

Build a simple WYSIWYG Web page editor

Skill Level: Intermediate

Gregory Michael Travis (mito@panix.com)

Software Engineer

Google

16 Sep 2008

Explore a simple Asynchronous JavaScript + XML (Ajax) system that lets your users assemble pages by adding and arranging pre-made widgets. Many sites provide this kind of functionality, but this easy-to-use system lets you do it on your own site and provides a simple library for creating new widgets.

This article describes a simple system that lets users of your Web site build their own Web pages. With it, they can place text and images on their page, arrange them to their liking, and save their work. The code in this article stands alone, without any third-party libraries. It's not necessarily what you would want to do in a real-world environment, but it covers a lot of ground for investigating implementation techniques.

Architecture

The code in this system is divided into five sections:

- **Widgets:** *Widgets* are the individual elements that make up a Web page. This article considers only two widget types: an editable text widget and an image widget. There are many other kinds of widgets you could create. However, here I am more interested in the infrastructure that supports the widgets rather than a wide variety of options.
- **Layout:** The whole point of this system is to let you create Web pages, which you do by creating, moving, and resizing text and image widgets. There's nothing really ground-breaking going on here—simply mouse event handlers, `<div>` resizing, and the like. These things are adequately covered by a myriad of articles and tutorials and are beyond the scope of

this article.

- **Persistence:** Users must be able to save their work and load it again later, so a persistence mechanism is needed. You'll use some basic serialization to turn the data into a savable form, and then you'll store it using Document Object Model (DOM) Storage. DOM Storage is defined in the HTML version 5 specification and is implemented in more recent versions of Mozilla Firefox.
- **Click-and-drag function:** There are plenty of articles and tutorials on the Web about implementing click-and-drag functions in JavaScript code, so I won't go into much detail here. Instead, I go over the basic structure of how to do it, because I think it's nicely modular.
- **Drag-and-drop function:** Like click-and-drag functions, you can implement drag-and-drop functions entirely in your JavaScript code. However, modern browsers and operating systems increasingly support native drag-and-drop functions, which can be both useful and a problem. This article covers both of these issues.

Widgets

According to Wikipedia, a widget is "a placeholder name for an object or, more specifically, a mechanical or other manufactured device." In the software world, widgets often refer to self-contained GUI elements that are freely placed on a page or combined to form a coherent whole.

This system contains two widgets: one for text and one for images. The text widget is editable, the image widget is not. This is a pretty basic set of widgets, and a far cry from the variety of widgets you get with commercial systems, but it's enough for demonstration purposes. This article is interested in what goes around the widgets.

Widgets are objects

JavaScript is an object-oriented language, so naturally, widgets are objects. The JavaScript language is, however, very flexible. It is prototype-based rather than class-based, and doesn't have a single, built-in inheritance technique. Both of the widgets are, first and foremost, GUI elements. The `Image` object is, in fact, an `HTMLImageElement`. Its constructor is shown in Listing 1.

Listing 1. The Image constructor

```
function Image( url ) {
  var self = elem( "img" );
  self.src = url;
  // ...
  return self;
}
```

This particular style of constructor is unusual in that it returns a value called `self`. In the JavaScript language, a constructor function initializes the underlying `Object` provided by the run time system, and this `Object` is returned to the caller of the constructor.

This constructor returns a value, however, which means that this value takes the place of the regular `Object`, which itself is thrown away. The custom object overrides the default `Object`.

Note: I called this value `self`, because I'm not permitted to reassign the traditional `this` value. The result of all this is that the `Image` object is really an `HTMLImageElement` created by the call to `elem()`. In a sense, this means that `Image` is a subclass of the `HTMLImageElement`, which you can put directly into the DOM tree.

The editable widget

The other kind of widget in this system is the text widget, called `EditText` because the text is editable. The `EditText` constructor also overrides the constructor return value, as Listing 2 shows.

Listing 2. The `EditText` constructor

```
function EditText( text )
{
  var self = Box();
  self.shower = elem( "div" );
  // ...
  return self;
}
```

The `self` object is created by the `Box` constructor, which also overrides the return value, as seen in Listing 3.

Listing 3. The `Box` constructor

```
function Box()
{
  var self = elem( "div" );
  sty( self, "border", "none" );
  // ...
  return self;
}
```

Thus, an `EditText` object is really a `Box` object, which in turn is really an HTML

`DivElement`. It, too, can be put directly into the DOM tree.

