Dashboard Reference

General

2009-02-04

Contents

Tables

Introduction to Dashboard Reference

This document contains reference material for creating Dashboard widgets. It documents the JavaScript objects available to widgets and the interfaces used to configure and extend the behavior of a widget. It also covers the Objective-C interface used to create widget plug-ins.

Who Should Read This Document?

Dashboard Reference is for any widget creator looking for detailed information about the various interfaces available to widgets.

Organization of This Document

This document contains the following chapters:

- "Widget Object" (page 9) describes the methods and properties of the Widget object—a JavaScript object you use to manage widget-specific behavior.
- "Regions" (page 15) describes the use of regions within your widget. Regions allow you to specify areas for specific uses.
- "Dashboard Info.plist Keys" (page 19) describes the custom and expected keys to include in your widget's information property list file.
- "Widget Plug-in Interface" (page 21) describes the interface used to create custom plug-ins for your widget.

See Also

For an introduction to Dashboard widgets, see "Widget Basics" in *Dashboard Programming Topics*. Read the other articles in *Dashboard Programming Topics* for information on the various technologies available to widget developers. To learn how to use Dashcode to create widgets, see *Dashcode User Guide*.

For more information on the HTML, CSS, and JavaScript capabilities found in WebKit, the technology behind Dashboard widgets, consult:

- *Safari HTML Reference*
- *Safari CSS Reference*
- *WebKit DOM Reference*

Widget Object

The Widget object is a JavaScript object that provides Dashboard-specific extensions. When your widget is loaded, Dashboard automatically creates an instance of this object for use in the JavaScript code of your widget. The name of this instance is `widget`.

Note: The properties and methods found in this chapter are those supported by Apple for developing widgets. Any widget properties and methods not found here are subject to change without notice.

Properties

The following sections describe the properties of the Widget object.

identifier

Contains a unique identifier for this instance of the widget.

```
widget.identifier
```

This read-only property contains a string value that is unique among all of the instances of a single widget. This value is assigned by Dashboard and persists between instantiations of each widget instance.

Note: The `identifier` property should not be confused with the `CFBundleIdentifier` property list key, which is described in "Dashboard Info.plist Keys" (page 19). The value of the `CFBundleIdentifier` property list key is used to differentiate between widgets from different projects, not between different instances of the same widget.

ondragstart

Contains the event handler to be called upon the start of a widget drag.

```
widget.ondragstart
```

Assign a function to this property if you want to be notified when your widget has begun a drag. You use this function to change your widget's user interface while it is being dragged. Your function declaration should look like the following:

```
function MyDragStartHandler() { ... }
```

ondragend

Contains the event handler to be called upon the finish of a widget drag.

```
widget.ondragend
```

Assign a function to this property if you want to be notified when your widget has ended a drag. You use this function to change your widget's user interface after it has been dragged. Your function declaration should look like the following:

```
function MyDragEndHandler() { ... }
```

onhide

Contains the event handler to be called when the Dashboard layer is hidden.

```
widget.onhide
```

Assign a function to this property if you want to be notified when your widget is hidden. You use this function to deactivate your widget and put it into a quiescent state. Your function declaration should look like the following:

```
function MyHiddenHandler() { ... }
```

onremove

Contains the function to be called when your widget is removed from the Dashboard layer.

```
widget.onremove
```

Assign a function to this property if you want to be notified when your widget is removed from the Dashboard layer. Upon receiving this event, your widget should perform any necessary cleanup operations, such as save its preferences, remove cache files, and release any resources it currently holds. Your function declaration should look like the following:

```
function MyRemoveHandler() { ... }
```

onshow

Contains the function to be called when the Dashboard layer is shown.

```
widget.onshow
```

Assign a function to this property if you want to be notified when your widget is shown. You use this function to activate your widget and begin processing data again after being quiescent. Your function declaration should look like the following:

```
function MyShowHandler() { ... }
```

Methods

The following sections describe the methods of the Widget object.

openApplication

Launches the application with the specified bundle identifier.

```
widget.openApplication(bundleId)
```

Use this method to launch the application indicated by `bundleId` on the target system. Calling this method dismisses the Dashboard layer.

openURL

Opens the specified URL in the user's preferred browser.

```
widget.openURL(url)
```

This method opens the specified URL and dismisses the Dashboard layer. This method does not permit the opening of URLs that use the `file:` scheme unless the `AllowFileAccessOutsideOfWidget` key is set in the widget's information property list file.

preferenceForKey

Returns the preference associated with the specified key.

```
widget.preferenceForKey(key)
```

Use this method to retrieve a preference value previously stored with a call to `setPreferenceForKey`. The method returns a string with the contents of the preference, or `undefined` if no such preference exists.

prepareForTransition

Notifies Dashboard that you are about to perform a transition to or from its reverse side.

```
widget.prepareForTransition(transition)
```

This method prepares your widget for either showing or hiding its reverse side.

