'm not going to go into parsing XML and JavaScript, as it's a bit involved. Allowing jQuery to give us a full-fledged JavaScript object is so much easier. And, that's about all we really need to know to start making some powerful, web-connected applications. In the final video for this lesson we'll again attempt to tie all of what we learned together, and make a really fun application. This time, using the Open Movie Database API to search for movies, and a couple of mustache templates to display the result. See you, when you're ready.

PROJECT: MOVIE FINDER

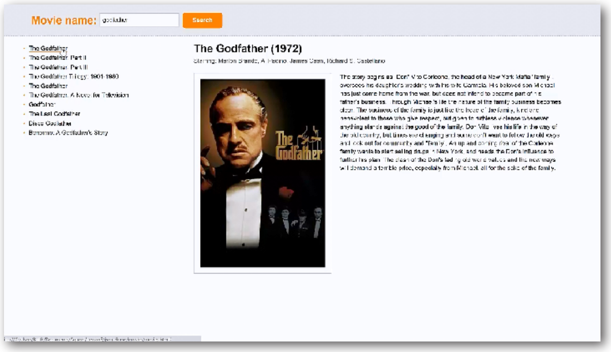
Now, we're going to make the most ambitious app in the course, so far. We'll have two separate templates and two different server calls.

Here's the app we'll be creating. Initially, it's just a form with a text input field and a button. The user can enter a search term and it will receive a list of results; any movies that have that search term in the titles.

You already know how to do that much. But, here we're displaying those results as a clickable list, using a template. When the user clicks one, another call is made to the open movie database to get details about that specific movie, and that detailed data is displayed, using another template.

Here you can see they were showing the title, date, list of actors, and a poster image, and a plot summary. Not bad. Ready to create this app?

As in the final project in lesson four, I've gotten a designer to help with the design of this application. So the HTML and templates are already set up for us, with lots of sweet CSS styling. We just need to write the code to make it all work.



- The Godfather
- The Godfather: Part II
- The Godfather: Part III
- The Godfather Trilogy: 1901-1980
- The Godfather
- The Godfather: A Novel for Television
- Godfather
- The Last Godfather
- Disco Godfather
- Bonanno: A Godfather's Story

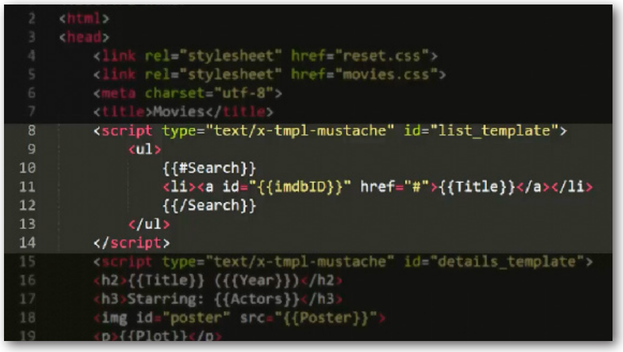
The Godfather (1972)

Starring: Marlon Brando, Al Pacino, James Caan, Richard S. Castellano

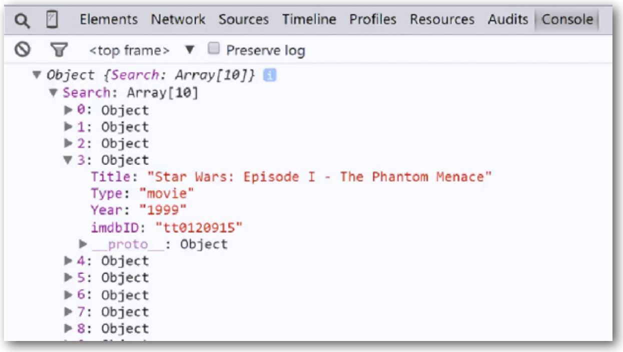
The story begins as "Don" Vito Corleone, the head of a New York Mafia "family", oversees his daughter's wedding with his wife Carmela. His beloved son Michael has just come home from the war, but does not intend to become part of his father's business. Through Michael's life the nature of the family business becomes clear. The business of the family is just like the head of the family, kind and benevolent to those who give respect, but given to ruthless violence whenever anything stands against the good of the family. Don Vito lives his life in the way of the old country, but times are changing and some don't want to follow the old ways and look out for community and "family". An up and coming rival of the Corleone family wants to start selling drugs in New York, and needs the Don's influence to further his plan. The clash of the Don's fading old world values and the new ways will demand a terrible price, especially from Michael, all for the sake of the family.

