A guide to writing unobtrusive JavaScript and Ajax

Employ good programming practices when creating your web applications

Skill Level: Intermediate

Joe Lennon (joe@joelennon.ie)
Lead Developer
Core International

02 Nov 2010

Unobtrusive JavaScript is the practice of separating the JavaScript, CSS, and HTML elements in your web applications. By keeping your applications organized in this way, it's easier to maintain them and to ensure that your applications behave consistently across various platforms and web browsers. In this article, learn how to employ techniques to reap the benefits of developing web applications in an unobtrusive manner.

Introduction

When writing JavaScript and Asynchronous JavaScript + XML (Ajax) applications, it is all too easy to focus on the interactive features they have to offer, while forgetting about the basic fundamentals of web application development. It is important to write JavaScript and Ajax applications in an unobtrusive manner for a number of reasons. First, doing so lets you keep the logic of the application separate from your content, making it easier to maintain your applications going forward. Additionally, it lets you ensure that your application behaves consistently across various platforms and web browsers in its most basic form, resulting in you only needing to worry about this issue when you add in the JavaScript and Ajax features. Most importantly, perhaps, is that developing web applications in this manner means that you are following the idea of *progressive enhancement*, meaning that your application will support users who are using browsers that do not support JavaScript or particular JavaScript features (including Ajax). If you have been writing JavaScript applications in an obtrusive manner up until now, this article will help you discover the best practices that allow you to create web applications that work for everybody, while providing all the bells and whistles to those users who can use them.

The term *unobtrusive JavaScript* has a relatively loose definition, but is generally accepted as being the process of creating web pages and applications using a collection of good programming practices. These include the following:

- Keeping separate the JavaScript, CSS, and HTML elements of your application
- Using JavaScript to progressively enhance your application—don't use JavaScript for core functions
- Maintaining your code structure in such a way that reduces repetition, is better organized, and is easier to read and maintain.
- Adhering to web and accessibility standards

Not only is it good practice to develop this way, but it also ensures that your application will work for a wide range of audiences using different web browsers and devices, even those with limited capabilities. Applications built in this fashion are also generally better organized and structured, perform faster, and are less prone to bugs.

In this article, you'll see how the presentation, style, and behavior layers of your application should be kept separate, with the goal of using no inline CSS or JavaScript event handling. You will also see some examples of obtrusive JavaScript code, and discover the attributes that they have that are considered to be poor programming practice. You will then learn how to correct these issues, writing the same code in an unobtrusive way, with guidelines on some best practices for this style of development. Ajax applications, in particular, are dangerous grounds for unobtrusive code. Just because your application has a rich Ajax interface does not mean that you cannot add this code in a progressively enhancing manner. You will learn how to approach Ajax functions in a way that will provide a fallback for users who cannot take advantage of the fluidity your Ajax features have to offer. Finally, you will see a detailed example of an application that provides dynamic Ajax loading that still works even with JavaScript switched off.

Separating behavior from content

Before the advent of Ajax and Web 2.0, JavaScript was largely used for basic things such as client-side form validation, rollover images, and showing/hiding content. Because each of these features was enabled by a small amount of JavaScript code, the general tendency was to create a couple of functions in a `<script>` block in the `<head>` section of your HTML document, and then attach those functions to events using attributes on HTML elements such as `onclick`, for example: `<input type="button" value="Click Me!" onclick="buttonPressed();">`.

JavaScript is not the only problem when it comes to separating the different parts of your code. CSS styling is also often included inline. Consider the following example: `<input type="button" value="Click Me!" onclick="buttonPressed();" style="background-color: #999" />`.

This style of coding usually starts off as an innocent effort to change the appearance or behavior of an element. The problem is, the more code that is created in this way, the more unmanageable it becomes, and the more difficult it becomes to develop in an unobtrusive manner. Changing JavaScript and CSS code that is written inline is a nightmare in large applications, as it may need to be changed in many, many places—making it extremely easy to miss something in the process.

Instead of coding inline, you should keep all JavaScript and CSS code out of your HTML mark-up. Instead, use reference attributes such as `id` and `class` to make it easy for your JavaScript and CSS to find and identify these elements in the DOM. In the case of the button, you could rewrite this as shown in Listing 1. This article uses the Prototype JavaScript library in all examples; you can substitute this with your favorite library or raw JavaScript should you so wish. See Resources for a link to an article comparing several JavaSript frameworks.