Passing the string "ToBack" for `transition` disables screen updates within your widget's user interface so that you can prepare it for displaying your widget's reverse side. Passing the string "ToFront" for `transition` freezes your widget's user interface so that you can prepare it for displaying the main contents again. When your HTML layers are ready, call `performTransition` to display them.

performTransition

Runs an animation to toggle between your widget's reverse and contents.

```
widget.performTransition()
```

You call this method after first calling `prepareForTransition`, which indicates whether you are displaying your widget's reverse side or contents. When you call `performTransition`, Dashboard begins an animation that makes the widget appear to flip over and display the new content.

Prior to calling this method, you should also adjust the style sheet properties of your HTML to reflect the change in what is about to be displayed. For example, before calling this method to show your reverse side, you should show the HTML elements associated with your reverse side and hide those elements associated with your widget's contents.

setCloseBoxOffset

Changes the location of the widget close box.

```
widget.setCloseBoxOffset(x, y)
```

Use this method to move your widget's close box. This method centers the close box *`x`* pixels from the left edge of the widget and `y` pixels down from the top of the widget. Only values between `0` and `100` are allowed for `x` and `y`.

setPreferenceForKey

Associates a preference with the given key.

```
widget.setPreferenceForKey(preference, key)
```

The *`preference`* and *`key`* parameters contain strings representing the preference you want to store and the key with which you want to associate it. Specifying `null` for the *`preference`* parameter removes the specified key from the preferences.

Preferences saved using `setPreferenceForKey` are saved as clear text and therefore are not recommended for saving passwords or other sensitive information.

system

Executes a command-line utility.

```
widget.system(command, endHandler)
```

The *`command`* parameter is a string that specifies a command utility to be executed. It should specify a full or relative path to the command-line utility and include any arguments. For example:

```
widget.system("/usr/bin/id -un", null);
```

The *endHandler* parameter specifies a handler to be called when the command has finished executing. If `NULL`, the entire method is run synchronously, meaning that all execution inside the widget halts until the command is finished. When running synchronously, these options are available:

Table 1-1 `widget.system` properties available during synchronous usage

Property	Definition	Usage
`outputString`	The output of the command, as placed on `stdout`.	`var output = widget.system("/usr/bin/id -un", null).outputString;`
`errorString`	The output of the command, as placed on `stderr`.	`var error = widget.system("/usr/bin/id -un", null).errorString;`
`status`	The exit status of the command.	`var status = widget.system("/usr/bin/d -un", null).status;`

If *endHandler* is specified, the command is run asynchronously, meaning that the command runs concurrently and the handler is called when execution is finished. When run asynchronously, `widget.system` returns an object that can be saved and used to perform other operations upon the command:

Table 1-2 `widget.system` properties and methods available during asynchronous usage

Option	Purpose	Description
`command.outputString`	Property	The current string written to `stdout` (standard output) by the command.
`command.errorString`	Property	The current string written to `stderr` (standard error output) by the command.
`command.status`	Property	The command's exit status, as defined by the command.
`command.onreadoutput`	Event Handler	A function called whenever the command writes to `stdout`. The handler must accept a single argument; when called, the argument contains the current string placed on `stdout`.
`command.onreaderror`	Event Handler	A function called whenever the command writes to `stderr`. The handler must accept a single argument; when called, the argument contains the current string placed on `stderr`.
`command.cancel()`	Method	Cancels the execution of the command.
`command.write(string)`	Method	Writes a string to `stdin` (standard input).
`command.close()`	Method	Closes `stdin` (`EOF`).

Note: To use `widget.system`, you need to set the `AllowSystem` key in your `Info.plist`. For more information, see "Dashboard Info.plist Keys" (page 19).

Regions

Regions are CSS properties that you use to set bounds. Currently, Dashboard specifies only one type of region, a control region. Specifiying a control region means that if a user attempts to drag a widget from within a specified region, the drag will not occur and the widget will not move.

Regions come in two shapes: rectangles and circles. Any combination of these shapes is allowed, letting you create complex control regions for use with odd shapes.

Once you have defined a region within a style element, you need to wrap an element within your markup with that style. For instance, a control region definition may look like this:

```
.control-circle-example {
    ...
    -apple-dashboard-region: dashboard-region(control circle 5px 5px 5px 5px);
    ...
}
```

Now that you've defined a style, you need to apply it to an element:

```
<div class="control-circle-example"><img src="foo.png"></div>
```

Properties

The following property is defined for use when specifying regions within a widget.

