Bing Bar 7 App Developer's Guide

A Bing Bar app uses HTML, CSS, JavaScript and asset files such as images to do something useful for the user. COM DLLs can also be used when JavaScript and the Bing Bar APIs are inadequate. A Bing Bar app resides in an environment (the Bing Bar) that gives it abilities beyond that of a standard web page. Bing Bar runs within 32-bit versions of Internet Explorer 7 and above on Windows XP (SP3) and above. Bing Bar runs on both 32-bit and 64-bit versions of Windows.

Requirements

Bing Bar app development only requires a text editor like Notepad. Development is easier using Visual Studio to debug JavaScript and to act as a test harness for unit testing of JavaScript code.

Other Resources

This section describes resources *not* included in this SDK you can use to develop Bing Bar apps.

Production Code

Bing Bar apps are published to user hard drives as source code. You can view the production source code of most shipping Bing Bar apps by viewing subdirectories within %localappdata%\Microsoft\BingBar\Apps\.

Note: Production apps contain real-world development patterns for Bing Bar apps. These source files are not intended to be code samples, do not contain detailed comments, and are not intended for code re-use within your apps.

Bing Bar App Development Discussion

Partners can reach Bing Bar team members for help using the Bing Bar App Development Discussion email alias at btbadd@microsoft.com. To coordinate translation handoffs with the Bing Bar Internationalization team, send email to bxcintl@microsoft.com.

Retail build of Bing Bar 7

To get the latest retail build of Bing Bar 7, visit this web address.

http://www.bingtoolbar.com/

Files included in this SDK

This section describes files and folders included in this SDK.

BingBarSetup.exe

This executable file installs the debug build of Bing Bar 7. The debug build includes a wing stay-open mode (see *Force the wing to stay open*), and is compatible with trace files (see *Use TraceView*). All other debugging features, including just-in-time debugging and the debug console, are available in both retail and debug builds of Bing Bar.

To install the debug build of Bing Bar 7:

1) Uninstall any existing version of Bing Bar. To uninstall, close all Internet Explorer windows, then from Control Panel, click **Programs and Features**, right-click **Bing Bar**, and click **Uninstall**.
2) Double-click BingBarSetup.exe.

WingApp Folder

This folder contains a simulation environment for wing development.

Shared Folder

Shared components you can include in your Bing Bar app, including CSS, JavaScript, and asset files you can use. This document includes descriptions of all shared components.

Registry Folder

This folder contains .reg files that turn debugging features on and off.

Design Folder

This folder contains PSD files that include colors and graphic elements designers can use to create derivative designs. These resources describe elements in shared\css\bingclient.css.

Tracing Folder

This folder contains files used by TraceView to interpret tracing output for this debug build of Bing Bar.

Samples Folder

This folder contains these samples.

Sample	Description
HelloWorld	Sample that includes an app button and an app canvas called a wing.
Linkomatic	Sample that contains a footer, a service, and a page task.
Simple	Simple app that contains a service and page task. Use this sample as a starting point for a new Bing Bar app.
Persistent	Sample that creates a persistent wing app. (v 7.1 only)
Unison	Complex app that uses a service and many page tasks to change the user's web page.

Install any sample by copying files into the Bing Bar app directory.

To install HelloWorld:

From a cmd prompt, navigate to the directory of the sample, and copy the sample files to %localappdata%\Microsoft\BingBar\Apps.

Example:

```
c:\sdk\samples\HelloWorld>xcopy /s /i *.*%localappdata%\Microsoft\BingBar\Apps\HelloWorld\7.0.0
```

If Internet Explorer is running, close all IE windows and run it again. The button of the Hello World sample app will appear at the right end of the Bing Bar.

Development environments

You can develop Bing Bar apps in two environments. You can start by prototyping in WingApp, a wing simulator. Later, when you need to interact with the Bing Bar APIs, you can develop your code in-place in the Bing Bar. This section describes both techniques.

Run WingApp

WingApp is a wing simulator that simplifies development of wing user interfaces. WingApp simulates a subset of Bing Bar APIs. WingApp primarily helps development of UI content rendering and user interaction in HTML, CSS, and JavaScript.

To run WingApp, double-click wingapp\wingapp.html.

Note: WingApp uses ActiveX controls to mock Bing Bar services. To run WingApp, approve security dialogs that appear in your browser. Security dialogs may appear at the top, bottom, or center of your browser window, and may disappear if not approved quickly.

To run HelloWorld in WingApp, type ..\samples\helloworld in the **App directory** field and click **Go**.

A log of code activity updates at the bottom of the page while apps run within WingApp.

In addition to HelloWorld, WingApp can run these sample apps located in the WingApp folder:

Sample	Description
App	A demonstration of jQuery and jQuery UI.
CSSExplorer	An overview of styles defined in shared\css\bingclient.css and a demonstration of the jscrollpane scroll bar.

Footer	A sample that demonstrates the shared wing footer.
Metrics	A sample that reports pixel sizes of wing components.

Use WingApp to debug wing UI

WingApp loads app HTML and JavaScript into the simulated wing at the top of the page. If you see nothing, there is probably a syntax error in app JavaScript. To debug JavaScript and view the app DOM using the built-in IE debugger, press F12. In the IE debugger, click the **Script** tab, then click the **Start debugging** button. The console will show more information about the error. You can set breakpoints and modify variables within the debugger. You can also modify the DOM in the **HTML** tab, and modify styles in the **CSS** tab. After you modify a source file, click the **Go** button in WingApp to see the change in the new wing.

To debug a service using WingApp, temporarily move the service source code to the wing source.

Code a Bing Bar app in-place

The WingApp simulator can show how user interfaces made with HTML, CSS, and JavaScript appear in an app wing. To call Bing Bar APIs, your app must be developed in the running Bing Bar, by writing code in-place. Source code for all running Bing Bar apps resides in subdirectories of %localappdata%\Microsoft\BingBar\Apps.

Caution: When you uninstall the Bing Bar, all app files are deleted. Copy your app source to a safe location before uninstalling.

Enable JavaScript debugging in Internet Explorer

You must enable script debugging in Internet Explorer to debug wing or page task JavaScript code in Visual Studio. To enable script debugging in Internet Explorer 9, click **Tools**, click **Internet options**, and click the **Advanced** tab. In the **Browsing** section, uncheck **Disable script debugging (Other)**, and click **OK**. Settings take effect after you close all Internet Explorer windows.

When debugging is enabled, a Visual Studio just-in-time debugging dialog will appear when JavaScript code reaches a JavaScript **debugger;** statement.

Note: A JavaScript **debugger;** statement in service code will always trigger a just-in-time debugging dialog regardless of script debugging settings in Internet Explorer.

Parts of a Bing Bar App

Bing Bar apps are made up of one or more code parts: a service, a wing that shows user interface elements, and a page task. Each part can access the others programmatically, but run in different processes. Most apps also provide a button. This section describes these elements.

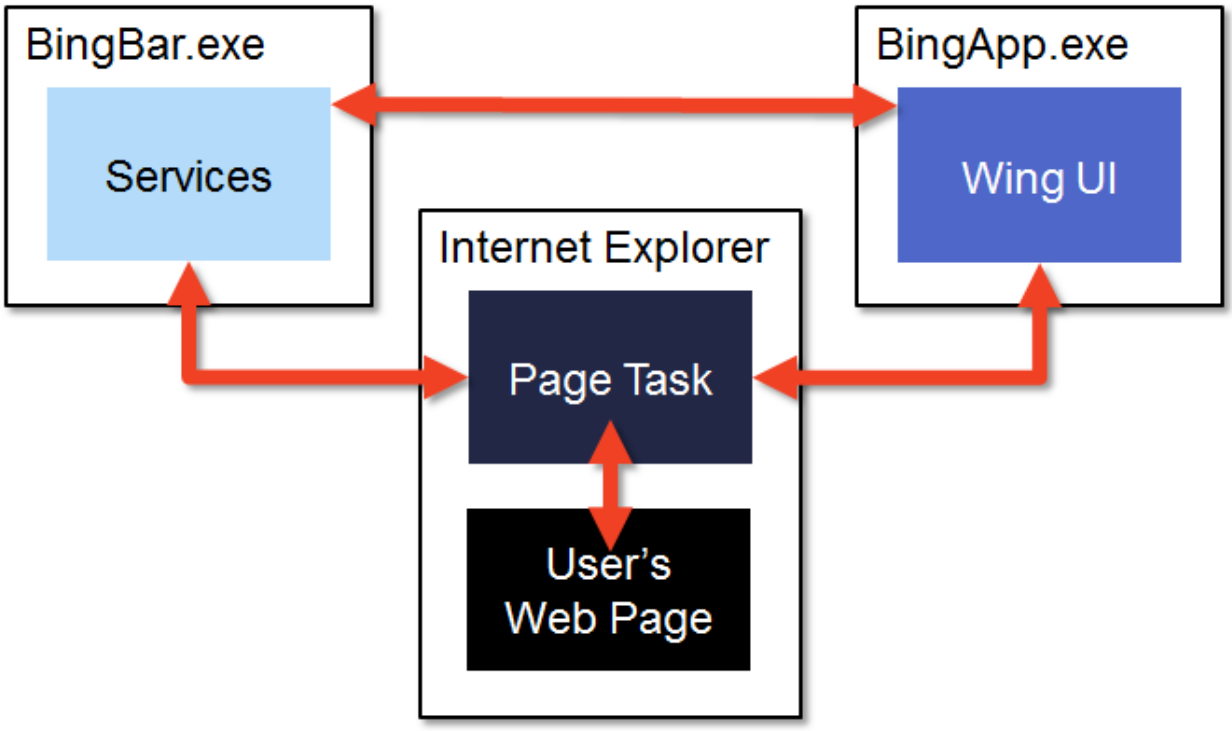


Figure: App parts at runtime

Note: An app wing includes an HTML tag that forces code and content running on a later browser version to run in Internet Explorer 7 compatibility mode. This limits wing features to browser features found in Internet Explorer 7. This limitation does not impact web pages viewed by the user in Internet Explorer.

