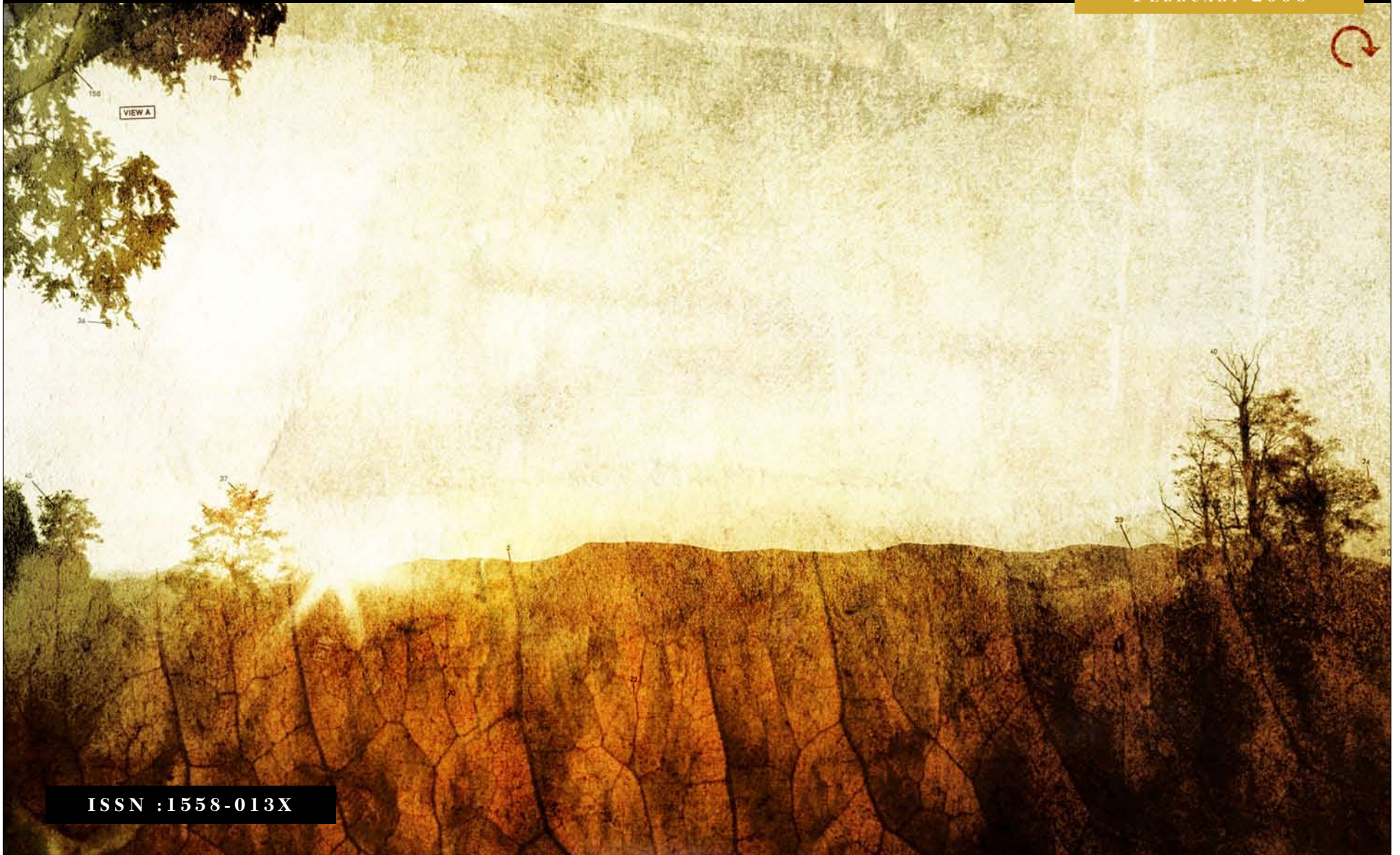


treehouse.

A Web Development Magazine



FEBRUARY 2006



ISSN : 1558-013X



Cover Notes

by Brian Campbell

For me, one of the brilliant things about artwork is that it can be as mysterious or as obvious as it wants to be. There is no proper or incorrect— way to look at a piece because so much of its beauty is in an individual's interpretation. Perhaps the 'rotate' icon in this month's cover is too heavy-handed, but I wanted an image that quite specifically could be viewed in more than one way. The most natural progression of that seemed to be in constructing an image that could viewed both vertically and horizontally. The icon and diagrams seek to give the piece context.

