

# jQuery Fundamentals Training

Ajax

## Lesson 1, Activity 2: Key Concepts

The XMLHttpRequest method (XHR) allows browsers to communicate with the server without requiring a page reload. This method, also known as Ajax (Asynchronous JavaScript and XML), allows for web pages that provide rich, interactive experiences.

Ajax requests are triggered by JavaScript code; your code sends a request to a URL, and when it receives a response, a callback function can be triggered to handle the response. Because the request is asynchronous, the rest of your code continues to execute while the request is being processed, so it's imperative that a callback be used to handle the response.

jQuery provides Ajax support that abstracts away painful browser differences. It offers both a full-featured `$.ajax()` method, and simple convenience methods such as `$.get()`, `$.getScript()`, `$.getJSON()`, `$.post()`, and `$.fn.load()`.

Most jQuery applications don't in fact use XML; instead, they transport data as plain HTML or JSON (JavaScript Object Notation).

In general, Ajax does not work across domains. Exceptions are services that provide JSONP (JSON with Padding) support, which allow limited cross-domain functionality.

Proper use of Ajax-related jQuery methods requires understanding some key concepts first.

### GET vs. POST

The two most common "methods" for sending a request to a server are GET and POST. It's important to understand the proper application of each.

The GET method should be used for non-destructive operations, that is, operations where you are only "getting" data from the server, not changing data on the server. For example, a query to a search service might be a GET request. GET requests may be cached by the browser, which can lead to unpredictable behavior if you are not expecting it. GET requests send their data in a query string appended to the URL.

The POST method should be used for destructive operations, that is, operations where you are changing data on the server. For example, a user saving a blog post should use a POST request. POST requests are generally not cached by the browser; the query string is usually sent as the body of the request, after the header block. POST data will be encrypted under HTTPS/SSL.

### The Same Origin Policy

The XMLHttpRequest object used for Ajax is subject to the *Same Origin Policy*, which prevents a request from being made to a server other than the one the page came from.

### Data Types

jQuery generally requires some instruction as to the type of data you expect to get back from an Ajax

request; in some cases the data type is specified by the method name, and in other cases it is provided as part of a configuration object. There are several options:

- `text` - For transporting simple strings, usually to be placed directly into the page
- `html` - For transporting blocks of HTML, usually to be placed directly into the page
- `xml` - For transporting blocks of XML, which can be parsed to provide data
- `script` - For adding a new script to the page -- this type of request uses a concept called *dynamic script tagging* instead of `XMLHttpRequest`, and is therefore not limited by the same origin policy
- `json` - For transporting JSON-formatted data, which can include JavaScript strings, arrays, and objects
- `jsonp` - A variant of JSON, which can be used to retrieve content from other servers -- this type of request also uses dynamic script tagging

Note: As of jQuery 1.4, if the JSON data sent by your server isn't properly formatted, the request may fail silently. See <http://json.org> for details on properly formatting JSON, but as a general rule, use built-in language methods for generating JSON on the server to avoid syntax issues.

It is recommended to use the JSON format in most cases, as it provides the most flexibility. It is especially useful when your server environment is PHP, since JSON is also native to that language.

## JSON and XML - what do they look like?

### JSON:

```
person = {"age" : 33, "name" : "Joshua"}
```

### XML:

```
<person>
  <age>33</age>
  <name>Joshua</name>
</person>
```

## A is for Asynchronous

The asynchronicity of Ajax catches many new jQuery users off guard. Because Ajax calls are asynchronous by default, the response is not immediately available. Responses can only be handled using a callback. So, for example, the following code will not work:

```
var response;
$.get('foo.php', function(r) { response = r; });
```

```
console.log(response); // undefined!
```

Instead, we need to pass a callback function to our request; this callback will run when the request succeeds, at which point we can access the data that it returned, if any.

```
$.get('foo.php', function(response) { console.log(response); });
```

## Same-Origin Policy and JSONP

In general, Ajax requests are limited to the same protocol (http or https), the same port, and the same domain as the page making the request. This limitation does not apply to scripts that are loaded via jQuery's Ajax methods.

The other exception is requests targeted at a JSONP service on another domain. In the case of JSONP, the provider of the service has agreed to respond to your request with a script that can be loaded into the page using a `<script>` tag, thus avoiding the same-origin limitation; that script will include the data you requested, wrapped in a callback function you provide.

