ADOBE® BRIDGE CS3

Br

JAVASCRIPT GUIDE

Adobe®

Contents

Welcome **6**
- About This Book 6
  - Who should read this book 6
  - What is in this book 7
  - Document conventions 7
    - Typographical conventions 7
  - Where to go for more information 7

1 Scripting Adobe Bridge **9**
- Scripting Overview 9
  - Documentation and sample code 9
  - Executing scripts for Adobe Bridge 10
- The Adobe Bridge Browser Window and Object Model 10
  - The Adobe Bridge browser window 11
  - Accessing the Adobe Bridge browser through scripts 12
- The Adobe Bridge Object Model 14
  - Basic node model 15
    - The application and documents 15
    - Thumbnails in documents 15
  - Scripting Adobe Bridge interactions 16
    - Application preferences 16
    - User-interaction events 16
  - Customizing the user interface 17
    - Customizing the browser window 17
    - Communicating with the user from a script 19
  - Extending browser behavior 21

2 Interacting with Adobe Bridge through Scripts **23**
- Accessing Thumbnails with Scripts 23
  - Thumbnails as node references 23
  - Using and accessing thumbnails 24
    - Accessing many thumbnails correctly 24
    - Ensuring valid thumbnail data 25
  - Metadata for thumbnails 25
  - Embedding metadata in a script as XML 26
- Event Handling in Adobe Bridge 27
  - Defining event handlers 27
  - Registering event handlers 28
- Communicating with Other Applications 28

3 Creating a User Interface for a Script **30**
- User Interface Options for Scripts 30
  - Navigation bars 31
  - Dialogs boxes 32
  - Content pane 32
  - Tabbed palettes 32
- Displaying ScriptUI in Adobe Bridge 32

Displaying ScriptUI dialogs ... 33
Displaying ScriptUI elements in a navigation bar ... 33
Displaying ScriptUI elements in a custom palette ... 33
Displaying HTML in Adobe Bridge ... 34
Defining callbacks for HTML scripts ... 34
Executing script functions defined on HTML UI pages ... 35
Displaying HTML in Adobe Bridge dialogs ... 36
Communicating with Adobe Bridge from dialog JavaScript ... 36
Using callbacks in an HTML dialog ... 37
Calling functions defined in an HTML dialog ... 37
Displaying HTML in a navigation bar ... 37
Calling functions defined in an HTML navigation bar ... 37
Displaying HTML in a custom palette ... 38
Passing Complex Values in Remote Calls ... 38
Scheduling Tasks from Callbacks ... 38

4 Customizing the Adobe Bridge Browser Window ... 40

Creating a Customized Tabbed Palette ... 40
Script-Defined Inspector Panels ... 41
Creating and displaying inspector panels ... 42
Specifying string values in an inspector panel ... 42
Panelette markup elements for dynamic text ... 43
Markup examples ... 43
Extending Adobe Bridge Menus ... 44

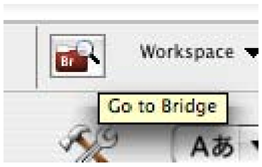
5 Extending Adobe Bridge Node-Handling Behavior ... 45

Creating Script-Defined Node-Handling Extensions ... 45
Registering your node-handling extension ... 46
Installing a node-handling extension ... 46
Installation structure ... 47
Extension workspaces ... 48
Shared startup scripts ... 48
Accessing the node-handling model and data ... 48
Defining an ExtensionHandler ... 49
Immediate handler operations ... 49
Long-running handler operations ... 49
Defining an ExtensionModel ... 50
Immediate model operations ... 51
Long-running model operations ... 53
Defining Node Data Sets ... 54
Managing the data cache ... 54
Core node data ... 55
Defining Long-Running Operations ... 56
Implementing an operation ... 56
Monitoring operation progress and status ... 58
Resolving conflicts ... 59
Defining Node Searches and Filters ... 61
Handling interactive node searches ... 61
Implementing getSearchDefinition() ... 62
Implementing getBridgeUriForSearch() ... 63
Adding filters ... 63
Implementing the getFilterCriteria() method ... 64

6 Porting Guide **66**
New Features in Adobe Bridge CS3 66
Node handling customization 66
Browser customization 66
Changes and Deprecations in Adobe Bridge CS3 67
Object model additions 67
Index **68**

Welcome

Welcome to the *Bridge JavaScript Guide*. This book describes how to use JavaScript to manipulate and extend Adobe® Bridge, a powerful file-browser available with many Adobe applications, which you can invoke from the **Go to Bridge** icon.



Adobe Bridge provides a highly customizable platform that can be extended using a rich JavaScript API, Flash® components, and HTML-based solutions. It includes the ExtendScript Toolkit, an interactive development environment for JavaScript.

About This Book

This document describes how to use the scripting API to extend and manipulate Adobe Bridge. Complete reference information for the JavaScript objects, properties, and functions defined by Adobe Bridge is provided in a companion document, the *Bridge JavaScript Reference*.

- Adobe provides the *ExtendScript Toolkit* an interactive development environment (IDE) for JavaScript, with all JavaScript-enabled Adobe applications. The ExtendScript Toolkit is documented in the *JavaScript Tools Guide*. This book also provides documentation for various utilities and tools that are part of ExtendScript, the Adobe extended implementation of JavaScript.
- The Bridge SDK, which contains this document, also contains a set of code samples ("Snippets"), that demonstrate how to use the features of the Bridge JavaScript API. This book refers to these samples by name for illustration of concepts and techniques. You can download the SDK from Adobe Developer Center, http://www.adobe.com/devnet/.

Who should read this book

This book is for developers who want to extend the capabilities of Adobe Bridge using JavaScript, call Bridge functionality from scripts, and use scripts to communicate between Bridge-enabled applications. It assumes a general familiarity with the following:

- JavaScript
- Adobe Bridge
- Any other Bridge-enabled applications you are using, such as Adobe Illustrator®, Adobe Photoshop®, or Adobe InDesign®. The scripting API details for each application are included with the scripting documentation for that product.

Note: If you have already been using the scripting interface to Bridge CS 2, see Chapter 6, "Porting Guide" for information about changes in this release.

What is in this book

This book contains the following chapters:

- Chapter 1, "Scripting Adobe Bridge," introduces some important concepts in Adobe Bridge scripting and describes the Bridge JavaScript document object model (DOM).
- Chapter 2, "Interacting with Adobe Bridge through Scripts," discusses the various ways of accessing `Thumbnail` objects, describes how Bridge generates user-interaction events, and shows how you can respond to these events by defining event handlers in your scripts.
- Chapter 3, "Creating a User Interface for a Script," discusses the various options available to scripts for interaction with Bridge users, such as dialog boxes and navigation bars, and describes how to use these means to display a user interface defined in ScriptUI or in HTML.
- Chapter 4, "Customizing the Adobe Bridge Browser Window" describes how to add a script-defined palette to Bridge, how to build a customized object-inspector pane that displays information related to the selected node in any way you choose, and how to customize Bridge menus.
- Chapter 5, "Extending Adobe Bridge Node-Handling Behavior," is intended for tool developers. It describes how to extend the node-handling behavior of Bridge and define your own node types.
- Chapter 6, "Porting Guide," summarizes changes and additions to this release, to help you in porting existing scripting applications.

