**Working with JavaScript in Drupal** <https://drupal.org/node/121997>

**Drupal Javascript API** <https://drupal.org/node/304258>

Adding JavaScript to your theme or module

The Drupal API function [drupal\_add\_js()](https://api.drupal.org/api/drupal/includes%21common.inc/function/drupal_add_js/7) lets you add a JavaScript file, setting or inline code to the page and it takes 5 parameters (see the api reference).

The first parameter is always going to be a path to a js file, an array, or a piece of JavaScript code. If the second parameter is 'module' (the default), 'theme', or 'core', then the first parameter must be a path and the only difference between these three is the order in which the script tag will be placed relative to other scripts, i.e. core scripts first, then module scripts and finally theme scripts.

If the second parameter is 'setting', then the first parameter must be an array of settings. This is very handy for telling JavaScript about the configuration settings for your module, for example, and is explained in more detail in the [next section](https://drupal.org/node/304258#drupal-settings).

The final possibility for the second parameter is "inline" and this means you are passing in straight JavaScript code as your first parameter, which you want written directly to your page between script tags, rather than a call to a file.

These two are the most important parameters for this function - indeed you may never even use the second one if all you ever want to do is add a js file for your module - but you can learn about the other parameters by reading this [function's documentation at api.drupal.org](http://api.drupal.org/api/function/drupal_add_js/).

## Adding JavaScript from within a module

So, let's say you have written some jQuery code that targets some of the elements that are output by your module or theme. You simply need to save your jQuery code as a JavaScript file (i.e. with the .js extension) and place it in your module or theme's directory. If it's for a module, you then need to make the call to drupal\_add\_js() as above, from wherever the module is about to output content (e.g. in hook\_block or hook\_nodeapi), or if the JS is not acting on content output by your own module you can call it from hook\_menu or hook\_init. Note, that putting this file into hook\_init means it will run on every page request, whether needed or not. Here is how you would ensure the file gets added:

drupal\_add\_js(drupal\_get\_path('module', 'mymodule') . '/mymodule.js');

The jquery.js file is included automatically so you don't need to worry about adding it.

To add JS (or CSS) file from form builder function the recommended way is to use Form API #attached property as below,

$form['#attached']['js'] = array(

drupal\_get\_path('module', 'ajax\_example') . '/ajax\_example.js',

);

See [Form API reference](http://api.drupal.org/api/drupal/developer%21topics%21forms_api_reference.html/7#attached) for more details.

## Adding JavaScript from within a theme

### With drupal\_add\_js

(drupal\_add\_js is deprecated in Drupal 8; use '#attached' instead.) If you are adding your .js file from within a theme, then you simply need to make the call from within **template.php**, you need to add the second parameter this time as 'theme' so that Drupal knows this script should be loaded after all core and module scripts are loaded:

drupal\_add\_js(drupal\_get\_path('theme', 'mytheme') . '/mytheme.js', array(

'type' => 'file',

'group' => JS\_THEME,

));

### Using the theme's .info file

You can add your script path in the .info file of your theme and Drupal will include them automatically.

name = My theme

description = Theme developed by me.

core = 7.x

engine = phptemplate

scripts[] = mytheme.js

### Using a preprocess\_page function

To conditionally add js to the theme, you can use a [preprocess\_page function](http://drupal.org/node/223430):

function mytheme\_preprocess\_page(&$vars, $hook) {

if (true) {

drupal\_add\_js(drupal\_get\_path('theme', 'mytheme') . '/mytheme.js');

$vars['scripts'] = drupal\_get\_js(); // necessary in D7?

}

}

# JavaScript API overview

This page is a description of how JavaScript is implemented in Drupal, including an in-depth look at the drupal.js file and in particular the Drupal js object initialized therein.

A couple of simple principles guide Drupal's JavaScript approach:

* All pages should be perfectly functional without scripts. JavaScript provides alternatives or supplements, not replacements, for standard elements.
* No JavaScript is hard-coded into pages. Rather, actions are attached dynamically to page elements, and only if the needed JavaScript support is present.

Note: The following examples are lacking JSDoc, only for clarity.

The very first line of JavaScript code in Drupal core, in [drupal.js](http://cgit.drupalcode.org/drupal/tree/misc/drupal.js?id=refs/heads;id2=7.x), is an Object declaration:

var Drupal = Drupal || { 'settings': {}, 'behaviors': {}, 'themes': {}, 'locale': {} };

In this code, Drupal is an Object declared to be equal to itself, or, if not yet set, equal to { 'settings': {}, 'behaviors': {}, 'themes': {}, 'locale': {} }which is an Object containing 4 properties (settings, behaviors, themes, and locale) each of which is itself an Object. This line of code is an [Object Initializer](https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide/Working_with_Objects#Using_Object_Initializers). This Drupal object and its properties can then be used and extended by other modules. The best way to understand this is to look at the different properties one by one and the ways they are used by Drupal modules.

Jump to a section:  
[Drupal.settings](https://www.drupal.org/docs/7/api/javascript-api/javascript-api-overview#drupal-settings)  
[Drupal.behaviors](https://www.drupal.org/docs/7/api/javascript-api/javascript-api-overview#drupal-behaviors)  
[Drupal.theme](https://www.drupal.org/docs/7/api/javascript-api/javascript-api-overview#drupal-theme)  
[Drupal.locale](https://www.drupal.org/docs/7/api/javascript-api/javascript-api-overview#drupal-locale)

### Drupal.settings

Drupal.settings is what enables us to pass information from our PHP code to our JavaScript code. This means you can change how your JavaScript behaves based on your module. For example, you may want to simply let JavaScript know what the base path is. In order to do this, you just create a PHP array of settings, as follows:

$my\_settings = array(

'basePath' => $base\_path,

'animationEffect' => variable\_get('effect', 'none')

);

**Note:** The array keys are set using CamelCasing according to the [JavaScript coding standards](https://drupal.org/node/172169).  
@see: <https://drupal.org/node/172169#camelcasing>

Then call drupal\_add\_js() and pass in this array, with "setting" as your second parameter:

drupal\_add\_js(array('myModule' => $my\_settings), 'setting');

(Drupal 8 deprecates drupal\_add\_js; refer to [Attaching Configurable JavaScript in Drupal 8](https://www.drupal.org/theme-guide/8/assets#configurable-javascript) for the D8 way to pass values from PHP to JavaScript.)

Note that it is further padded inside another array purely for namespacing purposes: another module might define the basePath setting as well. Now you can access these settings in your JavaScript code as follows:

var basePath = Drupal.settings.myModule.basePath;

var effect = Drupal.settings.myModule.animationEffect;

These are strings, but not string objects in JavaScript. The value of the array key you pass into drupal\_add\_js() will be concatenated to the end of this string separated by a comma.

**Note**: [Drupal 7 passes the settings locally](http://drupal.org/node/224333#local_settings_behaviors)

### Drupal.behaviors

When most of us learn jQuery for the first time, we learn to put all our code inside the $(document).ready() function, like this:

$(document).ready(function () {

// Do some fancy stuff.

});

This ensures that our code will get run as soon as the DOM has loaded, manipulating elements and binding behaviors to events as per our instructions. However, as of Drupal 6, we don't need to include the $(document).ready()function in our jQuery code at all. Instead we put all our code inside a function that we assign as a property of Drupal.behaviors. The Drupal.behaviors object is itself a property of the Drupal object, as explained above, and when we want our module to add new jQuery behaviors, we simply extend this object. The entire jQuery code for your module could be structured like this:

Drupal.behaviors.myModuleBehavior = {

attach: function (context, settings) {

$('input.myCustomBehavior', context).once('myCustomBehavior', function () {

// Apply the myCustomBehaviour effect to the elements only once.