-apple-dashboard-region

Specifies the property to be defined.

```
-apple-dashboard-region:
```

This property tells Dashboard that you are about to specify a region. Without any parameters, this property does nothing. As parameters to this method, you need to specify regions using the *`dashboard-region`* parameter.

Parameters

The following parameters are defined for use when specifying regions within a widget.

dashboard-region

Specifies the type and bounds of a region.

```
dashboard-region(label, geometry-type)
dashboard-region(label, geometry-type, offset-top, offset-right, offset-bottom,
 offset-left)
```

This parameter function is used in conjunction with the `-apple-dashboard-region` property. It specifies a region's bounds and type, in function form. Table 2-1 (page 16) shows the values expected by the *`dashboard-region`* parameter.

Table 2-1 `dashboard-region()` parameters

Parameter	Description
`label`	Required; specifies the type of region being defined; `control` is the only used value.
`geometry-type`	Required; specifies the shape of the region, either `circle` or `rectangle`.
`offset-top`	Optional; specifies the offset from the top of the wrapped area from where the defined region should begin, in pixels. Negative values not allowed.
`offset-right`	Optional; specifies the offset from the right of the wrapped area from where the defined region should begin, in pixels. Negative values not allowed.
`offset-bottom`	Optional; specifies the offset from the bottom of the wrapped area from where the defined region should begin, in pixels. Negative values not allowed.
`offset-left`	Optional; specifies the offset from the left of the wrapped area from where the defined region should begin, in pixels. Negative values not allowed.

If you specify `circle` for the `geometry-type` parameter, the control region created is centered in between the specified offsets (or the edges of the region, if no offsets are provided). Of the circle region's width and height (which ideally should be equal), the resulting control region's diameter is the smaller value.

When using the offset parameters, you either provide values for all four offsets or none of them. Note that if you do not specify values for the offset parameters, a default value of `0` is used for each of them.

You can chain multiple *`dashboard-region`* parameters together in one `apple-dashboard-region` property declaration, allowing you to create complex-shaped regions:

```
.equals-button-example {
    ...
    -apple-dashboard-region:
        dashboard-region(control circle 0px 0px 80px 0px)
        dashboard-region(control rectangle 10px 0px 10px 0px)
        dashboard-region(control circle 80px 0px 0px 0px);
    ...
}
```

Some elements have control regions assigned to them by default:

- `button`

- `input`
- `select`
- `textarea`

Whenever you use one of these elements you do not need to manually specify a control region them. The region specified on these elements extend to their edges:

```
 button, input, select, textarea {
    -apple-dashboard-region:dashboard-region(control rectangle);
}
```

none

Removes any regions on an element.

```
none
```

Setting the `-apple-dashboard-region` property to `none` removes any region applied to an element.

Dashboard Info.plist Keys

Dashboard widgets provide information to the system and to Dashboard through the use of an information property list (`Info.plist`) file. The keys in a widget's information property list file identify the type of the bundle and the location of the widget's main HTML file

Table 3-1 lists the custom keys associated with all widgets. Dashboard uses these keys to configure the widget and prepare it for display. To learn more about using these keys in a widget, including a sample `Info.plist` file that you can base your own widget's `Info.plist` file off of, read Widget Basics in *Dashboard Tutorial*.

Table 3-1 Custom property list keys

Key	Type	Description
`AllowFileAccess-OutsideOfWidget`	Boolean	Optional; specify if your widget requires access to the file system outside of your widget. Access is limited by the user's permissions.
`AllowFullAccess`	Boolean	Optional; specify if your widget requires access to the file system, WebKit and standard browser plug-ins, Java applets, network resources, and command-line utilities.
`AllowInternetPlugins`	Boolean	Optional; specify if your widget requires access to WebKit and standard browser plug-ins, such as QuickTime.
`AllowJava`	Boolean	Optional; specify if your widget requires access to Java applets.
`AllowNetworkAccess`	Boolean	Optional; specify if your widget requires access to any resources that are not file-based, including those acquired through the network.
`AllowSystem`	Boolean	Optional; specify if your widget requires access to command-line utilities using the widget script object.
`BackwardsCompatible-ClassLookup`	Boolean	Optional; specify if your widget uses the Apple-provided JavaScript classes known as Apple Classes in a backward compatible way. See Introduction to the Apple Classes for more information.
`CloseBoxInsetX`	Number	Optional; the offset for the location of the widget close box on the x-axis. Positive values move toward the right. Must be between 0 and 100.
`CloseBoxInsetY`	Number	Optional; the offset for the location of the widget close box on the y-axis. Positive values move toward the bottom. Must be between 0 and 100.
`Fonts`	Array	Optional; contains an array of strings. Each string is the name of a font included within the widget bundle, located at its root.