Let's look at the templates. First, the list template. This will display the list of movies found in the initial search. This template opens an unordered list, and then has an arrangement of tags that may not be familiar to you.

Remember, I said in lesson four that Mustache has a special syntax for handling arrays of data. Well, here it is.



```
<html>
<head>
    <link rel="stylesheet" href="reset.css">
    <link rel="stylesheet" href="movies.css">
    <meta charset="utf-8">
    <title>Movies</title>
    <script type="text/x-tmpl-mustache" id="list_template">
        <ul>
            {{#Search}}
            <li><a id="{{imdbID}}" href="#">{{Title}}</a></li>
            {{/Search}}
        </ul>
    </script>
    <script type="text/x-tmpl-mustache" id="details_template">
    <h2>{{Title}} ({{Year}})</h2>
    <h3>Starring: {{Actors}}</h3>
    <img id="poster" src="{{Poster}}">
    <p>{{Plot}}</p>
```

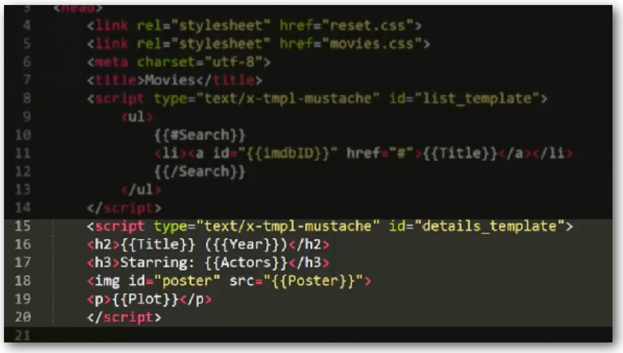

Elements Network Sources Timeline Profiles Resources Audits Console
<top frame>
Preserve log

```
▼ Object {Search: Array[10]}
  ▼ Search: Array[10]
    ▶ 0: Object
    ▶ 1: Object
    ▶ 2: Object
    ▼ 3: Object
        Title: "Star Wars: Episode I - The Phantom Menace"
        Type: "movie"
        Year: "1999"
        imdbID: "tt0120915"
      ▶ __proto__: Object
    ▶ 4: Object
    ▶ 5: Object
    ▶ 6: Object
    ▶ 7: Object
    ▶ 8: Object
```

Let's look again at the object we got back from the search. It has a property called, search, which is an array. So, these two tags, #search and /search, indicate that everything in between them will be done for every element of the array called, search. Inside there, for each result, we create a list item, and within that list item, an anchor tag with a dummy href, just so that it appears as a clickable link.

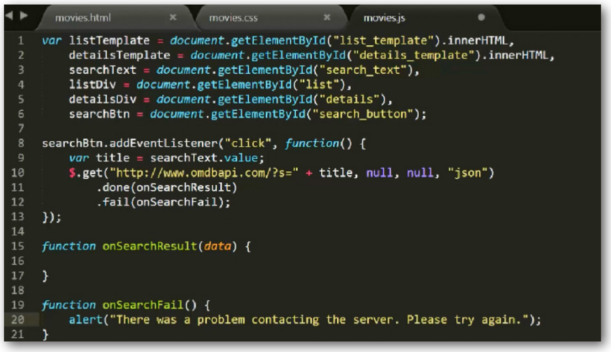
Each item in the search array will be an object, with a bunch of properties. We'll use the title property of that object, as the text of the anchor tag in the IMDB ID property, as the ID of the tag. This IMDB ID will be unique for every single movie, so this will be a good way to retrieve the details of a particular movie. You'll see how we use that, in a moment.