});

}

};

@see: <https://www.drupal.org/update/modules/6/7#jquery_once>

**Any function defined as a property of Drupal.behaviors will get called when the DOM has loaded**. drupal.js has a $(document).ready() function which calls the Drupal.attachBehaviors() function, which in turn cycles through the Drupal.behaviors object calling every one of its properties, these all being functions declared by various modules as above, and passing in the document as the context.

The reason for doing it this way is that if your jQuery code makes AJAX calls which result in new DOM elements being added to the page, you might want your behaviors (e.g. hiding all h3 elements or whatever) to be attached to that new content as well. But since it didn't exist when the DOM was loaded and Drupal.attachBehaviors() ran, it doesn't have any behaviors attached. With the above set-up, though, all you need to do is call Drupal.behaviors.myModuleBehavior(newcontext), where newcontext would be the new, AJAX-delivered content, thus ensuring that the behaviors don't get attached to the whole document all over again. There are full instructions on how to use this code on the [Converting 5.x modules to 6.x](http://drupal.org/node/114774#javascript-behaviors) or [Converting 6.x modules to 7.x](https://drupal.org/update/modules/6/7) page.

#### Drupal.behaviors practical example

The following is a more practical example. Drupal Behaviors are fired whenever attachBehaviors is called. The context variable that is passed in can often give you a better idea of what DOM element is being processed, but it is not a sure way to know if you are processing something again. **Passing the context variable as the second argument to the jQuery selector is a good practice** because then only the given context is searched and not the entire document. This becomes more important when attaching behaviors after an AJAX request. The following is an example of a Drupal.behavior that ensures that processing only happens once per DOM object.

Drupal.behaviors.myModuleBehavior = {

attach: function (context, settings) {

// This jQuery code ensures that this element

// is only processed once. It is basically saying:

// 1) Find all elements with this class, that do not

// have the processed class on it

// 2) Iterate through them

// 3) Add the myCustomBehavior-processed class (so that it will not

// be processed again).

$('input.myCustomBehavior', context).once('myCustomBehavior', function () {

// Apply the myCustomBehaviour effect to the elements only once.

});

}

};

Note: Your included JS file needs to have the function($) prototype definition which is not mentioned above! So my WHOLE file would look like

(function ($) {

Drupal.behaviors.myModuleBehavior = {

attach: function (context, settings) {

$('input.myCustomBehavior', context).once('myCustomBehavior', function () {

// Apply the myCustomBehaviour effect to the elements only once.

});

}

};

})(jQuery);

### Drupal.theme

Drupal.theme() is the client-side counterpart to the server-side theme()function. Here's what it looks like in Drupal 7:

Drupal.theme = function (func) {

for (var i = 1, args = []; i < arguments.length; i++) {

args.push(arguments[i]);

}

return (Drupal.theme[func] || Drupal.theme.prototype[func]).apply(this, args);

};

So, when you make a call to Drupal.theme(), you pass in a function name as your first argument and all subsequent arguments will be arguments to be passed to that function. The function you pass in will need to be a member of Drupal.theme() object, an example of which is below:

Drupal.theme.myThemeFunction = function (left, top, width) {

var myDiv = '<div id="myDiv" style="left:'+ left +'px; top:'+ top +'px; width:'+ width +'px;">';

myDiv += '</div>';

return myDiv;

};

And here's how you would call it:

Drupal.theme('myThemeFunction', 50, 100, 500);

### Drupal.locale

The Drupal.locale property works in conjunction with Drupal.t(), the JavaScript equivalent of the server-side t() function. It holds a collection of string translations so that Drupal.t() can then access the required string from Drupal.locale in order to translate what was passed into it.

# Introducing jQuery

[jQuery](http://jquery.com/) is a cross-browser JavaScript library, which means it provides some pre-defined functions to make your life easier. Drupal integrates jQuery by default, you can use it in any module or theme without any need to loading it.

The most current versions Drupal 7 is compatible with are jQuery 1.4.2 and jQuery UI 1.8.

This is a short overview of how you can use jQuery. For further guidance see [jQuery documentation](http://docs.jquery.com/) itself.

## Selectors

jQuery makes it easy to select DOM elements. It allows you to very simply "query" the DOM for the set of elements you wish to manipulate using selector. You can also use CSS or xPath selector expression.

Here are some examples of jQuery selector expressions:

**CSS selectors**

// select all 'a' elements

 $('a')

//select the element with the id 'container'

$('#container')

// select all div elements with the class 'ajaxContainer'

 $('div.ajaxContainer')

// select all li elements that are the first child of their parent

$('li:first-child')

**X-Path selectors**

// select all 'a' elements that have a title attribute

$('a[title]')

// select all 'a' elements whose href attribute begins with 'mailto'

 $('a[href^="mailto:"]')

**Custom selectors**

// select the second 'li' element

 $('li:eq(1)')

// select all odd 'tr' elements that don't contain a 'th' element

$('tr:not([th]):odd')

## Commands and effects

Commands is what you can do to your elements once you've selected them.

//hide the element #container

$('#container').hide();

## Events

Events correspond to user interaction of some sort, the most obvious being a mouse click. But the loading of your HTML page is also an event, and an extremely important one at that, as we shall see. As for commands, these fall into a few distinct categories, such as commands for DOM traversal, commands for manipulation of DOM elements, effects and AJAX commands.

//when we click on a link inside #menu

$('#menu a').click(function() {

//display an alert

alert('Handler for .click() called.');

//show the element #container

$('#container').show();

});

## Dom ready

**Drupal uses its own API "Drupal.behaviors" to do that, please read**[**The Drupal JavaScript API**](https://www.drupal.org/node/304258)

The DOM elements required won't even have loaded in the page yet by the time this code gets executed, so we need some way of ensuring that it doesn't get executed until the elements it acts upon are ready.

jQuery has a way of ensuring the code gets run as soon as just the DOM has loaded, i.e. just the HTML skeleton of the page, so it doesn't have to wait for large image and media files to load. Here's what it looks like:

$(document).ready(function(){

// your jQuery code goes here

});

# Managing JavaScript in Drupal 7

Drupal 7 has introduced several new techniques that allow you far greater flexibility and control in the scripts you can have on your Drupal site's pages.

### Adding JavaScript

In Drupal 7, there are four primary methods of adding JavaScript to Drupal.

#### Method 1: Module/Theme .info file

JavaScript files can be added to a module and/or theme .info files. When adding JavaScript using this method, it will be included on every page. This allows Javascript to be aggregated in an optimal way, and is the preferred method of adding Javascript that most visitors will need on most/all pages on a typical site visit. Example:

scripts[] = somescript.js

#### Method 2: drupal\_add\_js()

JavaScript can be added in any function using [drupal\_add\_js()](https://api.drupal.org/api/drupal/includes!common.inc/function/drupal_add_js/7). See the rest of this page and the documentation page for the various options available when calling this function. Example:

drupal\_add\_js('path/to/somescript.js');

#### Method 3: #attached

Drupal 7 makes heavy use of [render arrays](https://www.drupal.org/node/930760). Any element of a render array can have the [#attached](https://api.drupal.org/api/drupal/developer!topics!forms_api_reference.html/7#attached) property, and this property can be used to attach JavaScript to that element. Example:

$element['#attached']['js'][] = array(

'type' => 'file',

'data' => 'path/to/somescript.js',

);

Note that when possible, this method is preferred over drupal\_add\_js(), as the reference to the file is cached, whereas drupal\_add\_js() requires a function call to add the script to the page. In some instances, such as hook\_block\_view(), drupal\_add\_js() will not be called if the block is cached, and therefore #attached must be used to ensure your code is loaded with the block.