Listing 1. Keeping your JavaScript and CSS code out of your HTML mark-up

```
// HTML code

<input type="button" value="Click Me!" id="my_button" />

// JavaScript code

$("my_button").observe("click", buttonPressed);

// CSS code

#my_button { background-color: #999; }
```

Just as it's important not to mix CSS and JavaScript code in your HTML mark-up, it's also important not to blend CSS code into your JavaScript. The `buttonPressed` function in Listing 1, for example, may result in the style of an element being changed. Look at the example in Listing 2.

Listing 2. Example showing how the buttonPressed function could inadvertently change the style of an element

```
function buttonPressed() {
    $("my_div").setStyle({
        backgroundColor: "#FF6600",
        fontSize: "12px"
    })
}
```

As you can see, this code directly manipulates styles, which is not good, as that job should be left to CSS. Instead, you should use CSS to define the style properties for a given class name, and then use JavaScript to apply that class to the relevant object. So the code in Listing 2 could be improved, as shown in Listing 3.

Listing 3. Improving the code

```
// JavaScript code

function buttonPressed() {
    var my_div = $("my_div");
    if(!my_div.hasClassName("highlight"))
        my_div.addClassName("highlight");
}

// CSS code:

.highlight { background-color: #FF6600; font-size: 12px; }
```

Separating your code in this manner will lead to a highly organized code structure, making it far easier to diagnose and fix any problems that may arise. It is also suggested that you place your JavaScript and CSS code into external files, rather than including them in your HTML code (even if they just reside in the `<head>` section). In the majority of cases, this will also help speed up your application, as external files are cached the first time they are accessed, and don't need to be downloaded again on the next page.

Although you can technically build JavaScript applications that employ progressive enhancement while still using inline code, doing so will lead to messy code that is difficult to maintain. Where possible, you should always strive to keep your JavaScript, CSS, and HTML separated, for your own sake!

Obtrusive JavaScript in action

The best way to describe obtrusive JavaScript is to give an example. All too often you will see links created in the following way: `<a href="#" id="my_link">Click me</a>`.

You might then find an event handler like the following in the JavaScript code: `$("my_link").observe("click", validateAndSubmit);`.

The `validateAndSubmit` function would likely contain some validation rules, and if these are passed, a form is submitted or some other action is performed. While this might seem perfectly innocent, the link will not work at all if JavaScript is not enabled. The browser will look for the URL #, which is, of course, an anchor to the same page as you are currently on, and stop.

Note: As a rule, client-side validation should not be relied upon, as JavaScript can easily be switched off or circumvented, making it a very insecure and unreliable means of validating forms and input. Instead, client-side validation should only be used as a means of improving the user experience and lessening the number of invalid forms sent to the server. You should always validate and cleanse the input on the server side.

In this case, the client-side validation should only come into play if JavaScript is enabled. Fixing this link might be as simple as creating the link as follows: `<a href="page2.html" id="my_link">Click me</a>`.

Now, in your JavaScript code, you can determine whether the link should be followed (see Listing 4).

Listing 4. Determining whether the link should be followed

```
function validateAndSubmit() {
    //validation logic here
    if(valid) {
        return true;
    } else {
        alert("Error!");
        return false;
    }
}
```

In Listing 4, if `valid` is returned as false, an alert box will open, and the link will not be followed, as the event handler has returned a false value.

Another common use of unobtrusive JavaScript is in *quick jump* drop-down lists. See the example in Listing 5.

Listing 5. Quick jump drop-down lists

```
// HTML code

<select id="my_select">
    <option value="">Select...</option>
    <option value="http://www.google.com">Google</option>
    <option value="http://www.yahoo.com">Yahoo!</option>
    <option value="http://www.bing.com">Bing</option>
</select>

// JavaScript code

function goToSearch(e) {
    var el = Event.element(e);
    if($F(el).length > 0)
        window.location = $F(el);
}

document.observe("dom:loaded", function() {
    $("my_select").observe("change", goToSearch);
});
```

Again, in this example, if JavaScript is disabled, nothing will happen when you select an item from the drop-down list. The problem here is that there is no standard HTML fallback to navigate to one of the available pages. Let's rewrite this example, this time using progressive enhancement to include the JavaScript functions (see Listing 6).