Treehouse Logo

by Julius Santiago



From the Editors <i>by Kevin Hale</i>	3	Cartoon <i>Robot Tea Party</i>	26	Marketplace <i>Classifieds</i>	50
To the Editors <i>Letters & Comments</i>	4	Network News <i>Around 9rules</i>	27		

CODE

EASY AJAX WITH ASK <i>by Robert Nyman</i>	7	HOW TO EXTRACT A DESIGN STRATEGY <i>by Kyle Neath</i>	17
Interview SAM STEPHENSON	12	Interview COLE POELKER	20
Book Review <i>Web Application Development with PHP and MySQL</i>	14	Book Review <i>Flash Professional 8 Unleashed</i>	23
Best of the Web <i>Code Links</i>	15	Best of the Web <i>Design Links</i>	24

DESIGN

BUSINESS

CONVERTING WEB VISITORS INTO BUYERS <i>by Yaro Starak</i>	29	PRELOADING DATA WITH AJAX AND JSON <i>by Ryan Campbell</i>	39
Interview JESSICA LIVINGSTON	32	LIGHTBOX GONE WILD! <i>by Chris Campbell</i>	44
Book Review <i>All Marketers are Liars</i>	36		
Best of the Web <i>Business Links</i>	37		

DIGEST

SURFACE INNOVATION : A VALENTINE TO CUSTOMERS

By Kevin Hale

Last month, William Grosso synthesized a very thought-provoking look at the current attitudes following our generation of startups into Silicon Valley. On his blog ([Wander, Think, Repeat](#)), he believes there aren't enough people thinking about big ideas—about changing the world, about deep innovation.

He writes, "we're taking a generation of entrepreneurs and technologists and training them in the art of short-term thinking (when a swarm of people are pursuing the same goal, there's going to be a lot of small releases and jockeying for incidental advantages. Which means customer-driven and surface innovation, not deep innovation)."

My initial reaction to Grosso's writing was, for the most part, to agree with him. I look at the line-ups at the conferences and the symposiums this year and it feels like 2006 might be just a lot more 2005. If Web 2.0's mascot is the mashup (I wonder what that costume would look like—platypus?), then our song is most certainly a Bach fugue—re-configured variations on a common theme. It's just not a song I'd like to hear on repeat.

Our team is obsessed with the paths towards

innovation. We believe that old adage about innovation's momma and so everything we conjure up here derives from one source—trouble. And so after nights about thinking about how a small company can get on the path towards deep innovation, I realized that the biggest trouble of our predecessors was in their attitudes toward the inexperienced. They weren't bringing everyone along for the ride.

Now that I think of it, the greatest innovation of our era *is* that customer-driven surface innovation. Accessibility is the new king for a reason. If we're to get to where we want to go we have to bridge the paths for all our audiences. Peter Drucker, a man known for his strong beliefs about a company's need to innovate, believed that the primary purpose of any business was to create a customer.

And we haven't made that formulaic quite yet. Deep innovation comes with long term survival and I think any small company hoping to survive long enough to make that kind of impact could only do so with the support of their customers. With them happy, it's much easier to change the world.

treehouse.

FEBRUARY 2006

Volume II. Issue II.

Editor-in-Chief
Kevin Hale

Managing Editor
Ryan Campbell

Business Manager
Chris Campbell

Copy Editor
Elizabeth Dehnart



OUR MISSION

To make the web
easier to improve.

OUR MANTRA

Give them our best and toss in
a little fun while we're at it.



© 2005 Particletree Inc.
All rights reserved.

All scripts, however, are managed under a
[Creative Commons License](#) and individual articles
are the copyright of their respective owners.

Hacker → Founder

Can you write good software? Can you *design* good software? If you're good enough at hacking to build something users like, you already have 80% of what you need to start a startup. We can teach you the rest.

The deadline for applying to Y Combinator's Summer Founders Program is **February 13th**.



<http://ycombinator.com>

digest.

