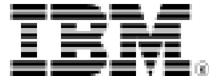


developerWorks®

Building Google gadgets, Part 2: Working with the user interface

Learn techniques for creating a compelling user experience.

Skill Level: Intermediate

John Muchow (Technical.tutorials@gmail.com)
Writer
#####

22 May 2007

Part 2 of the Building Google gadgets series introduces the advanced features of gadgets, including creating a tabbed user interface, drag and drop, and MiniMessages, and gets you started creating your own.

Section 1. Before you start

The following offers some background information to help you make the most of this tutorial.

About this series

This series provides a solid foundation for you to begin writing your own Google gadgets.

About this tutorial

This tutorial is the second in a two-part series on creating Google gadgets. Part 1 built the foundation for gadgets, teaching you about data types and the specifics of the gadget XML file. This tutorial addresses the more advanced gadget features,

© Copyright IBM Corporation 2007. All rights reserved.

including creating tabbed interfaces, adding drag-and-drop support, and displaying messages.

Objectives

In this tutorial, you'll be introduced to the programming APIs for various Google gadgets features, and you'll have a chance to view the code for several complete gadgets.

Prerequisites

This tutorial is written for developers who are familiar with XML, working application programming interfaces (APIs), and coding in JavaScript. To get the most from this tutorial, you should have a general familiarity with these concepts.

System requirements

To build and run the examples in this tutorial, you need nothing more than a text editor, an Internet connection, and a passion for coding and debugging.

Section 2. Housekeeping

Before going any further, I recommend that you add a specific gadget to your personalized home page. This gadget will be of great help as you write, test, and debug your own gadgets.

Introducing the developer gadget

Google created a *developer gadget* that allows you to see all the gadgets currently on your personalized home page. It also allows you to toggle whether or not a gadget is cached. Although caching your gadget definitions is fine once your gadget is functionally complete and debugged, during the development phase you'll almost always want your most recently uploaded gadget to be displayed. The developer gadget is nothing more than another gadget with some specific features of interest only to the developer crowd.

© Copyright IBM Corporation 2007. All rights reserved.

Add the developer gadget

From your personalized Google home page, click the **Add stuff** link in the upper-right corner. Select **Add by URL** and enter the path as shown in Figure 1.

Figure 1. Adding the gadget

Set preferences

After you've added the developer gadget, you'll notice a listing of all the current gadgets on your home page (see Figure 2).

Figure 2. Developer gadget preferences

During the development phase, when you're busy coding, loading, and testing ("rinse and repeat"), you'll want to uncheck the **Cached** preference for your gadget.

© Copyright IBM Corporation 2007. All rights reserved.

This will force the home page to show the most recently uploaded version of your gadget.

One final note: With the developer gadget on your home page, you can now add gadgets without having to navigate away from this same page. That is, you no longer need to click the **Add stuff** link and select **Add by URL**. Simply type in the URL (once you've uploaded your gadget to a server), and away you go.

Section 3. Tabbed UI: Includes

There are a few ground rules to cover before you can use tabs within your gadget. This section introduces the required code, including a short example of each.

Reference the tabs library

In the `ModulePrefs` section, you must include a reference to the tabs library through a call to `<Require feature="tabs"/>` (see Listing 1).

Listing 1. Tabbed UI declarations

```
<?xml version="1.0" encoding="UTF-8" ?>
<Module>
  <ModulePrefs title="Using Tabs" height="200">
    <Require feature="tabs" />
  </ModulePrefs>
...
```

Import the style sheet

In addition to loading the tabs library, you must also import the style sheet necessary for working with tabs (see Listing 2).

Listing 2. Import the tabs style sheet

```
<?xml version="1.0" encoding="UTF-8" ?>
<Module>
  <ModulePrefs title="Using Tabs" height="200">
    <Require feature="tabs" />
  </ModulePrefs>
...
  <Content type="html">
  <![CDATA[
    <style type="text/css">
```


© Copyright IBM Corporation 2007. All rights reserved.

```
        @import url(http://www.google.com/ig/tablib.css);
     </style>
...
```

Section 4. Tabbed UI: Coding

Tab contents are shown inside a `div` section (see the `div` sidebar for more information). You can create the `div` sections yourself or have them generated for you. The easiest means to get a handle on this is to walk through an example that highlights the basics of defining and displaying tabs.

Tab example

Figure 3 shows how your gadget will look once you add it to your personalized home page.

Figure 3. Tab 1

Listing 3 shows the complete gadget XML file for this tab example.

Listing 3. Tab example XML file

```
 1   <?xml version="1.0" encoding="UTF-8" ?>
 2   <Module>
 3     <ModulePrefs title="Tab example" height="100">
 4       <Require feature="tabs" />
 5     </ModulePrefs>
 6     <Content type="html">
 7     <![CDATA[
 8       <style type="text/css">
 9          @import url(http://www.google.com/ig/tablib.css);
10       </style>
11
12       <script type="text/javascript">
13       function doTabs()
14       {
15         var tabs = new _IG_Tabs(__MODULE_ID__);
16         var tabOne;
```


© Copyright IBM Corporation 2007. All rights reserved.

```
17
18        // Save tab id, add content inside the div
19        var tabOne = tabs.addTab("Tab 1");
20        _gel(tabOne).innerHTML = "Homebrew is your friend.";
21
22        // Create tab, add (static) content below inside the div declaration
23        tabs.addTab("Tab 2", "tabTwo");
24
25        // Create tab, as with tabTwo, add static content in the div defined below.
26        // We also add dynamic content by using a callback function
27        tabs.addTab("Time", "tabThree", dynamicContent);
28      }
29
30      // Callback function to add dynamic content, displaying the current time
31      function dynamicContent(id)
32      {
33        var date = new Date();
34        var hour = date.getHours();
35        var min = date.getMinutes();
36        var indicator;
37        var displaytime;
38
39        // Check for am or pm
40        if (hour > 12)
41        {
42          hour -= 12;
43          indicator = 'pm';
44        }
45        else
46          indicator = 'am';
47
48        // Add leading 0 to minutes, if necessary (e.g. 09 minutes)
49        if (min < 10)
50          min = '0' + min;
51
52        // Create string that will be displayed
53        displaytime = hour + ':' + min + ' ' + indicator;
54
55        _gel(id).innerHTML = displaytime;
56      }
57
58      // On load, initialize/display tabs
59      _IG_RegisterOnloadHandler(doTabs);
60
61      </script>
62
63     <div id="tabTwo" style="display:none">Inside Tab Two...</div>
64     <div id="tabThree" style="display:none">This text will never be seen...</div>
65    ]]>
66    </Content>
67  </Module>
```