App Services

A service is a long-lived piece of code that typically performs periodic operations. For example, the mail app service checks for new messages on an update schedule. Services have no user interface. Services run in a JavaScript runtime extended with the Bing Bar APIs and services can use all Bing Bar APIs except members in the bingclient.flyout namespace. Services can access a web page DOM, but cannot use jQuery because there is no window object in the service script host. A service can load a page task to access a DOM with jQuery. Services may fire alerts to the user. Services often subscribe to browser-related events, such as the bingclient.ui.browser.events.documentComplete event. All Bing Bar app services start when the first iexplore.exe process starts, and continue running until the last iexplore.exe process shuts down. When Internet Explorer starts, Bing Bar loads JavaScript files listed in the AppManifest.xml of each app. Don't include JavaScript files that are only used by wing or page task code in AppManifest.xml.

Note: Any app can call any function on any service.

Scalable services

Bing Bar reaches millions of computers. An app service might query remote servers regularly for updates. Intelligent service code should reduce contacts with remote servers for users who aren't actively using the app. A scalable service also centralizes refresh frequency and service URLs in the settings.js file so these settings can be modified without updating the entire app source.

Service Naming Conventions

Apps register services by name. To avoid name collisions with the services of other apps, services use the naming convention *CompanyName*.BingBar.*ServiceName.* For example, the service of the mail app is named Microsoft.BingBar.Mail.

Events

Services are in a unique position to monitor browser events, and take action on interesting events. For example, a service can subscribe to an event that fires when a browser document completes, in any tab. The Bing Bar mail app uses this event to notice when a user has navigated to Hotmail, and invites the user to monitor Hotmail with Bing Bar. For more information, see bingclient.ui.browser.events in *Bing Bar 7 API Reference*.

Periodic Updates

Services often refresh information from remote sources. Since jQuery is not loaded into the Bing Bar runtime, the Bing Bar API provides a set of analogous timer functions. For more information, see bingclient.scripting.timers in *Bing Bar 7 API Reference*.

Ajax, XML, JSON, settings and localization methods from serviceutility.js

The shared\utility\serviceutility.js file is included by nearly all Bing Bar services. This table describes features provided by this code.

Class / Method	Description
XmlHttpRequest class	Services cannot use the jQuery $.ajax, $.get, and $.post methods to send HTTP requests and perform asynchronous network I/O. A set of analogous functions are provided by the XmlHttpRequest class in shared\utility\serviceutility.js.
XMLParser class	Parses a string that contains valid XML using the msxml2.domdocument.6.0 ActiveX object.
String.toHexString method	Lets String variables in Bing Bar app code represent a large integer value in a hexadecimal string.
ServiceUtilities.loadJSONFile method	Loads a JSON file contents into a JavaScript object.
ServiceUtilities.loadSettings method	Loads settings.js file of a specific locale.
ServiceUtilities.loadLocStrings method	Loads locStrings.js file of a specific locale. Called from service code to load localized strings. See *Localize your app*.

Communication with Wing UI

Through most of its life, a service runs in the background refreshing data or monitoring browser events. When the app wing opens, the service may want to rapidly communicate updates to the wing. For example, in the Bing Bar mail app, when the user clicks the Refresh link to refresh the mail view, the wing UI updates as soon as the service obtains the updated mail.

These interactions require the wing and service to set up a callback relationship that lasts as long as the wing is open. When the wing closes, the callback relationship is broken. This is a common pattern among Bing Bar apps. For a code example, see *Set up and tear down service-to-wing callbacks*.

File I/O

Services often save and load data to local disk files. The Bing Bar API includes functions to use files and directories, and includes encryption methods for storing Personally Identifiable Information (PII) to disk. File I/O is most commonly used to store an app's state across sessions, but it can also be used for any other purpose. All file I/O in Bing Bar apps must appear within a try block with an exception handler.

JSON2.js

The shared\json\JSON2.js file is a standard implementation of JSON serialization and deserialization features. It is often included by both service and wing app code. JSON is a convenient way to store user preferences and other information to disk, and can also process web APIs that use JSON format for information interchange. Calls to the JSON.parse() method should be wrapped in a try-catch block.

App Wings

The wing is the visual app space that appears below the Bing Bar. A wing typically includes HTML, JavaScript, and CSS components. The filename and pixel dimensions of the app wing are defined in the <AppUI> element of the app AppManifest.xml file. Only one wing can appear at a time. Typically, wing code will contact an app service to get the latest information to show.

Wingless apps

It is possible to write an app that has no wing. A wingless app could just communicate using alerts, or register a service used by other apps. A wingless app sets AppType="**noui**" in AppManifest.xml.

jQuery

The jQuery library is a widely-used JavaScript library that streamlines the manipulation of dynamic HTML from JavaScript. Wing code can also use jQuery to send HTTP requests. For information about jQuery, see this web address.

http://jquery.com/

A copy of jQuery is provided in the shared\jquery directory. All Bing Bar apps with a wing use this library heavily. The shared\jquery directory also includes jQuery UI. This package includes controls you can use, including a calendar control and a color-picker control. For information about jQuery UI, see this web address.

http://jqueryui.com/

Thousands of jQuery extensions are available on the internet. Microsoft must review the use of any open-source code, but code licensed under the MIT license is typically approved.

Wing HTML file

Wing UI is typically defined in a single HTML file that contains many different wing UI elements separated into groups. By showing and hiding groups of UI elements, the HTML file supports many different visual states of the wing.

This code excerpt shows the top of every wing HTML file.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7">
        <link type="text/css" href="css/bingclient.css" rel="stylesheet">
        <!--your css inclusions here -->

        <script type="text/javascript" src="js/jquery-1.4.2.min.js"></script>
        <script type="text/javascript" src="js/myapp.js"></script>
        <!--your JavaScript inclusions here -->

    <!-- App code instantion -->
    <script type="text/javascript">
        var gApp = new MyApp();
        $(function ()
        {
            gApp.onReady();
        });
    </script>
</head>
```

Most apps use bingclient.css to provide element styles and colors that are consistent with other Bing Bar apps. Most apps also include jQuery and other JavaScript files from the shared\javascript directory, and one or more files that implement app logic in JavaScript. In this example, MyApp is the JavaScript class that implements the app. After Internet Explorer has fully loaded the JavaScript code included in this HTML, script code instantiates the app object and calls the onReady method in MyApp.

The HTML <body> element must contain three elements to provide a footer and Options view. See *Use footer.js to provide a wing footer* for details.

Ids and Classes

Most apps manipulate wing HTML from JavaScript code using jQuery with element identifiers. This example sets the identifier of a element.

```
<div id="links" class="linesBordersBorder"></div>
```

This jQuery sets the text of the element, addressing it using its identifier.

```
$("#links").text("Can't contact service.");
```

You can also apply CSS styles to an element using its identifier.

```
$("#header").addClass("primaryLightBackground");
```

Use element classes to access all similar elements. For instance, it might make sense to tag all the elements in a list with the same class. You can find and operate on all elements of the same class using jQuery, and can also apply CSS styles by class. This HTML applies the listItem class to 3 elements.

```
<div class='listItem'>Item 1</div>
<div class='listItem selectedItem'>Item 2</div>
<div class='listItem'>Item 3</div>
```

This jQuery code removes the selectedItem class from all elements and then applies it to just the first element.