Listing 6. Progressive enhancement

```
// HTML code:

<form name="redirect" method="get" action="redirect.php">
<select name="my_select" id="my_select">
    <option value="">Select...</option>
    <option value="http://www.google.com">Google</option>
    <option value="http://www.yahoo.com">Yahoo!</option>
    <option value="http://www.bing.com">Bing</option>
</select>
<input type="submit" value="Go!" />
</form>
```

In this example, you use an HTML `<form>` element, which allows you to define a server-side script that will take care of the redirection, should JavaScript not be available. If JavaScript is switched off, when the user clicks the **Go!** button the form will submit to redirect.php, passing the URL in the query string. The server can then validate that a valid option was selected and redirect the output based on the input received.

In this example, you could use JavaScript to progressively enhance this in two ways:

- Validate that the user has selected a search engine
- Perform the redirection without requiring another HTTP request to be sent to the web server

Let's go ahead and implement these improvements using JavaScript (see Listing 7).

Listing 7. Implementing improvement using JavaScript

```
// JavaScript code

function goToSearch(e) {
    Event.stop(e);

    var my_select = $F("my_select");
    if(my_select.length > 0)
        window.location = my_select;
    else
        alert("You must select a search engine!");
}
```

```
document.observe("dom:loaded", function() {
    document.redirect.observe("submit", goToSearch);
});
```

Listing 7 stops the default action from running (it prevents the form from submitting to the server) and then validates that a search engine has been selected. If it has, it redirects the user to the selected URL. If it has not, it displays an alert box with an error message.

You may notice that the example in Listing 6 added a **Go!** button, whereas the original example worked on the `onchange` event of the `<select>` element itself. The problem here is that you cannot submit a form that only contains this element without a submit button or JavaScript. If you were determined to have no **Go!** button, you could use JavaScript to dynamically set the style of the button so that it is hidden on browsers that have JavaScript enabled. That way, it is still present in browsers that don't have JavaScript enabled, but it is hidden in browsers that do. You could then attach the event handler to the `onchange` event of the `<select>` element and have the best of both worlds.

Writing unobtrusive code

In order to write code in an unobtrusive manner, you should start by developing a solution that will work for the lowest common denominator—a browser that does not support JavaScript. Most modern web browsers either have a built-in option to disable JavaScript or have a plug-in available. Use this to see your site through the eyes of a user who doesn't have the luxury of JavaScript available to them.

Start off by writing your HTML code without JavaScript, and write the HTML so that it performs all the basic functions the page needs to provide. If you need to submit input data, use a `<form>` tag with a real action attribute. If you need to link to other items, use good old-fashioned hyperlinks; you won't have the ability to attach events to unsuitable elements like `<img>` or `<div>`. Build a working application in this way. It doesn't have to be pretty or offer the greatest user experience ever known, but it must work.

When you are satisfied that your application or web page works without any JavaScript, while providing the basic requirements it sets out to fulfill, you can move on to progressively enhancing the page using JavaScript. By developing your application in this way, your page will have more semantic meaning, as it is more likely to use HTML elements for their intended purpose. One of the other great benefits of developing in this way is that features that you may have otherwise crippled will still be enabled.

A good example of this is using form fields without a `<form>` tag, particularly when it comes to Ajax applications. It is all too easy now to not bother including a `<form>`

element in your application, and use DOM methods like `document.getElementById` to get the values you need to submit, and submit them to the server using an `XMLHttpRequest`. The problem with building applications in this way is that default events are not taken into account, and the end result is your application becomes less usable. For example, say you have a search text box and a button that performs a search of your site using the `XMLHttpRequest` described in this section, with no `<form>` element used in the process. It would be fair to expect that when you press the Return key while typing your query in the text box, that the form would be submitted and the search performed. Unfortunately, this won't be the case, and you might find yourself running off to figure out how to capture the keypress event to mimic this function. The irony is, if you had used the `<form>` element in the first place, pressing the Return key would have actually worked. This is just one example of the many overlooked default behaviors that HTML elements provide, that continue to be abused and misused by developers every day.

Let's work through a simple example of writing a piece of code unobtrusively. Imagine you have a thumbnail image, and you want to allow the user to click on the thumbnail and show a larger version of the image in a lightbox (a kind of modal pop-up window that appears, with the page faded into the background using a mask overlay effect). Instead of just creating an `<img>` tag, attach an `onclick` event to it and call the lightbox to show the large image. Let's work through how this should be done to cater to those users who are without JavaScript support.

The first thing to do is to plan out what it is you are trying to do. You want to click on an image, and have a larger image appear, without leaving the page you are currently on. OK, so you want to add fancy lightbox effects, but stripped down to the basic requirements; this is essentially what you need to do. The HTML-only solution is very straightforward (see Listing 8).