div

The `div` tag defines a division (or section) in an HTML document. `div` allows you to define another container for additional HTML content. For instance, you can use the `div` tag, along with various attributes such as a font and background color, to draw attention to a specific area within a Web page. You can learn more by reviewing the HTML 4 Specification (see Resources).

Required code

© Copyright IBM Corporation 2007. All rights reserved.

As you learned in the previous section, you must include two declarations in any gadget XML file that uses tabs. First, you must include the tabs library in the `ModulePrefs` section (see line 4 in Listing 3). Second, you must reference the tabs style sheet (see lines 8 through 10 of Listing 3).

There is one final piece of business to attend to before declaring any tabs. Line 15 shows a call to the tab constructor, which creates an instance of a `tab` object. The variable name `tabs` is now your means for adding and otherwise working with tab content.

Static tab #1

Listing 3 shows the first tab declaration on lines 19 and 20. `tabOne` is the variable name assigned to the tab, and the string "Tab 1" is displayed on the tab (see Figure 3).

Line 20 adds static content to the `div` section for this tab. Beyond these lines (assuming you don't want to change the tab contents), no further code is required.

Static tab #2

Line 23 shows the declaration for the second tab. The variable name `tabTwo` is your means to reference this tab in your gadget. The string "Tab 2" is shown on the tab (see Figure 4).

Figure 4. Tab 2

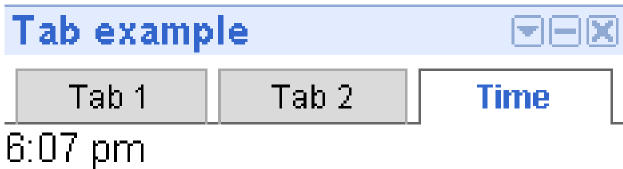
At this point, you have a tab declared, yet you have no corresponding `div` section in which to display the tab contents. This is taken care of on line 63, which defines a `div` section. Notice the reference to your tab using the variable name `tabTwo`. Also, line 63 adds static content to the tab with the string "Inside Tab Two...", which is shown when the user selects the tab.

Dynamic tab

© Copyright IBM Corporation 2007. All rights reserved.

Line 27 declares the third tab. The variable name to reference it is `tabThree`. The third parameter is a reference to a callback function, `dynamicContent()`, which is called when the tab is created or the user selects this tab. The string "Time" is displayed on the tab, as you can see in Figure 5.

Figure 5. Tab 3



Looking a little further, inside the function `dynamicContent()` you'll see code similar to the final example you created in Part 1 of this tutorial series -- namely, JavaScript code to display the current time.

The final line of code you need to concern yourself with is line 64. As with the previous tab, you need to associate the tab with a `div` section. Notice the text string "This text will never be seen..." in the `div` declaration. To understand why you'll never see this message, you need to look back up in the code to line 55, which is the last line of the callback for the third tab. This line sets the contents of the tab to the current time. Each time the user selects this tab, the callback function is invoked and the current time is displayed. Therefore, the message defined in the `div` section is overwritten with the current time.

In the "Dynamic and remote content" section, you'll revisit working with tabs and dynamic content.

Section 5. Tabbed UI: Library

Tabbed-based user interface components are now commonplace on Web pages. You can add the same look and feel to gadgets using the Tabbed UI Library.

Tab functions

The tabs library consists of seven functions for working with tabs. Table 1 shows each function and includes a short description of each.

© Copyright IBM Corporation 2007. All rights reserved.

Table 1. Tabs library functions

Function	Description
_IG_Tabs(module_id, opt_selected_tab)	Constructor
addDynamicTab(tabName, callback)	Adds a tab. A callback function will insert this tab's content dynamically.
addTab(tabName, opt_domId, opt_callback)	Adds a tab. Depending on the parameters used, this tab may have static or dynamic content.
currentTab()	Returns the index of the active tab.
moveTab(tabIndex1, tabIndex2)	Allows you to interchange the position of two tabs.
setSelectedTab(tabIndex)	Sets the active tab programmatically and invokes any associated callback.

You've already seen two of these functions in action. Line 15 in Listing 3 called the tabs library constructor `_IG_Tabs()`. Lines 19, 23, and 27 all invoked the function `addTab()` to create new tabs within the gadget. As you continue with this tutorial, you'll call additional functions from Table 1.

For more information about the specific parameters to each function, please see the Resources section for a link to the Google Gadgets API Developer Guide and the tabs library reference.

Section 6. Dynamic and remote content

Previously, you added dynamic content using the `addTab()` function. This section introduces an additional function for working with dynamic content and, at the same time, teaches you more about functions for working with remote content.

Online content functions

There are three functions for working with remote content. Table 2 shows each, along with a short description.

Table 2. Content functions

Function	Description
_IG_FetchContent(url, func)	Gets content through a URL.
_IG_FetchXmlContent(url, func)	Gets content from an XML document.

© Copyright IBM Corporation 2007. All rights reserved.

_IG_FetchFeedAsJSON(url, func, num_entries, get_summaries)	Gets content from RSS or an Atom feed.

Given the limited space in this tutorial, the bad news is that each function can't be covered in detail. The good news is that you'll learn how to build an interesting gadget in this section using `_IG_FetchContent(url, func)`. At the same time, you'll discover an additional function for working with dynamic content inside a tab.