```
$(".listItem").removeClass("selectedItem").find(":first").addClass("selectedItem");
```

JScrollPane

The shared\jscrollpane folder provides Bing Bar-styled scroll bar elements your wing code can use to provide content scrolling.

Note: This code does not meet accessibility standards set in MATS (Microsoft Accessible Technology Standards) 1.0 Level 2. In particular, this feature is invisible on some high contrast display themes, because high contrast themes make many images invisible. See *Assure accessibility* for ways to improve usability on high contrast themes.

Persistent wings (version 7.1 and higher)

By default, a wing is created each time the user clicks on the app button, and is destroyed when the wing closes. By responding to a bingclient.flyout.events.closing event, an app can make its wing persistent. Instead of being destroyed when it is closed, a persistent wing will be hidden and continues to run in the background. The app also receives bingclient.flyout.events.opening event when its wing is being re-shown. A persistent wing enables uninterrupted use of an audio media player or hosting of a control such as a game that needs to persist across wing invocations. See bingclient.flyout.events in *Bing Bar 7 API Reference* for more information.

App Page Tasks

A page task is a special, usually small piece of code that has efficient access to the DOM of the browser's current web page. Page tasks can run jQuery selections on the page DOM efficiently to find particular information, or even modify the page itself.

Wing code can use jQuery to access and modify the DOM of a web page. However, since the wing runs in a different process from the web browser, code that acts on the DOM incurs substantial remote-procedure-call traffic when running in wing code, negatively impacting performance. A page task runs in the same process as the DOM. In a page task, code in a service or a wing loads and runs a script in the same process as the DOM.

A page task runs in a private script host that has access to the DOM of the web page. It does *not* run in the same script host as the JavaScript on the web page. The script host of a page task only supports the window and document objects, which point to the current page. The script host does not support any other JavaScript objects, or any Bing Bar APIs.

This code example from the Linkomatic sample loads and runs two page tasks. One page task provides jQuery to the other page task.

```
var path = bingclient.context.app.getActiveVersionPath();
var jqpath = bingclient.io.path.append(path, "jquery-1.4.2.min.js");
var lnkpath = bingclient.io.path.append(path, "linkinator.js");

var session = bingclient.ui.browser.session(evt.sessionId);
var scope = bingclient.context.app.getScope();
session.scriptDebuggingEnabled = true;
session.loadScriptFile(scope, jqpath);
session.loadScriptFile(scope, lnkpath);

var linkinator = session.executeScript(scope, "new Linkinator()");
var links = linkinator.getPageLinks();
```

Note: Page tasks have to be written defensively since the user can navigate between the onDocumentComplete event and launch of a page task.

App buttons

An app button uses image overlays and changes to show the app state. Clicking an app button opens the app wing. An app with no wing can subscribe to and handle button events in service code. Button backgrounds should be transparent, so visual cues for hovering and pressing appear from behind the button image. Buttons can specify different default image files for different magnification. A button image can change, but real-time animation is not supported. Apps specify their default button image in the AppManifest.xml file, and change button image and state indicators using methods of the bingclient.ui.button namespace. To start quickly in Internet Explorer, Bing Bar caches overlay states between runs.

Development techniques

This section describes development techniques for Bing Bar apps.

Use the color and style classes in bingclient.css

The Bing Bar and its apps present a consistent experience to the user. The shared\css\bingclient.css file provides the easiest way to create an app with the Bing Bar appearance and behavior. This file contains classes that provide colors, text styles, and buttons used in Bing Bar apps. Whenever possible, avoid declaring colors explicitly. Instead, use the color classes in bingclient.css. You can add class names defined in bingclient.css to your elements at design time or at run time using jQuery. Note that more than one CSS class can be assigned to an HTML element.

Colors for any element

You can set the color of any element, element background, or element border by adding a CSS class that follows the bingclient.css naming convention for element coloring. Class name roots are listed in *Appendix A: Color Palettes* in *Bing Bar 7 Human Interface Guide*. The class name suffixes--Color, Background, or Border--specify where the color should appear on an element.

This jQuery call adds the primaryLightBackground class to the element, changing the background area of this element to a light blue color.

```
$("#header").addClass("primaryLightBackground");
```

This example uses HTML and CSS to show text in a rectangular area with a line divider at the bottom.

In HTML:

```
<div id='statusBox' class='primaryDarkBorder primaryExtraLightBackground paragraphTextColor'>
   Today's my birthday!</div>
```

In CSS:

```
#statusBox
{
  width: 308px;
  height: 54px;
  padding: 4px;
  border-bottom: 1px solid;          /* Note: not specifying color here, just metrics */
}
```

This jQuery call sets the background of the element with the header identifier to a purple color (the History color is purple in this release of Bing Bar).

```
$("#header").addClass("HistoryBackground");
```

This example removes one class and adds another.

```
$("#searchBox").removeClass("secondaryDarkColor").addClass("paragraphTextColor");
```

Text styles

CSS classes use a naming convention based on the text styles listed in *Appendix A: Color Palettes* in *Bing Bar 7 Human Interface Guide*. The bingclient.css file uses lowercase, and does not include dashes, so the TX-1 style for extra-large text becomes tx1 in bingclient.css.

Button styles

CSS classes use a naming convention based on the button styles listed in *Appendix A: Color Palettes* in *Bing Bar 7 Human Interface Guide*. Follow this naming convention for button styles in bingclient.css.

btn*n*[Direction][Side][State]

For example, to show a left-pointing Btn-5 style button in the hover state, set the button style to btn5LeftHover.

```
$("#myButton").toggleClass("btn5LeftHover");
```

Create an app GUID

A Bing Bar app has a unique GUID. This GUID appears in the app's appmanifest.xml file. Bing Bar GUIDs do not include dashes.

To create a GUID, use a tool like http://www.guidgen.com/. Be sure to remove dashes from the GUID string.

Use footer.js to provide a wing footer and Options view

The source files in shared\footer implement the footer control. A footer is a band across the bottom of the app wing that shows your app name on the left and an optional **Options** button on the right. When the **Options** button is pressed, the Options view appears. Footer.js also provides two kinds of delay indicators called spinners. Footer.js changes z-index ordering with a visual transition effect to switch between your main wing HTML and the Options view. Every app wing must include a footer.

To add the footer to your wing, follow these steps:

1. Copy the shared\footer directory and contents to your app source, at the same level as your appmanifest.xml file.
2. Add these lines to your app's wing HTML file:

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

3. Include these three <DIV> sections in your wing HTML.

HTML section	Description
	Put the HTML for the main user interface of your wing here.
	Put the HTML for your Options view here.
	Don't put any HTML here. Footer.js will insert HTML at runtime.

Initialize your footer control in your onReady method.

Example:

```
this._footer = new Footer({
  appNameStr: gLocStrings["appName"],
  optionsStr: gLocStrings["optionsButtonText"],
  optionsCloseStr: gLocStrings["optionsCloseText"],
  optionsOpen: function (footer)
  {
    // Called when options is opened.
    $("#options").append("Options opened!<br/>");
  },
optionsClose: function (footer)
{
  // Called when options is closed
  $("#main").append("Options closed!<br/>");
}
});
```

Footer object parameter	Type	Description
appNameStr	string	Localized friendly name of app.
optionsStr	string	Localized Options button text.
optionsCloseStr	string	Localized Close button text.
optionsPreclose	callback	Optional. Function called when options is open and user has clicked the Close button. If the callback function returns Footer.kAllowClose, the Options view closes and the optionsClose function is called. If the callback returns Footer.kDenyClose, the close is cancelled, and the Options view remains open. Signature: bool optionsPreclose(footer)
optionsOpen	callback	Function called when Options view opens. Signature: optionsOpen(footer)
optionsClose	callback	Callback called when options is closed. Signature: optionsClose(footer)
enableOptions	boolean	Optional. enable/disable Options view (default: true).

Validate user settings in the Options view

You can validate settings in the Options view synchronously or asynchronously. The Footer sample in the WingApp simulator demonstrates both techniques.

Code can validate settings in the Options view synchronously (without network calls) by following these steps.

1. Provide an optionsClose function in the Footer creation object that stores data pulled from the Options view.
2. Provide an optionsPreclose function in the Footer creation object that validates data pulled from the Options view. If valid, return Footer.kAllowClose. The Options view will visually close and optionsClose will be called.
3. If optionsPreclose determines the settings are invalid, modify the Options view HTML to show an error message, and return Footer.kDenyClose. The options panel will remain open, and optionsClose will *not* be called.

Code can validate settings in the Options view asynchronously, for example waiting on $.ajax calls to complete before validating settings, by following these steps.

1. Provide an optionsClose function in the Footer creation object that stores data pulled from the Options view.
2. Provide an optionsPreclose function in the Footer creation object that spawns your asynchronous operation and calls showSpinner(Footer.kShow) to show the spinner. Also call enableOptionsButton(Footer.kDisable) to disable the Options close button.
3. If your async operation returns success, call showSpinner(Footer.kHide) and call showOptions(Footer.kHide) to close the Options view. This will cause options to be closed and optionsClose to be called. Also call enableOptionsButton(Footer.kEnable) to turn the Options button back on.
4. If your async operation fails (or times out), call showSpinner(Footer.kHide) and modify your HTML to show an error message. Options will remain open with an enabled Close button, and optionsClose will *not* be called. Call enableOptionsButton(Footer.kEnable) to turn the Options button back on.

Make the app title a hyperlink

Some apps make the app name in the footer a hyperlink to a web site. To create a hyperlink title, create the Footer object, initialize it, and then run this code.