Listing 8. HTML-only solution

```
<a href="large.jpg" id="my_thumb" target="_blank"><img src="thumb.jpg"
width="50" height="50" alt="My Picture" /></a>
```

Listing 8 contains the basic requirements met using nothing but plain old HTML. This is fine, but it's not pretty. For those with JavaScript, you want to make the large image show up in a lightbox overlay in the current window, not in a new window. You can do this as shown in Listing 9 (assuming your lightbox is launched by a function `openLightbox`, which accepts the large image URL as an argument):

Listing 9. shownLargImage() function

```
function showLargeImage(e) {
    Event.stop(e);
    var link = Event.element(e).up("a");
    openLightbox(link.href);
}

document.observe("dom:loaded", function(e) {
    $("my_thumb").observe("click", showLargeImage);
});
```

To take this even a step farther, you could have a gallery of thumbnails, and may want to open a lightbox with the relevant large image for each thumbnail. This is also straightforward. Rather than use IDs, you could use a class name or as is commonly used in lightbox scripts, the `rel` attribute (see Listing 10).

Listing 10. Creating a gallery of thumbnails

```
// HTML code

<a href="large1.jpg" rel="lightbox" target="_blank"><img src="thumb1.jpg"
width="50" height="50" alt="Picture 1" /></a>

<a href="large2.jpg" rel="lightbox" target="_blank"><img src="thumb2.jpg"
width="50" height="50" alt="Picture 2" /></a>

<a href="large3.jpg" rel="lightbox" target="_blank"><img src="thumb3.jpg"
width="50" height="50" alt="Picture 3" /></a>

// JavaScript code

function showLargeImage(e) {
    Event.stop(e);
    var link = Event.element(e).up("a");
    alert(link.href);
}

document.observe("dom:loaded", function(e) {
    $$("a[rel=lightbox]").each(function(thumb) {
        thumb.observe("click", showLargeImage);
    });
});
```

As you can see, the end result is some very clean, readable, and easy to maintain code. You can attach events to multiple items without needing to use unnecessary `id` attributes or having to attach the handlers in an inline fashion. I am shuddering at the idea of how this might be handled if you wrote it obtrusively. I can picture no <a> tags, with nothing working in non-JavaScript browsers. I can see inline `onclick` handlers on the `<img>` elements, with the URL of the large image passed as an argument directly in the mark-up itself. By taking a step back and trying to map out what it is I was trying to achieve, I was able to provide a solution that is accessible, adheres to standards, has semantic meaning, and provides neat lightbox functions in a progressive manner.

Unobtrusive Ajax

The advent of Web 2.0 and Ajax applications has led to a new challenge for web developers when it comes to creating interactive web applications. The advantages of using asynchronous HTTP requests to retrieve data are clear—applications that take advantage of Ajax are more responsive, easier to use, and are edging ever closer to being as usable as a traditional desktop application might be (some would argue that well-designed web applications are even more usable and accessible than traditional applications).

The problem with using Ajax requests is that it can very quickly blur your vision when it comes to catering to those users who don't have the luxury of a JavaScript application. Take a typical registration form, for example. There are many ways you can *Ajaxify* this type of form. You could automatically check that a user name or email address is not already in use, validate the form without refreshing the page, or even submit the form and display a result dynamically.

Let's take a simple form that requests a user name and password, which sends an asynchronous HTTP request to a server-side script, which then returns a response of "ok" if the process was successful, and an error message if there was a validation issue. The code form of the HTML form might look like Listing 11.

Listing 11. Simple form

```
<form id="register">
        <label for="username">Username</label>
        <input type="text" name="username" id="username" />
        <label for="password">Password</label>
        <input type="password" name="password" id="password" />
        <input type="submit" value="Register" id="reg_button" />
</form>
```

The first problem should be very evident. The form in Listing 11 has no method or action attribute, which means that when you press the **Register** button, nothing will happen. Just because you can easily use JavaScript to get the form by its `id` attribute and submit it using Ajax does not mean you should forget about the standard method of submitting a form. Instead, you could write your server-side script to look for a extra field in the form. If it's not available, you know that the request is made through a regular form post, and you should serve back a full page. If it's set to 1, you know the request was made using an asynchronous Ajax call, and you should just send back a message with `OK` or an error. You would then set the value of the field to 1 in your Ajax request, which, of course, won't be executed if the browser does not have JavaScript support.