Next there's the details template. This will get another object, with a bunch of information about that movie. There's nothing really special here. It's just displaying selected pieces of that information. Note that it's using one property, poster, as the source of an image tag. That poster property should contain the URL to a picture of the movie poster for a given movie.



```html
<head>
    <link rel="stylesheet" href="reset.css">
    <link rel="stylesheet" href="movies.css">
    <meta charset="utf-8">
    <title>Movies</title>
    <script type="text/x-tmpl-mustache" id="list_template">
        <ul>
            {{#Search}}
            <li><a id="{{imdbID}}" href="#">{{Title}}</a></li>
            {{/Search}}
        </ul>
    </script>
    <script type="text/x-tmpl-mustache" id="details_template">
    <h2>{{Title}} ({{Year}})</h2>
    <h3>Starring: {{Actors}}</h3>
    <img id="poster" src="{{Poster}}">
    <p>{{Plot}}</p>
    </script>
```

Below the templates there are a couple of empty divs to hold the list, and the details. Then, a script statement for Mustache, jQuery, and our main JS file. Now, let's write some code.



```javascript
var listTemplate = document.getElementById("list_template").innerHTML,
    detailsTemplate = document.getElementById("details_template").innerHTML,
    searchText = document.getElementById("search_text"),
    listDiv = document.getElementById("list"),
    detailsDiv = document.getElementById("details"),
    searchBtn = document.getElementById("search_button");

searchBtn.addEventListener("click", function() {
    var title = searchText.value;
    $.get("http://www.omdbapi.com/?s=" + title, null, null, "json")
        .done(onSearchResult)
        .fail(onSearchFail);
});

function onSearchResult(data) {

}

function onSearchFail() {
    alert("There was a problem contacting the server. Please try again.");
}
```

Again, here we are with a blank file. And, as last time, I'm going to start by defining some variables, and grabbing some of the HTML elements, and the templates that we need to work with. We'll get the two templates by using get element by ID, and enter HTML. Then, the search text input field, the empty list, and details div, and the search button.

Next, we can think about how the app will be used. The user will enter some text and click on the search button, and they'll expect to see some results. So, we'll have to add a click event listener to the search button. Inside that listener, we'll first get the search term that the user entered. This will be the value property of the search text element.

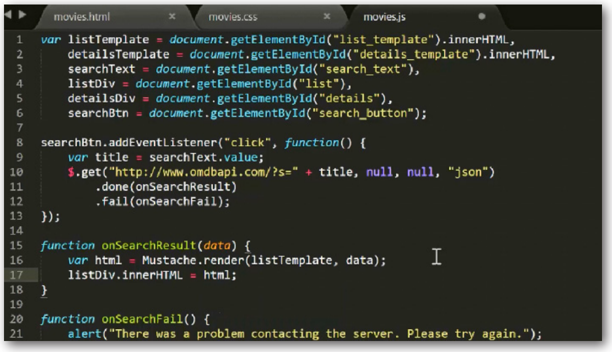
We can then make our first server call, using jQuery get. We'll pass in the server URL with s equals, and then concatenate the search term. Then we'll add null null and json so that jQuery knows to deserialize that json string we get, back into an object.

Then, we have our done and fail handling functions. Rather than make these inline, I'll specify them as separate named functions, onSearchResult and onSearchFail. Then, I'll create those two functions.

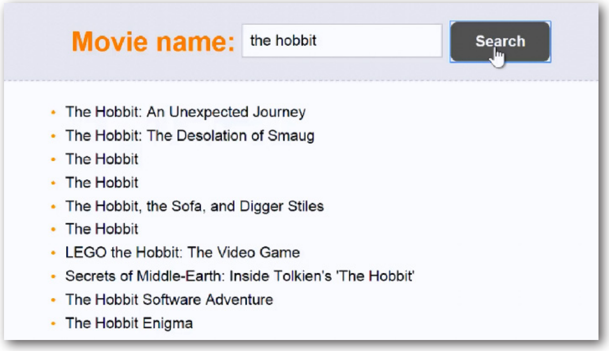
For failure, I'll just quickly throw up an alert. That's easy enough.

Now, for onSearchResult, we should get an object that has an array called, search. And, our template is all set up to handle this exact object.