```
$(this._footer.getAppNameElement())
  .mouseover(function (event)
  {
    $(this).css("text-decoration", "underline");
  })
  .mouseout(function (event)
  {
    $(this).css("text-decoration", "none");
  })
  .click(function (event)
  {
    window.open("http://www.bing.com");
  });
```

Use utility.js

The shared\utility\utility.js file contains methods you can use in your wing code.

Class / Method	Description
XMLParser class	Parses a string that contains valid XML using the msxml2.domdocument.6.0 ActiveX object.
String.toLocalPath method	Converts a local path from HTML into a path that can be used with members of the bingclient.io namespace.
String.folderOf method	Removes the file name from a full path and returns the directory portion.
String.stripHTML method	Strips HTML tags from a string of text.
String.toEllipses method	Truncates strings and adds an ellipsis. Use only when CSS text-overflow is insufficient.
String.singleWordtoEllipses method	Truncates a string that contains a single long word using an ellipsis. Use only when CSS text-overflow is insufficient.
getTextWidth method	Measures a string.
setDisplayWidth method	Set an element's width to fit a string.
Date.setISO8601 method	Set a date object to a date-time string in ISO-8601 format.
linkEnterKeyToClick method	Registers a keyup handler for all elements with the **tabStop** class. The keyup handler calls the element's click handler when the user presses Enter when the element has focus. Use for improved MATS compliance.
IsNullOrEmpty method	Returns **true** if a string is **null** or zero-length.
isKeyDownAndSpaceOrEnterKey method	Returns **true** if the jQuery event is a keyDown event and the key is Space or Enter.
$.flashClass method	jQuery extension to add a class to an element and remove the class after a delay. Useful for flashing a pressed state image when the user activates an element using the keyboard.
$.disableTextSelection method	jQuery extension that binds selectStart to prevent text selection in an element.
reportError method	Reports an error using the bingclient.debugging.tracing.traceError method.
Utilities.loadLocStrings method	Loads localized strings into wing code.
Utilities.getLocal method	Returns the current locale.
Utilities.loadSettings method	Loads settings into wing code.
Utilities.loadJSONFile method	Loads a JSON file and returns a JavaScript object.
window.linkify method	Generates link HTML from a URL.

Constants in utility.js

The utility.js file contains a class called *Constants* that contains a set of useful constants. Feel free to use these to help make your code more readable.

```
Constants.kForReading = 1;          // For FileSystemObject.OpenTextFile, et al
Constants.kForWriting = 2;
Constants.kForAppending = 8;
Constants.kNoCreate = false;        // For FileSystemObject.OpenTextFile, et al
Constants.kCreate = true;
Constants.kShallowCopy = false;     // For jQuery.extend()
Constants.kDeepCopy = true;
Constants.Keys =
{
    kBell: 7,
    kBackspace: 8,
    kTab: 9,
    kLF: 10,
```

```
    kCR: 13,
    kSpace: 32,
    kDownArrow: 40,
    kLeftArrow: 37,
    kUpArrow: 38,
    kRightArrow: 39,
    kEsc: 27,
    kR: 82,
    kA: 65
};
```

Use appsettings.js

The AppSettings object is defined in shared\utility\appsettings.js. Add an AppSettings object in to your service to create, load, and save a dictionary of key-value pairs to a file between user sessions.

Centralize settings to settings.js

The settings.js file contains settings you can change after your app has been deployed. It is easier to deploy an updated settings.js file than to update an entire app. Locate polling intervals in settings.js so you can respond if your app overwhelms a server-based resource. URLs should also reside in your settings.js file so they can be changed later.

Marketize your app

Each language-market group can have its own version of the settings.js file. If your app uses different URLs in different locales, these URLs should be specified in settings.js files specific to each language-market group. You can also use settings.js files specific to language-market groups to hold format pictures for bingclient.localization.formatDate and bingclient.localization.formatTime methods.

Usually you will load the correct settings.js file based on locale from within your app's _initialize() method.

```
var utilities = new Utilities();
var locale = utilities.getLocale();
utilities.loadSettings(locale);
```

By default, the Bing Bar team copies your default (en-us) settings.js into every market. If you want different behavior, contact the Bing Toolbar App Development Discussion alias at btbadd@microsoft.com.

Localize your app

To enable translation of text that appears in your app, do not include text strings in HTML. Instead, leave the HTML element text blank, and update the text at runtime using text from the localized locStrings.js file that corresponds to the current locale.

The locStrings.js file contains a dictionary of key-value pairs, where the value is the localizable string. Here's a simple locStrings.js file.

```
var gLocStrings =
{
    "appName": "Mail",
    "optionsLink": "Options",
    "optionsCloseLink": "Close",
    "optionsHeading": "Mail Options",
    "mainHeading": "Mail",
    "YahooMail": "Yahoo Mail",
    "Gmail": "Gmail",
```

```
    "Hotmail": "Hotmail"
};
```

Microsoft will translate text in your en-us (English-USA) locStrings.js file and appmanifest.xml file into translated files for the markets where you ship. Contact bxcintl@microsoft.com to coordinate handoff and completion dates for translations.

Load localized strings

An service typically loads strings from the correct locStrings.js file within its _initialize() method. This code example uses shared\utility\servicesutility.js to load localized strings.

```
ServiceUtilities.loadLocStrings(
  bingclient.context.app.getActiveVersionPath(),
  bingclient.configuration.getLocale());
```

A wing app loads strings using a slightly different method.

```
var utilities = new Utilities();
utilities.loadLocStrings(bingclient.configuration.getLocale());
MyApp.locStrings = gLocStrings;
```

This code example uses a localized string.

```
$("#myDiv").text(gLocStrings["myDivTitle"]);
```

Minimize cross-process traffic

The architecture diagram shows the BingBar.exe, BingApp.exe, and Internet Explorer processes. Calls between processes are expensive marshaled RPC COM calls. This includes every property access, variable reference, and method call between different processes. To maximize performance, make as few such calls as possible. For instance, to pass an array of data across, you could access the array directly. While this works, every array member access is a separate cross-process call.

A better solution is to use JSON2.js to convert the array into a big string, send the string across the process boundary using one cross-process call, and then re-convert the JSON string into the array in the receiving process. This technique is usually faster than making many cross-process calls. The mail app uses this technique to move the mail and account arrays between the service and the wing.

Avoid circular references between the DOM and JavaScript

DOM elements and JavaScript objects can refer to one another. In this code example, the JavaScript object saves a reference to the DOM object. Then it creates a circular reference using an expando property on the DOM object to refer back to the JavaScript object.

```
function MyComponent(element)
{
  ...

  // Reference from JavaScript to DOM
  this._panel = document.getElementById('uxPanel');

  // Reference from DOM to JavaScript
  this._panel.componentInstance = this;
}
```

If not explicitly broken by the app, this kind of circular reference can cause memory leaks. One way to avoid this is to refer to the DOM object by its identifier. If you must create a circular reference, make sure you break it explicitly when you're done using the objects.

Code for future versions

Bing Bar deploys apps in a version-aware directory structure. When a new app version is deployed, Bing Bar deletes all files in the bingclient.context.app.getActiveVersionPath() directory. Use bingclient.context.app.getVersionIndependentPath() to store files you want to keep between version upgrades. You must handle file format changes between versions, and users can skip verisons in some upgrade scenarios. To simplify upgrades across versions, you can store user settings in JSON data structures that can include version details.

Set up and tear down service-to-wing callbacks

To raise an event from your service to your wing, create a service-to-wing callback by following these steps.

1. Expose a callback method in your service code.

```
MyService.prototype.setDataChangedCallback = function(cb)
{
    this._dataChangedCallback = cb;
}
```

2. When data changes in the service, call your callback.

```
if (this._dataChangedCallback != null)
{
    try
    {
        this._dataChangedCallback();
    }
    catch(e)
    {
        // Don't let a failed callback stop the service.
    }
}
```

Always check for null, because the callback is only valid when the wing is open. Set this call in a try-catch block in case the reference is invalid or the wing throws an exception.

3. In the wing code, in onReady, set the callback.

```
this._service.setDataChangedCallback(this._myDataChangedCallback);
```

4. In the wing code, in window.unload, unset the callback.

```
this._service.setDataChangedCallback(null);
```

Provide instrumentation pings

Your Bing Bar app can contact your servers to track user activities, app failures, or other events. Microsoft can also share basic information about your app performance with you. If these options are not enough, you can seek agreement with Microsoft to use methods in the bingbar.instrumentation namespace to track events in your app.

Open a web page

There are (at least) four ways to open a web page in a browser.

Create a new tab navigated to a URL

```
window.open('http://www.msn.com');
```

Create a new tab navigated to a URL (alternative)

```
bingclient.ui.browser.session(bingclient.context.browserSessionId).create("http://www.msn.com");
```

Create a new window navigated to a URL

In this example, it's typical to close the open wing.