Remote content example

The gadget you'll write in this section retrieves its content from the CNNMoney.com Web site. Specifically, you'll access a Web page that lists information for the daily U.S. markets. The gadget will contain two tabs: one to show the current market change for the Dow Jones, and one to show the change for the NASDAQ.

Begin by looking at the screen shots of the gadget you'll put together in this section. Figure 6 shows the Web site and the related market data you're after -- that is, the Dow Jones and NASDAQ values in the Change column.

Figure 6. Market Web site

The gadget will feature nothing more than two tabs, one each for the Dow and NASDAQ changes (see Figure 7 and Figure 8).

© Copyright IBM Corporation 2007. All rights reserved.

Figure 7. Dow tab

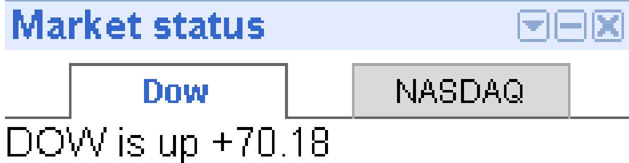
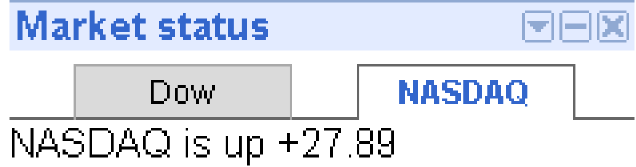


Figure 8. NASDAQ tab



Market status tab code

Listing 4 shows the code for this gadget. As before, I'll step through each aspect of the gadget after the listing, further clarifying how the gadget accesses, formats, and displays its content.

Listing 4. Market tab

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <Module>
3   <ModulePrefs title="Market status" height="100">
4     <Require feature="tabs" />
5   </ModulePrefs>
6   <Content type="html">
7   <![CDATA[
8     <style type="text/css">
9        @import url(http://www.google.com/ig/tablib.css);
10      </style>
11
12      <script type="text/javascript">
13
14      function doTabs()
15      {
16        // Tab constructor, tab labeled "NASDAQ" will be the active tab
17        var tabs = new _IG_Tabs(__MODULE_ID__, "NASDAQ");
18
19        tabs.addDynamicTab("Dow", getDow);        // Add tab
20        tabs.addDynamicTab("NASDAQ", getNasdaq);  // Add tab
21      }
22
23      function getDow(tabID)
24      {
25        // Pass in the string "dow"
26        _IG_FetchContent('http://money.cnn.com/data/us_markets/',
                           _IG_Callback(marketQuery, tabID, "dow"));
27      }
28
```


© Copyright IBM Corporation 2007. All rights reserved.