“Software and cathedrals are much the same
—first we build them, then we pray”

Sam Redwine, Jr.



PRELOADING DATA WITH AJAX AND JSON

By Ryan Campbell

Web applications have made huge leaps and bounds in improving user experience thanks to a lot of recently developed fancy-pants technology. In an effort to help you take it up a notch, we'd like to share a method for helping your site anticipate a user's next move.

On most sites, there are usually only a handful of options that can be done at any given moment and often some of these options are more likely than others. By determining what is important on each page, we can preload the data of the user's next action and store it on the client with JSON. When the user decides to perform their next action, they'll see their results almost instantly because the info was loaded in the background.

As always, we've created a little demonstration for you to observe, illustrate, and dissect. It's a paging demo that shows how we can preload in the background the next and previous queries from a database. When the user pages through, they can do so quickly without having to wait for an Ajax call to finish.

[Download the code](#)

Also note that this demo and the tutorial that follows use the [Prototype.js](#) framework along with our own custom database class. You'll find both in the zip file above.

JSON

Before we get started on the paging, let's take a quick look at [JavaScript Object Notation](#),

or JSON. JSON is a data interchange format, similar to XML, which is built using JavaScript. The easiest way to get your head around it is to imagine the format as a variable containing multiple arrays that can be nested endlessly. Implementing JSON in JavaScript is simple and the [official documentation](#) actually does a great job illustrating lots of examples. For this tutorial, we will be returning a string of JSON from the server, and that string will contain our preloaded data. The only tricky part is that once the data is fetched, we'll have to convert it into a JavaScript object after the client receives it. To do that, we'll use `eval()`.

```
currentPage = eval('(' + response.responseText + ')');
```

The JavaScript line above makes use of the

DIGEST

SEE DEMO

DOWNLOAD FILES

VISIT ORIGINAL

`eval` function, which converts the properly formatted string from the server into a JavaScript object. Once that is ready to go, the rest is easy. Now, if you're not comfortable using JSON, you can always use XML instead. To compare the two, check out PPK's review. That man never stops doing his homework.

Determining What to Preload

Once we know how to store our data, we'll want to determine what kind of data we'd like to store. As far as I am aware, any amount of data can be stored on the client side. It is all dependent on the client computer (however, do correct me if I am wrong). Chances are, we don't wish to crash the client, so the choice of and amount of data becomes important. For example, an online store with hundreds of thousands of products may not want to load every single product into client memory. Likewise, loading millions of records of data that need to be paged is also not ideal, so for this example we just set up a simple next and previous page to always be preloaded.

Getting Started

To set up a paging system, we need to know a few things: the total amount of records, the current page, the next page, the previous page, and the paging size. Starting from the initial page load, here is what needs to happen:

- Grab the total amount of records so we know how many total pages there might be.
- Grab the first page of data and display it to the user.
- Grab the next page of data and store it as JSON on the client.

No previous page of data is needed, since record 0 is always loaded first. Let's take a look at each step individually.

Total Records

To get the total amount of records possible, we need to create a basic query that returns the value to the client.

```
function getRecordCount() {
    $db      = new database();
    $sql     = "SELECT COUNT(*) AS recordCount FROM Accounts";
    $rs      = $db->query($sql);
    $row     = mysql_fetch_array($rs, MYSQL_ASSOC);
    echo    $row['recordCount'];
}
```

This function simply gets the record count and prints the value. The following Ajax call retrieves the value:

```
function getRecordCount() {
    var myAjax = new Ajax.Request(
        'flow.php?action=count',
        {
            method: 'get',
            parameters: '',
            onComplete: function(response) {
                recordCount = response.responseText;
            }
        }
    );
}
```

The global variable, `recordCount`, is now initialized. Also, note that the parameter, `action`, in the querystring, is used to tell the server what action to perform. In this case, it told the server to return the record count.