The `Box` object is a `<div>` that contains another GUI element. It has a pair of methods for getting and setting this contained element. The `EditText` element needs these elements, because `EditText` has two modes. It is a text area when the text is being edited and a regular `<div>` when it is just being displayed. As the `EditText` object goes from one state to another, it calls the `Box.set()` method to change what is being shown.

Persistence

Persistence on the Web has always been tricky, because it is the browser's responsibility to keep the code running in Web pages from doing anything untoward to your hard disk. But persistence is a needed function, so there have been many ways of doing it.

DOM Storage

The code in this system uses DOM Storage, a relatively new technique defined in the HTML version 5 specification. DOM Storage provides a simple set of JavaScript variables that are persistent across page loads. DOM Storage, however, can only store strings. Therefore, you must convert your saved data to strings.

The widgets in this system are simple. The `Image` widget has an image URL; the `EditText` widget has a string. A widget of either kind has a position, a width, and a height. The `gather()` function iterates through all the widgets on the page and gathers this information into a single data structure. This data structure is then turned into a string, which you can store using DOM Storage.

The stored string is actually JavaScript source code, which, when evaluated, produces the original data structure. This is nice, because rather than requiring two new functions (`serialize()` and `unserialize()`), it only requires one new one—`.toSource()`—and the regular `eval()` already included in the language. When you've evaluated the code and produced the data structure, simply iterate through it and create each widget in turn.

Click-and-drag functions

Making draggable objects in the JavaScript language is a time-honored technique that only requires a few carefully implemented mouse handlers. However, getting everything right can be tricky.

GUI programming is notoriously non-modular; it's difficult to build up complicated interactions from a collection of simpler ones. But modularity is always worth

pursuing, so I go over what I did here.

What follows is a series of convenience functions, each of which takes care of one aspect of the click-and-drag process. Not only do they save a bit of development time, but they also provide a consistent approach that works well in the presence of other handlers—even other click-and-drag handlers. Each function is relatively small, but all together, the functions serve to hide a lot of annoying details.

The CND object

A *CND*, or Click-and-Drag, is a trio of functions that forms an interface to click-and-drag code functions. The three methods are:

- `start()`
- `move(x, y)`
- `end()`

The `start` method is called after you click the mouse button, and the `end` method is called after you release the button. In between, the `move` method receives the coordinates of every mouse movement.

It's nice to bundle these three functions together, because it lets you define the entire process in one place rather than in separate event handlers that might be implemented in different parts of your code. It's also nice because it hides the details of the event objects and handler return values. It just provides you with the basics.

Installing the handlers

Another tricky aspect of implementing click-and-drag functions or any other multistep GUI operation is that you're implementing handlers that may well be taking the place of other handlers. Under the hood, the click-and-drag process starts with an `onclick` method of the target object. The drag handling, however, must be running in the document object, because you want to get all mouse-move events, not just those that might happen with the cursor over the target object.

Messing with handlers in the document object, however, is dangerous business. Such handlers receive events from the entire page, which means that other page elements might stop receiving the events they need. Also, the document object might already have handlers installed for some other purpose, and you don't want to overwrite them.

Cleaning up after yourself

This system uses a set of handler functions that can clean up after themselves. This is done with a call to `install_mouse_handlers_into_target()`, which installs

a set of mouse event handlers into any object and returns a restorer function. The restorer function, when called, puts the original handlers (if any) right back where they were. If you always add a function like this to install your handlers, you never have to worry about your handlers clobbering other parts of the event-handling system.

Automatic cleanup

For the particular case of click-and-drag functions, there's an ever easier—and safer—way to install the handlers. Each click-and-drag operation ends with the release of the mouse button, at which point you want to remove all the handlers you installed at the beginning of the click-and-drag operation.

The `4()` function takes care of all this. Like `install_mouse_handlers_into_target()`, `4()` adds a set of mouse handlers and an object to install them in. But before it does this, it alters the `onmouseup` handler so that in addition to whatever it already does, it invokes the restorer, cleaning everything up automatically. Listing 4 shows this code.

Listing 4. The `4()` function

```
var orig_up = onmouseup;
onmouseup = function( e ) {
  restorer.restore();
  return orig_up( e );
};
```

After this modification to the `onmouseup` handler, all the handlers are passed to `install_mouse_handlers_into_target()`.

The modified `onmouseup` handler is a composite handler. It first calls the restorer, which cleans up all the click-and-drag handlers, and then it calls the original `onmouseup` handler that the caller supplied. This way, the mouse handlers don't have to worry about cleaning up after themselves—they just need to take care of the click-and-drag operation itself.

Starting it all off

Now that you've simplified the process of cleaning up your handlers at the end, turn your attention to simplifying the beginning of the click-and-drag process.