So, we can say, HTML = Mustache.render(listTemplate, data). And, then, listDiv.innerHTML = HTML. And, that should be that for displaying the list of results. Let's try it.



```
var listTemplate = document.getElementById("list_template").innerHTML,
    detailsTemplate = document.getElementById("details_template").innerHTML,
    searchText = document.getElementById("search_text"),
    listDiv = document.getElementById("list"),
    detailsDiv = document.getElementById("details"),
    searchBtn = document.getElementById("search_button");

searchBtn.addEventListener("click", function() {
    var title = searchText.value;
    $.get("http://www.omdbapi.com/?s=" + title, null, null, "json")
        .done(onSearchResult)
        .fail(onSearchFail);
});

function onSearchResult(data) {
    var html = Mustache.render(listTemplate, data);
    listDiv.innerHTML = html;
}

function onSearchFail() {
    alert("There was a problem contacting the server. Please try again.");
```



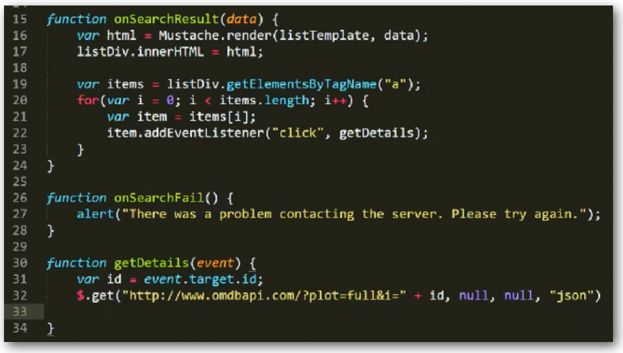
- The Hobbit: An Unexpected Journey
- The Hobbit: The Desolation of Smaug
- The Hobbit
- The Hobbit
- The Hobbit, the Sofa, and Digger Stiles
- The Hobbit
- LEGO the Hobbit: The Video Game
- Secrets of Middle-Earth: Inside Tolkien's 'The Hobbit'
- The Hobbit Software Adventure
- The Hobbit Enigma

I'll enter The Hobbit, and do the search. And, sure enough, a bunch of movies containing those words. Cool. If you're not convinced of the power of templates now, I'm not sure what else I can do to make you see the light.

Next, we need to make it so that when the user clicks on one of those results, the details for that particular movie are shown. So, we need to add a click listener for each of those items that were just created in the template. That would be really easy to do with jQuery, since it's already in the project. But, in keeping with the spirit of the course, I'm going to do it manually.

First, I'll say, list get elements by tag name A. This will give me a list of all those anchor elements that contain the title of each movie result. Remember that, each one also has an ID, which is the IMDB ID of that movie.

I can just loop through this array of results, and add a click event listener to each item. Again, I'll use a separate named function for this, just for clarity's sake. This get details handler function will get an event object as a parameter. We've covered some of the properties of event objects, much earlier in the course.



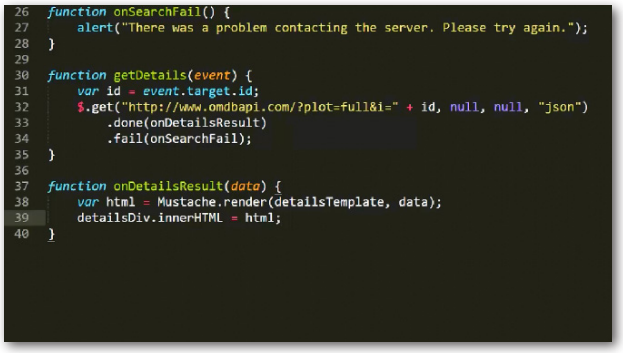
```
function onSearchResult(data) {
    var html = Mustache.render(listTemplate, data);
    listDiv.innerHTML = html;

    var items = listDiv.getElementsByTagName("a");
    for(var i = 0; i < items.length; i++) {
        var item = items[i];
        item.addEventListener("click", getDetails);
    }
}

function onSearchFail() {
    alert("There was a problem contacting the server. Please try again.");
}

function getDetails(event) {
    var id = event.target.id;
    $.get("http://www.omdbapi.com/?plot=full&i=" + id, null, null, "json")

}
```

One useful property for click events is target. This gives you the actual element that was clicked. So, event.target.ID will contain that IMDB ID that will help us make the next server call.