Get Initial Data

Once the user first hits the page, we need to show them the first page of data. To do that, we have to do a few things. First, let's look at the function that will draw out the HTML that the user sees.

```
function drawTable(page, contain) {
    table = '<table>';
    alt = '';
    for(i = 0; i < page['players'].length; i++) {
        table += '<tr class="'+alt+'">' +
            '<td>' + page['players'][i].lastName + '</td>' +
            '<td>' + page['players'][i].firstName + '</td>' +
            '<td>' + page['players'][i].position + '</td>' +
            '</tr>';
        (alt == '')
            ? alt = 'alt'
            : alt = '';
    }
    table += '</table>';
    contain.innerHTML = table;
}
```

This function creates a table, and inserts it as the innerHTML of an element with the ID of container. You'll notice when the table is drawing, it is looping through an object named page. The next step is to take a look at the page object. The page object is a string of data that the server returned to the client, and the client then converted into a JSON object. On the server side, the data is retrieved and displayed like this:

```
function getTableData() {
    $ret = '{"players" :[';
    $db = new database();
    $sql = "SELECT * FROM Accounts LIMIT ".$_
```

```
GET['current'].", "._GET['size'];
    $rs = $db->query($sql);

    while($row = mysql_fetch_array($rs, MYSQL_ASSOC)) {
        $ret .= '{ "firstName" : "'.$row['FirstName'].'",
            "lastName" : "'.$row['LastName'].'", "position" :
            "'.$row['Position'].'"} , ';
    }

    $ret = rtrim($ret, ', ').'>';
    echo $ret;
}
```

Then, using the eval() function mentioned earlier:

```
currentPage = eval('(' + response.responseText + ')');
```

We convert the server response into a JavaScript object that looks like this:

```
{ "players" : [
  { "firstName" : "Ryan", "lastName" : "Campbell", "position" : "S" },
  { "firstName" : "Chris", "lastName" : "Campbell", "position" : "QB" },
  { "firstName" : "Kevin", "lastName" : "Hale", "position" : "DT" }
]}
```

Once the object is created, the drawTable() function just loops through all players and displays their information. So now the user is viewing the initial data, and we wish to preload the next set of data. To do this, we only have to recreate what we just did, and not call the drawTable() function. Also, we need to store the response in a variable called nextPage instead of currentPage.

Demonstration

Preloading Data with Ajax and JSON

Preloading Data with Ajax and JSON

Preloaded Previous	<u>Previous</u>	<u>Next</u>	Preloaded Next
Adams, Blue CB	Clinkscale, Jonathan G/C	Graham, Earnest RB	
Allen, Jared QB	Colmer, Chris T	Heller, Will TE	
Allen, Will S	Cook, Jameel FB	Hilliard, Ike WR	
Alstott, Mike FB	Cooper, Marquis LB	Hovan, Chris DT	
Barber, Ronde CB	Cox, Torrie CB	Jackson, Dexter S	
Becht, Anthony TE	Davis, Anothony T	Jackson, Scott C	
Beitia, Xavier K	Davis, Carey FB	Johnson, B.J. WR	
Bidwell, Josh P	Douglas, Robert FB	Jones, Mark WR	
Bogle, Phil G	Galloway, Joey WR	Kelly, Brian CB	
Bolden, Juran CB	Gessner, Chas WR	Lewis, Jacque RB	

Record 10

This is a paging demo that shows how to preload the next and previous page of data using JSON. When the user pages through, they can do so quickly and see results without having to wait for an Ajax call to finish.

[Download Source](#) · [Return to Tutorial](#)

Preload data to improve the user experience. By guessing the next move of the user, we can preload the data associated with it. This example shows how the previous and next page of table data can be preloaded as the user pages through.

Paging

At this point, we have the user viewing the current page of data, and that data is stored in a JSON object named `currentPage`. We also have the next page of data stored in a JSON object named `nextPage`. So, when the user clicks the “next page” button, we want to do a few things:

```
function getNextPage() {
    currentRecord += pagingSize;
```

Increase the current record by the paging size. If they were viewing records 0 - 20, and they hit “next page,” the current record will be increased by 20, making the next set 20 - 40.

```
    showNavigation();
```

Explained below, this function hides and shows the previous and next buttons as necessary.

```
    previousPage = currentPage;
```

We know the previous page will become the current page, since the current page is advancing one. No need to hit the server to get previous data—just change the variables.

```
    currentPage = nextPage;
```

`nextPage` was preloaded into a JSON object,

so we can now set the current page to the next page.

```
drawTable(currentPage, $('view'));
```

Draw the new currentPage to the screen.

```
getNextData();  
}
```

Preload the new nextPage. This will work until the last page is hit. When a user clicks “previous page,” the same thing happens in reverse order.

