net.tutsplus.com

Fun with jQuery Templating and AJAX

In this tutorial, we'll take a look at how jQuery's beta templating system can be put to excellent use in order to completely decouple our HTML from our scripts. We'll also take a quick look at jQuery 1.5′s completely revamped AJAX module.

What's Templating?

> Templating is a newish (it's still in beta, so it's likely to change slightly as it matures and migrates into core, but it's been around for the best part of a year), hugely powerful jQuery feature that allows us to specify a template to use when building DOM structures via script, something that I'm sure we all do on an almost daily basis.

It's always been incredibly easy when using jQuery to do something like this:

```
$("#someElement").children().each(function() {
        $(this).wrap($("<div></div>"));
});
```

> Templating allows us to remove these string-based snippets of HTML from our behaviour.

This will simply wrap each child element of `#someElement` in a new `<div>` element. There's nothing particularly wrong with doing this; it's perfectly valid and works well in countless situations. But, that's HTML there in our script — content mixed up with behaviour. In the simple example above it's not a huge issue, but real-world scripts could contain many more snippets of HTML, especially when building DOM structures with data obtained via an AJAX request. The whole thing can quickly become a mess.

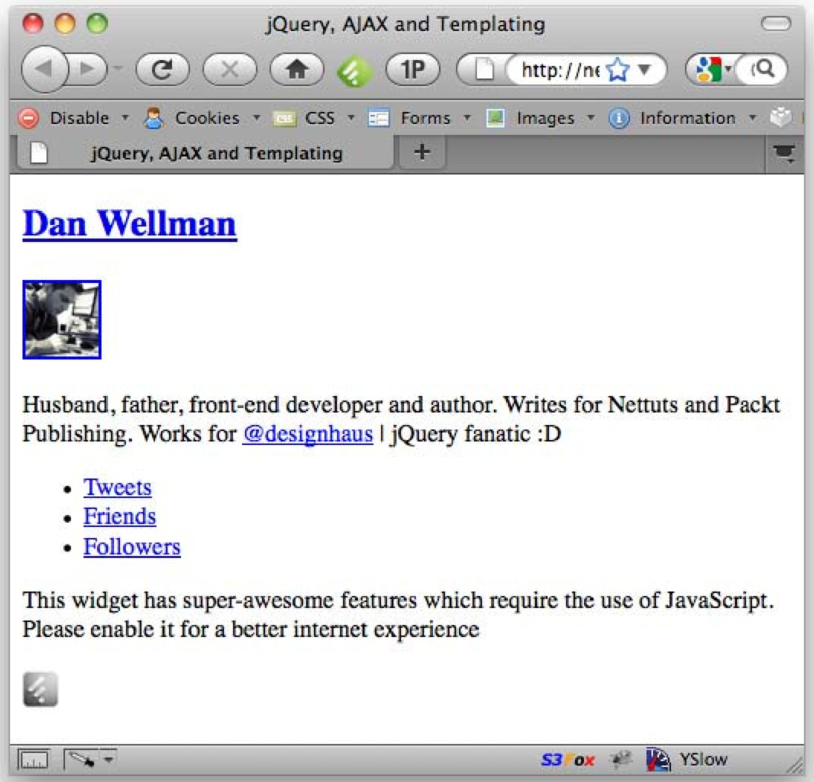
Templating allows us to remove these string-based snippets of HTML from our behaviour layer, and put them back firmly where they belong in the content layer. While we're doing that, we can also check out one of the brand new, super-cool AJAX features of jQuery 1.5 – deferred objects.

Getting Started

In this example, we'll build a Twitter widget that will not only load some of our recent tweets, but also list some friends, followers and suggestions. I chose Twitter for this example because it outputs JSON in the format we require; it's easy and fun.