```
bingclient.ui.browser.create("http://www.msn.com");
bingclient.flyout.close();
```

Navigate the current tab to a new URL

```
bingclient.ui.browser.session(bingclient.context.browserSessionId).navigate("http://www.msn.com");
```

Run code when the wing is closing

To run code when the wing is closing (and is not just hiding by returning bingclient.flyout.events.kPersistent from a closing event handler), set a handler for the .unload function.

```
$(window).unload(function (event)
{
  instance._onExit();
});
```

Get the relative path in a script

A script can get the path where its own source code is located. The script can use this path to reference file resources in the same directory.

```
var scriptPath =
  document.getElementsByTagName("script")[document.getElementsByTagName("script").length -
1].src;
```

Assure accessibility

The *Human Interface Guide* specifies design patterns apps use to maintain accessibility for all users. App code can use the bingclient.accessibility.isHighContrast property to determine when code is running in a high contrast visual theme. To detect when code is running in a high dots-per-inch setting, app code can examine the screen.deviceXDPI property. See more information at this web address.

http://msdn.microsoft.com/en-us/library/ms533721(v=vs.85).aspx

Load tab sequence

The tabindex attribute in HTML helps keyboard users navigate your interface. You can often set tab index at design time, but sometimes you need to set tab index values at runtime. When testing the tab sequence in the UI, you may get unexpected results. For example, while tabbing through elements, at some point it may appear that no element has focus. To help diagnose these kinds of issues, add this code to your app's onReady() method.

Note: This code requires the debug console. To add the debug console to your wing, see *Use DebugConsole.js to call and evaluate JavaScript at runtime*.

```
$(document).keydown(function (e)
{
  try
  {
    if (e.which == Constants.Keys.kTab)
    {
      var name, elem = document.activeElement;
      if (!elem)
      {
        name = "[nothing]";
      }
    else if (elem.id && typeof (elem.id) == "string" && elem.id.length > 0)
```

```
        {
          name = "id: " + elem.id;
        }
      else name = "no-id &lt;" + elem.tagName + "&gt;";
      gConsole.writeLine("Focused before tab: " + name);
      }
    }
    catch (e)
    {
      gConsole.writeLine("Tab detect exception: " + e.message);
    }
  });
```

To use this code, tab through the tab sequence, then open the debug console to see the ids of the elements that had focus during the sequence.

Add image alternatives for high-contrast themes

Windows can be set to a high contrast theme. In a high contrast theme, CSS-based background-image attributes do not appear. (HTML-based <img> graphics still appear in high contrast themes.) This can make user interface elements that are drawn using pre-rendered graphics disappear.

Apps can determine if a high-contrast color theme is active.

```
var isHighContrast = bingclient.accessibility.isHighContrast;
```

Apps can add text that appears when elements that use background-image disappear in high contrast scenarios. For example, this HTML creates a menu button.

```
<div id="acctMenuButton"></div>
```

An app can show a down-arrow button in normal themes, and can show a similar image in high contrast scenarios.

Visual image	Description
	Image that appears in normal visual themes.
	Symbol that appears in high contrast themes.

This CSS code implements this alternative symbol.

```
#acctMenuButton
{
    width: 14px;
    height: 14px;
    background-image: url('../images/acctbtn.png');
    font-family: 'Symbol';
    font-size: 10px;
    font-weight: bold;
    line-height: 10px;
    text-align: center;
}
```

This code sets up text attributes for the element even though there is no text present. At runtime, if high contrast is detected, set the text of the element.

```
$("#acctMenuButton").html(bingclient.accessibility.isHighContrast ? "&#xda;" : "");
```

The character represented by 0xda is a down-arrow in the Symbol font. Using this technique, the normal themes show the graphic and high contrast themes show the symbol.

When a symbol can't replace an element that doesn't appear in high contrast themes, consider these alternatives.

- Set the element text to a descriptive string.
- Set the title attribute of the element so the element has a tooltip when the mouse hovers.
- Use the CSS text-overflow attribute to provide ellipsis that prevent text from wrapping.

Use an HTML template for dynamic lists

The HTML template pattern can simplify showing a list of similarly-styled content generated at runtime in an app wing.

This example shows a list of names and telephone numbers in static HTML.

```
<div id='contacts'>
  <div class='contact'>
    <div class='contactField contactName'>Don Hall</div>
    <div class='contactField contactPhone'>425-555-1212</div>
  </div>
  <div class='contact'>
    <div class='contactField contactName'>Guido Pica</div>
    <div class='contactField contactPhone'>425-555-1212</div>
  </div>
  <div class='contact'>
    <div class='contactField contactName'>Mor Hezi</div>
    <div class='contactField contactPhone'>425-555-1212</div>
  </div>
</div>
```

This CSS defines the visual style of the static HTML.

```
#contacts
{
    width: 542px;
    height: 308px;
    overflow-x: hidden;
    overflow-y: auto;
}
.contact
{
    position: relative;
    width: 100%;
    height: 15px;
    background-color: whitesmoke;
    border-bottom: 2px solid white;
}
.contactField
{
    position: absolute;
    display: inline-block;
    overflow: hidden;
}
.contactName
{
    left: 0;
    top: 0;
}
.contactPhone
{
    top: 0;
    right: 0;
    width: 100px;
    background-color: gainsboro;
}
```

This new HTML and CSS makes this list dynamic.

```
<div id='contacts'>
  <div class='contact contactTemplate'>
    <div class='contactField contactName'></div>
    <div class='contactField contactPhone'></div>
  </div>
</div>
```

New CSS class:

```
.contactTemplate
{
    display: hidden;
}
```

This JavaScript code populates the table at runtime.

```
var contactData = [
  ["Don Hall", "425-555-1212"],
  ["Guido Pica", "425-555-1212"],
  ["Mor Hezi", "425-555-1212"]];

for (var i = 0; i < contactData.length; i++)
{
  $(".contactTemplate")
    .clone()
    .removeClass("contactTemplate")
    .find(".contactName")
      .text(contactData[i][0])
      .end()
    .find(".contactPhone")
      .text(contactData[i][1])
      .end()
    .appendTo("#contacts");
}
```

Load remote JavaScript at runtime

Most apps ship JavaScript files and include them using a <script> tag in HTML. You can also load JavaScript that resides at a remote web site at runtime. This code loads a JavaScript code file from a remote web site asynchronously and calls a method when the load completes.

```
$.ajax({
  url: gAppSettings['mapscontrolurl'],
  dataType: 'script',
  async: true,
  cache: true,
  timeout: 120000,
  success: function (data) { instance._mapsControlScriptsLoaded(); },
  error: function (data) { instance._mapsUnhandledError(); }
});
```

Handle and report errors

Error handling in a Bing Bar app can be complicated. Errors may be caused by user interaction, such as an incorrect password. External factors like network unavailability may cause errors without user interaction. Errors may occur in service code running in the background. And program errors such as exceptions may occur.

For errors caused by user interaction, provide a mechanism in the app wing for alerting users of error states. Here are a few examples of error reporting in a wing.

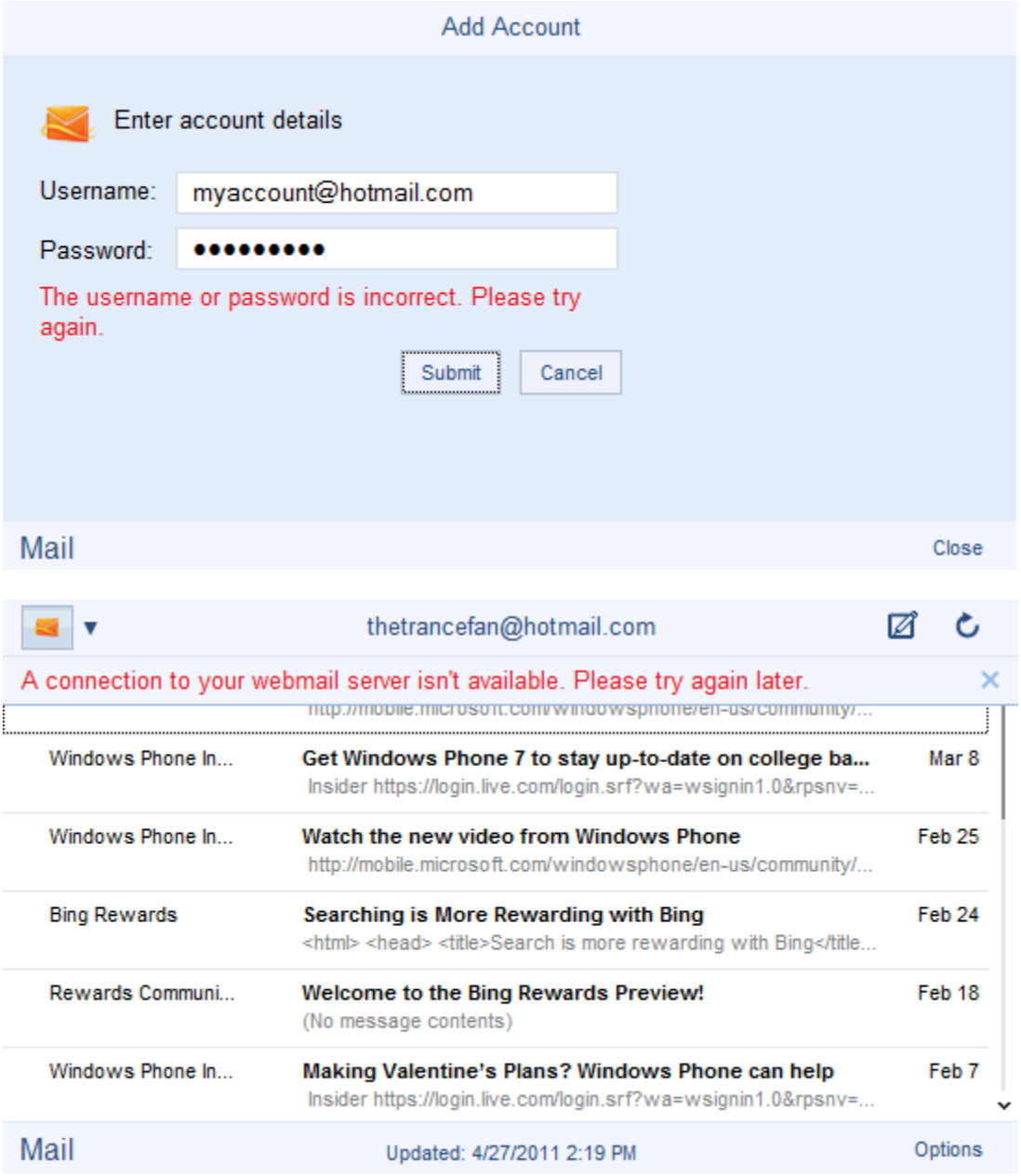


Figure: Two error UI presentations

Errors caused by external factors may occur in wing or service code. Apps can choose to handle these errors and attempt some fallback behavior, or report the error to the user. To notify the user of errors that occur in a service, create a mechanism to queue and forward the error information to the wing UI the next time the user opens the app wing.

Be proactive when handling program errors. For errors that you can anticipate in advance, including file access or network errors, wrap code in try-catch blocks and react with grace. Some API descriptions in *Bing Bar 7 API Reference* specify exception-handling requirements. For example, methods in the bingclient.io namespace can throw exceptions from any method.

Debugging with the debug console

You can add code in shared\js\debugconsole\debugconsole.js to wing code to add a debug console to your wing to debug wing UI, service, and page tasks. Using the debug console, you can log debugging

text, inspect your app's state, call methods on your app and service, run unit tests, and evaluate JavaScript expressions.

To enable the debug console in the Linkomatic sample, remove the HTML comment around this line in linkomatic.html.