Controlling the Navigation

The last thing to do is hide the “next page” button if the user is on the last page, and hide the “previous page” button if they are on the first. The following function should do the trick:

```
function showNavigation() {  
  (currentRecord == 0)  
    ? $('previousLink').style.visibility = 'hidden'  
    : $('previousLink').style.visibility = 'visible';  
  ((currentRecord + pagingSize) >= recordCount)  
    ? $('nextLink').style.visibility = 'hidden'  
    : $('nextLink').style.visibility = 'visible';  
}
```

Basically, this function just changes the style based on the currentRecord. If we’re at 0, then we’re on the first page. If the currentRecord equals the pagingSize (or total number of pages), then we’re on the last. Hide and show as deemed fit.

Improvements and Possibilities

Lately, I have found JavaScript to be fairly stable except when a user spams an action. The same problem applies here. If a user spams the “next page” button, the code may get thrown off. In order to prevent this problem, I sometimes find it necessary to create global variables, such as `isActive`, that prevents anything from happening until the variable is false. Other than that, everything should work out fine when preloading data.

I also mentioned earlier on that a store could use this technique. Right now, Ajax is fast, but it’s not instant since it still has to hit the server. By storing the current page of cart items on the client, you could add to the cart instantly and the user would have no wait while browsing through products. The store example and the example I used here both emphasize heavy sets of data, but it’s important to also realize that this can be used for trivial things like preparing the next set of HTML the user will see, or allowing for mass saving at the end of multiple actions rather than saving each individual action as it happens.

As always, we’d love to hear about some more possible uses of this technique, and suggestions for improving it are always welcome.





LIGHTBOX GONE WILD!

By Chris Campbell

In user interface design, a modal window (sometimes referred to as a modal dialog) is a window that blocks input to other windows. It has to be closed before the user can continue to operate the application and are frequently an element of Multiple Document Interface (MDI) applications or desktop applications like Windows or OS X. One of their purposes is to prevent the software from being operated in an ambiguous state.

While researching the best way to recreate a modal window for our current project, we ran into Lokesh Dhakar's lightbox.js and we knew we found a winner delivery container. Dhakar's method, however, while fantastic, was a bit too specific for our purposes and so we created our own implementation that we think is a bit more flexible for extending a web site's interface.

In this tutorial, we'll take a look at how to create a

modal window using some nifty JavaScript and CSS. After a quick walkthrough, we'll take a look at the JavaScript and CSS files that make it all tick.

Implementation

For those of you at work wanting to impress the boss, here's the low down on getting the script working. After downloading the files, upload the lightbox.css, lightbox.js and prototype.js files into your web directory and include them in the head of your html document like so:

```
<link rel="stylesheet" type="text/css"
href="css/lightbox.css" />
<script type="text/javascript"
src="scripts/prototype.js"></script>
<script type="text/javascript"
src="scripts/lightbox.js"></script>
```

DIGEST

SEE DEMO

DOWNLOAD FILES

VISIT ORIGINAL

Note:

This demo and the tutorial that follows use the Prototype.js framework. You'll find a compressed version of Prototype in the zip file above.

Apologies if you're not a Prototype fan, but it saves a lot of time around here and it's our flavor of choice. Obviously, you're always free to reverse engineer any Prototype class into normal JavaScript.

Create some Lightboxes

Create an external document that contains the markup for whatever you want to be loaded into a lightbox. For example, we created a file called `text.html` and included the following markup:

```
<h3>what is a Lightbox?</h3>
<p class="highlight">Lightbox is an
unobtrusive script used to overlay
content on the current page.</p>
```

Because we're just inserting the HTML snippets into the lightbox you could display a login form, a photo, additional settings for your web app, help documentation, etc. The possibilities are endless.

Activating Lightbox

To call your lightbox interface, just link to the external file and set the class to `lbOn`.

```
<a href="form.html" class="lbOn">
Email This</a>
```

Deactivating Lightbox

If you want people to be able to close the lightbox after it's open, include a link with class `lbAction` and a `rel` of `deactivate` in the external file:

```
<a href="#" class="lbAction"
rel="deactivate">
Close Lightbox.</a>
```

Linking to a Another Lightbox within a Lightbox

If you want to load a different lightbox within an already open lightbox set the `rel` attribute to `insert` and your `href` to the file you want to load instead.

```
<a href="confirm.html" class="lbAction"
rel="insert">Go to Another Lightbox</a>
```

And you're done.