#### Method 4: Libraries

Drupal Libraries provide a standardized method of adding collections of JavaScript and CSS. Libraries can, for example, be used to integrate jQuery plugins with Drupal. If you have a set of JavaScript and/or CSS that could be considered a package, it can be provided as a library to other modules by implementing [hook\_library()](http://api.drupal.org/api/drupal/modules!system!system.api.php/function/hook_library/7). It can then be included by using either ['#attached']['library'] or [drupal\_add\_library()](http://api.drupal.org/api/drupal/includes!common.inc/function/drupal_add_library/7). This is the preferred way to integrate with JavaScript and CSS which might be used by other modules.

An example of a system provided library can be found in system.module, which defines the Vertical Tabs library. This library includes one JavaScript file and one CSS file. The library is defined as follows:

function system\_library() {

...

// Vertical Tabs.

$libraries['vertical-tabs'] = array(

'title' => 'Vertical Tabs',

'website' => 'http://drupal.org/node/323112',

'version' => '1.0',

'js' => array(

'misc/vertical-tabs.js' => array(),

),

'css' => array(

'misc/vertical-tabs.css' => array(),

),

);

...

return $libraries;

}

The library can then be added using one of the following methods.  
drupal\_add\_library() example:

// Module name: system

// Library name: vertical-tabs

drupal\_add\_library('system', 'vertical-tabs');

#attached example:

// Module name: system

// Library name: vertical-tabs

$element['#attached']['library'][] = array('system', 'vertical-tabs');

#### Do not directly add library files

drupal\_add\_js() and [#attached]['js'] should not be used for JavaScript which is part of a library (as all core JavaScript libraries are). Use drupal\_add\_library(), or ['#attached']['library'] instead.

### Weighted JavaScript

When adding JavaScript, you can add a weighted "group" value to each script so you can control the order in which JavaScript is outputted on the page. The following constants are available to be used as values when using the "group" value:

* JS\_LIBRARY: Any libraries, settings, or jQuery plugins.
* JS\_DEFAULT: Any module-layer JavaScript.
* JS\_THEME: Any theme-layer JavaScript

Groups will be added to the page in the above order. All files in the JS\_LIBRARY group will be added first, followed by files in the JS\_DEFAULT group, and finally files in the JS\_THEME group. JavaScript files added in modules are automatically added to the JS\_DEFAULT group, and JavaScript files added in themes are automatically added to the JS\_THEME group.

You can also use the "weight" property to fine-tune the output order of JavaScript files. The weight will affect files that have the same 'scope', 'group', and 'every\_page' levels.

drupal\_add\_js() example:

drupal\_add\_js(

'jQuery(document).ready(function () { alert("Hello!"); });',

array(

'type' => 'inline',

'scope' => 'footer',

'group' => JS\_THEME,

'weight' => 5,

)

);

#attached example:

$element['#attached']['js'][] = array(

'type' => 'inline',

'data' => 'jQuery(document).ready(function () { alert("Hello!"); });',

'scope' => 'footer',

'group' => JS\_THEME,

'weight' => 5,

);

### External JavaScript

External scripts can be included as follows.  
drupal\_add\_js() example:

drupal\_add\_js('http://example.com/example.js', 'external');

#attached example:

$element['#attached']['js'][] = array(

'type' => 'external',

'data' => 'http://example.com/example.js',

);

### Passing values from PHP to Javascript with "Drupal.settings"

You can easily make variables from PHP available to Javascript on the front end with Drupal.settings.

drupal\_add\_js() example:

drupal\_add\_js(array('myModule' => array('key' => 'value')), 'setting');

#attached example:

$element['#attached']['js'][] = array(

'type' => 'setting',

'data' => array('myModule' => array('key' => 'value')),

);

This will be available in Javascript as:

if (Drupal.settings.myModule.key === 'value') {

alert('Got it!');

}

**NB:** Drupal.settings may not be available until after your javascript has been attached to the page, which can result in undefined errors if it's being immediately called.

### Overriding JavaScript

[hook\_js\_alter()](http://api.drupal.org/api/drupal/modules%21system%21system.api.php/function/hook_js_alter/7) allows you to modify the path referenced by one of the scripts added by core or another module. An obvious example is if you want to use a newer version of jQuery than what comes with core:

function hook\_js\_alter(&$javascript) {

$javascript['misc/jquery.js']['data'] = drupal\_get\_path('module', 'jquery\_update') . '/jquery.js'; // Swap out jQuery to use an updated version of the library

}

### Behaviors

Behavior handling has [changed](http://drupal.org/update/modules/6/7#drupal_behaviors) in Drupal 7, with modules now required to explicitly define their attach handler, and optionally specify a detach handler.

Instead of the settings being a global object, [settings are now passed to your handlers directly](http://drupal.org/update/modules/6/7#local_settings_behaviors), after the context.

In Drupal 6, Drupal.behaviors was structured as follows:

Drupal.behaviors.exampleModule = function (context) {

// Code to be run on page load, and

// on ajax load added here

}

In Drupal 7, Drupal.behaviors is structured as follows:

(function ($) {

Drupal.behaviors.exampleModule = {

attach: function (context, settings) {

// Code to be run on page load, and

// on ajax load added here

}

};

}(jQuery));

These behaviors are instigated as you declare them in JavaScript so to clarify, there is no need for defining behaviors within a theme's or module's PHP file.

#### Additional References

Lullabot published a comprehensive [article](https://www.lullabot.com/blog/article/understanding-javascript-behaviors-drupal) regarding Drupal behaviors.

### Using jQuery

[jQuery is now namespaced](http://drupal.org/update/modules/6/7#javascript_compatibility) to avoid conflicts with other Javascript libraries such as Prototype. All your code that expects to use jQuery as $ should be wrapped in an outer context like so.

(function ($) {

// All your code here

}(jQuery));

If you don't, you may see the error Uncaught TypeError: Property '$' of object [object DOMWindow] is not a function or similar.

### jQuery $.once()

Drupal.behaviors will often be called multiple times on a page. For example, every time a form performs some Ajax operation, all Drupal behaviors will be executed again after page load, in order to attach any relevant JavaScript to the newly loaded elements. This can have the undesired affect of applying JavaScript to elements each time Drupal behaviors are executed, resulting in the same code being applied multiple times. To ensure that the JavaScript is applied only once, we can use the jQuery $.once() function. This function will ensure that the code inside the function is not executed if it has already been executed for the given element.

[Using jQuery $.once()](http://plugins.jquery.com/once/) (integrated into D7 core), the developer experience of applying these effects is improved. Note that there is also the $.removeOnce() method that will only take effect on elements that have already applied the behaviors.

Drupal.behaviors.mybehavior = {

attach: function (context, settings) {

$('#some\_element', context).once('mybehavior', function () {

// Code here will only be applied to $('#some\_element')

// a single time.

});

}

};

For best-practice examples, it's helpful to look at (comparatively) simple examples such as misc/collapse.js to see how it's done.