Document conventions

Typographical conventions

`Monospaced font`	Literal values; code, such as JavaScript or HTML code; file and path names.
Italics	Variables or placeholders in code. For example, in `name="`*`myName`*`"`, the text *`myName`* represents a value you are expected to supply, such as `name="Fred"`. Also indicates the first occurrence of a new term.
Blue underlined text	An active link to a related section in this book or to a URL in your web browser.
Sans-serif bold font	The names of Bridge UI elements (menus, menu items, and buttons). The **>** symbol is used as shorthand notation for navigating to menu items. For example, **Edit > Cut** refers to the **Cut** item in the **Edit** menu.

Note: Notes highlight important points that deserve extra attention.

Where to go for more information

- For complete reference information for the JavaScript objects, properties, and functions defined by Adobe Bridge, see the *Bridge JavaScript Reference*.
- Adobe provides an extended version of JavaScript, used in many Adobe products, called ExtendScript. ExtendScript is a complete implementation of ECMA JavaScript, plus additional tool and utilities.

  The JavaScript API includes rich functionality such as building blocks for user interface elements and network connectivity. In addition, it allows you to to develop C and C++ native extensions.

  For a description of the objects, tools, and utilities defined by Adobe ExtendScript, including the ExtendScript Toolkit, see:

  JavaScript Tools Guide

- For a general introduction to scripting as a tool for working with Adobe applications, see:

  Adobe Creative Suite: Introduction to Scripting

This book does not describe the JavaScript language. For documentation of the JavaScript language or descriptions of how to use it, see any of numerous works on this subject, including the following:

- The public JavaScript standards organization web site: www.ecma-international.org
- *JavaScript: The Definitive Guide, 4th Edition*; Flanagan, D.; O'Reilly 2001; ISBN 0-596-00048-0
- *JavaScript Programmer's Reference*; Wootton, C.; Wrox 2001; ISBN 1-861004-59-1
- *JavaScript Bible. 5th Edition*; Goodman, D. and Morrison, M.; John Wiley and Sons1998; ISBN 0-7645-57432

1 Scripting Adobe Bridge

This chapter introduces some important concepts in Adobe Bridge scripting and describes the Adobe Bridge JavaScript API object model. For detailed descriptions of the objects and their properties and methods, see the *Bridge JavaScript Reference*.

Note: If you have already been using the scripting interface to Adobe Bridge CS 2, see Chapter 6, "Porting Guide" for information about changes in this release.

Scripting Overview

Adobe Bridge provides a configurable, extensible browser platform that allows users to search for, select, and organize files by navigating among files and folders in the local file system, those on remote file systems, and also web pages accessible over the Internet.

Adobe Bridge is integrated with Adobe applications, which bring up the Adobe Bridge browser window in response to specific user actions that require file selection, or through a Browse button or command. You can also bring up a browser window independently, by invoking it interactively or through a script.

The browser is highly configurable and extensible, using JavaScript. Adobe Bridge supports ExtendScript, the Adobe extended implementation of JavaScript. ExtendScript files are distinguished by the `.jsx` extension. ExtendScript offers all standard JavaScript features, plus additional features and utilities, such as:

- Platform-independent file and folder representation
- Tools for building a user interface to a script
- An interactive development and debugging environment, the ExtendScript Toolkit

For details of these features, see the *JavaScript Tools Guide*.

You can use JavaScript to manipulate browser windows and their contents programmatically, and to change and extend their functionality. This manual describes what you can do, and provides an overview of the JavaScript objects and functions that you can use to program Adobe Bridge.

Documentation and sample code

The Adobe Bridge Software Developer's Kit (SDK), available from Adobe Developer Center, http://www.adobe.com/devnet/, contains many code samples for Adobe Bridge and the JavaScript tools. You can download and install the SDK in a folder with a name and location of your choice, referred to here as *`sdkInstall`*. The SDK contains:

`sdkInstall``/docs/`	This document, and the companion documents *Adobe Bridge JavaScript Reference* and *JavaScript Tools Guide*.
`sdkInstall``/javascript/docs/`	Documentation for all of the JavaScript sample code in HTML format.
`sdkInstall``/javascript/sdksamples/`	A set of JavaScript code samples that illustrate Adobe Bridge scripting concepts and techniques.

The sections in this manual that discuss particular concepts list the code samples that demonstrate the related techniques.

Executing scripts for Adobe Bridge

Adobe Bridge executes JavaScript scripts in any of these ways:

- You can load and run a script in the ExtendScript Toolkit, specifying Adobe Bridge as the target application. This is how to run the example code Snippets. The Toolkit is a development environment for JavaScript, in which you can see debugging output and step through execution.

  If you run a script for the ExtendScript Toolkit that has Adobe Bridge as its target, the Toolkit automatically launches Adobe Bridge if it is not already running. Similarly, if another application sends a JavaScript message to Adobe Bridge, the messaging framework launches Adobe Bridge if necessary. For details of the Toolkit and messaging framework, see the *JavaScript Tools Guide*.
- You can create a menu command that runs a script and add it to a menu or submenu in the Adobe Bridge browser, using the `MenuElement` object. See 'Extending Adobe Bridge Menus' on page 44.
- When the browser window displays a JSX file, you can double-click that file thumbnail to run the script in its target application. It runs in Adobe Bridge if the script specifies Adobe Bridge as its target application by including the directive:

  ```
  #target bridge-2.0
  ```

  If the script specifies another application as its target, ExtendScript prompts the user to start that application if necessary. If the script does not specify a target application, it opens in the ExtendScript Toolkit. For details of application specifier syntax, see the *JavaScript Tools Guide*.
- You can pass a script to the Adobe Bridge executable, to be executed on startup, by dragging the JSX file icon onto the Adobe Bridge executable file icon or shortcut. This script is executed after all startup scripts found in the startup folders.
- Adobe Bridge has a dedicated location for user-installed startup scripts. This is the location referred to by the **Reveal** button in the Preference/General panel. All JSX files found in these locations are automatically run when Adobe Bridge is launched.
  - In Windows, the user startup folder is:

    ```
    %APPDATA%\Adobe\Bridge CS3\Startup Scripts\
    ```
  - In Mac OS, the user startup folder is:

    ```
    ~/Library/Application Support/Adobe/Bridge CS3/StartupScripts/
    ```

Other locations are used for the Adobe startup scripts that are supplied on installation of Adobe Bridge-enabled applications. As a third-party developer, you must be very careful when installing anything in these locations. They should be used only for extensions to the Adobe Bridge node-handling model that support companion applications. For details, see Chapter 5, "Extending Adobe Bridge Node-Handling Behavior.