## Ajax and Firebug

Firebug (or the Webkit Inspector in Chrome or Safari, or IE's Developer Tools) is an invaluable tool for working with Ajax requests. You can see Ajax requests as they happen in the *Console* or *Net* tabs of Firebug (and in the *Resources* > *XHR* panel of Webkit Inspector), and you can click on a request to expand it and see details such as the request headers, response headers, response content, and more. If something isn't going as expected with an Ajax request, this is the first place to look to track down what's wrong.

## Lesson 1, Activity 3: jQuery's Ajax-Related Methods

While jQuery does offer many Ajax-related convenience methods, the core `$.ajax` method is at the heart of all of them, and understanding it is imperative. We'll review it first, and then touch briefly on the convenience methods.

The `$.ajax` method offers features that the convenience methods do not. Once you have gained some experience with Ajax in jQuery, it is easiest to just use this one method.

### `$.ajax`

jQuery's core `$.ajax` method is a powerful and straightforward way of creating Ajax requests. It takes a configuration object that contains all the instructions jQuery requires to complete the request. The `$.ajax` method is particularly valuable because it offers the ability to specify both success and failure callbacks. Also, its ability to take a configuration object that can be defined separately makes it easier to write reusable code.

```
$.ajax(options)
$.ajax(url, options)
```

### Options for `$.ajax`

There are many, many options for the `$.ajax` method, which is part of its power. For a complete list of options, visit <http://api.jquery.com/jQuery.ajax/>.

Note that the url can either be provided as a separate parameter or as part of the options object.

Here are some of the more commonly used options:

- `url` - The URL for the request. Required either as an option or as a separate parameter.
- `type` - The type of the request, "POST" or "GET". Defaults to "GET". Other request types, such as "PUT" and "DELETE" can be used, but they may not be supported by all browsers.
- `async` - Set to `false` if the request should be sent synchronously. Defaults to `true`. Note that if you set this option to `false`, your request will block execution of other code until the response is received.
- `cache` - Whether to use a cached response if available. Defaults to `true` for all data types except `script` and `jsonp`. When set to `false`, the URL will simply have a cachebusting parameter appended to it.
- `success` - A callback function to run if the request succeeds. The function receives the response data (converted to a JavaScript object if the data type was JSON), as well as the text status of the request and the raw request object.
- `error` - A callback function to run if the request results in an error. The function receives the raw request object and the text status of the request.
- `complete` - A callback function to run when the request is complete, regardless of success or failure. The function receives the raw request object and the text status of the request. This function

will run after any error or success function, if those are also specified.

- **context** - The scope in which the callback function(s) should run (i.e. what `this` will mean inside the callback function(s)). By default, `this` inside the callback function(s) refers to the object originally passed to `$.ajax`.
- **data** - The data to be sent to the server. This can either be an object, like `{ foo: 'bar', baz: 'bim' }`, or a query string, such as `foo=bar&baz=bim`.
- **dataType** - The type of data you expect back from the server. By default, jQuery will look at the *MIME-type* (from the `Content-Type` header) of the response if no data type is specified.
- **jsonp** - The callback parameter name to send in a query string when making a JSONP request. Defaults to `callback`. (jQuery will add a parameter `callback=XXXXXX` to the query string; this option sets the parameter name, and the following option sets the parameter value).
- **jsonpCallback** - The value of the callback parameter to send in a query string when making a JSONP request. Defaults to an auto-generated random value.
- **timeout** - The time in milliseconds to wait before considering the request a failure.

The `url` option is the only required property of the `$.ajax` configuration object; all other properties are optional.

## Using the Core `$.ajax` Method

```
$.ajax({
  // the URL for the request
  url : 'post.php',

  // the data to send
  // (will be converted to a query string)
  data : { id : 123 },

  // whether this is a POST or GET request
  type : 'GET',

  // the type of data we expect back
  dataType : 'json',

  // code to run if the request succeeds;
  // the response is passed to the function
  success : function(json) {
    $('<h1/>').text(json.title).appendTo('body');
    $('<div class="content"/>')
      .html(json.html).appendTo('body');
  },

  // code to run if the request fails;
  // the raw request and status codes are
  // passed to the function
  error : function(xhr, status) {
    alert('Sorry, there was a problem!');
  },

  // code to run regardless of success or failure
  complete : function(xhr, status) {
```

```

    alert('The request is complete!');
  }
});

```

Note: A note about the `dataType` setting: if the server sends back data that is in a different format than you specify, your code may fail, and the reason will not always be clear, because the HTTP response code will not show an error. When working with Ajax requests, make sure your server is sending back the data type you're asking for, and verify that the `Content-Type` header is accurate for the data type. For example, for JSON data, the content type should be `application/json`.