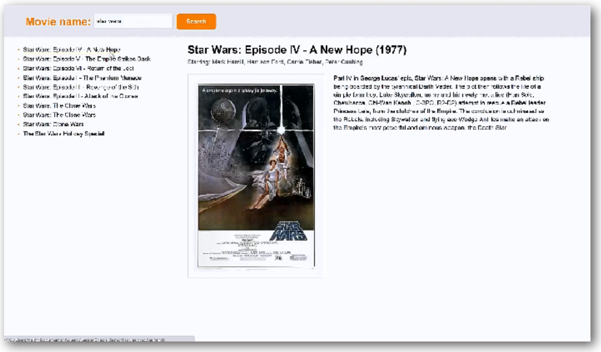
So, we'll call the open movie database server again, this time using the i parameter. This i parameter allows you to pass an IMDB ID, and it will return the single movie that matches that ID.

I'm going to throw in another parameter, there. Plot equals full. This will just give us a more detailed plot in the results. The done function will be called, onDetailsResult, and I'll reuse the onSearchFail function for failure.

Now, all that's left is to define this onDetailsResults function. Again, that's going to be super simple. The details template is all set to work with the data that we get back from the server call. All we need to do is, use the template and data in a call to Mustache render, and again assign the result of that to the details div in our HTML.



```
function onSearchFail() {
    alert("There was a problem contacting the server. Please try again.");
}

function getDetails(event) {
    var id = event.target.id;
    $.get("http://www.omdbapi.com/?plot=full&i=" + id, null, null, "json")
        .done(onDetailsResult)
        .fail(onSearchFail);
}

function onDetailsResult(data) {
    var html = Mustache.render(detailsTemplate, data);
    detailsDiv.innerHTML = html;
}
```



- Star Wars: Episode IV - A New Hope
- Star Wars: Episode V - The Empire Strikes Back
- Star Wars: Episode VI - Return of the Jedi
- Star Wars: Episode I - The Phantom Menace
- Star Wars: Episode III - Revenge of the Sith
- Star Wars: Episode II - Attack of the Clones
- Star Wars: The Clone Wars
- Star Wars: The Clone Wars
- Star Wars: Clone Wars
- The Star Wars Holiday Special

Star Wars: Episode IV - A New Hope (1977)

Starring: Mark Hamill, Harrison Ford, Carrie Fisher, Peter Cushing

Part IV in George Lucas' epic, Star Wars: A New Hope opens with a Rebel ship being boarded by the tyrannical Darth Vader. The plot then follows the life of a simple farm boy, Luke Skywalker, as he and his newly met allies (Han Solo, Chewbacca, Obi-Wan Kenobi, C-3PO, R2-D2) attempt to rescue a Rebel leader, Princess Leia, from the clutches of the Empire. The conclusion is culminated as the Rebels, including Skywalker and flying ace Wedge Antilles make an attack on the Empire's most powerful and ominous weapon, the Death Star.

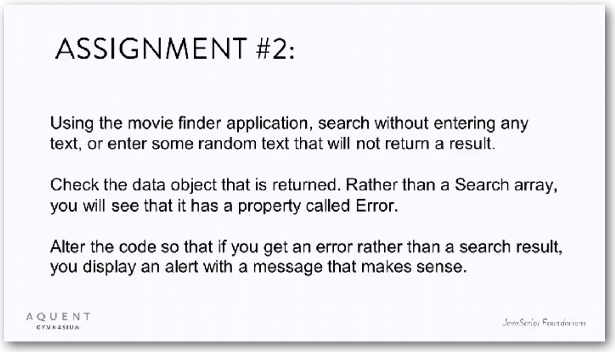
Let's try it out. I'll enter Star Wars again, and we get a list of results. Click on one, and we have all the details for that movie, including a poster and a detailed plot. So, we have the six original episodes, some Clone Wars stuff, and even the holiday special. How cool is that?

And while this is a fully functional application, take another look at the code that it took to create it. Less than 40 lines. It could be a lot shorter, too, if you used jQuery more, and wrote those functions in line.

I'm really hoping that you understand everything that's going on in this code. While it has a few more steps than the previous examples, it's all stuff that we've covered before. So, if there's anything you don't fully get, please review the earlier material, and ensure that you understand it all, before moving on. In the final lesson for this course, we'll be looking at even more complex application concepts, using a framework specifically made for building applications, backbone.js.