The Adobe Bridge Browser Window and Object Model

An object model is an application programming interface (API), which allows you to programmatically access various components of a *document* (as defined for that application) through a scripting language such as JavaScript. In the case of Adobe Bridge, a "document" is defined as the browser window. This section shows how each aspect of the API relates to the parts of the browser. A more detailed description of the objects of the Adobe Bridge model follows in 'The Adobe Bridge Object Model' on page 14.

The Adobe Bridge browser window

The browser window is highly configurable. A browser configuration is called a *workspace*. There are predefined workspaces, and you can save and set the workspace interactively through the **Window > Workspace** menu, or through a script. User-defined workspaces are saved as XML files with the `.workspace` extension, in these locations:

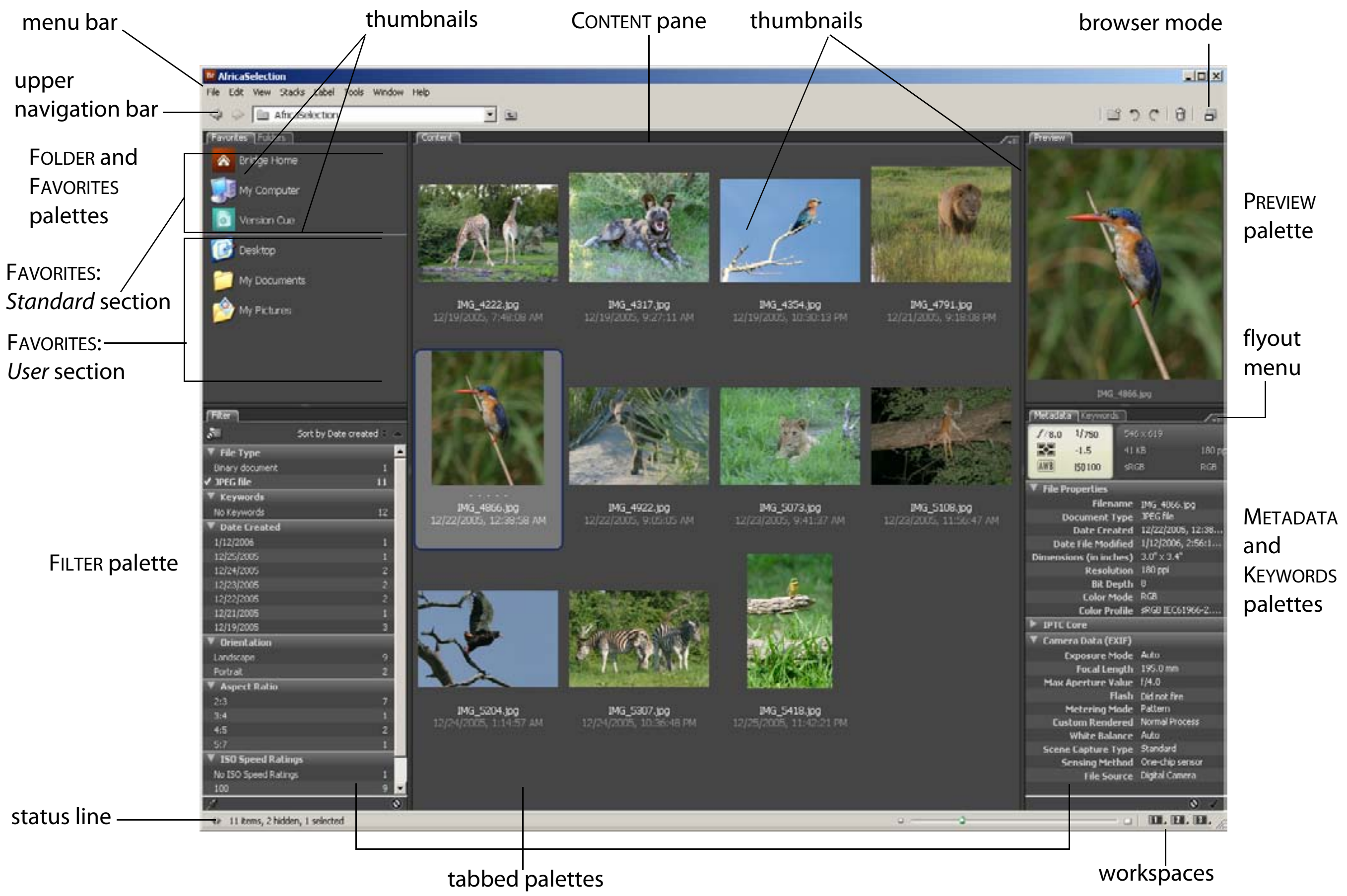
- In Windows, the user workspace folder is:

  `%APPDATA%\Adobe\Bridge CS3\Workspaces\`

- In Mac OS, the user workspace folder is:

  `~/Library/Application Support/Adobe/Bridge CS3/Workspaces/`

The following figure identifies parts of the browser window in a default configuration and viewing mode.



The tabbed palettes are arranged in three columns . You can change the arrangement interactively. Columns can be resized, and palette groups can also be resized within the column. Individual tabbed palettes can be opened and closed, and dragged into different relative positions. Dragging a palette to the bottom of an existing set adds a new tab set at the bottom of that column.

Your scripts can add customized palettes and menus, and control various aspects of the window configuration and display.

Accessing the Adobe Bridge browser through scripts

The following table describes how the Adobe Bridge JavaScript API maps to the various parts and features of the browser window, and how a script can access each part or feature through the Adobe Bridge object model.

Window area	Purpose	Scripting control
Browser window	Displays files, folders, and web pages, along with related file information and tools.	Represented by the `Document` object. Current browser window (the one with the operating system focus) is in `app.document`. All open browser windows are in `app.documents`. A script can create a new browser window by creating a new `Document` object. The browser mode (full or compact) is controlled by `app.document.browserMode`. The overall configuration of palettes is controlled by `app.document.workspace`. All defined configurations are in `app.workspaces`.
Thumbnails	The labeled icons that appear in the Folders/Favorites panes as navigation nodes, and in the Content pane and Preview palette to represent files and folders.	The `Thumbnail` object represents a node in the browser navigation hierarchy. Thumbnails can represent `File` or `Folder` objects, URLs, or script-defined nodes associated with script-defined node-handling extensions. Adobe Version Cue® nodes are an example of script-defined nodes with their own behavior.
Favorites palette	Provides a place for users to drag and drop favorite items (in the bottom half of the pane). Displays only top-level containers and one level of subnodes.	Access the visible thumbnails through the `app.favorites` property, and traverse the hierarchy through the `Thumbnail.children` properties. The `Favorites` object allows you to add thumbnails to this pane. User interaction with a thumbnail in this pane generates an `event` with a `Thumbnail` target object and a location of `favorites`.
Folder palette	Displays the file-system navigation hierarchy. Displays only containers (such as folders and subfolders). The content of the selected container appears in the Content pane.	Access the currently selected thumbnail in the Folders pane through the `Document.thumbnail` property. Traverse the hierarchy through the `Thumbnail.parent` and `Thumbnail.children` properties. User interaction with a thumbnail in this pane generates an `event` with a `Thumbnail` target object and a location of `document`.