## Convenience Methods

If you don't need the extensive configurability of `$.ajax`, and you don't care about handling errors, the Ajax convenience functions provided by jQuery can be useful, terse ways to accomplish Ajax requests. These methods are just "wrappers" around the core `$.ajax` method, and simply pre-set some of the options on the `$.ajax` method.

The convenience methods provided by jQuery are:

- `$.get` - Perform a GET request to the provided URL.
- `$.post` - Perform a POST request to the provided URL.
- `$.getScript` - Add a script to the page.
- `$.getJSON` - Perform a GET request, and expect JSON to be returned.

In each case, the methods take the following arguments, in order:

- `url` - The URL for the request. Required.
- `data` - The data to be sent to the server. Optional. This can either be an object or a query string, such as `foo=bar&baz=bim`.
  - Note: This option is not valid for `$.getScript`.
  - Also, this parameter is truly optional - if you do not want to supply it, you don't have to supply a placeholder value (jQuery looks at the type of the second parameter to determine if it is your parameters; if the second parameter is a function, it uses it as the success callback described below)
- `success` - A callback function to run if the request succeeds. Optional. The function receives the response data (converted to a JavaScript object if the data type was JSON), as well as the text status of the request and the raw request object.
- `dataType` - The type of data you expect back from the server. Optional. (Note: This option is only applicable for methods that don't already specify the data type in their name.)

## Using jQuery's Ajax Convenience Methods

```
// get plain text or html
```

```
$.get('/users.php', { userId : 1234 }, function(resp) {
    console.log(resp);
});

// add a script to the page, then run a function defined in it
$.getScript('/js/myScript.js', function() {
    functionFromMyScript();
});

// get JSON-formatted data from the server
// resp will be a single object parsed from the incoming JSON
$.getJSON('/details.php', function(resp) {
    $.each(resp, function(k, v) {
        console.log(k + ' : ' + v);
    });
});
```

## \$.fn.load

The `$.fn.load` method is unique among jQuery's Ajax methods in that it is called on a selection. The `$.fn.load` method fetches HTML from a URL, and uses the returned HTML to populate the selected element(s). In addition to providing a URL to the method, you can optionally provide a selector as part of the URL parameter; jQuery will fetch only the matching content from the returned HTML.

## Using \$.fn.load to Populate an Element

```
$('#newContent').load('/foo.html');
```

## Using \$.fn.load to Populate an Element Based on a Selector

```
$('#newContent').load('/foo.html #myDiv h1:first', function(html) {
    alert('Content updated!');
});
```

## Ajax Examples

### Code Sample:

---

#### [jqy-ajax/Demos/ajax-text.html](#)

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<title>Testing Ajax with HTML</title>
<script src="../../jqy-lib/jquery.js"></script>
<script>
$(document).ready(function() {
```



```

$.ajax("data/blurb.html", {
  type: "get",
  dataType: "html",
  success: function(data) {
    $('#results1').html(data);
  },
  error: function(xhr, status, err) {
    $('#results1').html("Error " + status + ", " + err);
  }
} );
$('#results2').load("data/blurb.html");
});
</script>
</head>
<body>
<h1>Here is the Ajax we received via $.ajax:</h1>
<div id="results1"></div>
<h1>Here is the Ajax we received via $.load:</h1>
<div id="results2"></div>
</body>
</html>

```

We load the HTML text two ways, using `$.ajax` and `$.fn.load`.

Using `$.ajax`, we supply an options object specifying the GET method, HTML data, and both `success` and `error` functions. You can test the error capability by changing the URL to a non-existent file name.

The `$.fn.load` approach is simpler -- we identify the receiving element, and simply load the file.

## Code Sample:

---

### [jqy-ajax/Demos/ajax-json.html](#)