So let's get started; the widget itself will be built from the following underlying mark-up:

```
<!DOCTYPE HTML>
<html>
    <head>
        <meta charset="utf-8">
        <title>jQuery, AJAX and Templating</title>
        <link rel="stylesheet" href="tweetbox.css">
        <!--[if lte IE 8]>
                <script src="http://html5shim.googlecode.com/svn/trunk/html5.js"></scr
            <![endif]-->
        </head>
            <body>
            <aside id="tweetbox">
                <div id="user">
            <h2><a href="http://twitter.com/danwellman" title="Visit Dan Wellman on Twitte
                <a href="http://twitter.com/danwellman" title="Dan Wellman"><img src="img/
                        <p>Husband, father, front-end developer and author. Writes for
                </div>
                <ul id="tools">
                        <li><a href="#tweets" title="Recent Tweets" class="on">Tweets<
                        <li><a href="#friends" title="Recent Friends">Friends</a></li>
                        <li><a href="#follows" title="Recent Followers">Followers</a><
                </ul>
                <div id="feed">
                        <div id="tweets">
                                <noscript>This widget has super-awesome features which
                        </div>
                        <div id="friends"></div>
                        <div id="follows"></div>
                </div>
            </aside>
            <script src="jquery.js"></script>
            <script src="jquery.tmpl.min.js"></script>
            <script src="tweetbox.js"></script>
        </body>
</html>
```


Dan Wellman

Husband, father, front-end developer and author. Writes for Nettuts and Packt Publishing. Works for @designhaus | jQuery fanatic :D

- Tweets
- Friends
- Followers

This widget has super-awesome features which require the use of JavaScript. Please enable it for a better internet experience

We're using HTML5 and have included the simplified `DOCTYPE` and `meta charset` element. We link to a custom style sheet, which we'll create in a moment, and in order to support current versions of IE8 and lower, we use a conditional comment to link to the Google-hosted `html5shiv` file.

Using `aside`

This widget would probably go into a sidebar, and be distinct from the actual content of the page it is featured on, but related to the site as a whole. With that in mind, I feel an `<aside>` is an appropriate outer container in this case. We give it an `id` for easy selecting and styling.

Continuing on with our markup, we have some details about the Twitter user whose tweets are listed, including the name in a `<h2>`, an image and the bio in a standard `<p>`. Feel free to change these to your own details when reconstructing the example. We could get all of these items from the JSON call that we'll make when we request the data, however, if there is a slight delay in the request at page load, the visitor could be left staring at a bunch of empty boxes, so hard-coding this info into the widget is again, appropriate. If we were making a plugin for other developers to consume, we certainly couldn't do this, but when adding this to our own site, or a specific client's site, this is a feasible approach.

Next, we have the tabs that will be used to switch between the tweets, friends and followers. These are built from a simple collection of `<ul>`, `<li>` and `<a>` elements. The friends tab will be displayed by default, so the link for this tab has the class *on* attached to it. In a larger project, we could of course use jQuery UI tabs, but I didn't want the tutorial to lose focus, and it's no bother to add this functionality ourselves manually.

> Notice that we're also using a plugin – this is the `tmpl` (templating) plugin, which gives us the ability to use jQuery templates.

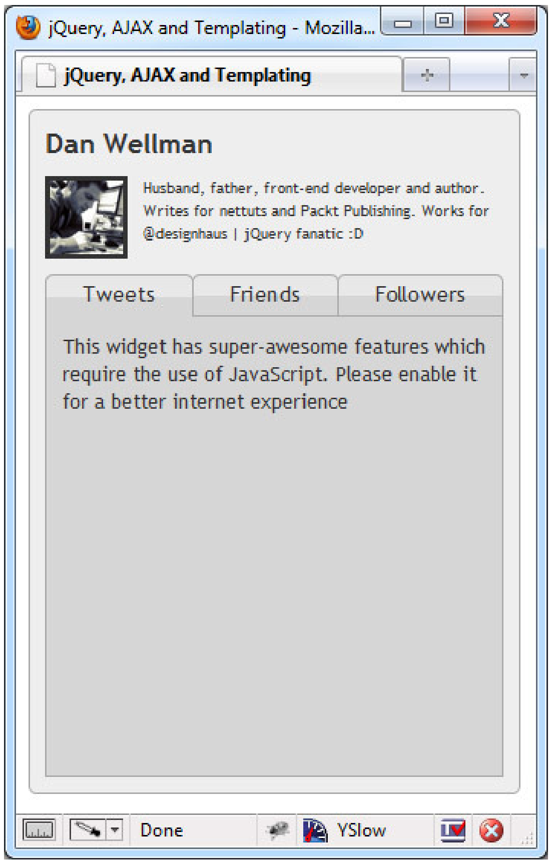
Finally, we have the elements that will hold each stream of data; we have an outer container with an `id` of `feed`, and three containers for the tweets, friends and followers respectively, which also have `id` attributes for easy selecting. We also include a `<noscript>` element for visitors that may have scripting disabled (if any actually exist anymore), which is within the default tab content area. Notice that we're also using a plugin – this is the `tmpl` (templating) plugin, which gives us the ability to use jQuery templates. This file can be downloaded from here