```
29     function getNasdaq(tabID)
30     {
31       // Pass in the string "nasdaq"
32       _IG_FetchContent('http://money.cnn.com/data/us_markets/',
                           _IG_Callback(marketQuery, tabID, "nasdaq"));
33     }
34
35     function marketQuery(response, tabID, marketStr)
36     {
37       var start;
38       var buffer;
39       var plus;
40       var minus;
41
42       // If no data was found, we're done
43       if (response == null)
44         return;
45
46       // Partial string we retrieve and search from the website...
47       // "...(dowURL,'');">Dow</a></td><td class=change><
                        span class=textpositive>+102.30</span>..."
48
49       // Look for the dow or nasdaq string (i.e. "dowURL" or "nasdaqURL")
50       start = response.search(marketStr + "URL");
51
52       // If not found, we're done
53       if (start == -1)
54         return;
55
56       // Extract 80 characters starting from where we found the first string.
57       // We do this to limit what we need to search
58       buffer = response.slice(start, start + 80);
59
60       // Search 'buffer' for '+', indicating a positive market change
61       plus = buffer.search("\\+");
62
63       // If '+' is not found...
64       if (plus == -1)
65       {
66         // Search 'buffer' for '-'
67         minus = buffer.search("\\-");
68
69         // If not found, we're done
70         if (minus == -1)
71           return;
72
73         // '-' found, extract the value and format string
74         _gel(tabID).innerHTML = marketStr.toUpperCase() + " is down " +
                      buffer.substr(minus, buffer.indexOf("<", minus) - minus);
75
76       }
77       else
78       {
79         // '+' found, extract the value and format string
80         _gel(tabID).innerHTML = marketStr.toUpperCase() + " is up " +
                      buffer.substr(plus, buffer.indexOf("<", plus) - plus);
81       }
82     }
83
84     // On load, initialize/display tabs
85     _IG_RegisterOnloadHandler(doTabs);
86
87     </script>
88   ]]>
89   </Content>
90 </Module>
```


© Copyright IBM Corporation 2007. All rights reserved.

Creating tabs

The function `doTabs()` is where it all begins. Within this function, you call the tab constructor and add two dynamic tabs. The function call `addDynamicTab()` is an additional means of creating tabs (beyond what was introduced earlier). Using this function, you pass in as the first parameter the name you want displayed on the tab, as well as the callback function that will be invoked.

Callback functions

Lines 23 to 33 are the callback functions for the tabs. Each time the user selects a tab, the corresponding callback is invoked.

For each of these functions, you pass in as the first parameter the URL you want to visit. The second parameter is also a callback function. However, using the function `_IG_Callback`, you can pass additional information to the callback. The first parameter is the function to call, the second parameter is the unique identifier for the tab, and the third parameter is a string that represents which market value you're searching for. I'll talk more about the third parameter momentarily.

Search results

The function `marketQuery()` is where the bulk of the action takes place. The first check, line 43, looks to see if the query returned any results. If there is data, you keep moving on; otherwise, exit the function.

Line 47 shows an example of the type of data you expect from the Web site. I came about this string by looking at the View > Page Source option from within my Web browser (View > Page Source in Mozilla Firefox or View > Source in Internet Explorer).

You can find the market quotes you're after by searching for either "dowURL" or "nasdaqURL". You can see how you build this search string on line 50. If the requested string is found within the result set, line 50 extracts an additional 80 characters and stores it in the variable named `buffer`.

Within the string of 80 characters, you now search for a "+" or "-" sign, which is the start of the data you're after. For example, looking back at line 47, you'll see the string "+102.30", which represents an increase in the Dow of 102.30. Likewise, you could also run into a string such as "-10.10", which would indicate a market decrease.

© Copyright IBM Corporation 2007. All rights reserved.

Format results

The code in lines 73-74 and 79-80 is used for formatting the output, based on whether you find a number that starts with a "-" or "+". You capitalize the incoming string (for example, "dow" becomes "DOW"), add the appropriate message (" is up " or "is down"), and extract the number you're after.

The last step is to assign the formatted string to the current tab, which in turn results in the market value being displayed. For more information about working with remote content, see the Resources section for a link to additional information.

Section 7. Drag and drop

Beyond tabs, another powerful feature available when creating gadgets is drag and drop. With drag and drop, a user can move HTML elements around within a gadget using a mouse. Figure 9 and Figure 10 show examples from the Google Gadgets API Developer Guide on using drag and drop. Although hard to represent in screen shots, the idea behind the gadget is that you can drag the names of colors (on the left) onto the image (on the right) and the image will change accordingly.

Figure 9. The original image

© Copyright IBM Corporation 2007. All rights reserved.

Figure 10. Dropping green on the image

© Copyright IBM Corporation 2007. All rights reserved.

Terminology

Before going any further, you need to learn the terminology used by gadgets when working with the drag-and-drop library (see Table 3).

Table 3. Terminology

Term	Description
Source	An object a user can click and drag
Target	An object that can have other objects dragged onto it
Surrogate	An object used to visually follow the mouse as an object is dragged

A few notes: An object can be both a source and target. Also, the primary reason for using a surrogate is to provide visual feedback to users to let them know that they've clicked on an object and that the object is being moved to follow the mouse.

© Copyright IBM Corporation 2007. All rights reserved.

Drag-and-drop functions

You can work with drag and drop using five key functions. Table 4 shows each, along with a short description. These functions are callbacks, where each is invoked when the corresponding action occurs.

Table 4. Drag-and-drop functions

Function	Description
onDragClick = function(source) {}	User clicked but did not move the mouse.
onDragEnd = function(source, target) {}	User let up on the mouse button (stopped dragging).
onDragStart = function(newSource) {}	User clicked on the mouse and started to drag an object.
onDragTargetHit = function(newTarget, lastTarget) {}	User has dragged an object onto a target.
onDragTargetLost = function(lastTarget) {}	User is no longer on a target.

Section 8. Drag-and-drop coding

This section walks through the drag-and-drop code example included in the Google Gadgets API Developer Guide. For reference, I've included the screen shots from the previous section that show how this gadget works. Essentially, you can drag a color preference onto an image, and the image will be updated to reflect your color choice (see Figure 11 and Figure 12).

Figure 11. The original image

© Copyright IBM Corporation 2007. All rights reserved.

Figure 12. Dropping green on image

© Copyright IBM Corporation 2007. All rights reserved.

Drag-and-drop code

Listing 5 shows the code for this basic drag-and-drop gadget.

Listing 5. Drag-and-drop code

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <Module>
3   <ModulePrefs title="Special Effects" scrolling="true" height="320">
4     <Require feature="drag" />
5   </ModulePrefs>
6   <Content type="html">
7   <![CDATA[
8     <table border=0>
9       <tr>
10          <td>
11            <p id="id1" style="font-weight:bold;">original</p>
12            <p id="id2" style="color:brown; font-weight:bold;">sepia</p>
13            <p id="id3" style="color:red; font-weight:bold;">red</p>
14            <p id="id4" style="color:green; font-weight:bold;">green</p>
15          </td>
16          <td id="id5" style="padding-left:60;">
17            <img src="http://doc.examples.googlepages.com/Rowan-small.gif" alt="Rowan"/>
```


© Copyright IBM Corporation 2007. All rights reserved.

```
18        </td>
19      </tr>
20    </table>
21
22    <script type="text/javascript">
23    // Preload the images
24    var images = new Object();
25    images["id1"] = new Image();
26    images["id1"].src = "http://doc.examples.googlepages.com/Rowan-small.gif";
27    images["id2"] = new Image();
28    images["id2"].src = "http://doc.examples.googlepages.com/Rowan-small-sepia.gif";
29    images["id3"] = new Image();
30    images["id3"].src = "http://doc.examples.googlepages.com/Rowan-small-red.gif";
31    images["id4"] = new Image();
32    images["id4"].src = "http://doc.examples.googlepages.com/Rowan-small-green.gif";
33
34    // drag constructor
35    var drag_obj = new _IG_Drag();
36
37    // Add sources and target. In this example, there is one target: the photo.
38    drag_obj.addSource("original", _gel("id1"),
                       "<p style=color:orange;>Restore original...</p>");
39    drag_obj.addSource("sepia", _gel("id2"),
                       "<p style=color:orange;>Change to sepia...</p>");
40    drag_obj.addSource("red", _gel("id3"),
                       "<p style=color:orange;>Change to red...</p>");
41    drag_obj.addSource("green", _gel("id4"),
                       "<p style=color:orange;>Change to green...</p>");
42    drag_obj.addTarget("image", _gel("id5"));
43
44    // When user drags source onto target and releases it,
45    // display appropriate photo.
46    drag_obj.onDragEnd = function(source, target) {
47      if (target == null) return;
48      target.innerHTML = "<img src='" + images[source.id].src + "'/>";
49    }
50    </script>
51  ]]>
52  </Content>
53 </Module>
```

Includes

As when working with tabs, you need to include the appropriate library to enable the use of drag and drop within your gadgets. Line 4 shows the syntax for including the drag-and-drop library.

HTML table

The purpose of the table defined between lines 8 and 20 is twofold. First, the table represents the visual layout of the gadget. That is, it represents the text labels and their associated attributes (font weight and color) as well as the source and placement of the image.

The second purpose of the table is to create identifiers for each of the source objects. Notice the identifier (for example, `id="id1"`) assigned to each of the text labels. You'll refer to these values throughout the gadget.