But, before that, you have an assignment to do. You know this one's coming. Take the quiz.

Second, using the movie finder application, search without entering any text, or enter some random text that won't return any real result. Check the data object that's returned. Rather than a search array, you'll see that it has a property called error. Alter the code, so that if you get an error result rather than a search result, display an alert or some kind of message that makes sense in that case.



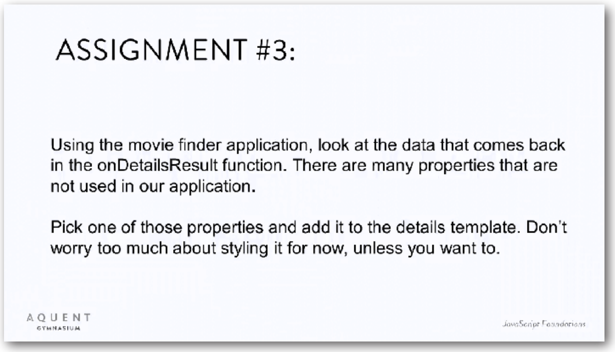
ASSIGNMENT #2:

Using the movie finder application, search without entering any text, or enter some random text that will not return a result.

Check the data object that is returned. Rather than a Search array, you will see that it has a property called Error.

Alter the code so that if you get an error rather than a search result, you display an alert with a message that makes sense.

Three, again, using the movie finder application, look at the data that comes back in the onDetailsResult function. There are many properties that are not used in our application. Pick one of those properties, and add it to the details template. Don't worry too much about styling it for now, unless you want to.



ASSIGNMENT #3:

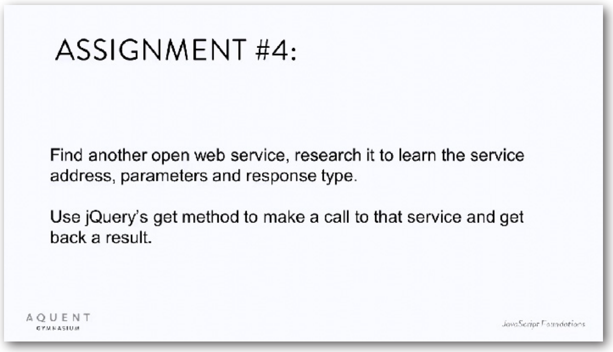
Using the movie finder application, look at the data that comes back in the onDetailsResult function. There are many properties that are not used in our application.

Pick one of those properties and add it to the details template. Don't worry too much about styling it for now, unless you want to.

AQUENT GYMNASIUM
JavaScript Foundations

Four, find another open web service. Research it to learn the service address, parameters, and response type. Use jQuery's get method to make a call to that service, and get back a result.

A good place to look for web services is the Programmable Website. This site lists, and reviews many web services, and provides information, and links about each service. You can search for sites with various filters for categories, and data types, et cetera. Without too much effort, you should be able to find an open service that interests you. You might even want to build out a full application, using one of them.



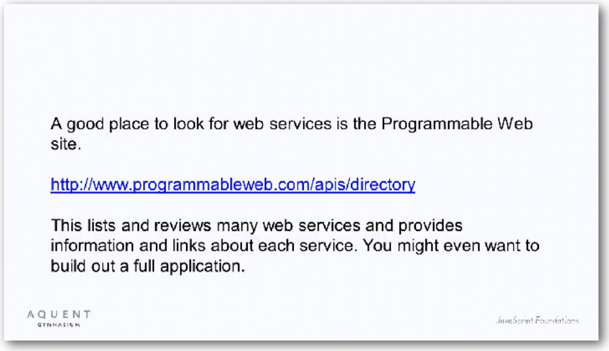
ASSIGNMENT #4:

Find another open web service, research it to learn the service address, parameters and response type.

Use jQuery's get method to make a call to that service and get back a result.

AQUENT GYMNASIUM

JavaScript Foundations



A good place to look for web services is the Programmable Web site.

http://www.programmableweb.com/apis/directory

This lists and reviews many web services and provides information and links about each service. You might even want to build out a full application.

AQUENT GYMNASIUM

JavaScript Foundations