Grab a copy now and stick it in the same folder as the web page we just created.

I mentioned a custom style sheet earlier; let's add that in right now; in a new file in your text editor add the following code:

```
#tweetbox {
        display:block; width:300px; padding:10px; border:1px solid #aaa; -moz-border-radius:5p
        border-radius:5px; font-family:"Trebuchet MS", Arial, Helvetica, sans-serif;
        background-color:#eee;
}
#tweetbox img { display:block; }
#user { margin-bottom:10px; float:left; }
#user h2 { margin:0 0 10px 0; position:relative; font-size:18px; }
#user img { float:left; }
#user p { width:230px; margin:0; position:relative; float:left; font-size:10px; color:#333; }
#user img { display:block; margin-right:10px; border:3px solid #333; }
#tools { margin:0; *margin-bottom:-10px; padding:0; clear:both; list-style-type:none; }
#tools li {  float:left; }
#tools a {
        display:block; height:20px; padding:3px 24px; border:1px solid #aaa; border-bottom:non
        -moz-border-radius:5px 5px 0 0; border-radius:5px 5px 0 0; margin-right:-1px;
        position:relative; font-size:14px; outline:none; background-color:#d6d6d6;
        background-image:-webkit-gradient(linear, left top, left bottom, color-stop(0.5, #E8E8
        background-image: -moz-linear-gradient(center top, #E8E8E8 50%, #DBDBDB 0%, #D6D6D6 50
}
a { text-decoration:none; color:#333; }
#tools .on { height:21px; margin-top:-1px; top:1px; }
#feed { width:298px; border:1px solid #aaa; clear:both; background-color:#d6d6d6; }
#feed > div { display:none; }
noscript { display:block; padding:10px; font-size:13px; color:#333; }
```

Save this file as `tweetbox.css` in the same directory as the HTML page. This is just a bit of layout styling for our widget. There's a couple of CSS3 niceties for capable browsers: some rounded-corner action (notice we no longer need the `–webkit–` vendor prefix for rounded corners in the latest webkit browsers!) and some gradients for the tabs. A point to note is that we hide all of the containers within the feed element, except for the one with the class `active`. At this point (and with JS disabled) the widget should look like so:


jQuery, AJAX and Templating - Mozilla...
jQuery, AJAX and Templating
Tweets
Friends
Followers
Done
YSlow

Dan Wellman

Husband, father, front-end developer and author. Writes for nettuts and Packt Publishing. Works for @designhaus | jQuery fanatic :D

This widget has super-awesome features which require the use of JavaScript. Please enable it for a better internet experience

Adding the Script

Let's put the basic script together and get those tabs working. In another new file in your text editor, add the following code:

```
(function($) {
        //tabs
        var tweetbox = $("#tweetbox"),
                tweetData = null,
                friendData = null,
                followData = null;

        tweetbox.find("#tools a").click(function(e) {
                e.preventDefault();

                var link = $(this),
                        target = link.attr("href").split

                tweetbox.find(".on").removeClass("on");
                link.addClass("on");
                tweetbox.find("#feed > div").hide();
                tweetbox.find("#" + target).show();
        });
})(jQuery);
```

Save this file as `tweetbox.js` in the same directory as the HTML page. It's all pretty straight-forward, and, as it's not really the main focus of this tutorial, I won't go into it too much. All we do is alias the string character within an anonymous function, which we execute straight away – more for good-practice than sheer necessity in this example – and then cache a selector for the main outer container for the widget. We also initialize three variables for use later on, and set their values to `null`.