# Debugging JavaScript

If you don't have an IDE that can debug JavaScript for you, then one alternative is the Firefox add-on called [Firebug](http://getfirebug.com/). Firefox now also has inbuilt developer tools, and Chrome similarly has its own set of developer tools.

Firebug can examine scripts and set breakpoints so that the next time the script is run you can stop execution and manually step through it. In addition, Firebug has a command-line feature that can run bits and pieces of code in the page, using any scripts that have been provided by that page. This can be used to examine the DOM data structures or local script variables, or you can test snippets to see what their output will be.

Not only is Firebug great for debugging JavaScript, but also for examining HTML. Firebug automatically takes convoluted HTML code and straightens it out so that the indentation is correct and tags appear correctly. You can use the "Inspect" feature, which allows you to click on any part of the page and see its HTML representation. And Firebug also performs a little magic with CSS code. When you examine an HTML element, it will show all the CSS that applies to it-- even CSS that was overridden by other CSS! Another tab lets you see the layout of the selected block, visually displaying the borders, margins, and padding.

In addition to Firebug, you may also want to use the [Drupal for Firebug module](http://drupal.org/project/drupalforfirebug), which adds Drupal-specific features to Firebug. As documented on the project page, you'll need to fetch the [Drupal for Firebug add-on](https://addons.mozilla.org/en-US/firefox/addon/8370) separately and install it into Firefox.

There is an annoyance when debugging scripts that use jQuery or any similar JavaScript library. While stepping through your code, Firebug will also step into the library files. This can be a nightmare as jQuery-- even when uncompressed-- is hard to read. So a good rule of thumb is to simply step over the jQuery functions during debugging and save yourself a headache.

# Ajax Forms in Drupal 7

## Introduction to Ajax-enabled Forms

Ajax-enabled forms in Drupal 7 offer dynamic form behavior with no page reloads and are easy to create and manipulate. They are a simple extension of the Drupal Form API.

What is dynamic behavior? Traditional web behavior has the user fill in a form, click a button, and the entire page is rebuilt and sent back to the browser. Ajax-enabled forms update or replace part of the page or part of the form without doing a full page reload - only the part that needs to be changed is changed. It's more responsive to the user and typically faster than the traditional page reload approach.

Some facts about Ajax:

* Ajax forms provide dynamic form behavior without page reloads.
* They're significantly simplified in Drupal 7.
* As a developer you don't use or touch any JavaScript to create an Ajax-enabled form. Drupal does all the work for you.
* Ajax forms are a close relative of multistep forms.
* Most of the time, Ajax-enabled forms are just dynamic replacement of an HTML region on the page, which is most often a piece of a rebuilt form.

Some background:

* Before Drupal 7, Ajax forms were referred to as AHAH forms because Ajax (Asynchronous JavaScript and XML) implied that XML was involved but the Drupal technique doesn't use any XML (and the JavaScript part is behind the scenes). In Drupal 7 the terminology was changed to use the common and recognizable "Ajax" even though it is not literally exact. Also before Drupal 7, there was quite a lot of black magic required to get the background form submission to work correctly. All of that has been standardized and moved into Drupal core so you don't have to think about it any more.
* There are plenty of examples of Ajax behavior in Drupal that have nothing to do with Ajax forms. For example, the [Fivestar module](http://drupal.org/project/fivestar) uses its own Ajax implementation to communicate a vote from the browser to the server without a page reload.

## The Big Idea

The big idea here is that:

1. Your form gets rebuilt when you manipulate a form element (e.g. select, submit etc.)
2. Your form builder function builds it a different way based on that input ($form\_state)
3. Your #ajax settings and your callback function arrange to deliver all or a part of the newly rebuilt form to replace or otherwise enhance some part of the page.

## The Basics

To create an Ajax-enabled form, you:

* Mark a form element as an Ajax-enabled using the [#ajax](http://api.drupal.org/api/drupal/developer--topics--forms_api_reference.html/7#ajax) property. This form element will now trigger a background Ajax call when it is changed or clicked.
  + The #ajax['wrapper'] property includes the HTML ID of a page section that should be replaced (or altered in some other way).
  + The #ajax['callback'] tells the Form system what callback should be called after the Ajax call happens and the form is rebuilt.
* Create a callback function (named by the #ajax['callback']). This is generally a very simple function which does nothing but select and return the portion of the form that is to be replaced on the original page.
* In the form definition, access the values of one or more elements of the form (usually the triggering element, $form\_state['values']['howmany\_select']in the example below) to conditionally affect the form element with the wrapper ID ($form['checkboxes\_fieldset'], in the example, and how it is modified is only shown in the In More Detail section below.

In the [Examples Module](http://drupal.org/project/examples) "Ajax Example: generate checkboxes" example, the Ajax-enabled element is a select, $form['howmany\_select'], which causes replacement of the HTML ID 'checkboxes-div' (named in #ajax['wrapper']), which is a wrapper around a the fieldset $form['checkboxes\_fieldset']:

/\*\*

\* Ajax-enabled select element causes replacement of a set of checkboxes

\* based on the selection.

\*/

function ajax\_example\_autocheckboxes($form, &$form\_state) {

$default = !empty($form\_state['values']['howmany\_select']) ? $form\_state['values']['howmany\_select'] : 1;

$form['howmany\_select'] = array(

'#title' => t('How many checkboxes do you want?'),

'#type' => 'select',

'#options' => array(1 => 1, 2 => 2, 3 => 3, 4 => 4),

'#default\_value' => $default,

'#ajax' => array(

'callback' => 'ajax\_example\_autocheckboxes\_callback',

'wrapper' => 'checkboxes-div',

'method' => 'replace',

'effect' => 'fade',

),

);

$form['checkboxes\_fieldset'] = array(

'#title' => t("Generated Checkboxes"),

// The prefix/suffix provide the div that we're replacing, named by

// #ajax['wrapper'] above.

'#prefix' => '<div id="checkboxes-div">',

'#suffix' => '</div>',

'#type' => 'fieldset',

'#description' => t('This is where we get automatically generated checkboxes'),

);

// Complete example below!

When the 'howmany\_select' element is changed, a background request is issued to the server requesting that the form be rebuilt. After the form is rebuilt, using the changed 'howmany\_select' field as input on how to rebuild it, the callback is called:

/\*\*

\* Callback element needs only select the portion of the form to be updated.

\* Since #ajax['callback'] return can be HTML or a renderable array (or an

\* array of commands), we can just return a piece of the form.

\*/

function ajax\_example\_autocheckboxes\_callback($form, $form\_state) {

return $form['checkboxes\_fieldset'];

}

The callback in this case (and in many cases) just selects the portion of the form which is to be replaced on the HTML page and returns it. That portion of the form is later rendered and returned to the page, where it replaces the #ajax['wrapper'] which was provided.

That's Ajax forms in a nutshell. A form element with the #ajax property submits a background request to the server when it is triggered by a click or change. The form gets rebuilt (on the server) by the form-builder function, and then the callback named in #ajax['callback'] is called, which selects the portion of the form to return for replacement on the original page.

## In more detail

Here is the complete example discussed above, from the Ajax Examples in the [Examples](http://drupal.org/project/examples) module. (This example is live and maintained. You can download the Examples modules and experiment with this example.)

/\*\*

\* Ajax-enabled select element causes replacement of a set of checkboxes

\* based on the selection.