The click-and-drag operation is started by an `onmousedown` handler, which notes the click and installs all the handlers. You can abstract this out with yet another convenience function, shown in Listing 5.

Listing 5. The `5()` function

```
function install_onmouse_installer onclick( target,
onmouse ) {
  target.onmousedown = function() {
    install_mouse_handlers_until_mouseup( document,
onmouse );
    return false;
  };
}
```

This code takes the same arguments as before: a target object and a set of mouse handlers. It installs an `onmousedown` handler that starts everything off by calling your previous convenience function, 4.

Now, you've automated most of the plumbing that enables the click-and-drag mouse handlers to work. What about the mouse handlers themselves? They, too, can be made simpler and more modular.

CND and mouse handlers

Earlier, I mentioned the CND object, which contains three methods: `start()`, `move()`, and `stop()`. These methods capture the essence of click-and-drag functions. However, they are not event handlers. An event handler is more general; it takes an event object that contains the event type, the event coordinates, and so on.

CNDs are simpler, because they're more specific to click-and-drag functions and don't bother with any of the information you're not interested in. If you want to use CNDs in a system that wants mouse handlers, you need a way to convert CNDs into mouse handlers. This function is `6()`, shown in Listing 6.

Listing 6. The `6()` function

```
function cnd_to_onmouse( cnd ) {
  return {
    down: function( e ) { cnd.start(); },
    move: function( e ) { cnd.drag( e.clientX, e.clientY
); },
    up: function( e ) { cnd.stop(); },
  };
}
```

Again, it's a small, simple function, but it makes things quite a bit neater. Once again, you've removed a bit more of the annoying, boilerplate plumbing that click-and-drag handlers need.

Click-and-move functions

You can use a click-and-drag operation for various purposes. In this application, you use it to perform two tasks: to move widgets and to resize widgets. Each task can be encapsulated in a general-purpose CND. Listing 7 shows the `7()` function.

Listing 7. The `7()` function

```
function move_element_cnd( elem ) {
  var yet = false;
  var dx, dy;

  return make_cnd(
    function() { yet = false; },
    function( x, y ) {
      if (!yet) {
        dx = sz( elem.style.left ) - x;
        dy = sz( elem.style.top ) - y;
        yet = true;
      }
      elem.style.setProperty( "left", x+dx,
"" );
      elem.style.setProperty( "top", y+dy,
"" );
    },
    null);
}
```

This function does a good bit of position-munging, but it all boils down to a simple idea: Keep constant the precise distance between the mouse cursor and the target object. This is, of course, perfectly general, so it can be used on any object.

Click-and-resize functions

Resizing can be done the same way as moving. Resizing doesn't drag the object itself, but it does drag a corner of the object as well as the resize widget, if any. Listing 8 shows the `8()` function.

Listing 8. The `8()` function

```
function resize_element_cnd( elem ) {
  var yet = false;
  var dx, dy;

  return make_cnd(
    function( x, y ) {
      yet = false;
    },
    function( x, y ) {
      if (!yet) {
        dx = width( elem ) - x;
        dy = height( elem ) - y;
        yet = true;
      }
      setsize( elem, x + dx, y + dy );
    },
    null);
}
```

This function looks a lot like `7()`. In fact, you could summarize the `8()` function as "the total motion of the mouse should always equal the total change in the size of the target object." Once again, this function is entirely generic and can be used on any DOM element.

The whole shebang

Using all this stuff together is just about as easy as can be, as shown in Listing 9.

Listing 9. A single line sets up all the click-and-drag handlers

```
install_cnd_installer onclick( target, move_element_cnd(
movee ) );
```

After this, you can click `target` and start dragging. The object that is dragged, however, is not `target`, but rather `movee`. These objects are often, but not always, the same.

For example, resizing might need them to be different objects. In the system described in this article, you resize an object by clicking and dragging the resize icon. This, in turn, resizes the widget. This is best implemented by two CND objects. The resize icon needs a move CND, while the widget needs a resize CND. You can combine these two CNDs using the `10()` function, shown in Listing 10.

Listing 10. The 10() function

```
function compose_cnds( a, b ) {
  return make_cnd( function() { a.start(); b.start(); },
                  function( x, y ) { a.drag( x, y );
b.drag( x, y ); },
                  function() { a.stop(); b.stop(); } );
}
```

The `10()` function takes two CNDs and returns a CND that combines them by running them in sequence. Any call to the composite CND just calls the first CND, then calls the second one. Listing 11 shows the `10()` function in action.