A revised form might look like Listing 12.

Listing 12. Revised form

```
<form id="register" method="post" action="register.php">
    <label for="username">Username</label>
    <input type="text" name="username" id="username" />
    <label for="password">Password</label>
    <input type="password" name="password" id="password" />
    <input type="submit" value="Register" id="reg_button" />
</form>
```

Now you can take this form and submit your Ajax request to the same server-side script, which will easily be able to distinguish how it should send its response (redirect back to register page with an error if the Ajax field doesn't exist, or just output a simple message if the value is 1). The JavaScript code (with a little help from the Prototype library) to perform this action might be as follows in Listing 13.

Listing 13. Using the form to submit your Ajax request to the server-side script

```
<script type="text/javascript">
    function registerSuccess(transport) {
        if(transport.responseText == "ok")
            alert("Success!");
        else
            alert(transport.responseText);
    }

    function registerFailure(transport) {
        alert(transport.status+' '+transport.statusText);
    }

    function submitUsingAjax(e) {
        Event.stop(e);
        var options = {
            parameters: {
                ajax: "1"
            },
            onSuccess: registerSuccess,
            onFailure: registerFailure
        }
        $("register").request(options);
    }

    document.observe("dom:loaded", function() {
        $("register").observe("submit", submitUsingAjax);
    });
</script>
```

In Listing 13, you add an event handler to the `submit` event of the form. This handler prevents the default event (submitting the form) from being fired, sets the value of the extra `ajax` argument to 1, and makes an `ajax` Request using the original form for the URL and data. The `registerSuccess` function will be called if a successful HTTP response code is received, or in the case of an error, the `registerFailure` function will be called.

Another unobtrusive Ajax example

Another scenario where Ajax is particularly useful is in grouping large sets of data into pages, usually referred to as pagination. If you have a set of search results, for example, rather than show several hundred or more search results at a time, you'll more likely show a smaller subset of this data, for example, 10 records, and give the user the option to move forward and back between the pages. An example of this is Google's search results. At the bottom of the page you get the option to navigate between pages of the results.

Typically, the standard way of moving between pages is to pass a parameter to the server-side script loading the data, telling it what page of the result set it should output. The problem is, each time the user needs to move to a different page, you must send a request back to the server, which would cause the page to reload with the new set of data.

With Ajax, you can ensure that the user doesn't have to watch the entire page reload when a new page of data is being fetched, but instead just replace part of the page with the new set of data. Let's see an example of how this might work.

First and foremost, as always, you need to ensure that you cater to those users without JavaScript. To do this, you need to ensure that your application works in a traditional manner, with the page refreshing with a new set of data (see Listing 14). Again, you'll use the concept of passing an `ajax` flag to the server-side script to let it know whether it is being called by a regular `GET` request or by an Ajax call.

Listing 14. Ensuring that the application works for users without JavaScript

```
//HTML code

<div id="results">
    <ol>
        <li>Result 1</li>
        <li>Result 2</li>
        ...
        <li>Result 10</li>
    </ol>

    <a class="paging" href="results.php?page=2">Next Page</a>
    <a class="paging" href="results.php?page=10">Last Page</a>
</div>
```

The results.php server-side script would produce a page like the one in Listing 14 for page 1, and clicking the links would bring you to the second page or the last page. If you were not on page 1, you'd also see the Previous Page and First Page links, and if you were on the last page you wouldn't see the Next page or Last Page links. You could even show a list of pages to make it easy to jump to a particular page. So how do you *Ajaxify* this paging section without breaking it for non-JavaScript browsers? It's simple really, you just get a reference to all paging links, stopping the default action from firing when they are clicked (the browsers replace the current page with the page in the `href` attribute of the link). You then get the value of the `href` attribute and use this as the URL for your Ajax request. Finally, you tag your `ajax` parameter so the server-side script will know that it's an Ajax request (see Listing 15).

The server would then either return an entire page, or just the section you want to replace with the requested set of data, depending on whether it was called using Ajax or a regular `GET` request. Your function then replaces the contents of the `results` div with the HTML returned by the server, containing the `<ol>` list of results and the pagination links (which may change as you move from page to page).