Content pane	Displays container contents when you select a container node in the Folders or Favorites palette, or when you double-click a navigable node (such as a folder) in the Content pane itself. Displays both containers (such as subfolders) and leaf nodes (such as files and images). Can also display web pages.	Controlled by the `Document.presentationMode`, which you set with `Document.setPresentationMode()` to one of these values: ● `browser`: Displays file and folder nodes, represented by `Thumbnail` objects, found at the path specified in `Document.presentationPath`. When a folder is selected in the Folders pane, access the current contents of the Content pane through `app.document.thumbnail.children[]`. ● `html`: Displays an HTML page (local or remote) specified by a URL in `Document.presentationPath`. User interaction with a thumbnail in this pane generates an `event` with a `Thumbnail` target object and a location of `document`. The selected thumbnails are available through `Document.selections`.
Preview palette	Displays image previews.	User interaction with a thumbnail in this pane generates an `event` with a `Thumbnail` target object and a location of `preview`.
Filter palette	Allows you to sort thumbnails in the Content pane, or limit the nodes shown to those matching selected criteria.	The `FilterDescription` object allows you to specify filtering criteria for script-defined node types. When displaying a handled container node, Adobe Bridge builds the list of filters by calling the developer-defined `getFilterCriteria()` method of the node's `ExtensionModel`.
Metadata palette	Displays embedded file metadata information.	Embedded metadata is displayed and can be modified for a specific thumbnail. Access XMP metadata, encapsulated in a `Metadata` object, from the `Thumbnail.metadata` property. Set `app.synchronousMode` before accessing metadata in a script, to ensure that Adobe Bridge retrieves the latest valid data.
Keywords palette	Displays keyword information.	Not directly accessible to scripts, except insofar as you can modify keywords defined in embedded metadata.

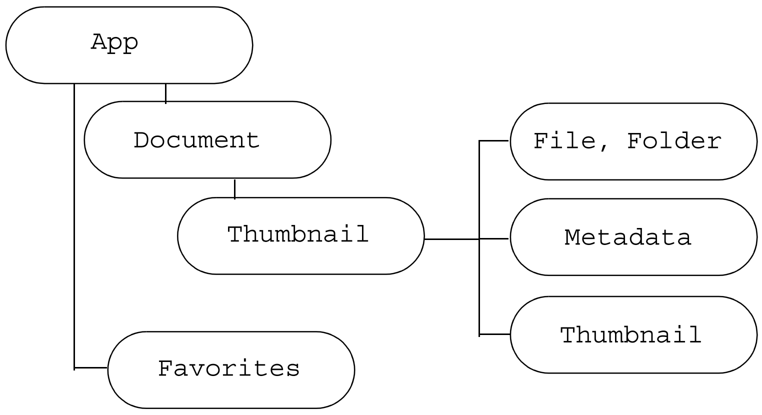
Custom tabbed palettes (not shown)	Script-defined tabbed palettes that can be added to the default palettes.	A new palette is represented by the `TabbedPalette` object. A palette can display HTML or ScriptUI user-interface elements; ScriptUI can display Flash animation. The `TabbedPalette.title` value is shown in the tab.
Inspector (not shown)	An object-inspection palette that displays further information related to a selected thumbnail in the Content pane. This type of palette is script-defined; there is no default version.	Each palette is represented by the `InspectorPanel` object, which contains one or more subpanels. The subpanels display information related to the focus thumbnail, directly or indirectly through `Thumbnail` or `Metadata` properties. Various types of subpanel are represented by subclasses of the `Panelette` base class.
View mode controls	Various controls at the bottom right of the browser that set the viewing mode to predefined pane configurations, and set the display size of thumbnails in the Content pane.	The view mode is controlled by `app.document.thumbnailViewMode`. You can control various aspects of the view with other `Document` properties such as `displayInspectorView`, `showFolders`, `showThumbnailName`, and so on.
Status line	The bottom bar on the browser window that displays current status information.	Set text to be shown in the status line with `Document.status`.
Menubar	The menubar at the top of the browser window that contains Adobe Bridge commands.	While this is not an object under direct `Document` control, you can add menus and commands using the `MenuElement` object, by referring to existing menus and commands.
Context menus	The right-click menus associated with thumbnails, and flyout menus for some palettes, containing context-specific commands.	You can add submenus and commands to these menus using the `MenuElement` object, by referring to existing commands.
Browser window upper navigation bar	The navigation bar immediately under the menubar. Not configurable.	Not accessible to scripts.
Top and bottom navigation bars	Two configurable navigation bars that can appear above and below the Content pane.	Represented by predefined `NavBar` objects, accessed through the `Document.navbars` property. By default, the navigation bars are invisible. You can make a bar visible, and add ScriptUI or HTML UI elements to it.

The Adobe Bridge Object Model

Each application has its own object model, which consists of a hierarchical representation of the application, and of the documents used in the application. The object model allows you to programmatically access and manipulate the document and its components. Since the use of a document varies for each application, the object model terminology varies for each application. For example, each application's object model includes a `Document` class, but the object referred to is different for each application, and the `Document` class has different properties and methods for each application.

Applications typically define a `Document` class to deal with files of a particular type, such as HTML pages, images, or PDF documents. However, Adobe Bridge uses a different approach. In the object model, the `Document` class refers to a browser window, and the properties and methods of `Document` refer to various components of the user interface (UI). The browser window displays icons that reference the files that other applications consider documents—HTML pages, images, PDFs, and so on. In the object model, these icons are represented by the `Thumbnail` class.

Basic node model



The application and documents

The Adobe Bridge `App` object is the root of the hierarchy, and represents the Adobe Bridge application. A single global instance of the `App` class, named `app`, is created when the application is started, and provides access to global values. Even though the user can create multiple browser windows by selecting the **File** > **New Window** command, making it appear that separate Adobe Bridge applications are running in parallel, only a single instance of the application is running, which is reflected by a single instance of the `app` object.

The `Document` object represents a browser window. Each time a user selects **File** > **New Window**, a new `document` object is created. When multiple browser windows are open, the user can select which window to use by clicking the window to make it active. In a script, you can access the active, or most recently used, browser window through the `app.document` property. The set of all open browser windows is available through the `app.documents` array.

Thumbnails in documents

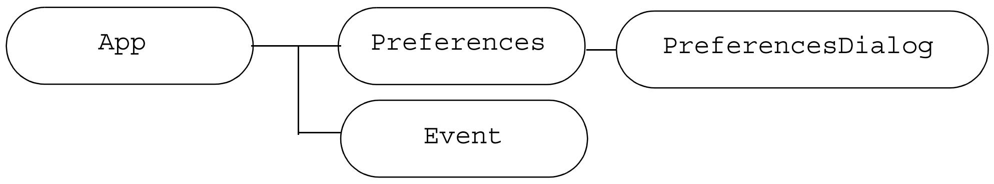
The `Thumbnail` object type represents a *node*, or browsable element in the browser navigation hierarchy. It typically represents a a file or folder, but can also be associated with a web page. A document contains various collections of `Thumbnail` objects.