\*/

function ajax\_example\_autocheckboxes($form, &$form\_state) {

$default = !empty($form\_state['values']['howmany\_select']) ? $form\_state['values']['howmany\_select'] : 1;

$form['howmany\_select'] = array(

'#title' => t('How many checkboxes do you want?'),

'#type' => 'select',

'#options' => array(1 => 1, 2 => 2, 3 => 3, 4 => 4),

'#default\_value' => $default,

'#ajax' => array(

'callback' => 'ajax\_example\_autocheckboxes\_callback',

'wrapper' => 'checkboxes-div',

'method' => 'replace',

'effect' => 'fade',

),

);

$form['checkboxes\_fieldset'] = array(

'#title' => t("Generated Checkboxes"),

// The prefix/suffix provide the div that we're replacing, named by

// #ajax['wrapper'] above.

'#prefix' => '<div id="checkboxes-div">',

'#suffix' => '</div>',

'#type' => 'fieldset',

'#description' => t('This is where we get automatically generated checkboxes'),

);

$num\_checkboxes = !empty($form\_state['values']['howmany\_select']) ? $form\_state['values']['howmany\_select'] : 1;

for ($i = 1; $i <= $num\_checkboxes; $i++) {

$form['checkboxes\_fieldset']["checkbox$i"] = array(

'#type' => 'checkbox',

'#title' => "Checkbox $i",

);

}

$form['submit'] = array(

'#type' => 'submit',

'#value' => t('Submit'),

);

return $form;

}

/\*\*

\* Callback element needs only select the portion of the form to be updated.

\* Since #ajax['callback'] return can be HTML or a renderable array (or an

\* array of commands), we can just return a piece of the form.

\*/

function ajax\_example\_autocheckboxes\_callback($form, $form\_state) {

return $form['checkboxes\_fieldset'];

}

1. The form is presented to the user, as any form would be.
2. In the form, a div with an HTML ID of 'checkboxes-div' wraps $form['checkboxes']. This is done with $form['checkboxes']['#prefix'] and $form['checkboxes']['#suffix'].
3. If the user changes $form['howmany\_select'], a background request is made to the server, causing the form to be rebuilt.
4. The form is rebuilt, with as many checkboxes as were requested by $form['howmany\_select'].
5. ajax\_example\_autocheckboxes\_callback() is called. It selects the piece of the form which is to be replaced on the page (almost always the same as what's in #ajax['wrapper']).
6. The portion returned is rendered, sent back to the page, and the div with id 'checkboxes-div' is replaced on the page.

## Details and Warnings

* Changes to the form must only be made in the form builder function (ajax\_example\_autocheckboxes() in the example here), or validation will fail. The callback function must not alter the form or any other state.
* **About Ajax callbacks and #default\_value:** When Ajax replaces form elements on the page, the form field values are not automatically populated with #default\_value. However, there are other ways to set default values when using Ajax callbacks. See these links for further discussion/hints and example code: [Form API: default value does not change](http://drupal.org/node/1082818) and [Default\_value not working for Radio Buttons in Ajax Callback](http://drupal.org/node/1446510).
* It is possible to replace any HTML on the page, not just a form element. This is just a matter of providing a wrapper ID.
* You can easily replace the entire form if that is easiest. Just add a #prefixand #suffix to the entire form array, then set that as the #ajax['wrapper']. (This will allow you to change multiple form elements via a single ajax call.) The only reason not to do this is that the process is faster if less information is transferred.
* Keep in mind that the $form you're dealing with in your callback function has already been sent through all the form processing functions (but hasn't yet been sent to [drupal\_render()](http://api.drupal.org/api/function/drupal_render/7)). So while adjusting, say, the markup of an element is straightforward:
* $elements['some\_element']['#markup'] = 'New markup.';
* return $elements;

Changing a value that has already been converted into the #attributesproperty means digging deeper into the $form array, as well as also changing that element's corresponding property.

// You need to do both

$elements['some\_element']['#disabled'] = TRUE;

$elements['some\_element']['#attributes']['disabled'] = 'disabled';

return $elements;

## Graceful degradation when the browser does not support JavaScript

It is considered best practice to provide graceful degradation of behaviors in the case the browser does not support JavaScript. Ajax forms are built for this, but it may take considerable effort to make a form behave correctly (and easily) in either a JavaScript or non-JavaScript environment. In most cases, a "next" button must be provided with the Ajax-enabled element. When it is pressed, the page (and form) are rebuilt as they are when the Ajax-enabled element is changed. The [Examples module](http://drupal.org/project/examples) provides several examples of Ajax with graceful degradation in [ajax\_example\_graceful\_degradation.inc](http://api.drupal.org/api/drupal/developer--examples--ajax_example--ajax_example_graceful_degradation.inc/7):

* An add-more button
* A dependent dropdown example
* Dynamic sections
* Wizard (classic multistep form)

## More extensive Ajax features

The [Ajax Framework](http://api.drupal.org/api/group/ajax/7) provides many more features and options in addition to basic forms behavior.

* [Ajax Framework Commands](http://api.drupal.org/api/group/ajax_commands/7) may be used on the server side to generate dynamic behaviors on the page. In fact, the #ajax['callback'] function may return an array of these commands instead of returning a renderable array or an HTML string. These allow general dynamic page functions that go well beyond simple Form API operations. [Views module](http://drupal.org/project/views), for example, makes heavy use of these in its user interface.
* The #ajax['callback'] does not have to return a portion of the form. It can return any renderable array, or it can return an HTML string.
* The replace method is the default and most common, but it is also possible to do other things with the content returned by the #ajax['callback'], including prepending, appending, etc.
* It is possible to replace [ajax\_form\_callback()](http://api.drupal.org/api/function/ajax_form_callback/7) with your own functions. If you do so, ajax\_form\_callback() would be the model for the replacement. In that case, you would change #ajax['path'] from the default 'system/ajax' and set up a menu entry in hook\_menu() to point to your replacement path.

## Additional resources

* The [Examples module](http://drupal.org/project/examples) provides the example given here, an Ajax-enabled dependent dropdown, and several other examples, including an example of graceful degradation when JavaScript is not enabled.
* See the [Ajax Framework](http://api.drupal.org/api/group/ajax/7) documentation and the [Form API Reference](http://api.drupal.org/api/drupal/developer--topics--forms_api_reference.html/7#ajax) discussion of the #ajax property.

# Ajax in Drupal using jQuery

Ajax is a particular type of functionality that enables your pages to retrieve and display information from a resource on the server without the need to reload. jQuery provides a few different Ajax commands, depending on your exact requirements. Here is the simplest Ajax call you can make with jQuery:

$('#someDiv').load(url);

What this is saying is "Find the div with the id of 'someDiv' and load the html that you find at the location 'url' into this div".

(Technical aside: Actually, this example is really AHAH and not Ajax, because it returns straight html and does not need to be parsed. On the other hand, Ajax, in its strictest sense, retrieves XML data from a server resource, which then needs to be parsed before being displayed on your page. But in fact, very few Ajax applications actually return XML data these days. A much more common format for data to be sent back from the server is JSON. And this is how we do it in Drupal.)

The jQuery utility functions below provide for the extra flexibility you need when dealing with data coming back from the server that needs to be parsed.