How it Works

In a nutshell, when the user clicks on a link with a class of `lbOn`, a transparent div is positioned on top of the webpage to present a visual cue that the attention is now focused on our lightbox modal window. After the overlay is set, a div (`lightbox`) is positioned on top of the transparent overlay and loaded with infor-

mation the user can interact with. When the page first loads, our script inserts the following markup right before the closing body tag.

```
<div id="overlay"></div>
<div id="lightbox">
  <div id="lbLoadMessage">
    <p>Loading</p>
  </div>
</div>
```

The `overlay` div is responsible for holding dimming the rest of the page. We'll be inserting our different lightbox interfaces inside the `lightbox` div. Feel free to change the information inside the `lbLoadMessage` if you want to present something more exciting than just some text. When the page loads, we're attaching the lightbox object to each element with a class of `lbOn`.

```
function initialize(){
  lbox = document.getElementsByClassName('lbOn');
  for(i = 0; i < lbox.length; i++) {
    valid = new lightbox(lbox[i]);
  }
}
```

One of the differences between the original `lightbox.js` script and ours is that Dhakar was using JavaScript to determine the size of the HTML content so he could find and fix the position of the lightbox and `overlay.png` at the center of the screen. Wanting to move as

much of the presentation to CSS, we decided to use `position:fixed` to center our lightbox so that it simplifies things in all modern browsers. Unfortunately, “all modern browsers” doesn’t really include IE6 and below and so we’ll still have to use some JS to help them out.

In IE6, there’s no easy way to stretch the overlay div, which dims our page across the entire content of the HTML document to achieve a position fixed for our lightbox div. (This is a non-problem in the recent IE7 beta, but until that becomes popularly adopted, we’ll need to do the following.) To remedy this problem, we’re just going to use `position:absolute` instead in the CSS and hide that we’re using `position:fixed` for IE browsers.

```
#lightbox{
  display:none;
  position: absolute;
  top:50%;
  left:50%;
  z-index:9999;
  width:500px;
  height:400px;
  margin:-220px 0 0 -250px;
}
#lightbox[id]{ /* IE Can't See This */
  position:fixed;
}
#overlay{
  display:none;
  position:absolute;
  top:0;
  left:0;
  width:100%;
  height:100%;
  z-index:5000;
```

```
background-color:#000;
-moz-opacity: 0.8;
opacity:.80;
filter: alpha(opacity=80);
}
#overlay[id]{ /* IE6 Can't See This */
  position:fixed;
}
```

If you’re familiar with Dhakar’s method, you’ll notice that we’ve decided not to use a transparent png to get our effect and instead use CSS for our transparency effect. This makes it easier to change the percentage of transparency and the color of the transparency in the overlay CSS rules. To make the overlay stretch out to the full screen for IE users, we’ll need to set the body and html elements’ height to 100% and the overflow to hidden for those browsers.

```
prepareIE: function(height, overflow){
  bod = document.getElementsByTagName('body')[0];
  bod.style.height = height;
  bod.style.overflow = overflow;

  htm = document.getElementsByTagName('html')[0];
  htm.style.height = height;
  htm.style.overflow = overflow;
}
```

To make everything seamless for IE users, we’ll also use a `scrollTop()` function to find the current position of the scrollbar to jump the user to the top of the page where the lightbox is located (because it’s position absolute rather than fixed) and then use `setScroll()` to bring them back to their location when they deactivate the lightbox.

scrollTop() & setScroll()

```

scrollTop:function (){
    var yScroll;

    if (self.pageYOffset) {
        yScroll = self.pageYOffset;
    } else if (document.documentElement && document.
documentElement.scrollTop){ // Explorer 6 Strict
        yScroll = document.documentElement.scrollTop;
    } else if (document.body) { // all other Explorers
        yScroll = document.body.scrollTop;
    }

    this.yPos = yScroll;
},

setScroll:function(x, y){
    window.scrollTo(x, y);
},