```
<!--<script type="text/javascript" src="debugconsole/debugconsole.js"></script>-->
```

Copy debug console directory to your app source

To add the debug console to your app, follow these steps:

4. Copy the shared\js\debugconsole directory and contents to your app source, at the same level as your appmanifest.xml file.
5. Add this line to your app's wing HTML file:

```
<script type="text/javascript" src="debugconsole/debugconsole.js"></script>
```

Remove debug console before shipping

When preparing your app to ship, confirm there are no references to the gConsole variable in your app source, and remove the <script> line from your wing HTML file.

Open and use debug console

The debugconsole.js file adds a button in the upper-right corner of your app wing.

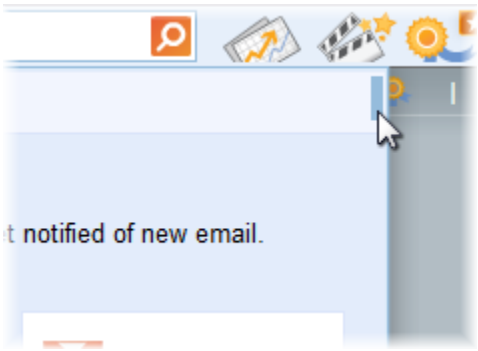


Figure: Button on wing that opens debug console

When you press the button, the debug console appears as a window over your wing.

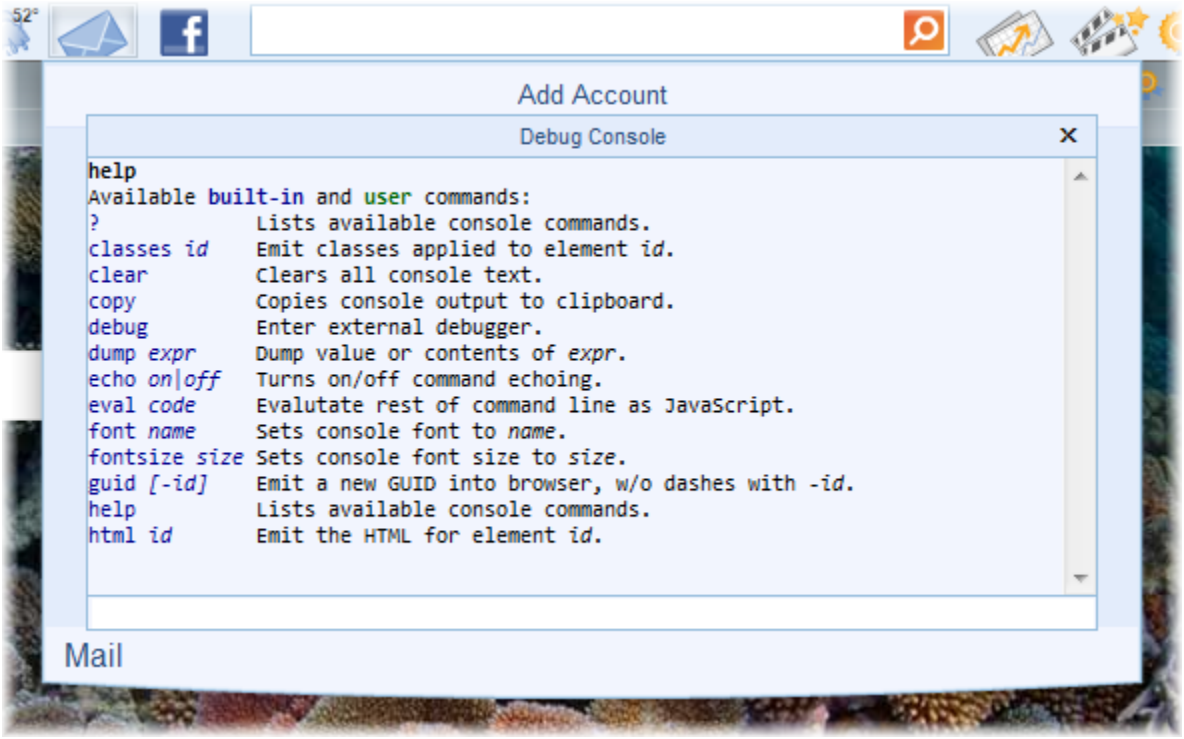


Figure: Debug console after entering ***help***

The console window shows a text-based output log of command results, with a text entry area at the bottom where you can type commands. To close the debug console, click its close box in the upper-right corner.

Built-in Commands

The debug console provides these built-in commands.

Command	**Description**
classes *id*	Lists the CSS commands that apply to the *id* element.
Clear	Clears the debug console screen.
Copy	Copies text on the debug console screen to the clipboard. **Note:** Only available when running in WingApp.
debug	Breaks into the Visual Studio debugger. **Notes:** Debugging must be enabled; see *Enable JavaScript debugging in Internet Explorer*. Wing stay-open mode helps debug wing code; see *Force the wing to stay open*.
dump *expr*	Shows a variable or object.
echo *on*|*off*	Turns command echoing on or off. Default: on.
eval *code*	Evaluate JavaScript. The debug console evaluates JavaScript entered in the text field. This command lets you evaluate more complex JavaScript.
font *name*	Set debug console font. Default: Consolas.
fontsize *size*	Set debug console font size.
guid [-id]	(v7.1 and up) Places a new GUID in the browser. When -id is included, the GUID contains no dashes.
help or ?	Lists commands with usage descriptions.
html *id*	Shows unrendered HTML of the *id* element.

The dump command shows a variable value

The **dump** command dumps a text representation of a variable or object. This is very useful for inspecting the state of your app. If gApp stores the app object, **dump gApp** shows the app object contents. To inspect members of the app object, type the object and member.

Example:

```
dump gApp._myMember
```

The html command shows raw HTML

The **html** command shows the unrendered HTML of the *id* element. If you use jQuery to get HTML by entering $('#myId')[0].outerHTML, the debug console shows *rendered* content of the myId element, as it would appear in a browser. The **html** command shows the raw HTML with HTML tags visible.

Debug console API

When an app includes debugconsole.js, app code can use gConsole, a global object, to call these methods during development.

gConsole methods to control the console window

Method	Description
showConsole()	Shows the debug console window.
hideConsole()	Hides the debug console window.
showUsage(cmd)	Shows usage details for a command. The command should be provided with no parameters. You might call this function from with your own custom commands to display your usage information.

This code catches all exceptions within the onReady method. If an exception occurs, the code opens the debug console and logs information about the exception. This provides more information during development rather than showing the sad robot error.

Note: Only call gConsole methods during app development. Never ship references to gConsole in production code.