We'll need to select elements repeatedly throughout the code; so caching a reference to the outer container helps minimise the number of jQuery objects we need to create. We then set a click handler for the tab links which gets the `id` of the tab to show from the `href` of the link that was clicked, removes the class name `on` from the tab links and then adds it back to the link that was clicked. We then hide all of the tab panels, before showing the selected tab panel.

Getting the Data

Now the real fun begins; we can make the requests to Twitter to get our three data sets and make use of jQuery's templating plugin to create the required DOM elements using the data we obtain from the requests. We'll get the data first and then add the templates. After the click-handler for the tool links, add the following code:

```
$.ajaxSetup({
        dataType: "jsonp"
});
```

```
function getTweets() {
        $.ajax("http://api.twitter.com/statuses/user_timeline/danwellman.json", {
                success: function(data) {
                        var arr = [];

                        for (var x = 0; x < 5; x++) {
                                var dataItem = {};
                                dataItem["tweetlink"] = data[x].id_str;
                                dataItem["timestamp"] = convertDate(data, x);
                                dataItem["text"] = breakTweet(data, x);
                                arr.push(dataItem);
                        }

                        tweetData = arr;
                }
        });
}
function getFriends() {
        return $.ajax("http://api.twitter.com/1/statuses/friends/danwellman.json", {
                dataType: "jsonp",
                success: function(data) {
                        var arr = [];

                        for (var x = 0; x < 5; x++) {
                                var dataItem = {};
                                dataItem["screenname"] = data[x].screen_name;
                                dataItem["img"] = data[x].profile_image_url;
                                dataItem["name"] = data[x].name;
                                dataItem["desc"] = data[x].description;
                                arr.push(dataItem);
                        }

                        friendData = arr;
                }
        });
}
function getFollows() {
        return $.ajax("http://api.twitter.com/1/statuses/followers/danwellman.json", {
                dataType: "jsonp",
                success: function(data) {
                        var arr = [];

                        for (var x = 0; x < 5; x++) {
                                var dataItem = {};
                                dataItem["screenname"] = data[x].screen_name;
                                dataItem["img"] = data[x].profile_image_url;
                                dataItem["name"] = data[x].name;
                                dataItem["desc"] = data[x].description;
                                dataItem["desc"] = data[x].description;
                                arr.push(dataItem);
                        }

                        followData = arr;
                }
        });
}

//execute once all requests complete
$.when(getTweets(), getFriends(), getFollows()).then(function(){

        //apply templates
});
```

First, we use jQuery's `ajaxSetup()` method to set the `dataType` option to `jsonp` for all subsequent requests. As this will be the `dataType` used by each of our requests, it makes sense to just set the option once.

We then define three standard functions; within each function we use jQuery's `ajax()` method to make a request to the web service that returns each set of data we'll be working with, the `user_timeline`, `friends` and `followers` respectively. In the settings object for each request, we define a `success` handler which will be executed once each individual request returns successfully. Each of these requests will return a JSON object that potentially contains up to 100 objects packed full of Twitter data.

> To ensure that the data is stored in the correct format for JSON we use square-bracket notation.

We don't need that much data, so in each `success` handler we create a new array, which in turn will contain a series of objects that hold just the data we are actually going to use. To ensure that the data is stored in the correct format for JSON, where every property name must be a string, we use square-bracket notation to set the property names in string format.

The `user_timeline` request stores the `id string` of the tweet which can be used as part of a URL that points to the tweet, as well as storing the result of two utility functions. The first of these functions creates a formatted data string which converts the date returned by Twitter into something that is a little prettier, and localised to the viewer's machine. We also format the tweet text so that we can atify any `@usernames` found in the text. We'll look at both the date and tweet formatter functions shortly.

The functions to retrieve the friends and followers lists are pretty much identical. Again, we rename the properties that we'll be working with when we build our objects and store them in each array. All three of our success handlers store the resulting 5-item arrays in the `null` variables we set at the top of the script.

Notice that we don't invoke each of our `getTweets()`, `getFriends()` and `getFollowers()` functions manually. Instead, we use jQuery's new `when()` method to invoke all of them at the same time. This method will completely handle running these functions and will keep track of when each one has returned. We chain the `then()` method after the `when()` method. Once all of the specified functions have returned successfully, the callback function we pass to the `then()` method will be executed.