- `Thumbnail` objects can contain other `Thumbnail` objects, as for example, when a folder contains files. In this case, the `container` property of the `Thumbnail` is `true`, and the `children` property contains a hierarchy of `Thumbnail` objects that represent files and folders contained in the folder.

- The Folders pane shows the full navigation hierarchy of folders and subfolders for the local file system, and for virtual file systems such as that defined by Version Cue. Scripts can add nodes to the Folders pane using `app.addCustomRoot()`. A script can access the currently selected thumbnail through the `app.document.thumbnail` property, and can walk the navigation hierarchy by accessing the `parent` and `children` properties of each `Thumbnail` object.
- The Favorites pane shows a selection of high-level nodes, some predefined (in the upper Standard section) and some chosen by the user (in the lower User section). These nodes can represent web pages, virtual locations such as Stock Photos, and remote folders, as well as local folders. The `Favorites` object represents the navigation nodes in the Favorites pane. A document contains a single `Favorites` object, which contains two arrays of `Thumbnail` objects, for the Standard and User sections. Access the `Favorites` object through `app.favorites`.
  - A script can add thumbnails to the User section of the Favorites pane by using the `Favorites` object's `insert` method, and one level of sub-nodes using the `addChild` method. A subnode can be any thumbnail; it does not have to be part of the root node's `children` hierarchy.
  - A script cannot remove thumbnails in the Standard section, but they can be shown or hidden with a Preference, and a script can modify the Preference and display state using `Favorites.enable()` and `Favorites.disable()`.
- You can add customized panes or palettes to the browser window which can display thumbnails, or information directly or indirectly contained in selected thumbnails; see 'Customizing the user interface' on page 17.

For additional information about how to work with `Thumbnail` objects, see 'Accessing Thumbnails with Scripts' on page 23.

Scripting Adobe Bridge interactions



Application preferences

The `Preferences` object allows a script to access Adobe Bridge application preferences. These are the values that can be viewed interactively in the Preferences dialog, in response to the **Edit** > **Preferences** command. The settings are stored and persist across sessions. Your script can use the `Preferences` object to view or set existing preferences, or to add new preference fields. In some cases, when you modify persistent preference values, the new settings are not reflected in the browser window until the application is restarted.

When the user brings up a Preferences dialog, Adobe Bridge invokes a ScriptUI dialog window, and generates a `create` event with the `PreferencesDialog` object as its target. You can define and register an event handler for this event that uses the object's `add` method to add a ScriptUI panel containing ScriptUI controls that show and allow the user to modify script-defined preference values.

User-interaction events

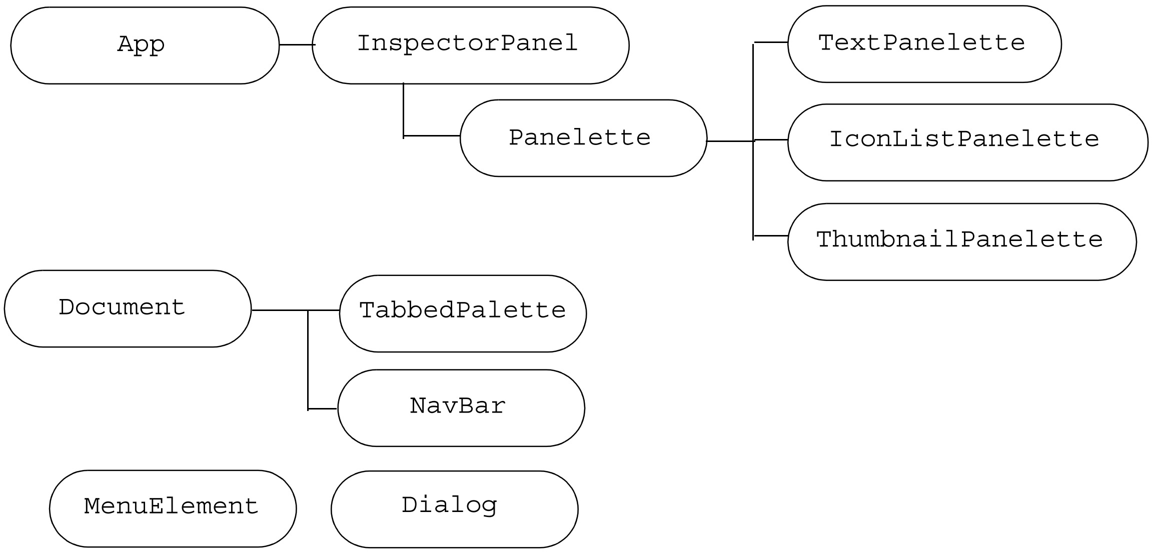
When interacting with Adobe Bridge, a user takes actions such as changing the thumbnail selection, or creating a new browser window. For most of these actions, Adobe Bridge triggers a *user-interaction event*, represented by an `Event` object of a particular *event type*, with a particular *target object*, such as an `App`, `Document`, or `Thumbnail` object. Some function calls can also trigger events.

Adobe Bridge defines default behavior for user-interaction events. You can extend or override the default behavior by defining and registering an *event-handler function*. This function receives the `Event` object as an argument, and returns a value that tells Adobe Bridge whether to continue with the default behavior or to ignore it.

For more information, see 'Event Handling in Adobe Bridge' on page 27.

Customizing the user interface

You can add user interfaces to your scripts or browser interactions, and your scripts can customize the browser window.