```

<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<title>Testing Ajax with JSON</title>
<script src="../../jqy-lib/jquery.js"></script>
<script>
$(document).ready(function() {
  $.ajax("data/people.json", {
    type: "get",
    dataType: "json",
    success: function(data) {
      var $list = $('<ol/>').appendTo('#results');
      $.each(data, function() {
        $list.append(
          '<li>' + this.firstName + ' ' + this.lastName + '</li>');
      });
    },
  },

```

```

    error: function(xhr, status, err) {
        $('#results').html("Error " + status + ", " + err);
    }
} );
});
</script>
</head>
<body>
<h1>Here are the contents of our JSON data:</h1>
<div id="results"></div>
</body>
</html>

```

This example is similar to the previous one, but we can't use `$.fn.load` for JSON data.

This time we supply an options object specifying the GET method, JSON data, and again both `success` and `error` functions.

We based our treatment of the JSON data upon our knowledge that it is an array of objects, each with two fields: `firstName` and `lastName`.

You can test the JSON-related aspects of the error capability by munging the data file.

## Lesson 1, Activity 5: Ajax and Forms

jQuery's Ajax capabilities can be especially useful when dealing with forms. The jQuery Form Plugin is a well-tested tool for adding Ajax capabilities to forms, and you should generally use it for handling forms with Ajax rather than trying to roll your own solution for anything remotely complex. That said, there are two jQuery methods you should know that relate to form processing in jQuery: `$.fn.serialize` and `$.fn.serializeArray`.

### Turning Form Data into a Query String

```
$('#myForm').serialize();
```

### Creating an Array of Objects Containing Form Data

```
$('#myForm').serializeArray();

// creates a structure like this:
[
  { name : 'field1', value : 123 },
  { name : 'field2', value : 'hello world' }
]
```

## Lesson 1, Activity 6: Working with JSONP

The advent of JSONP -- essentially a consensual cross-site scripting hack -- has opened the door to powerful mashups of content. Many prominent sites provide JSONP services, allowing you access to their content via a predefined API. An example source of JSONP-formatted data is Twitter, which we'll use in the following example to fetch tweets about jQuery.

### Using JSONP

#### Code Sample:

---

[jqy-ajax/Demos/ajax-jsonp.html](http://jqy-ajax/Demos/ajax-jsonp.html)

```

---- C O D E   O M I T T E D ----
<body>
<ul id="results">
</ul>
<script>
$.ajax({
  url : 'http://search.twitter.com/search.json',

  // the name of the callback parameter,
  // as specified by the Twitter service
  jsonp : 'callback',

  // tell jQuery we're expecting JSONP
  dataType : 'jsonp',

  // tell Twitter what we want using their specified parameters
  data : {
    q : 'jQuery',
    rpp : 5
  },

  // work with the response, knowing that the returned object
  // contains an array called results, and each result element
  // has, among other things, a field named text
  success : function(response) {
    var $results = $('#results');
    var data = response.results;
    for (var i = 0; i < data.length; i++) {
      $results.append('<li>' + data[i].text + '</li>');
    };
  },
  // work with the response
  error : function(response) {
    console.log(response);
  }
});
</script>
</body>

```

```
</html>
```

jQuery handles all the complex aspects of JSONP behind-the-scenes -- all we have to do is tell jQuery the name of the JSONP callback parameter specified by Twitter ("callback" in this case), and otherwise the whole process looks and feels like a normal Ajax request.

## Lesson 1, Activity 8: Ajax Events

Often, you'll want to perform an operation whenever an Ajax requests starts or stops, such as showing or hiding a loading indicator. Rather than defining this behavior inside every Ajax request, you can bind Ajax events to elements just like you'd bind other events. For a complete list of Ajax events, visit [http://docs.jquery.com/Ajax\\_Events](http://docs.jquery.com/Ajax_Events).

### Creating a Loading Indicator Using Ajax Events