```
MyApp.prototype.onReady()
{
  try
  {    ...
  }
  catch(e)
  {
    // FIXME - for development only - remove before ship
    if (gConsole)
    {
      gConsole.showConsole();
      gConsole.writeLine("MyApp.onReady: unhandled exception - " + e.message);
    }
  }
}
```

gConsole methods to log to the console window or browser

These functions allow your wing UI to write debugging output to the DebugConsole window or current browser tab.

Method	Description
write(str)	Writes a string to the debug console.

writeLine(str)	Writes a string to the debug console followed by a new line.
writeErrorLine(str)	Writes a string to the debug console followed by a new line. The string is prefixed by ERROR: in red text.
clear()	Clears the DebugConsole log. Equivalent to the **clear** command.
browserWrite(str)	(v7.1 and up) Appends a string to the content in the <body> tag of the current browser tab.
browserWriteLine(str)	(v7.1 and up) Appends a string to the content in the <body> tag of the current browser tab, followed by a tag.
browserClear()	(v7.1 and up) Clears the current browser tab by navigating to about:blank.

Notes:

- Strings sent with write() and browserWrite() are not terminated with a new line, so multiple calls append text to the existing line.
- The debug console renders HTML tags in strings. For example, this text will appear in **bold**.
- Existing HTML and CSS formatting rules apply to strings appended to text in the existing <body> tag of the browser.

gConsole methods to measure intervals of time

The debug console API contains a single timer you can use to measure the duration of service calls or UI events. The timer resolution is close to one millisecond.

Method	**Description**
startTiming()	Starts the debug console timer.
intermediateTiming(description)	Shows the timer state with a description string. This method does not stop the timer.
endTiming()	Stops the debug console timer.
getLastTiming()	Shows the total time between startTiming() and entTiming(), in milliseconds.

You can add timing calls to your wing UI to time, for example, how long between the time your app object is constructed and the end of your onReady() function. Or you could put timer calls around a call to your service to see how much time you spend waiting for the service to return.

gConsole methods to call or add a debug console command

Method	**Description**
executeCommand(cmd)	Calls any debug console command from your code.
addCommand(cmdName, usage, help, cookie, commandFunction)	Add a command to the debug console.

This code causes the contents of the gApp object to show in the debug console.

```
gConsole.executeCommand("dump gApp");
```

How to use the addCommand method

The gConsole.addCommand() method adds a custom command to the debug console. This table describes parameters of the addCommand method.

Parameter	Description
cmdName	New command name.
usage	Usage description. Can include HTML.
help	Details about what the command does. Can include HTML.
cookie	Value to pass to command function. Often a reference to **this**, the wing UI app object.
commandFunction	Reference to static JavaScription function to call when command is invoked.

The function specified in commandFunction has this signature.

```
MyClass._myCmd = function(console, cookie, commandLine)
```

This table describes the parameters to the custom command function.

Parameter	Description
console	Reference to gConsole, the debug console object. Use this object to call writeLine() or other gConsole methods.
cookie	Value you passed to addCommand().
commandLine	Complete command line used to invoke your custom command.

Example:

```
gConsole.addCommand("findclass", "findclass <em>className</em>",
  "Finds elements with class <em>className</em>.", this,
  Simple._cmdFindClass);
```

Here's the implementation of the findclass command.

```
Simple._cmdFindClass = function (console, cookie, commandLine)
{
    var words = commandLine.split(" ", 2);
    if (words[1])
    {
        console.writeLine("Elements containing class <em>" + words[1] + "</em>:");
        var match = $("." + words[1]).each(function (index)
        {
            var id = this.id ? this.id : "<none>";
            console.writeLine("  &lt;" + this.tagName + " id='" + id + "'&gt;");
        });
        console.writeLine("Total elements found: " + match.length);
    }
    else
    {
        console.showUsage("findclass");
    }
}
```

Here's example results from the findclass command.

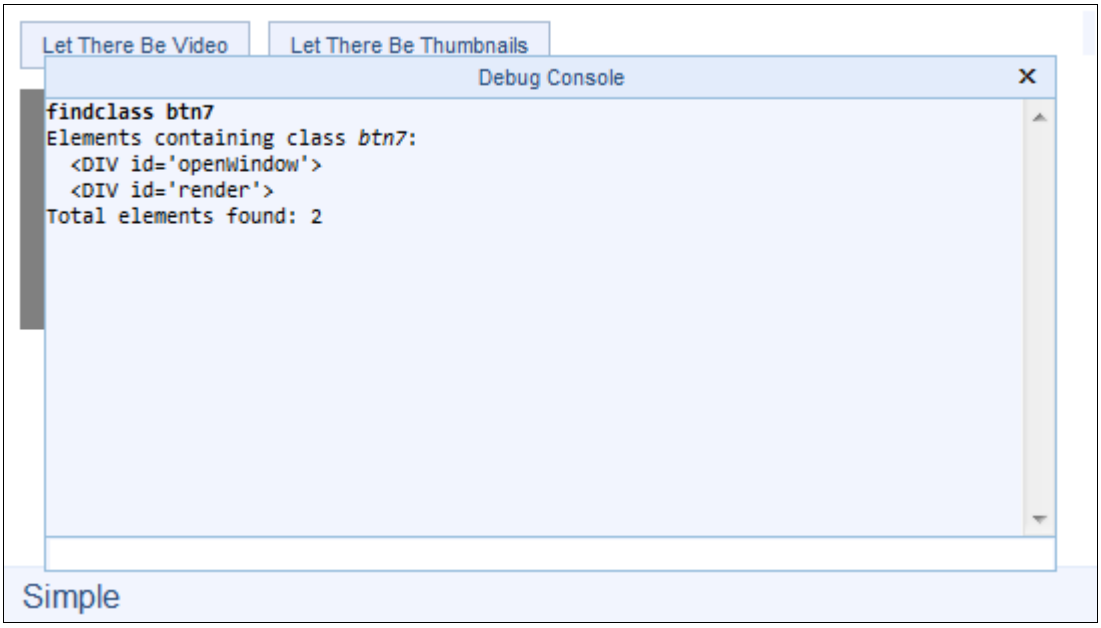


Figure: Results of custom findclass command

Cool tricks using the debug console

The debug console shows debugging log information. While it does that well, its real value comes from its ability to run arbitrary Javascript while your wing UI is running. Here are some handy tricks that leverage this capability. When your entry is not a console command, the debug console passes your console entries to the JavaScript eval() function and shows the result.

These examples assume a global variable called gApp contains your app instance, that your app has a reference to its service at gApp._service, and that your service contains a page task in a _pageTask member.

Attach a Debugger

To quickly attach a debugger to your app, type the **debug** command into the debug console. If your browser is configured to allow JavaScript debugging, you can approve the Visual Studio Just-In-Time debugging dialog, and set breakpoints in your app using Visual Studio.

Note: Typically, activation of any other window (including Visual Studio) will cause the wing to close, which destroys the wing UI. To prevent the wing from closing during debugging, see *Force the wing to stay open*.

Call your app methods

While your wing UI is under development, you can call member functions of your app directly. This can help when there is no user interface in place. To call a myMemberFunction() method in your app object, type gApp.myMemberFunction() in the debug console.

Confirm your service is running

To see if your service has loaded, type gApp._service in the debug console. If the value is undefined, then the service didn't start, and probably contains a syntax error or uncaught exception.

Call service methods

To call methods of your service, use the reference stored in the app object. Enter gApp._service.myServiceMethod() in the debug console.

Add a break-to-debugger function in a page task or service

To launch a debugger into a page task or service code, add a function to the page task or service code that breaks into the debugger.

```
SimpleService.prototype.debugService = function ()
{
    debugger;
}
```

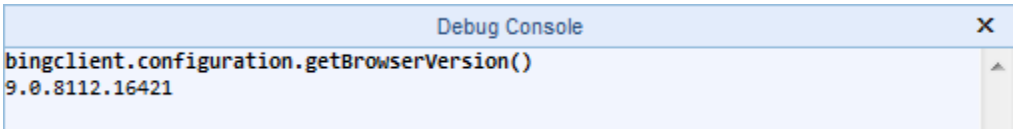
From your wing UI or from the debug console, you can enter gApp._service.debugService() to attach the debugger to the service. Then you can set breakpoints in your service.

Call page task methods

To call a page task method that takes a sessionId as a parameter, enter something like gApp._service._pageTask.findInputTextElems(bingclient.context.browserSessionId) in the debug console.

Call Bing Bar APIs

You can run bingclient.* methods directly from the debug console. For example:



Modify HTML and CSS at runtime

Any jQuery that can run in wing code can also run in the debug console at runtime. To test HTML or CSS changes, try changes in the debug console. Enter this code in the debug console to modify CSS of the running wing.