$.get(url, parameters, callback);

$.post(url, parameters, callback);

$.ajax(options);

The only difference between $.get and $.post is the http request method used to send your parameters (an array passed as the second argument) to the server. Very often in Drupal, you won't need to send any parameters because the url you will be calling will be a menu callback you have set up as, for example, 'ajax/get/node\_details' taking one argument, 'nid' and so you would simply make the call to 'ajax/get/node\_details/123' and not need to send your nid parameter as a parameter in the second argument.

So, here's a very simple example of how this would work. Suppose you have a slideshow page on your site, similar to what you often see on news sites, where underneath the main image are numbered buttons and clicking on these instantly changes the image displayed, without reloading the page. Well, to set up something like this you would first of all set up the page so that it outputs a container for the first image (for the purposes of this example we'll assume it loads the first image as normal; also, all images are nodes) and then all the required numbered buttons. Then, to add some ajax action, you'll need to build a custom module that defines your ajax callback. Set up a menu callback in your .module file, something like this:

/\*\*

\* Implements hook\_menu().

\*/

function mymodule\_menu() {

$items['photos/get/photos'] = array(

'page callback' => 'mymodule\_get\_photos\_ajax',

'type' => MENU\_CALLBACK,

'access arguments' => array('access content'),

);

return $items;

}

/\*\*

\* Callback to return JSON encoded image for given nid.

\*/

function mymodule\_get\_photos\_ajax($nid) {

$photo = mymodule\_get\_photo($nid); // returns the filepath of my photo

$image = theme('image', $photo);

drupal\_json(array('status' => 0, 'data' => $image));

}

Our JavaScript is going to make a request to the path photos/get/photos/123, where 123 is the nid of the photo it's looking for, and the function mymodule\_get\_photos\_ajax() will take that nid and do the required php to be able to send back the image path in a JSON object, to be parsed by the JavaScript.

Here's what our JavaScript will look like:

(function($) {

Drupal.behaviors.myModule = {

'attach': function(context) {

$('a.photo\_button:not(.mymodule-processed)', context)

.addClass('mymodule-processed')

.bind('click', function() {

$.get('/photos/get/photos/' + parseInt(this.id, 10), null, imageDetails);

return false;

});

}

}

var imageDetails = function(response) {

var result = Drupal.parseJson(response);

$('div.field-type-image div.field-item').html(result.data);

}

})(jQuery);

The first function here is binding the ajax behaviour to the onclick event of all buttons with the class "photo\_button". It's using $.get, meaning it will make a http GET request to the path passed in as the first argument. Notice that it's getting the nid of the required node by parsing the id of the button - an easier way would have been to make the href of the button the ajax path itself but that would have nasty results for users without JavaScript. So the href is just going to go to the node and the JavaScript retrieves the nid by other means. We pass in null as the second argument because we don't need to pass in any additional parameters, we're already sending the nid in the url. And finally we specify the callback function, imageDetails, to be called when the request has been successful. The imageDetails function parses the JSON object that our PHP function has sent back and loads the 'image' portion of the result into our container div.

# Multiple different versions of jQuery co-existing

## Resolving jQuery plugin dependencies in Drupal without touching the version in core (or using jQuery Update)

NOTE: The [jQuery Multi](http://drupal.org/project/jqmulti) module provides a way to implement this method using a UI or hooks.

Recent versions of jQuery include a noConflict() function which restore the previous definition of the $() function and, optionally, the jQuery() function.

This provides a way of using later versions of jQuery for particular purposes, without affecting the version that core and contrib are using.

## Example

Problem: We want to use the "foo" jQuery plugin in our theme, but it needs jQuery 1.3.

Solution: In our theme's .info file, carefully specify the order of the scripts:

1. Load the jQuery 1.3 library
2. Load the "foo" plugin library
3. Load and execute a custom "noConflict" script

e.g.:

scripts[] = [...]

scripts[] = scripts/jquery-132.js

scripts[] = scripts/jquery.foo.js

scripts[] = scripts/jquery-noconflict.js

scripts[] = [...]

jquery-noconflict.js looks like this:

var jQuery13 = jQuery;

jQuery.noConflict(true);

## How it works

### Drupal loads its own jQuery library

Before any of our javascript loads, Drupal has already loaded its default jQuery library.

### Any other javascript runs

It's okay for other javascript to run at this stage, just so long as the next three steps all happen in direct sequence:

### Load the new jQuery library

When jQuery 1.3 loads, it copies any pre-existing definitions of jQuery() and $() to 'backup' variables \_jQuery and \_$, before defining its own replacement versions.

The global jQuery() and $() now refer to jQuery 1.3

### Load any plug-ins which require jQuery 1.3

jQuery plug-ins attach themselves as methods to the global jQuery object. Because the global jQuery is (for the moment) jQuery 1.3, this means the plug-in will attach itself as a method available to jQuery 1.3 (and not available to the original jQuery).

### Run jquery-noconflict.js

var jQuery13 = jQuery; copies the current definition of jQuery to a non-conflicting name.

jQuery.noConflict(true); tells jQuery to restore the global definitions of jQuery() and $() to whatever they were before jQuery 1.3 loaded.

This means that they once again refer to the original core jQuery library supplied with Drupal, while our new jQuery13 object remains in existence, along with the plugins that attached themselves to it.

### Use the plugin via the jQuery13 object

Because the global names have already been tidied up, it doesn't matter when this code runs.

Just use jQuery13() instead of using $() or jQuery()

$(document).ready(function(){

jQuery13("div.foo").foo();

}

# Creating custom Drupal.ajax object 'on the fly' and attach it to any DOM element on the page (div, td, span etc.)

## Add AJAX capability to a new DOM element

From [Drupal documentation](https://api.drupal.org/api/drupal/includes%21ajax.inc/group/ajax/7) we know how to add 'AJAX' for links and form element, from PHP with '#ajax' element attribute, but this technics propose to create AJAX object on client side.

To create new Drupal.ajax object we need to do the following:

1. Select the element to attach Drupal.ajax object to. This element should have 'id' HTML attribute. AJAX behaviour can be attached to every HTML elements on the page which has 'id' HTML attribute.
2. Build object of options for new AJAX element.
3. Add new AJAX object to global Drupal.ajax object.

For example we have a table with data which is loaded from another server with drupal\_http\_request(). This could take a long time or finish with fail. To avoid waiting for response from other server, we can load page, and after that trigger an AJAX request to **local** server, where callback function will do request to **remote** server (some kind of lazy load). Lets see how it is works.

### STEP BY STEP

1. **Select the element to attach Drupal.ajax object to.**  
   Lets create HTML element first.
2. /\*\*
3. \* Menu callback.
4. \*
5. \* Build container to load ajax content to.
6. \*/
7. function my\_module\_remote() {
8. $wrapper = array();
9. $wrapper['#attached']['js'] = array(
10. drupal\_get\_path('module', 'my\_module') . '/my\_module\_ajax.js'
11. );
12. $wrapper['#attached']['library'] = array(
13. array('system', 'drupal.ajax')
14. );
15. $wrapper['container'] = array(
16. '#type' => 'container',
17. '#attributes' => array(
18. 'id' => 'remote-content-wrapper',
19. 'title' => t('click to reload')
20. )
21. );
22. $wrapper['container']['content'] = array(
23. '#markup' => t('Loading data...')
24. );
25. return $wrapper;
26. }

This callback will create simple page with div container and text "Loading data..." on it. Pay attention to element with id "remote-content-wrapper". This div container we will be base DOM element for our Drupal.ajax object.  
In JS code we select it is with #:

$('#remote-content-wrapper').once('remote-content-wrapper', function() {

var base = $(this).attr('id');

...