Listing 11. The 11() function

```
install_cnd_installer onclick( resize_icon,
compose_cnds(
  move_element_cnd( widget.resizer ),
  resize_element_cnd( widget ) ) );
```

This section contains a lot of small functions, each of which takes care of a bit of the annoying boilerplate that usually comes with implementing drag-and-drop functions. There are several advantages to all of this.

Convenience is quite underrated in programming. If you can set up a click-and-drag feature with a single line of code, you're much more likely to try it when prototyping a GUI. Designing GUIs is subtle, and it's important to be able to try many things and to refine the design as you go.

This kind of approach is also safer. The convenience functions aren't just convenient, they're clean and safe. Each function takes care not to overwrite an existing handler with a new one and to make sure that things are cleaned up when a click-and-drag process is done. When you know that this stuff is working, you're much less likely to have subtle bugs caused by the interaction between different handlers that serve different purposes.

Drag-and-drop functions

Drag-and-drop functions are a much-desired feature of individual applications and of entire operating systems. It's difficult to do, though, because it involves sharing data between unrelated applications, and unrelated applications are notorious for having unrelated data representations.

Nevertheless, the function is growing all the time. Recent versions of browsers and operating systems let you drag text selections and images onto various kinds of targets, such as text fields or the desktop. This is very convenient for users, because it corresponds well with the essential design of that ubiquitous input device, the mouse.

Two aspects of native drag-and-drop functions are relevant to the Build-Your-Page system: using it and not using it.

Using drag-and-drop functions

You can drag both images and text into a regular form text field. When you drag an image, the URL is inserted; when you drag text, the text itself is inserted.

In the Build-Your-Page system, there are two text boxes at the top of the page: one for images and one for text. After dragging something into one of these text boxes, you click the button next to it, and that something is turned into a widget.

There really isn't much special plumbing for this feature. The browser or operating system takes care of inserting the URL or text into the text field, and the button click invokes an `onclick` handler that calls `place_new_widget()`.

Not using drag-and-drop functions

There is a hitch, however. The native drag-and-drop feature can interfere with the rest of the interface. When I started writing the code in this article, I found that the resize button didn't always work. The browser sometimes seemed to think that what I really wanted to do was drag the resize icon itself somewhere. What's more, it wasn't clear under which conditions my click would be interpreted this way.

After some debugging, it turned out that the problem was that I wasn't returning a value from my handlers. Handlers are supposed to return values that indicate whether they handled the event or ignored it. This, in turn, determines whether the event should be propagated to the next handler.

Adding a `return false` to the end of the handler created in 5 did the trick. This handler is the trigger for the whole click-and-drag sequence, and by returning `False`, it told the browser that the event should not be passed along to anyone else—specifically, in this case, to the native drag-and-drop handler.

Summary

GUIs are difficult to develop. They involve many subtle issues surrounding the psychology of the humans who use them. And because GUIs form the boundary between two different autonomous agents—the user and the computer—they suffer from the special status of having two masters. This generally results in code that is intricate and non-modular and that can often be quite fragile during the long development process.

But all hope is not lost. By carefully paring the general-purpose plumbing from the application-specific functions, it is possible to create libraries of GUI actions that can be combined into larger systems of relatively independent interactions.

This article has covered a lot of ground. It has touched on several different elements, all of which are relevant to the Build-Your-Page system. Build-Your-Page is a toy system, but it does involve most of the elements of a substantial GUI application. It hints at a more sophisticated system, one that could fulfill the dream of a read-write Web.

Downloads

Description	Name	Size	Download method
Source code with example ¹	build-your-page-src.zip	4KB	HTTP

[Information about download methods](#)

Note

1. This file contains the source code for the Build-Your-Page system.

Resources

Learn

- The [Wikipedia entry for widget](#) shows how ubiquitous this concept is in computing.
- The [HTML version 5 specification](#) contains the official specification of the DOM Storage system.
- The [DOM Storage page](#) at mozilla.org explains in detail how to use this persistence mechanism.
- The [developerWorks Web development zone](#) is packed with tools and information for Web 2.0 development.
- [IBM technical events and webcasts](#): Stay current with developerWorks' Technical events and webcasts.
- The [developerWorks Ajax resource center](#) contains a growing library of Ajax content as well as useful resources to get you started developing Ajax applications today.
- Browse the [technology bookstore](#) for books on these and other technical topics.

Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

About the author

Gregory Michael Travis

Greg Travis is a software engineer at Google. He has been a Web programmer, an independent contractor, and a game developer. You can reach Greg at mito@panix.com.