```
$('#mydiv').css('margin-left', '2px')
```

Implement unit tests

Your app might have methods you want to test with unit tests. You can use the gConsole.addCommand() method to add a custom command that runs your unit tests. The tests can output their success, failure, or status to the debug console log.

More debugging techniques

This section describes more debugging techniques for Bing Bar apps.

Force the wing to stay open

The debug build of Bing Bar includes a feature that can keep the wing open until the user moves the underlying Internet Explorer window. This feature makes it easier to debug wing apps where activating Visual Studio would normally make the wing go away, and for test cases where automation wants to keep the wing open. This feature is only available in debug builds.

To force the wing to stay open, double-click registry\ Stay-open_ON.reg in your SDK files.

Note: This feature lets users create scenarios that cannot occur in the release build of Bing Bar. If you find a bug while using stay-open mode, verify that the bug also occurs in normal mode before reporting.

Use TraceView

You can use TraceView to examine real-time and logged sessions of traced code activity. TraceView is available in the Windows Driver Kit. To download the Windows Driver Kit, open this web address.

http://msdn.microsoft.com/en-us/windows/hardware/gg487463.aspx

To monitor tracing output, follow these steps:

1. Confirm you are running the debug build of Bing Bar shipped in this SDK.
2. Run the registry\Tracing_ON.reg file to enable tracing.
3. As administrator, run traceview.exe.
4. In TraceView, click **File**, then click **Create New Log Session**.
5. Click **Add Provider**.
6. In the Provider Control GUID Setup dialog, choose **Manually Entered Control GUID** and paste 3133F9EC-A554-497B-96D9-A43FEA2F1E12 in the text field.
7. Click **OK**.
8. Choose **Select TMF Files** and click **OK**.
9. Click **Add**.
10. Navigate to the tracing folder in the SDK files.
11. Select all files and click **Open**.
12. Click **Done**.
13. Click **Next >**.
14. Click **Finish**.

Now trace information will appear in TraceView as you use Bing Bar apps.

To log your app activity in TraceView, use trace* methods in the bingclient.debugging.tracing namespace.

Add a debugger; statement to script to break into Visual Studio

Internet Explorer can open Visual Studio when JavaScript code reaches a debugger statement. This code sample opens Visual Studio when an exception occurs.

```
try
{
    ...
```

```
}
catch (e)
{
  debugger;
}
```

Note: You must configure Internet Explorer to allow just-in-time debugging. See *Code a Bing Bar app in-place* for details.

Diagnose sad robot error on first press of app button

If your app shows a sad robot error when you press the app button, there is probably a syntax error in JavaScript. In a cmd window, use cscript to examine JavaScript source for syntax errors. To step through wing initialization in Visual Studio, add a debugger statement between script inclusions in the wing HTML file.

```
<script type="text/javascript" src="js/linkomatic.js"></script>
<script>debugger;</script>
<script type="text/javascript" src="js/anothermodule.js"></script>
```

If your code runs all the way to your app's script file, try adding a debugger statement as the first line of your app's script file, then stepping through the code line-by-line.

Diagnose sad robot error after first press of app button

If your wing opens, but later a sad-robot appears, you have an uncaught exception. To investigate this problem, wrap the onReady() method call in a try-catch block.

Diagnose a service that won't start

The most likely causes of a service that won't start are an exception early in code execution or a syntax error in service code. To diagnose, use the techniques described in *Diagnose sad robot error on first - press of app button*. Also try adding a **debugger;** statement at the start of service source code, and stepping through until an exception occurs.

Diagnose a page task that doesn't work

Page tasks can fail for the same reasons as wing code and service code, including syntax errors and uncaught exceptions. An uncaught exception in a page task is consumed by the runtime without additional information, so this kind of error won't cause a sad robot to appear. Use cscript from a cmd box to examine page task source for syntax errors. If the syntax is ok, add a **debugger;** statement at the beginning of the page task and step through the page task code until an exception occurs.

Debug a service by running service code as wing code

To debug interactions between wing and service code, you can relocate service code to the wing and run the debugger. This service code registers a service when not running in a wing. When running in a wing, this code creates an object and stores it in a global variable.

```
var gService = new MyService();
if (bingclient.flyout == undefined)
{
  // Register service if code is running as service (i.e. not in the wing)
  bingclient.services.create("MyCompany.BingBar.MyService", gService);
}
```

Wing code can refer to the local or registered service object.

```
if (gService)
{
  this._service = gService;
}
else
{
  this._service = bingclient.services.get("MyCompany.BingBar.MyService");
}
```

View ReadyState in ActiveVersion.xml

Bing Bar creates and updates an ActiveVersion.xml file for every app. The ReadyState attribute in this file can provide a clue about app state. For example, if the app button does nothing, check the ReadyState attribute. This table describes values of the ReadyState attribute.

Value	Description
unknown	App state has not been determined.
ready	App is ready to be displayed.
downloading	App is downloading or installing.
incompatible	App is incompatible with this version of Bing Bar.
blocked	App has been blocked and cannot run
downloadError	App failed to download or install.
loadError	App failed to load.

Test alerts

Bing Bar monitors alert frequency, and may throttle alerts to avoid overwhelming the user. Alerts only appear when the browser has focus, and when Bing Bar observes indications that the user is actively using the browser.

These mechanisms complicate testing alerts Consider these testing techniques.

- Add a simple method to service code that calls bingclient.ui.notifications.enqueue for each type of alert your app produces.
- Use a timer to queue the type of alert you're testing every five or ten seconds.
- Close all browser windows and open a new one to start testing alerts. This resets the state of the throttling mechanisms.
- Interact with the browser by creating tabs and clicking links to create the conditions for alerts to appear.
- In Bing Bar version 7.1 and later, change your alert type to instantToast (instead of generalToast) to fire the alert immediately.

Use mocks to simulate features

Files in the shared\mocks folder provide simulated services for testing and development scenarios. Many mocks cannot perform the tasks they're simulating, and only emit debug output so the developer knows the calls were made. Examine mock source to see if it can be modified to suit your testing or development scenario. Mocks emit debugging output using the gWingApp object. You can modify mock code to emit debug output to the debug console's gConsole object.

Unit tests

Visual Studio provides a test harness for JavaScript code. Your unit tests should:

- Call all public APIs with full coverage of parameter variations, including null values, empty strings, long strings, and edge case values.
- Examine API results for both successful and unsuccessful call outcomes.
- Cover at least 80% of the app code blocks.

AppManifest.xml File

The AppManifest.xml file specifies information about a Bing Bar app, including information about the app button, app services, and app wings. One AppManifest.xml file resides in each version directory of the app. An app will include localized versions of this file.

AppManifest attributes

Attribute	Description
Name	App name.
Publisher	Publisher of app.
Copyright	Optional. Copyright information.
Description	Description of app.
Version	App version in **major#.minor#.build#** format . This value is not used.
MinRuntimeVersion	Minimum version of Bing Bar where this app can run, in **major#.minor#** format.
MaxRuntimeVersion	Maximum version of Bing Bar where this app can run, in **major#.minor#** format.
Locale	Language-market group of this app.
AppType	Optional, default **standard**. App type. Set to **noui** for an app with no wing or button that does not appear in the settings user interface.
Id	Unique identifier of app. See *Create an app GUID*.
TargetFolder	Prefix of the app folder name on the user's computer. This value can contain letters, spaces, digits, dashes and underscores. The folder name combines this value with the app ID.

Button attributes

Attribute	Description
Icon	Path of image file to use as initial app button.
Icon125	Optional. Path of image file to use as initial app button when display is set to 125% in Control Panel.
Icon150	Optional. Path of image file to use as initial app button when display is set to 150% in Control Panel.
Tooltip	Optional. Text of initial app button infotip.

AppServices sub-element

Optional. Contains one or more <Script> nodes. Each <Script> node specifies a path to JavaScript code that contains a service.

AppUI element and attributes

Optional. The <AppUI> node specifies the size and source file for a wing.

Attribute	Description
Html	Relative path to the wing HTML file. If the file is in the same directory as the appmanifest.xml file, just the bare filename appears here.
Width	Wing width in pixels. This value should be 546 unless you've received approval for an exception.
Height	Wing height in pixels. The standard value is 308, maximum 440.

This appmanifest.xml file specifies a wing in the <AppUI> element, and includes a service in the <Script> element.

```
<?xml version="1.0" encoding="utf-8"?>
<AppManifest Name="Linkomatic"
             Publisher="Microsoft Corporation"
             Copyright="© 2011"
             Description="Collect the links from a web page."
             Version="7.0.444"
             MinRuntimeVersion="7.0"
             MaxRuntimeVersion="7.0"
             Locale="EN-US"
             Id="5EF9C381866B49889481A0429E78B57E"
             >

  <Button Icon="images\linkomatic.png" Tooltip="Your links. We've got 'em." />

  <AppServices>
    <Script>js\service.js</Script>
  </AppServices>

  <AppUI Html="linkomatic.html" Width="546" Height="308" />

</AppManifest>
```