Listing 15. Ajaxifying the paging section

```
<script type="text/javascript">
    function movePageSuccess(transport) {
        $("results").innerHTML = transport.responseText;
    }

    function movePage(e) {
        Event.stop(e);
        var el = Event.element(e);
        var url = el.href;
        var options = {
            method: "get",
            parameters: {
                ajax: "1"
            },
            onSuccess: movePageSuccess
        }
    }

    document.observe("dom:loaded", function() {
        $$(".paging").each(function(link) {
            link.observe("click", movePage);
        });
    });
</script>
```

There you have it—another straightforward Ajax example. As I'm sure you can see, it's relatively simple to work with Ajax in a way that progressively enhances your application rather than making JavaScript a compulsory requirement. In addition, by employing better coding practices, you'll actually find that working with Ajax in an unobtrusive way is much easier and efficient than using a large amount of JavaScript to do the same thing in a broken manner.

Conclusion

This article introduced you to the concept of unobtrusive JavaScript, progressive enhancement, and the idea of designing your application without JavaScript first, but then adding it to improve the user experience for those who benefit from it. It is by no means an exhaustive resource on how to develop applications in an unobtrusive manner. But if, like me, you have spent a lot of time developing web applications that aren't necessarily working with JavaScript and Ajax unobtrusively, you might be surprised by just how beneficial it is to embrace this web development philosophy.

Resources

Learn

- "Ajax overhaul, Part 1: Retrofit existing sites with Ajax and jQuery" (developerWorks, March 2008) shows how you can eliminate pop-up windows and navigational dead-ends with simple modal windows.
- "Mastering Ajax, Part 2: Make asynchronous requests with JavaScript and Ajax" (developerWorks, January 2006) explains how to use Ajax and the XMLHttpRequest object to create a request/response model that never leaves users waiting for a server to respond.
- "Where and when to use Ajax in your applications" (developerWorks, February 2008) describes how you can use Ajax to improve your websites while avoiding bad user experiences.
- *Unobtrusive Ajax* by Jesse Skinner (from O'Reilly Media) is about making web applications that work for everyone all the time, even if you have JavaScript turned off.
- Behavioral Separation by Jeremy Keith looks at separating content, style, and behavior in website design.
- Unobtrusify.com provides a clever demonstration of unobtrusive JavaScript in action.
- Progressive Enhancement with JavaScript by Aaron Gustafson discusses how to apply the progressive enhancement philosophy to client-side scripting.
- The seven rules of unobtrusive JavaScript were developed by Christian Heilmann through years of teaching and implementing JavaScript in an unobtrusive manner.
- Separating Behavior From Structure from Adobe labs provides background and examples on unobtrusive JavaScript.
- Check out the Unobtrusive JavaScript website.
- "Improve the performance of Web 2.0 applications" (developerWorks, December 2009) explores different browser-side cache mechanisms.
- "Create Ajax applications for the mobile web" (developerWorks, March 2010) explains how to build cross-browser smartphone web applications using Ajax.
- "Ajax performance analysis" (developerWorks, April 2008) examines toolsets that find and correct performance problems within your Ajax-enriched application.
- "Compare JavaScript frameworks" (developerWorks, February 2010) provides an overview of the frameworks that greatly enhance JavaScript development.

- The developerWorks Web Development zone specializes in articles covering various web-based solutions.
- To listen to interesting interviews and discussions for software developers, check out developerWorks podcasts.
- developerWorks technical events and webcasts: Stay current with developerWorks technical events and webcasts.

Get products and technologies

- Prototype is a JavaScript Framework that aims to ease development of dynamic web applications. The latest version is 1.6.
- Innovate your next development project with IBM trial software, available for download or on DVD.

Discuss

- Create your My developerWorks profile today and set up a watchlist on JavaScript or Ajax. Get connected and stay connected with My developerWorks.
- Find other developerWorks members interested in web development.
- Share what you know: Join one of our developerWorks groups focused on web topics.
- Roland Barcia talks about Web 2.0 and middleware in his blog.
- Follow developerWorks' members' shared bookmarks on web topics.
- Get answers quickly: Visit the Web 2.0 Apps forum.
- Get answers quickly: Visit the Ajax forum.

About the author

Joe Lennon

Joe Lennon is a 25-year-old mobile and Web application developer from Cork, Ireland. Joe works for Core International, where he leads the development of Core's mobile HR self service solutions. Joe is also a keen technical writer, having written many articles and tutorials for IBM developerWorks on topics such as DB2 pureXML, Flex, JavaScript, Adobe AIR, .NET, PHP, Python and much more. Joe's first book, *Beginning CouchDB* was published in late 2009 by Apress. In his spare time, Joe enjoys travelling, reading and video games.