> The `when()` method creates a deferred object that manages the functions we specify as arguments.

Once the functions have all returned, the deferred object is resolved and any functions registered with `then()` are called. Other handlers may also be chained to the `when()` deferred object, such as `fail()`, which would be executed if one or more of the functions passed to the deferred object failed.

This is incredible; we want to make all three requests, but we have no way of knowing beforehand which of these will be returned last, so it is impossible to use any one request's callback function if we wish to process the data returned from all of the functions at the same time. In the past, we would probably have had to setup an interval which repeatedly polled each function to check whether it had returned, and wait until all of them had before proceeding. Now we can delegate all of this manual tracking to jQuery to handle for us automatically.

Utility Functions

We use two utility functions in this example: `convertDate()` and `breakTweet()`. The code for these functions is as follows:

```
//format date
convertDate = function(obj, i) {

        //remove time zone offset in IE
        if (window.ActiveXObject) {
                obj[i].created_at = obj[i].created_at.replace(/[+]\d{4}/, "");
        }

        //pretty date in system locale
        var days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"],
                date = new Date(obj[i].created_at),
                formattedTimeStampArray = [days[obj[i].created_at], date.toLocaleDateString(),

        return formattedTimeStampArray.join(" ");
}

//format text
breakTweet = function(obj, i) {

        //atify
        var text = obj[i].text,
                brokenTweet = [],
                atExpr = /(@[\w]+)/;

        if (text.match(atExpr)) {
                var splitTweet = text.split(atExpr);

                for (var x = 0, y = splitTweet.length; x < y; x++) {
                        var tmpObj = {};

                        if (splitTweet[x].indexOf("@") != -1) {
                                tmpObj["Name"] = splitTweet[x];
                        } else {
                                tmpObj["Text"] = splitTweet[x];
                        }

                        brokenTweet.push(tmpObj);
                }
        } else {
                var tmpObj = {};
                        tmpObj["Text"] = text;
                brokenTweet.push(tmpObj);
        }

        return brokenTweet;
}
```

The `convertDate()` function is relatively straight-forward: we first check whether the browser in use is a variant of IE by looking for `window.ActiveXObject`. If this is found, we use the JavaScript `replace()` method to remove the Timezone Offset supplied as part of the string contained within the `created_at` property of the JSON object returned by Twitter. This method takes the regular expression pattern to look for, and an empty string to replace it with. This stops IE choking on the + character when the string is passed to the `new Date` constructor.

Next we create some variables; we set an array containing shortened day names, with Sunday (or Sun) as the first item in the array. Days in JavaScript dates are zero-based, with Sunday always appearing as day 0. We then create a *Date* object using the `new Date()` constructor, and pass in the date string stored in the `created_at` property of the object we passed in to the function.

We then create another array containing three items: the first item gets the correct day of the week from the first array we created within this function, the next item gets the localised date, and the last item gets the localised time. Finally, we return the contents of the array after we have joined it. We could simply use string concatenation to build this date string, but joining array items is much faster than building strings manually.

The `breakTweet()` function is slightly more complex. What we need to do is convert the plain text into a JSON array where each array item is an object containing either a `Name` or `Text` property so that we can use the data with a template (more on the templates next). First we store the text from the object returned by Twitter (which we pass into the function). We also create an empty array to store the objects in and define the regular expression that will match `@usernames`.

We then test the text to see whether it contains any usernames; if it does we then split the text string on each occurrence of a username. This will give us an array which contains items that are either plain text, or a username. We then cycle through each item in this array and check whether each item contains the `@` symbol; if it does, we know it's a username and so store it in an object with the key `Name`. If it doesn't contain the `@`

symbol we save it with the key `Text`. The object is then pushed into the array. Also, if the whole text doesn't contain an `@` character we store it with the key `Text`.

That's it; once we have stored our objects the function returns the new `brokenTweet` array to the `user_timeline` success function and is stored in the main JSON object for use with our templates. As well as atifying the text, we could also linkify and hashify if we wanted. I'll leave that up to you to implement.