```

hideSelects()

Due to a bug in IE, select elements tend to position themselves on top of the overlay (on top of everything actually, including flash). To fix this, we're just going to hide them. We've also had some problems with Firefox and Flash 8. If you're using the latest Flash in your application, you can modify the script to hide them too.

```

hideSelects: function(visibility){
    selects = document.getElementsByTagName('select');
    for(i = 0; i < selects.length; i++) {
        selects[i].style.visibility = visibility;
    }
}

```

The Lightbox Class

initialize()

The most important thing initialize() does is attach activate() to the link onclick, which gets the lightbox process rolling.

```

initialize: function(ctrl) {
    this.content = ctrl.href;
    Event.observe(ctrl, 'click', this.activate.bindAsEventListener(this), false);
    ctrl.onclick = function(){return false;};
},

```

activate()

activate() calls the methods responsible for setting the png overlay, manipulating the scrollbar position, and displaying the empty lightbox div.

```

activate: function(){
    if (browser == 'Internet Explorer'){
        this.scrollTop();
        this.prepareIE('100%', 'hidden');
        this.setScroll(0,0);
        this.hideSelects('hidden');
    }
    this.displayLightbox("block");
},

```

displayLightbox()

The last method called in activate() is displayLightbox(). This method sets the overlay and lightbox classes to display:block, and makes the png overlay and lightbox visible. displayLightbox() then calls loadInfo() to populate the empty lightbox div with information.

```
displayLightbox: function(display){
    $('overlay').style.display = display;
    $('lightbox').style.display = display;
    if(display != 'none') this.loadInfo();
}
```

loadInfo()

During initialize(), a member variable, content, was created in order to hold a file location. Using this file location, loadInfo() pulls information into the lightbox. After the info is loaded into the lightbox, processInfo() is called.

```
loadInfo: function() {
    var myAjax = new Ajax.Request(
        this.content,

        {method: 'post', parameters: "", onComplete: this.
processInfo.bindAsEventListener(this)}
    );
},
```

processInfo()

Responsible for actually inserting the information into the lightbox class.

```
processInfo: function(response){
    info = "<div id='lbContent'>" + response.responseText + "</div>";
    new Insertion.Before($('lbLoadMessage'), info)
    $('lightbox').className = "done";
    this.actions();
}
```

actions()

If you want to trigger an event inside of the lightbox, create a link with a class 'lbAction'. Also, set the link's rel to the function you want

called inside of the lightbox class. For example, we often want the user to have the ability to close the lightbox by clicking the cancel link. deactivate() is the method responsible for closing a lightbox and can be triggered with the following code.

```
<a href="#" class="lbAction" rel="deactivate">cancel</a>
```

actions() makes the previous link meaningful with the following code.

```
actions:function(){
    lbActions = document.getElementsByClassName('lbAction');
    for(i = 0; i < lbActions.length; i++) {
        Event.observe(lbActions[i], 'click',
this[lbActions[i].rel].bindAsEventListener(this), false);
        lbActions[i].onclick = function(){return false;};
    }
}
```

deactivate()

As we saw previously, deactivate() is called when we want to close the lightbox. deactivate() is similar to activate(), but instead of displaying the lightbox and overlay, it removes them. Take note that the information loaded into the lightbox must be removed during deactivate() since the lightbox div is only hidden and not removed.

```
deactivate:function (){
    Element.remove($('lbContent'));
    if (browser == "Internet Explorer"){
        this.setScroll(0,this.yPos);
        this.prepareIE("auto", "auto");
        this.hideSelects("visible"); }
    this.displayBlock("none");
}
```

insert()

As we explained in the intro you can easily link to another lightbox from inside a lightbox. `insert()` is the method which makes this possible and is seen below.

```
insert: function(e){
  link = Event.element(e).parentNode;
  Element.remove($('lbContent'));

  var myAjax = new Ajax.Request(
    link.href,
    {method: 'post', parameters:
      ""}, onComplete: this.processInfo.bindAsEventListener(this)}
  );
}
```