© Copyright IBM Corporation 2007. All rights reserved.

Image array

Your next task is to build an array that will contain each of the four images' colors (original, sepia, red, and green). Lines 24-32 show the code for defining the image array.

The identifiers you declared in the table (for example, `id="id1"`) are used as the index values in the array. With this approach, you're "mapping" a text label (for example, `"red"`) to an image (for example, Rowan-small-red.gif).

Source objects

You instantiate a new drag object in line 35 and follow this by adding the four source objects (the text labels) and the target object (the image on the right).

Drag-and-drop function

With the source and target objects defined, you're ready to go. Line 46 defines a callback function that is invoked when the user drops a source object onto the target object. Line 48 updates the HTML (display) with the appropriate image, based on the source object that was dropped.

That's it. Now mind you, this gadget won't change the world. However, it offers a fair amount of functionality in just over 50 lines of code. It also provides a foundation you can use to create more complex gadgets with additional drag-and-drop functionality.

There's a great deal more drag-and-drop functionality to explore. However, due to this tutorial's limited space, you'll be on your own to explore the more advanced features. See the Resources section for a link to the Google Gadgets API Developer Guide and related drag-and-drop reference material.

Section 9. MiniMessages

MiniMessages offer a broad array of options for communicating with your gadget user. For instance, you can use messages to show status information (for example, "Please wait..."), error messages (for example, "You must select an option..."), or a prompt for an action (for example, "Click here to win...").

© Copyright IBM Corporation 2007. All rights reserved.

Message types

The three types of messages are: dismissible, static, and timer. Table 5 describes each of these types.

Table 5. MiniMessage types

Message type	Description
Dismissible	A message that the user can dismiss or close
Static	A message that can only be dismissed programmatically
Timer	A message that is dismissed after a specified number of seconds

Although there are just three message types, as you'll see, they offer a great deal of flexibility when working with messages. You can use the MiniMessage library to include most any type of message and functionality in your gadget.

Includes

As with both tabs and drag-and-drop functionality, you need to include a reference to the appropriate library in order to use messages. Listing 6 shows the required library declaration, `<Require feature="minimessage"/>`.

Listing 6. MiniMessage library

```
<?xml version="1.0" encoding="UTF-8" ?>
 <Module>
   <ModulePrefs title="Message Example">
     <Require feature="minimessage" />
   </ModulePrefs>
...
```

MiniMessage functions

Five message functions are available to create and dismiss messages. Table 6 shows each, along with a short description.

Table 6. MiniMessage functions

Function	Description
_IG_MiniMessage(moduleId, opt_container)	Constructor
createDismissibleMessage(msg,	Creates a message the user can dismiss

© Copyright IBM Corporation 2007. All rights reserved.

opt_callback)	
createStaticMessage(msg)	Creates a message that can only be dismissed programmatically
createTimerMessage(msg, seconds, opt_callback)	Creates a message that is dismissed after a specified number of seconds
dismissMessage(msg)	Used to dismiss a static message

MiniMessages code

The best way to get a feel for the flexibility and options when working with messages is to see a few examples. Listing 7 creates five different types of messages. Following the code listing are the corresponding screen shots, which are in turn followed by line-by-line descriptions of the code.

Listing 7. MiniMessage examples

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <Module>
3   <ModulePrefs title="Message examples" height="100">
4     <Require feature="minimessage" />
5   </ModulePrefs>
6   <Content type="html">
7   <![CDATA[
8
9     <script type="text/javascript">
10
11      // Constructor
12      var msg = new _IG_MiniMessage(__MODULE_ID__);
13
14      // User dismissible message
15      msg.createDismissibleMessage("You can dismiss this by clicking the X");
16
17      // User dismissible message, with custom colors
18      var msg1 = msg.createDismissibleMessage("Custom message colors.");
19      msg1.style.backgroundColor = "white";
20      msg1.style.color = "blue";
21
22      // Message that creates a callback when dismissed
23      msg.createDismissibleMessage("Generates a callback...try it.", msgAck);
24
25      // A timed message, shown for 20 seconds
26      msg.createTimerMessage("Timed message...", 20);
27
28      // Static message that is programmatically dismissed
29      var msg2 = msg.createStaticMessage("Message dismissed programmatically.");
30
31      // Call back function
32      function msgAck()
33      {
34        alert("Message acknowledged. This message and the
                static message will now be dismissed!");
35        msg.dismissMessage(msg2);
36      }
37
38      </script>
39    ]]>
40    </Content>
```


© Copyright IBM Corporation 2007. All rights reserved.