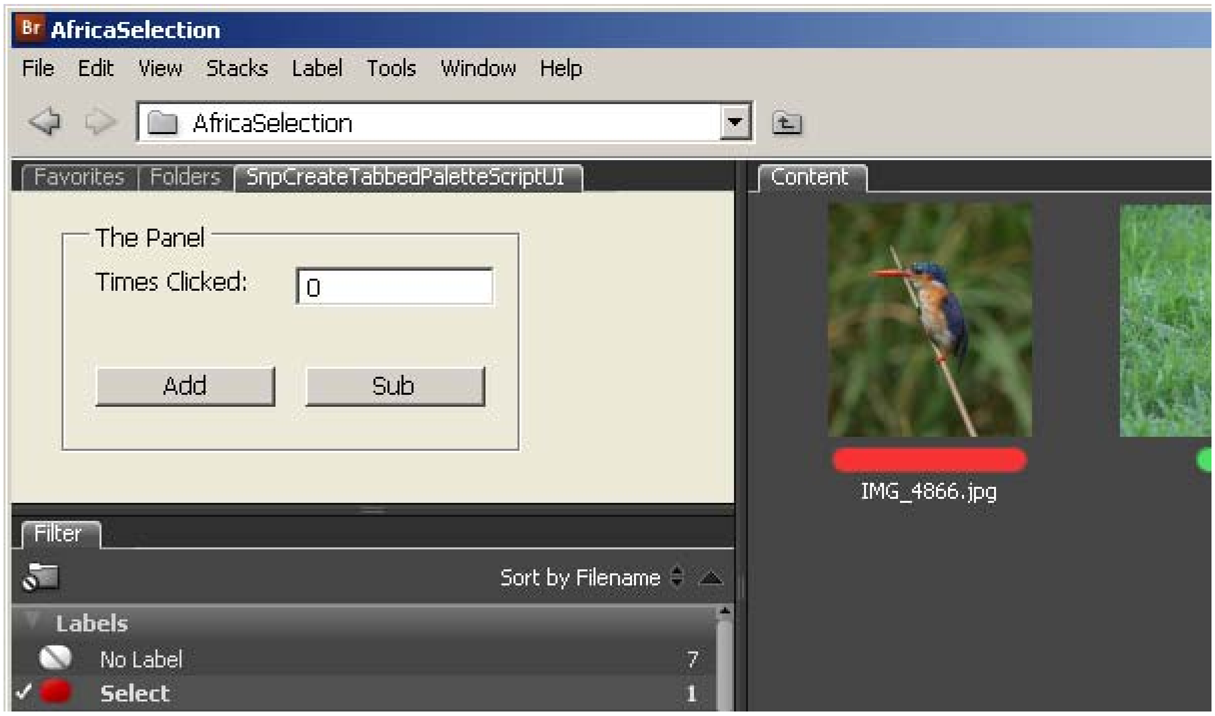


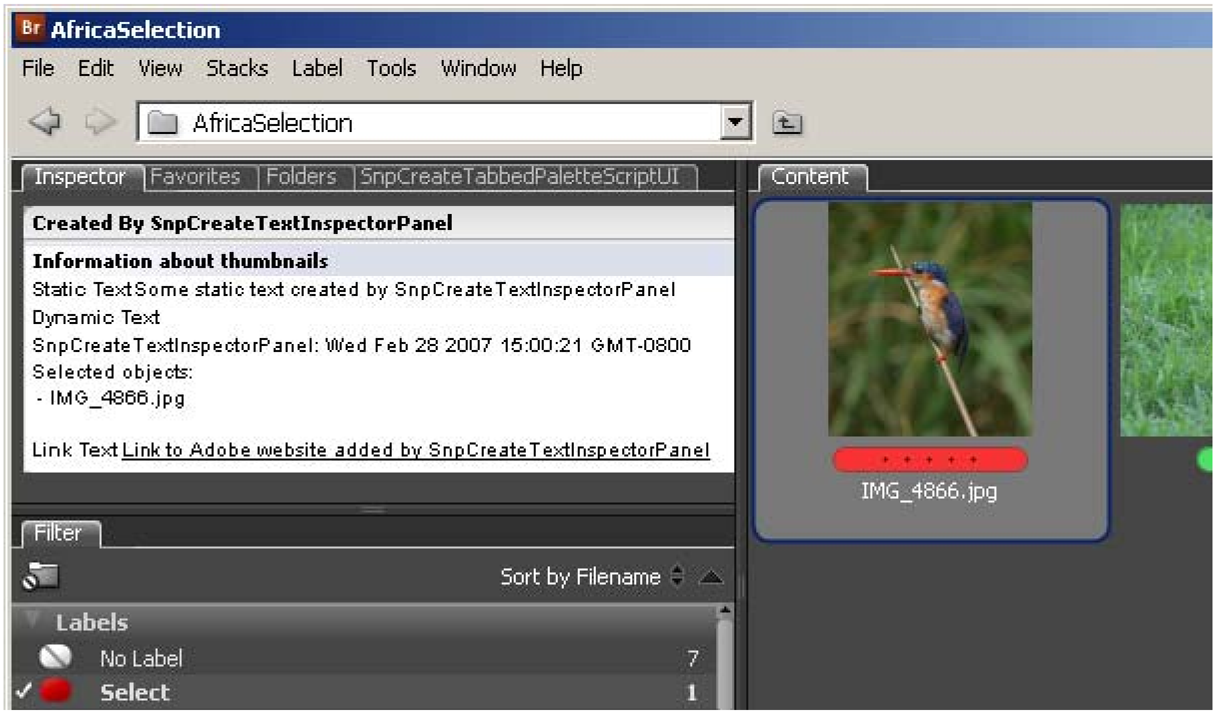
Customizing the browser window

- You can define additional tabbed palettes like those provided by default, using the `TabbedPalette` object. These can display HTML or ScriptUI content. Script UI provides a complete set of user-interface components that includes a Flash movie player.
- You can define object-inspection palettes using the `InspectorPanel`, which contains customized subpanels that display information related to the currently selected thumbnail. The selected thumbnail is the inspection focus of the display.

By default, these script-defined palettes appear in the upper left of the browser, but once they are displayed, you can resize them and drag them to new locations, like the predefined palettes.

The following figures show a script-defined palette with ScriptUI content, and a script-defined Inspector panel, both created by the example scripts in the Adobe Bridge SDK.




AfricaSelection
File Edit View Stacks Label Tools Window Help
AfricaSelection
Inspector Favorites Folders SnpCreateTabbedPaletteScriptUI
Content
IMG_4866.jpg
Filter
Sort by Filename
Labels
No Label 7
Select 1

Created By SnpCreateTextInspectorPanel

Information about thumbnails

Static Text Some static text created by SnpCreateTextInspectorPanel
Dynamic Text
SnpCreateTextInspectorPanel: Wed Feb 28 2007 15:00:21 GMT-0800
Selected objects:
- IMG_4866.jpg

Link Text Link to Adobe website added by SnpCreateTextInspectorPanel

The displayed information in an Inspector can be related to the selected node through embedded file metadata (in associated `Metadata` objects) or through Adobe Bridge- or script-defined properties of the `Thumbnail` object (see Chapter 5, "Extending Adobe Bridge Node-Handling Behavior").

For information on how to make these additions, see Chapter 4, "Customizing the Adobe Bridge Browser Window.

Communicating with the user from a script

Your script can display information to or collect information from the user by configuring the supplied Navigation bars, or by creating and displaying Dialogs.

ExtendScript provides a set of user-interface objects in the ScriptUI module, which defines windows and user-interface controls. You can use these objects to define a user interface for your application, in the form of popup dialogs, persistent dialogs (called palettes), or as part of navigation bars. The usage of the ScriptUI objects is discussed in the *JavaScript Tools Guide*.

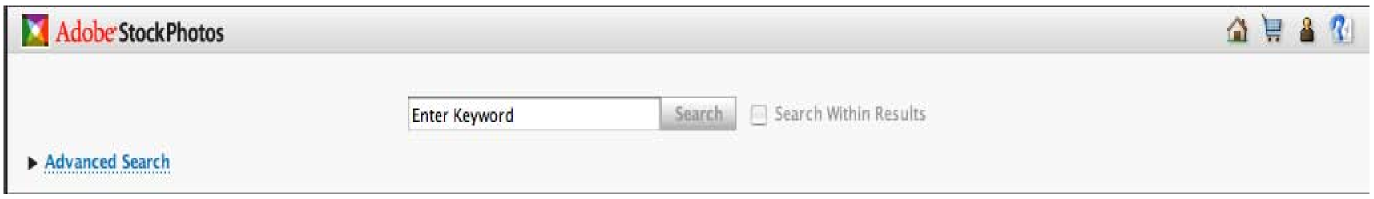
You can also define a user interface using standard HTML. When integrating your user interface with the browser, you can use either ScriptUI or HTML controls for any window or pane, but cannot mix the two. For a complete discussion, see Chapter 3, "Creating a User Interface for a Script."