Templating

Now that we have our processed JSON data, we can move on to the final part of the example: templating. In the anonymous function passed to the `then()` method in the last code section, there was a comment and nothing else. Directly after this comment, append the following code:

```
//apply templates
tweetbox.find("#tweetTemplate").tmpl(tweetData).appendTo("#tweetList");
tweetbox.find("#ffTemplate").tmpl(friendData).appendTo("#friendList");
tweetbox.find("#ffTemplate").tmpl(followData).appendTo("#followList");

//show tweets
tweetbox.find("#tweets").show();
```

This code simply applies the templates using the jQuery templating plugin method `tmpl()`. The method accepts the JSON containing the data that the template will use. We then specify where in the document to insert the template elements. Each set of templates appends the elements to the respective empty container within the widget. The `tmpl()` method is called on three elements, but these elements don't yet exist. We'll add these next.

Adding the jQuery Templates

Switch back to the HTML and first add the following `<script>` element directly after the empty `<ul>` with the id `tweetList`:

```
<script id="tweetTemplate" type="text/x-jquery-tmpl">
        <li>
                <p>
                        {{each text}}
                                {{if Name}}
                                        {{tmpl(Name) "#atTemplate"}}
                                {{else}}
                                        ${Text}
                                {{/if}}
                        {{/each}}
                        <a class="tweet-link" href="http://twitter.com/danwellman/status/${twe
                </p>
        </li>
</script>
<script id="atTemplate" type="text/x-jquery-tmpl">
        <a href="http://twitter.com/${$item.data}">${$item.data}</a>
</script>
```

jQuery templates are added to the HTML page using `<script>` elements. These elements should have `id` attributes set on them so that they can be selected and have the `tmpl()` method called on them. They should also have the custom type `x-jquery-tmpl` set on them.

In the first template, we add the mark-up that we want to new DOM structure to be built from, which in this case is an `<li>`, a `<p>` and an `<a>` to create each tweet. To insert the data from the JSON object passed into the `tmpl()` method we use a series of templating tags. First we use the `{{each}}` tag to go through each item in the `text` array.

This is the array containing the broken up tweet, so for each object we check whether it has a `Name` key; if it does we use the `{{tmpl}}` tag, which allows us to use a nested template. We specify the data to pass the nested function within parentheses after the `tmpl` tag and also specify the `id` of the template to select (this is the second template that we just added, which we'll look at in more detail in a moment). If the object does not contain a `Name` property, we know that we are dealing with a plain text portion of the tweet and just insert the data using `${Text}`. This conditional is achieved using the `{{else}}` template tag. We should also close the conditional using `{{/if}`, and close the iteration using `{{/each}}`.

Finally, we create a new anchor element that links directly to the tweet on the Twitter website using `${tweetlink}` as part of the `href`, and `${timestamp}` properties. These properties are the ones we created in the success handler for the `user_timeline` request.

In the *atTemplate* we also create a link; this time it links to the user that was mentioned. As this is a nested template, we need to access the actual data slightly differently; the data passed into the nested template by the `{{tmpl}` tag will be stored in a property of the `$item` object called `data`.

We still need to add the template for our friends and followers tabs. Both of these will be built from the same template, which should be as follows:

```
<script id="ffTemplate" type="text/x-jquery-tmpl">
        <li>
                <p>
                        <a class="img-link" href="http://twitter.com/${screenname}"><img src="
                        <span class="username"><a href="http://twitter.com/${screenname}">${sc
                        <span class="bio">${desc}</span>
                </p>
        </li>
</script>
```