```
$('#loading_indicator')  
  .ajaxStart(function() { $(this).show(); })  
  .ajaxStop(function() { $(this).hide(); });
```

Note: the `$.fn.ajaxStart` and `$.fn.ajaxStop` functions are not invoked for JSONP requests.

## Lesson 1, Activity 9: Load External Content

Duration: 20 to 30 minutes.

Open `ClassFiles\jqy-ajax\Exercises\index.html` in your browser. Edit the file `js\load.js`. Your task is to load the content of a blog item when a user clicks on the title of the item. The current behavior is that clicking the link takes you from this page to the blog page, but we would prefer to stay on this page and use Ajax to retrieve that content.

1. Create a target `div` after the headline for each blog post and store a reference to it on the headline element. You can either use `$.fn.data`, or create a closure on a local variable when you define the click handler in the next step. (It will be easiest to use the `$.fn.each` function to do this step.)
2. Bind a click event to the headline that will use the `$.fn.load` method to load the appropriate content from `exercises/data/blog.html` into the target `div`. Don't forget to prevent the default action of the click event.

Note that each blog headline in the page includes a link to the post. You'll need to leverage the `href` of that link to get the proper content from `blog.html`. Once you have the `href`, here's one way to process it into an ID that you can use as a selector in `$.fn.load`:

```
var href = 'data/blog.html#post1';
var url = href.replace('#', ' #');
```

As a challenge, try to:

1. use event delegating so that there is only one click handler.
2. merge this with the sliding code from the effects chapter (slide the Ajax content up and down, as opposed to the excerpts).

### Solution:

---

<jqy-ajax/Solutions/js/load.js>

```
$(document).ready(
function() {
  $('#blog h3').each(
function() {
  var $target = $('<div />');
  var $this = $(this);
  $this
    .after($target)
    .click(
function(e) {
  e.preventDefault();
  var url = $this.find('a:first')
    .attr('href').replace('#', ' #');
  $target.load(url);

```

```

    }
  );
}
);
}
);

```

For each headline in the blog, we create an empty target, insert it after the headline, and then establish a click handler.

The click handler finds the hyperlink within the headline, modifies the url to insert a space before the # (our page was set up specifically for this purpose -- the original version would take the user to that page, scrolled to that anchor. The anchor is inside a `div` with the same value as its id, so jQuery can use the same # to pull out just that div's content.

Because of the hyperlink in the original code, we prevent the default behavior to avoid actually going to the blog page now that we can retrieve it via Ajax. (Note that even though we caught the event in the hyperlink's container tag, the entire bubbling process takes place before the default behavior, and it doesn't matter which element we actually use to invoke `preventDefault`).

### Challenge Solution:

---

[jqy-ajax/Solutions/js/load-delegate-slide.js](http://jqy-ajax/Solutions/js/load-delegate-slide.js)

```

$(document).ready(function() {
  var $blogTitles = $('#blog>ul>li>h3:first-child');
  $blogTitles.each(function() {
    $.data(this, 'content', $('').insertAfter(this));
  });
  $('#blog>ul').delegate('li>h3:first-child', 'click', function(e) {
    e.preventDefault();
    var $content = $.data(this, 'content');
    $content.load(
      $(e.target).attr('href').replace('#', ' #'),
      function() {
        $content.slideDown(500, 'linear');
        $content.parent().siblings().find('div.ajaxContent:visible').slideUp(500, 'linear');
      }
    );
  });
});

```

We set up a delegating on the a click handler on the unordered list within the blog section. The subquery then finds the first h3 child of each list item. Essentially, we split the previous query into two sections.



To handle the sliding, we provided a callback function to `$.fn.load`, which slides down the content and slides up sibling content.

## Lesson 1, Activity 10: Load Content Using JSON

Duration: 20 to 30 minutes.

Open [ClassFiles\jqy-ajax\Exercises\index.html](#) in your browser. Uncomment the tag for [specials.js](#). Your task is to show the user details about the special for a given day when the user selects a day from the select dropdown.

1. Append a target div after the form that's inside the `#specials` element; this will be where you put information about the special once you receive it.
2. Bind to the change event of the select element; when the user changes the selection, send an Ajax request to [ClassFiles\jqy-ajax\Exercises\data\specials.json](#).
3. When the request returns a response, use the value the user selected in the select (hint: `$.fn.val`) to look up information about the special in the JSON response.
4. Add some HTML about the special to the target div you created.
5. Finally, because the form is now Ajax-enabled, remove the submit button from the form.

Note that we're loading the JSON every time the user changes their selection. How could we change the code so we only make the request once, and then use a cached response when the user changes their choice in the select?