In addition to displaying a user interface for your script, you can script user interactions by extending the Adobe Bridge Menus.

Navigation bars

The Adobe Bridge navigation bar immediately below the menubar cannot be scripted, but there are two configurable navigation bars, above and below the Content pane. They are represented by `NavBar` objects, which you can access through the `Document` object's `navbars` property.

The following figure shows a navigation bar defined by Adobe StockPhotos.



By default, the navigation bars are hidden and empty.

- You can show and hide a navigation bar by setting the object's `visible` property.
- You can configure a navigation bar to display either ScriptUI user-interface controls, or straight HTML controls. It cannot mix the two.
  - To display ScriptUI controls, set the `type` property to `"scriptui"`, then use the `NavBar.add` method to add controls.
  - To display HTML controls, set the `type` property to `"html"`, and the `file` property to the HTML file that defines the page you want to display.

You can program the controls to display information to or collect information from the user. For additional details, see 'Navigation bars' on page 31.

Dialogs

Your script can define dialogs to display information to or get information from the user. There are two ways to define these:

- **ScriptUI Dialogs**: Use the ScriptUI `Window` object to define a dialog that displays ScriptUI controls. For details of programming ScriptUI dialogs, see the *JavaScript Tools Guide*.

- **HTML Dialogs**: The Adobe Bridge `Dialog` object represents a window that displays an HTML page, rather than ScriptUI controls. For details of programming Adobe Bridge dialogs, see 'Displaying HTML in Adobe Bridge dialogs' on page 36.

You can invoke ScriptUI or Adobe Bridge dialogs from a script as *modal* or *nonmodal* dialogs.

- A modal dialog retains the input focus, and does not allow the user to interact with any other application windows until the dialog is dismissed. The function that invokes it does not return until the dialog is dismissed.
- A nonmodal dialog (known in ScriptUI as a *palette*), does not keep the input focus. The user can interact with other application windows while the dialog is up. The function that invokes it returns immediately, leaving the dialog on screen until the user or script closes it.

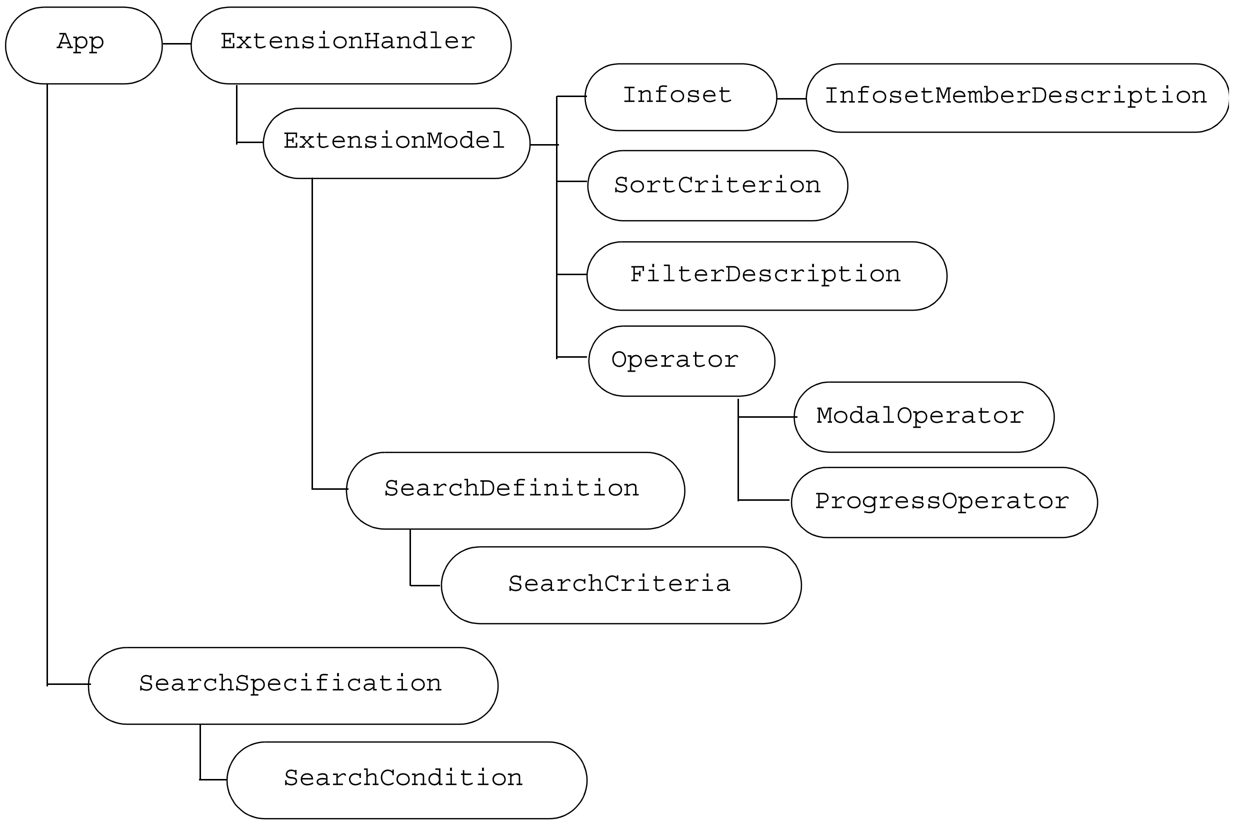
Menus

The `MenuElement` object allows you to extend Adobe Bridge menus. A script cannot remove or alter the behavior of predefined menu items, but you can add, remove, and modify script-defined menus, submenus and commands.

The menu bar, and most menus, submenus, and context menus can be extended by creating new `MenuElement` objects that reference existing menus and menu items. The identifiers of all menus and menu items that are accessible to scripts are listed with the description of the `MenuElement` object in 'Extending Adobe Bridge Menus' on page 44.

Extending browser behavior

Adobe Bridge provides a development infrastructure for advanced developers who want to create plug-in type extensions to basic Adobe Bridge functionality, in order to integrate Adobe Bridge with other applications and systems.