});

**NOTE:** I use '$.once()' method here to attach behaviour only once. [Read more](https://drupal.org/node/756722).  
So, the HTML element selected and it is div#remote-content-wrapper.

1. **Build object of options for new AJAX element.**
2. ...
3. var element\_settings = {
4. url: 'http://' + window.location.hostname + settings.basePath + settings.pathPrefix + 'ajax/remote',
5. event: 'click',
6. progress: {
7. type: 'throbber'
8. }
9. };
10. ...

The only required keys is:  
'url' - the path to make ajax call to, any drupal full path.  
'event' - the javascript/jQuery event to react to, any string or one of the registered like 'click', 'change', 'keypress'  
'progress' - the progress type. Object contain 'type' key, which can be either 'throbber' or 'bar'.

Other settings:

keypress: true,

effect: 'none', // jQuery effect.

speed: 'none', // setting for jQuery effect.

method: 'replaceWith', // Will be rewritten with ajax command.

// The additional data to submit.

submit: {

'js': true

}

1. **Add the new AJAX object to global Drupal.ajax object.**  
   It is simple easy:
2. Drupal.ajax[base] = new Drupal.ajax(base, this, element\_settings);

Drupal.ajax() function-constructor takes 3 parameters:  
- element id,  
- DOM object,  
- settings object  
**NOTE:** The new element should be stored in global Drupal.ajax array of ajax objects under the element id key. Every Drupal.ajax object on the page are stored in this array.

That it! The Drupal.ajax element created.  
To trigger new ajax reaction we should call registered in element\_settings object event:

$(this).click(); // $('#remote-content-wrapper').click();

I use here click, but you could pass as parameter any event you want, and then trigger it with .trigger() method.  
AJAX callback should return array of Drupal commands or any custom registered AJAX commands which will be executed after success ajax call.

**The complete js part:**

(function ($) {

/\*\*

\* Load remote content after the main page loaded.

\*/

Drupal.behaviors.my\_module\_load\_remote\_content = {

attach: function(context, settings) {

$('#remote-content-wrapper').once('remote-content-wrapper', function() {

var base = $(this).attr('id');

var element\_settings = {

url: 'http://' + window.location.hostname + settings.basePath + settings.pathPrefix + 'ajax/remote',

event: 'click',

progress: {

type: 'throbber'

}

};

Drupal.ajax[base] = new Drupal.ajax(base, this, element\_settings);

$(this).click();

});

}

};

})(jQuery);

**The complete PHP part:**

/\*\*

\* Implements hook\_menu().

\*/

function my\_module\_menu() {

$items['ajax/remote'] = array(

'title' => 'Data',

'page callback' => 'my\_module\_remote\_ajax',

'access arguments' => array('access content'),

'type' => MENU\_CALLBACK,

);

$items['page'] = array(

'title' => 'Data',

'page callback' => 'my\_module\_remote',

'access arguments' => array('access content'),

'type' => MENU\_NORMAL\_ITEM,

);

return $items;

}

/\*\*

\* Menu callback.

\*

\* Output the page with the container to load AJAX content to.

\*/

function my\_module\_remote() {

$wrapper = array();

$wrapper['#attached']['js'] = array(

drupal\_get\_path('module', 'submit\_ajax') . '/submit\_ajax.js'

);

$wrapper['#attached']['library'] = array(

array('system', 'drupal.ajax')

);

$wrapper['container'] = array(

'#type' => 'container',

'#attributes' => array(

'id' => 'remote-content-wrapper',

'title' => t('click to reload')

)

);

$wrapper['container']['content'] = array(

'#markup' => t('Loading data...')

);

return $wrapper;

}

/\*\*

\* Ajax callback.

\*

\* Load remote information.

\*/

function my\_module\_remote\_ajax() {

$commands = array();

$selector = '#remote-content-wrapper';

$html = my\_module\_function\_to\_request\_data\_form\_remote\_server\_and\_build\_html();

// Debug messages, we can use dpm() here.

$messages = theme('status\_messages');

$commands[] = ajax\_command\_prepend('body', $messages);

// Add command with insert new requested content

$commands[] = ajax\_command\_html($selector, $html);

print ajax\_render($commands);

drupal\_exit();

}

# Add More Button for Text field /file field using AHAH with drupal 7 form API

Example Code snippet to create ADD MORE BUTTON with drupal 7 form API

function addmore\_menu() {

$items = array();

$items['admin/snippet/addmore'] = array(

'title' => t('Example form'),

'type' => MENU\_CALLBACK,

'page callback' => 'drupal\_get\_form',

'page arguments' => array('simple\_form'),

'access callback' => TRUE,

);

return $items;

}

function simple\_form($form, $form\_state, $no\_js\_use = FALSE) {

$form = array();

$form['#tree'] = TRUE;

$form['#attributes']['enctype'] = 'multipart/form-data';

$form['example\_entry'] = array(

'#type' => 'fieldset',

'#title' => t('Example form'),

"#prefix" => '',

"#suffix" => '',

'#collapsible' => FALSE,

);

// --------------- Add more text box starts ------------------------------

$form['example\_entry']['entry\_fieldset'] = array(

'#type' => 'fieldset',

'#title' => t('Participants Name'),

'#prefix' => '',

'#suffix' => '',

);

if (empty($form\_state['pnum\_names'])) {

$form\_state['pnum\_names'] = 1;

}

for ($i = 0; $i < $form\_state['pnum\_names']; $i++) {

$form['example\_entry']['entry\_fieldset']['users'][$i] = array(

'#type' => 'textfield',

'#title' => '',

'#size' => 22,

'#description' => t('User Name'),

'#attributes' => array('multiple' => 'multiple'),

);

}

$form['example\_entry']['entry\_fieldset']['add\_user\_name'] = array(

'#type' => 'submit',

'#value' => t('Add more user'),

'#submit' => array('users\_add\_more\_add\_one'),

'#ajax' => array(

'callback' => 'users\_add\_more\_callback',

'wrapper' => 'entry-fieldset-wrapper',

),

);

if ($no\_js\_use) {

if (!empty($form['entry\_fieldset']['remove\_name']['#ajax'])) {

unset($form['entry\_fieldset']['remove\_name']['#ajax']);

}

unset($form['entry\_fieldset']['add\_name']['#ajax']);

}

// --------------- Add more text box ends ---------------------------------

// File Attachement form element starts

$form['example\_entry']['upload\_fieldset'] = array(

'#type' => 'fieldset',

'#title' => t('Attachments'),

'#prefix' => '',

'#suffix' => '',

);

if (empty($form\_state['num\_names'])) {

$form\_state['num\_names'] = 1;

}

for ($i = 0; $i < $form\_state['num\_names']; $i++) {

$form['example\_entry']['upload\_fieldset']['attachement'][$i] = array(

'#type' => 'managed\_file',

'#title' => '',

'#size' => 22,

'#description' => t('Upload File'),

'#attributes' => array('multiple' => 'multiple'),

);

}

$form['example\_entry']['upload\_fieldset']['add\_name'] = array(

'#type' => 'submit',

'#value' => t('Add more'),

'#submit' => array('attachment\_add\_more\_add\_one'),

'#ajax' => array(

'callback' => 'attachment\_add\_more\_callback',

'wrapper' => 'upload-fieldset-wrapper',

),

);

if ($no\_js\_use) {

if (!empty($form['upload\_fieldset']['remove\_name']['#ajax'])) {

unset($form['upload\_fieldset']['remove\_name']['#ajax']);

}

unset($form['upload\_fieldset']['add\_name']['#ajax']);

}

// File Attachement form element ends

$form['example\_entry']['submit'] = array(

'#type' => 'submit',

'#value' => t('Submit'),

);

return $form;

}

// --------------- AJAX CALLBACK FUNCTIONS FOR UPLOAD FIELD-----------------

/\*\*

\* Submit handler for the "add-one-more" button.