```
41 </Module>
```

Screenshots

Figure 13 shows all the messages on the gadget. Notice there is one timer message with the text "Timed message...".

Figure 13. All messages

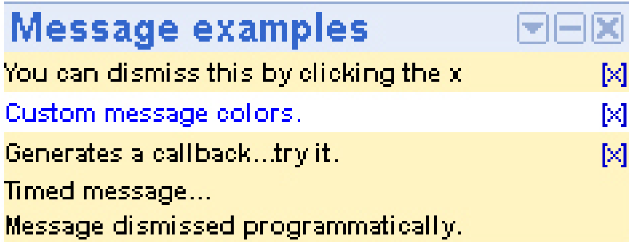
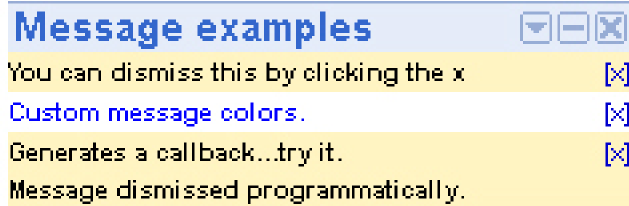


Figure 14 shows nothing more than the timed message no longer visible on the gadget. Remember, no user acknowledgement is required in this case. Timed messages are handy for such things as brief, temporary reminders (for example, "Don't forget the TPS reports!").

Figure 14. Timer message dismissed



To demonstrate both a callback and programmatically dismissing a message, you click on the "X" to dismiss the message titled "Generates a callback...try it." When you dismiss the message, a callback is generated, and the alert shown in Figure 15 is displayed. Within the callback, you can also programmatically dismiss the last (static) message on the display. For the end result, look ahead to Figure 16. You'll see the code for accomplishing all these steps in the next section.

Figure 15. Callback alert

© Copyright IBM Corporation 2007. All rights reserved.

Upon clicking **OK** in the alert message, the two messages on the bottom of the gadget are dismissed. Figure 16 shows the two user-dismissible messages remaining.

Figure 16. Static message removed

Initialization

Now take a moment to review the code, starting at line 12. As when working with tabs and the drag-and-drop libraries, your first step is to instantiate a new object. With the variable `msg` now referencing a `MiniMessage` object, you can proceed to create various types of messages.

Creating messages

You begin by creating a user-dismissible message on line 15. The only parameter is the text to be displayed. You follow this message with another user-dismissible message. However, this time you customize the text foreground and background colors (see lines 18-20).

Line 23 introduces a user-dismissible message with a callback. Once the user opts to dismiss the message, the callback function `msgAck()` is invoked -- more on this function in a minute. Line 26 is a timer message; the second parameter specifies that the text will be displayed for 20 seconds.

The last message on line 29 is a static message, which by definition, is a message that can only be dismissed programmatically. Continue on to see how this message is handled.

© Copyright IBM Corporation 2007. All rights reserved.

Callback

Two things are left to address. First is the callback function referenced on line 23. Second, you need to add code to programmatically dismiss the static message declared on line 29. The function `msgAck()` handles both cases for you. `msgAck()` is called when the user dismisses the message defined on line 23. Inside this same function, you dismiss the static message through a call to `msg.dismissMessage(msg2)`. At this point, there are just two messages remaining on the display, as shown in Figure 16.

If you'd like to learn more about working with messages, including code samples of additional message options, check out the Google Gadgets API Developer Guide. Please refer to the Resources section for a link to MiniMessage reference material.

Section 10. Tips and tricks

This section offers a few tried-and-true tips and tricks to ease the development process when creating your own gadgets.

Debugging with alert

When you're in the throes of writing a gadget, you'll have times when you'll need a few ideas for tracking down bugs. One approach for tracing program flow is to insert several calls to the JavaScript `alert()` function. I used this approach frequently when developing the gadgets within the tutorial. Listing 8 shows a partial block of code that should give you the idea of how to verify if the code inside one method is working. Adding the output of a variable to the `alert()` message is also a handy way to check your logic.

Listing 8. Debugging with the alert() type

```
<?xml version="1.0" encoding="UTF-8" ?>
<Module>
  ...

  <script type="text/javascript">
  function doTabs()
  {
    var buffer = "0";

    alert("Inside doTabs()");

    ... do stuff here
```


© Copyright IBM Corporation 2007. All rights reserved.

```
    alert("Leaving doTabs() - Value of buffer is: " + buffer);
  }

  ...
</Module>
```

Debug with div

The Google Gadgets API Developer Guide has an excellent section on writing error messages into a `div` section within a gadget. This is a unique approach to debugging, as the messages will appear inside the gadget (instead of a popup window, as in the previous section).

Listing 9 shows a slightly modified version of the example in the Google Gadgets API Developer Guide. Pay attention to line 12, which defines the debug `div` section; lines 17 and 18, which declare the debug variables; and lines 33-40, where you can find the debug message function.

Listing 9. Debugging with div

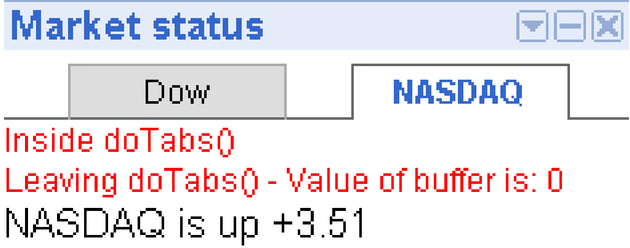
```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <Module>
3   <ModulePrefs title="Market status" height="100">
4     <Require feature="tabs" />
5   </ModulePrefs>
6   <Content type="html">
7   <![CDATA[
8     <style type="text/css">
9       @import url(http://www.google.com/ig/tablib.css);
10     </style>
11
12     <div id="debug_div" style="font-size:9pt; color: red;"></div>
13
14     <script type="text/javascript">
15
16     // Enable debugging?
17     var debugOn = 1;
18     var debugStr = "";
19
20     function doTabs()
21     {
22       var buffer = "0";
23
24       // Debug output
25       debugMsg("Inside doTabs()<br>");
26
27       ...
28
29       debugMsg("Leaving doTabs() - Value of buffer is: " + buffer + "<br>");
30     }
31
32     // Display debug messages
33     function debugMsg(msg)
34     {
35       if (debugOn == 1)
36       {
37         debugStr += msg;
```


© Copyright IBM Corporation 2007. All rights reserved.

```
38          _gel("debug_div").innerHTML = debugStr;
39        }
40      }
41
42      ...
43
44      </script>
45    ]]>
46    </Content>
47 </Module>
```

One thing to point out in this code: Inside the `debugMsg()` function, you append the incoming message to the `debugStr` variable. This is necessary to show the results of multiple calls to the debug function. Without this, only the last message will be displayed. Figure 17 shows how various messages might look within the market gadget you wrote previously.

Figure 17. Debug with div



Debug with Firefox

If you're using Firefox as your Web browser and you're working with a gadget that contains JavaScript, you can view the output of error messages from within the browser. In version 2.x, click on the **Tools** menu and choose **Error Console**.

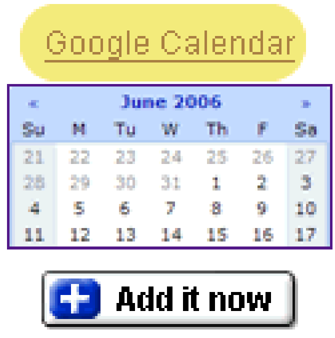
Sample code

One of the beauties of gadgets is that you can view the work of others. The best way to learn is to review what others have done and how they went about it.

To look at the code of other gadgets, visit the Google content directory (see Resources) and click the text link above the gadget image (see Figure 18).

Figure 18. Content directory

© Copyright IBM Corporation 2007. All rights reserved.



When the information page is displayed (see Figure 19), follow the link labeled **View source** to download the gadget XML file.

Figure 19. View source

© Copyright IBM Corporation 2007. All rights reserved.

One caveat: Not all gadget code is viewable as XML. If you recall from Part 1 of this tutorial, gadgets can be written in various scripting languages. Listing 10 shows an example of a hypothetical gadget written in Python and referenced through a URL.

Listing 10. Gadget URL