Key	Type	Description
`Height`	Number	Optional; contains a number value indicating the height of the widget, measured in pixels.
`MainHTML`	String	Required; contains a string with the relative path to the widget's main HTML file. This file is the implementation file of the widget.
`Plugin`	String	Optional; contains a string with the name of a custom plug-in used by the widget. Plug-ins are located inside the widget bundle.
`Width`	Number	Optional; contains a number value indicating the width of the widget, measure in pixels. This key is optional.

In addition to the preceding keys, the following keys are required and should be included in your widget's information property list file:

- `CFBundleIdentifier`
- `CFBundleName`
- `CFBundleDisplayName`

You may also include other keys such as `CFBundleVersion` or other keys that provide information to entities such as the Finder. For detailed descriptions of how these keys are used, see Property List Key Reference in *Runtime Configuration Guidelines* in Mac OS X Documentation.

Widget Plug-in Interface

If you want to define a custom plug-in to use with a widget, the principal class of your plug-in must support the Widget Plug-in interface. This interface provides basic initialization support for your plug-in code along with access to the web view of your widget. You can also use this interface to register custom JavaScript objects for use by your widget code.

While the Widget Plug-in interface is required of all widget plug-ins, implementing the optional WebScripting protocol provides you with the ability to bridge data between your plug-in's native code and your widget's JavaScript code. Learn more about the WebScripting protocol by reading Using Objective-C From JavaScript and `WebScripting`.

Methods

The following sections describe the methods of the Widget Plug-in interface.

initWithWebView:

Default initializer for your plug-in.

```
- (id) initWithWebView:(WebView*)webView
```

Use this method to perform basic initialization of your widget's principal class. The `webView` parameter contains the view object used to display the widget contents. This method is called before your widget's HTML page is fully loaded.

If you need to do additional initialization after your plug-in is loaded, you should expose a method from your plug-in object to perform that initialization. You can then call that method from the onload event handler of your widget's HTML page. See "windowScriptObjectAvailable:" (page 21) for information on how to create a bridge between your Objective-C classes and JavaScript.

Implementation of this method is required.

windowScriptObjectAvailable:

Indicates that a scriptable object is now available.

```
- (void) windowScriptObjectAvailable:(WebScriptObject *) windowScriptObject
```

This method notifies you that the window scripting object is available for your use. You can use this method to expose your Objective-C classes as JavaScript objects so that they can be accessed by your widget code.

Note: The objects you expose to JavaScript using this technique may conform to the WebScripting interface of WebKit, which is available in Mac OS X version 10.4. WebKit uses the methods of this interface to allow you to control which methods you expose from your Objective-C class. For more information, see "Using Objective-C From JavaScript" in *WebKit DOM Programming Topics*.

When your plug-in receives the `windowScriptObjectAvailable:` message, call the `setValue:forKey:` method of *`windowScriptObject`* to associate your Objective-C object (the value) with the object name JavaScript clients should use.

The following example registers an instance of the MyScriptObject class:

```
- (void) windowScriptObjectAvailable:(WebScriptObject *) scriptObj
{
    MyScriptObject* myObj = [[MyScriptObject* alloc] init];

    [scriptObj setValue:myObj forKey:@"MyScriptObj"];
}
```

After you publish your object in this manner, you can refer to it in JavaScript code by the name you gave it. From the preceding example, if the object exposed a method called `finishInitialization`, you could call that method using the following JavaScript code:

```
function MyWebPageLoadHandler()
{
    if (MyScriptObj)
    {
        MyScriptObj.finishInitialization();
    }
...
}
```

Document Revision History

This table describes the changes to *Dashboard Reference*.

Date	Notes
2009-02-04	Made minor corrections.
2008-10-15	Fixed description of the Fonts Info.plist key; clarified definition of widget identifier.
2006-05-23	Revised descriptions on Info.plist keys.
2006-01-10	Added information about Apple Classes-related Info.plist keys and links to more documentation about the WebScripting protocol.
2005-07-07	Added links to Safari Reference documentation and clarified acceptable close box inset Info.plist values, close box offset parameter values, and saved preference security.
2005-05-20	Revised Info.plist key and Dashboard Region information.
2005-04-29	Revised explanation for the Widget object and added link for additional documentation for the Webscripting protocol .
	Updated for public release of Mac OS X v10.4. First public version.
2004-11-18	Includes revised Widget object, Info.plist, and Dashboard region reference.
	Updated Info.plist chapter and widget.onshow/onhide definitions.
2004-11-02	Updated Widget object reference. Relocated WebKit Canvas Extension chapter to Safari JavaScript Reference.
2004-10-04	Updated Canvas, Info.plist, and Widget object reference and added Regions chapter. Renamed document to *Dashboard Reference*.
2004-06-28	First version of *Dashboard Developer Reference*.