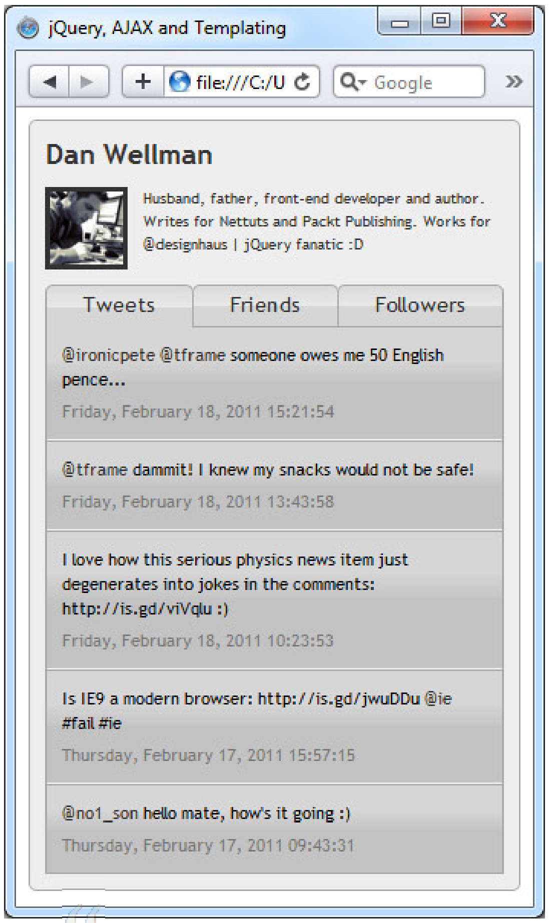
This template is much simpler as we aren't using nested templates or doing any iteration, we're simply inserting the data from each JSON array using the `${data}` template item format.

Finishing Up

Now that we applied the templates and populated our widget, we can add a few more CSS styles to tidy up the new elements that have been added; in tweetbox.css add the following code to the end of the file:

```
#feed ul { padding:0; margin:0; }
#feed li { padding:0; border-bottom:1px solid #aaa; list-style-type:none; font-size:11px; }
#feed li:last-child, #feed li:last-child p { border-bottom:none; }
#feed p { padding:10px; margin:0; border-bottom:1px solid #eee; background-image:-webkit-gradi
#feed p:after { content:""; display:block; width:100%; height:0; clear:both; }
.tweet-link { display:block; margin-top:5px; color:#777; }
.img-link { display:block; margin:4px 10px 0 0; float:left; }
#feed .username a { font-size:14px; font-weight:bold; }
#feed .bio { display:block; margin-top:10px; }
```

Save the file, our page should now appear as follows:



Dan Wellman

Husband, father, front-end developer and author. Writes for Nettuts and Packt Publishing. Works for @designhaus | jQuery fanatic :D

Tweets	Friends	Followers

@ironicpete @tframe someone owes me 50 English pence...
Friday, February 18, 2011 15:21:54

@tframe dammit! I knew my snacks would not be safe!
Friday, February 18, 2011 13:43:58

I love how this serious physics news item just degenerates into jokes in the comments: http://is.gd/viVqIu :)
Friday, February 18, 2011 10:23:53

Is IE9 a modern browser: http://is.gd/jwuDDu @ie #fail #ie
Thursday, February 17, 2011 15:57:15

@no1_son hello mate, how's it going :)
Thursday, February 17, 2011 09:43:31

There's still one more thing we should probably do: at the moment, our tweet formatting function doesn't work in IE because of how IE treats the `split()` regular expression method. To fix this issue, we can use an excellent JavaScript patch created by Steven Levithan. It can be downloaded from: http://blog.stevenlevithan.com/archives/cross-browser-split and can be included in the page using a conditional comment in the same way that we added the `html5shiv` file:

```
<!--[if IE]>
        <script src="fixSplit.js"></script>
<![endif]-->
```

This should be added directly before the script reference to our `tweetbox.js` file.

Summary

In this tutorial we looked at some of the more advanced features of jQuery templating, such as item iteration with `{{each}}` and conditionals with `{{if}}` and `{{else}}`.

“ A key point is that the new templating system from jQuery allows us to better separate our behaviour and presentation layers, moving all HTML snippets out of the `.js` file and back into the `.html` file.

Some might argue that we now just have the opposite situation and simply have more `<script>` in our page.

However, I think this is infinitely preferable to the alternative. Elements in the format `$("<div>")` are simply strings with no implicit meaning (until jQuery is run by the browser and they are created and inserted into the DOM).

We also looked at the excellent new `when()` method provided in jQuery 1.5, which allows us to manage a set of asynchronous requests and execute code once they have all completed. This involves using the `then()` method to specify a function to execute once the asynchronous functions have all returned.

Thank you so much for reading and let me know if you have any questions!