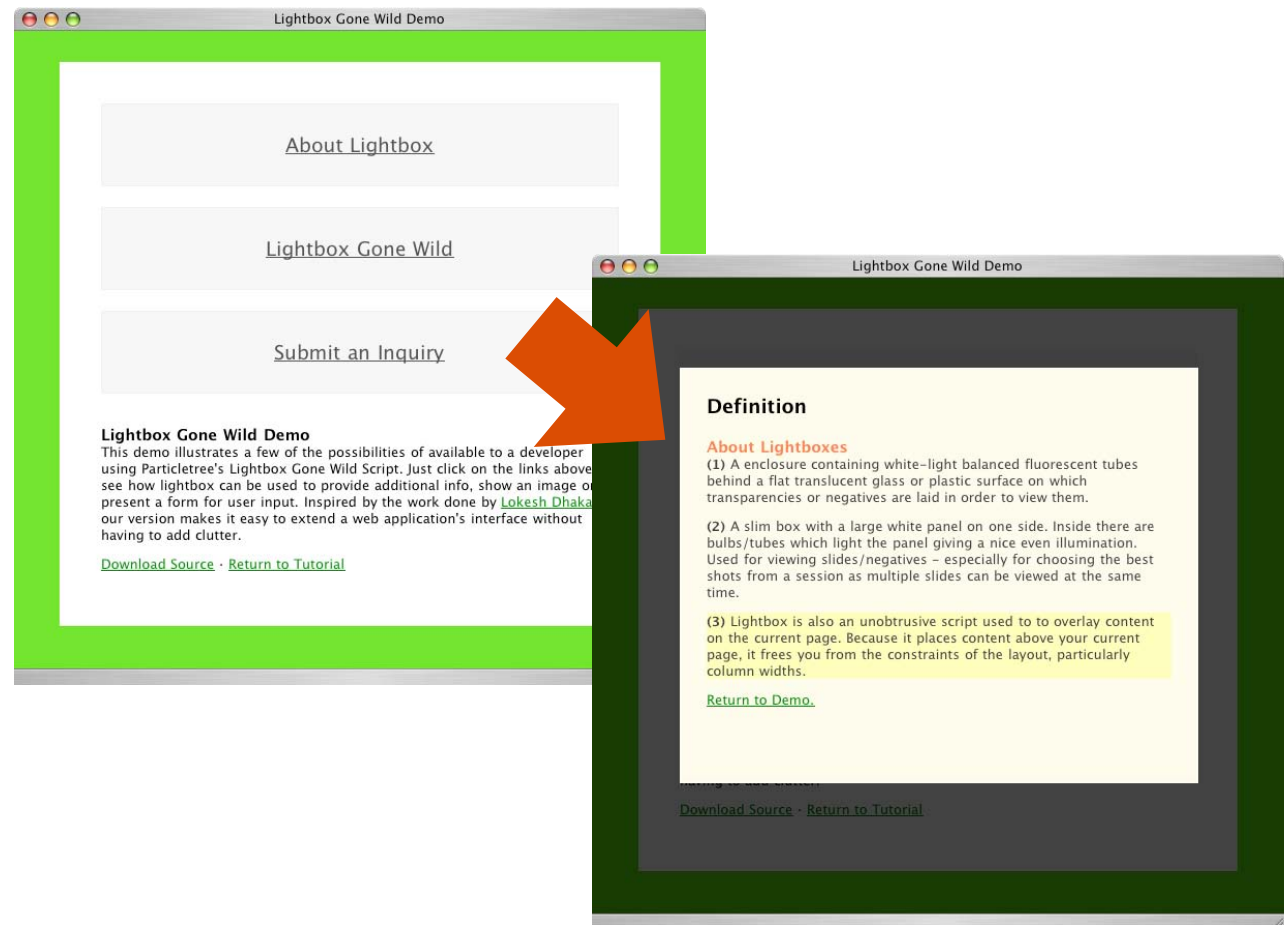
Conclusion

Well, there you have it. We believe the lightbox to be a great tool for presenting more information to a user without taking them to a new page or forcing a pop-up. As I said in the tutorial, there have been some problems with Flash 8 on Firefox and select elements in IE. If you find any other problem areas, please let us know.



Demonstration

Lightbox Gone Wild!



Dynamically load a modal window. Mimic desktop functionality by providing a window that sits on top of the page. When the window is present, the rest of the site is disabled.