```
<Module>
  <ModulePrefs ... />
  <Content type="url" href="http://www.somewebsite.com/scripts/gadget.py" />
</Module>
```

Scratchpad

A discussion about writing and debugging gadgets wouldn't be complete without a reference to the Gadget Scratchpad (see Resources). Beyond testing your own code, this tool is great for viewing the source code of gadgets written by other developers. Figure 20 shows the Scratchpad editor.

Figure 20. Scratchpad editor

© Copyright IBM Corporation 2007. All rights reserved.

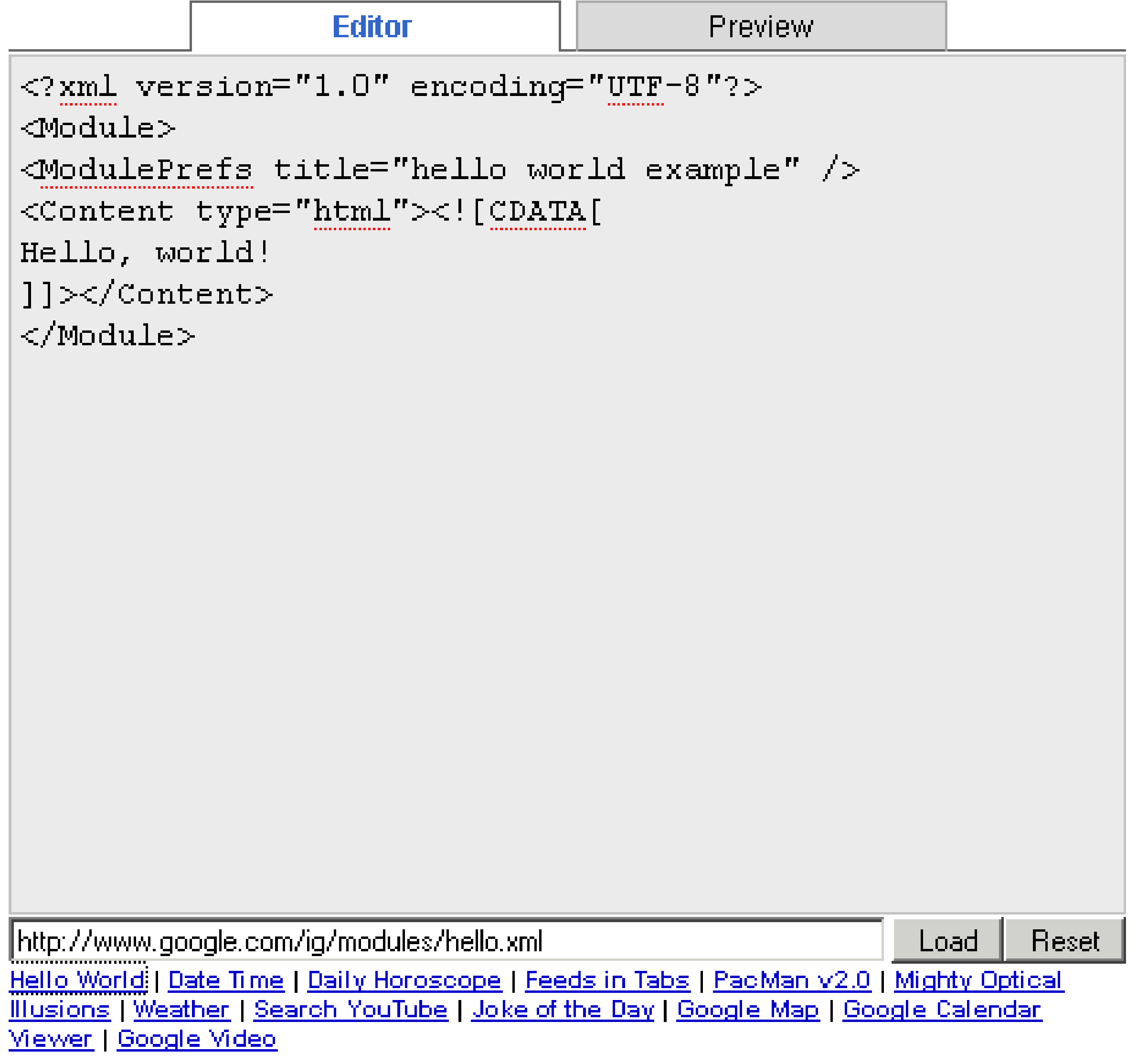
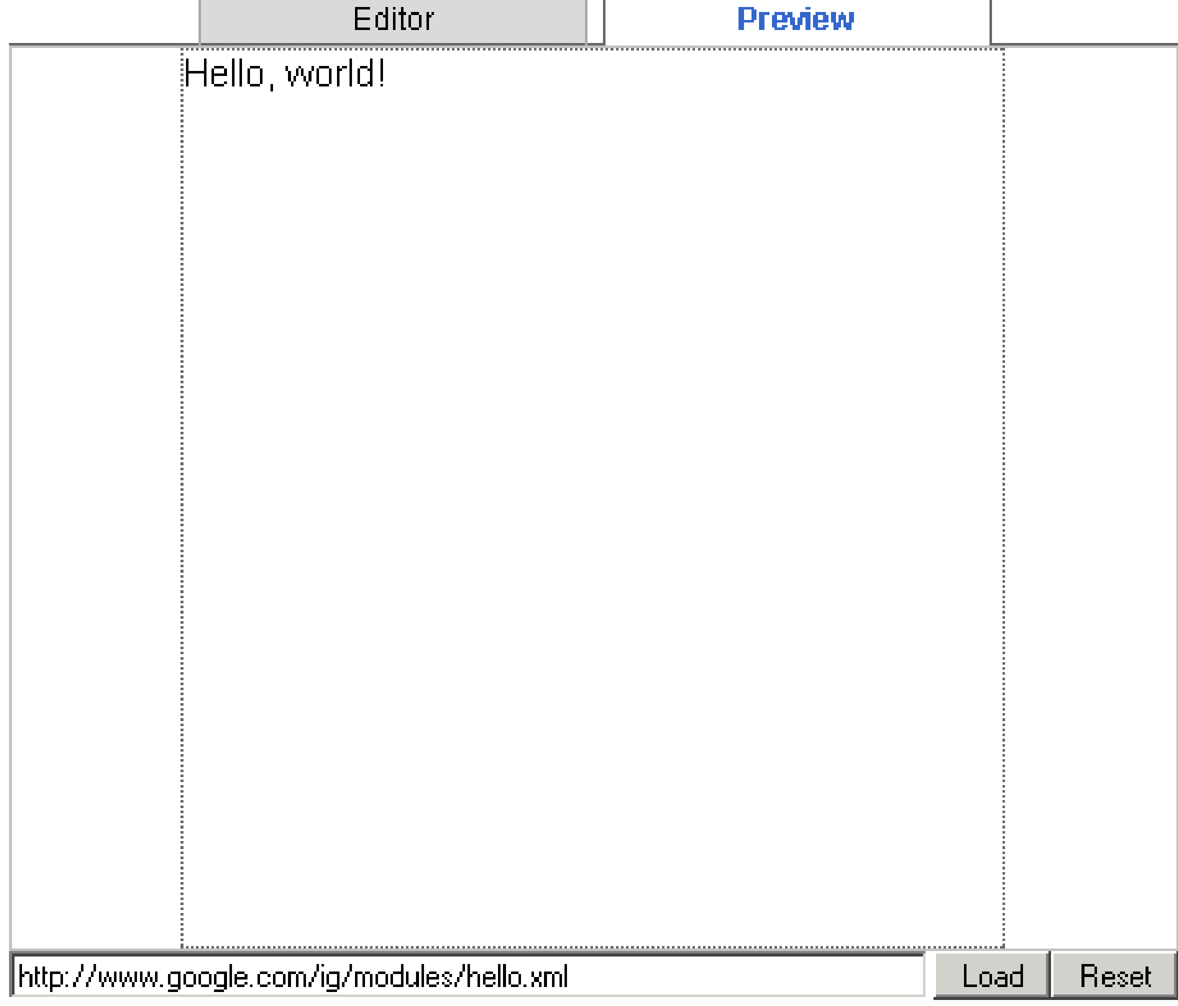


Figure 21 shows the code preview in the Gadget Scratchpad.

Figure 21. Scratchpad preview

© Copyright IBM Corporation 2007. All rights reserved.



You can also load gadgets from a URL into the Scratchpad using the text field shown on the bottom of the Scratchpad. One last suggestion: Try some of the links listed below the text field, as this is a great way to view the source (and output) of other interesting gadgets.

Section 11. Summary

Wrap up

© Copyright IBM Corporation 2007. All rights reserved.

This two-part tutorial on creating Google Gadgets has covered everything you need to know to get started working with, developing, and debugging gadgets. As was mentioned in the introduction of the first tutorial, what you can do with a gadget is limited only by your imagination (and your time to write the code, of course).

I recommend you start with one of the working gadgets covered in either tutorial and simply make a few changes to see how things work. As you gain more experience, delve into the code of gadgets written by others to learn and integrate more advanced features.

If you write a gadget you think is particularly cool, drop me an e-mail and tell me about it. Being a gadget freak (about both Google Gadgets and electronic stuff in general), I'd be happy to hear from you.

© Copyright IBM Corporation 2007. All rights reserved.

Resources

Learn

- The Google gadgets API Developer Guide presents all you ever wanted to know about writing Google gadgets.