\*

\* Increments the max counter and causes a rebuild.

\*/

function attachment\_add\_more\_add\_one($form, &$form\_state) {

if (!isset($form\_state['num\_names'])) {

$form\_state['num\_names'] = 0;

$form\_state['num\_names']++;

}

$form\_state['num\_names']++;

$form\_state['rebuild'] = TRUE;

}

/\*\*

\* Callback for both ajax-enabled buttons.

\*

\* Selects and returns the fieldset with the names in it.

\*/

function attachment\_add\_more\_callback($form, $form\_state) {

return $form['example\_entry']['upload\_fieldset'];

}

// --------------- AJAX CALLBACK FUNCTIONS FOR TEXT FIELD-----------------

function users\_add\_more\_add\_one($form, &$form\_state) {

if (!isset($form\_state['pnum\_names'])) {

$form\_state['pnum\_names'] = 0;

$form\_state['pnum\_names']++;

}

$form\_state['pnum\_names']++;

$form\_state['rebuild'] = TRUE;

}

function users\_add\_more\_callback($form, $form\_state) {

return $form['example\_entry']['entry\_fieldset'];

}

function simple\_form\_submit($form, &$form\_state) {

$num\_files = COUNT($form\_state['values']['example\_entry']['upload\_fieldset']['attachement']);

if (($num\_files > 0)) {

for ($i = 0; $i < $num\_files; $i++) {

$attachment[$i] = file\_load($form\_state['values']['example\_entry']['upload\_fieldset']['attachement'][$i]);

}

}

echo '';

print\_r($form\_state);

}

# Simple Drupal AJAX load with jQuery and delivery callback

**NOTE:** This tutorial requires knowledge about custom module development or function altering. (Link to [how to build a module](https://www.drupal.org/node/1074360).)

### Introduction

Most AJAX code in the Drupal documentation focuses on traditional json/\_get() applications, which require json manipulation of the information when it arrives on the client side. Using the [jQuery.load()](http://api.jquery.com/load/) function you can load HTML from a URL that will render directly in a target element with only a single line of JavaScript code.

The real trick here is to get only the HTML you need instead of rendering the header, footer, and every other element that comes with a full page load. This is where the 'delivery callback' setting in [hook\_menu](https://api.drupal.org/api/drupal/modules!system!system.api.php/function/hook_menu/7) comes into play. The 'delivery callback' setting defaults to [drupal\_deliver\_html()](https://api.drupal.org/api/drupal/includes!common.inc/function/drupal_deliver_html_page/7) which returns a fully rendered HTML page. We need to define our own delivery callback function returning just the HTML we want to render via AJAX on the client side.

### We need to

* Change the markup of the links which will trigger the AJAX request
* Add an additional container to the markup where the AJAX content should be rendered
* Write/alter some code in a [custom module](https://www.drupal.org/node/1074360).

### Add the link markup and target container to your site

1. Define a link to execute the AJAX call:  
   <a class="btn" href="#" onclick="myModule\_ajax\_load()">Ajax Test</a>
2. Set the element that will receive the AJAX callback:  
   <div id="ajax-target">Ajax goes here!!!</div>

### The custom module code

1. Create a JavaScript function that will handle the call using jQuery.load(). The $nid in the path is just an example. It could be any variable.
2. <script>
3. function myModule\_ajax\_load() {
4. jQuery("#ajax-target").load("/node/get/ajax/11");
5. }
6. </script>
7. Register the URL that will render the HTML by implementing hook\_menu():
8. /\*\*
9. \* Implements hook\_menu().
10. \*/
11. function myModule\_menu() {
12. $items['node/get/ajax/%'] = array(
13. 'page callback' => 'myModule\_ajax\_get\_ajax', // Render HTML.
14. 'page arguments' => array(3),
15. 'type' => MENU\_CALLBACK,
16. 'access arguments' => array('access content'),
17. 'delivery callback' => 'myModule\_ajax\_callback', // Magic goes here.
18. );
19. return $items;
20. }
21. Create page callback and delivery callback functions:
22. function myModule\_ajax\_get\_ajax($nid) {
23. // This example loads a node and returns the teaser.
24. // You can return whatever you want, including forms.
25. $node = node\_load($nid);
26. return node\_view($node, 'teaser');
27. }
28. function myModule\_ajax\_callback($page\_callback\_result) {
29. // Only render content
30. $content = drupal\_render($page\_callback\_result);
31. // Add CSS and JS files, add some markup
32. $html = '<html><head><title></title>' . drupal\_get\_css() . drupal\_get\_js() . '</head><body class="jquery-ajax-load">' . $content . '</body></html>';
33. print $html;
34. // Perform end-of-request tasks.
35. drupal\_page\_footer();
36. }

### Now lets load any page in your site with just 3 lines of code

The example above lets you render anything from your site, but some things are already there, like views, forms or webforms. So how can we get them without need to build them again.

For the start we need the function that renders any URL into a page using the menu.api, its name is [menu\_execute\_active\_handler()](https://api.drupal.org/api/drupal/includes!menu.inc/function/menu_execute_active_handler/7), this function by default returns a full page set on its first argument, but if you set the second argument to FALSE it returns just the content part of your page, no matter how or who generates it.

So now we will rebuild our myModule\_ajax\_get\_ajax() function in the step 5 to return only the content of any page declared in your site.

1. Define a link to execute the AJAX call:  
<a class="btn" href="#" onclick="myModule\_ajax\_load()">Ajax Test</a>

2. Set the element that will receive the AJAX callback:  
<div id="ajax-target">Ajax goes here!!!</div>

3. Create the JavaScript function that will do the call, using jQuery.load(), I am using an arbitrary URL "node/11", that can be a variable, with a prefix "/get/ajax" that will trigger hook\_menu:

<script>

function myModule\_ajax\_load() {

jQuery("#ajax-target").load("/get/ajax/node/11");

}

</script>

4. Create the URL that will render HTML with hook\_menu():

/\*\*

\* Implementation of hook\_menu().

\*/

function myModule\_ajax\_menu() {

$items['get/ajax'] = array(

'page callback' => 'myModule\_ajax\_get\_ajax', // Render HTML

'type' => MENU\_CALLBACK,

'access arguments' => array('access content'),

'delivery callback' => 'myModule\_ajax\_callback', // Magic goes here

);

return $items;

}

5. Create page callback and delivery callback functions:

function myModule\_ajax\_get\_ajax() {

// The function will receive the array of arguments after the "get/ajax" prefix

$path = implode('/', func\_get\_args());

$render\_string = menu\_execute\_active\_handler($path, FALSE);

return $render\_string;

}

function myModule\_ajax\_callback($page\_callback\_result) {

// Only render content and validates return

$content = is\_string($page\_callback\_result) ? $page\_callback\_result : drupal\_render($page\_callback\_result);

$html = '

' . drupal\_get\_css() . drupal\_get\_js() . '' . $content . ''; print $html; drupal\_page\_footer();   
}