- The tabs library presents the gadget tab API.
- For an inside look at the drag-and-drop API, check out the drag library.
- Take a look at the MiniMessage API in the MiniMessage library.
- Google offers a comprehensive introduction to working with various content types, as you can see in Working with Remote Content.
- Take a look at the code of other gadgets by visiting the Google content directory.
- You can use the Gadget Scratchpad to test your own code, as well as view the source code of gadgets written by other developers.
- If you're in need of a good JavaScript reference, check out the Mozilla developer center.
- If you need a good reference for CSS, start with the W3Schools CSS2 Reference.
- For an introduction to working with XML, read Working XML: Fundamentals of Web publishing with XML by Benoit Marchal (developerWorks, July 2003).
- Review the HTML 4 Specification to learn more about the `div` and `span` elements.
- Browse the technology bookstore for books related to XML, JavaScript, and other Web technologies.

Get products and technologies

- Download Eclipse and browse this comprehensive list of Eclipse plugins all related to XML.

Discuss

- Participate in the discussion forum for this content.
- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

© Copyright IBM Corporation 2007. All rights reserved.

John Muchow

John Muchow has been immersed in software development since 1988, working in roles from developer to chief software architect to CTO. Since 2000, John has focused on mobile technology, and during this time has had the opportunity to work alongside many high profile businesses including Nokia, Sony Ericsson, Symbian, Sprint/Nextel, Sun Microsystems, among others. He is also the author of Core J2ME Technology a best-selling wireless developer book that is still published in several foreign languages. John writes about mobile technology, from the developer perspective, on his blog 360Mobile.us.

© Copyright IBM Corporation 2007. All rights reserved.