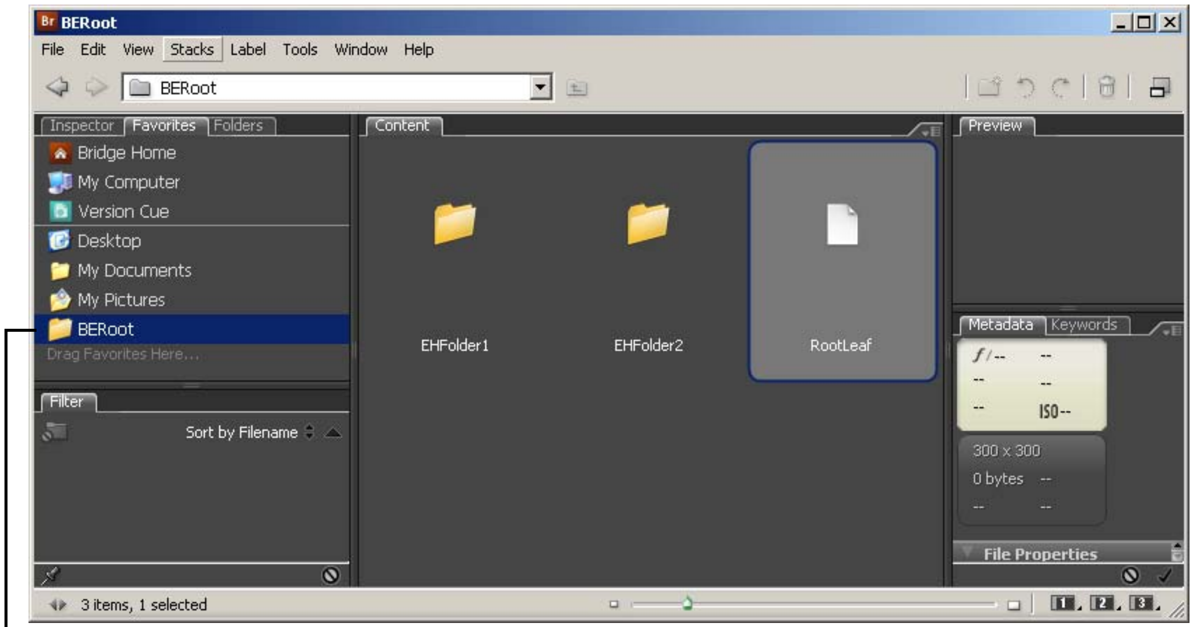
An advanced developer can extend the default node-handling behavior by defining a new node type, and a *node-handling extension* that manages that node type. When you define a node type, you assign an identifying prefix (or more than one), and register that prefix to associate it with the `ExtensionHandler` object that implements the handler.

To implement a node-handling extension, you define `ExtensionHandler` and `ExtensionModel` classes, providing the node-handling methods.

- You can create script-defined properties in the `Thumbnail` object for your handled nodes, using the `Infoset` object. You can access and display this kind of node data in a custom Inspector panel; see 'Script-Defined Inspector Panels' on page 41.
- You can define node search and filter criteria that consider the values of metadata and node data properties associated with your node types.
- You can define sorting criteria that determine how your nodes are ordered in the Content pane or Inspector panels.
- For time- or resource-intensive tasks such as those involving file-system access, your model methods must define an `Operator` object to execute and monitor the operation.

For a complete discussion, see Chapter 5, "Extending Adobe Bridge Node-Handling Behavior."

The following figure show the result of running the basic node-handling extension example provided with the Adobe Bridge SDK. Additional examples show how to add node-specific information to your own node types, and how to search and filter nodes of the handled type, based on the node-specific information.



Node of handled type added to Favorites palette

Container and leaf nodes of handled type displayed in Content pane

2 Interacting with Adobe Bridge through Scripts

The `Thumbnail` object is the basic JavaScript representation of the entities that your Adobe Bridge browser displays; see 'Accessing Thumbnails with Scripts' on page 23.

As a user interacts with the browser window, Adobe Bridge generates *user-interaction events*. Your scripts can detect and respond to these events in order to extend or override the default behavior; see 'Event Handling in Adobe Bridge' on page 27.

Your scripts can communicate with other Adobe JavaScript-enabled applications, such as Photoshop, Illustrator, and InDesign. You can, for example, send a file that a user selects in the Adobe Bridge browser to be opened in Photoshop. See 'Communicating with Other Applications' on page 28.

Accessing Thumbnails with Scripts

The `Thumbnail` object represents a *navigation node*. Thumbnails can represent entities such as files and folders, accessed through a local or remote file system, or web pages accessed over the internet and displayed in an embedded web browser. A thumbnail can also represent a service, such as Stock Photo or Adobe Media Gallery.

Thumbnails as node references

The `Thumbnail` object represents a node in a navigation tree. The basic behavior of a thumbnail icon is determined by a *node handler*; the core node handler provides the default behavior. Node handling is generally internal, although it is extensible by tool developers. By default, a node is associated with a file-system path or a `File` or `Folder` object, or with a URL. Clicking the node displays the associated folder contents, file, or page in the standard Adobe Bridge panes.

- Any scripter can define an event handler that responds to these standard thumbnail interaction events; see 'Event Handling in Adobe Bridge' on page 27.

A `Thumbnail` object is identified by a unique identifying string called a *Bridge URI*, which is available to scripts through the `Thumbnail.uri` property. The URI consists of a path or URL string, with a prefix string that serves as a node identifier. The node identifier determines how the node is handled when the user selects the icon. The default behavior is determined by the default node handler, which is identified by the default prefix `bridge:`.

- A tool developer can extend the default node-handling behavior, defining new node types and adding properties to the `Thumbnail` object; see 'Extending browser behavior' on page 21.

Here are some examples of complete, or *canonical*, Bridge URIs, including the prefix registered for the node handler:

Standard file, with default node handling	`bridge:fs:file:///C:/BridgeScripts/icons/2.jpg`
HTML page in Content pane, with default node handling	`bridge:html:file://C:\myWebPage.html`

Script-defined node-handling extension (from the Bridge SDK example)	`bridge:beNode:/BERoot/EHFolder1`
Adobe Stock Photos Favorite	`bridge:script:stockphoto://home`
Version Cue project file	`bridge:vc:vcstable:project/44940038-11e4-104c-bd8a-9211234f21d6/44940038-11e4-104c-bd8a-9211234f21d6_1_1`
Version Cue asset	`bridge:vc:vcstable:file/44940038-11e4-104c-bd8a-9211234f21d6/44940038-11e4-104c-bd8a-9211234f21d6_1_1/Folder%20One/thumbTest.jpg`

Using and accessing thumbnails

Thumbnails are used in a number of ways within a browser window, and the objects are referenced according to their use. For example:

- Access thumbnails that appear in the Favorites pane through `app.favorites`
- Access a thumbnail that is selected in the Folders pane through `app.document.thumbnail`
- Access thumbnails that appear in the Content pane through `app.document.thumbnail.children`
- Access thumbnails that are associated with a context menu through `app.document.context`
- Access thumbnails that have been selected in the Content pane through `app.document.selections`, or `app.document.getSelection()`.
- Access those thumbnails that are currently visible in the Conent pane through `app.document.visibleThumbnails`.

Note: Be careful when accessing selected or visible thumbnails, as it is possible for very large numbers of thumbnails to be selected or visible, and accessing a large array very often can affect performance; see Accessing many thumbnails correctly below.

Accessing many thumbnails correctly

Accessing many thumbnails can be a time-intensive operation for your scripts. Performance can suffer if you do so unnecessarily.

- Check the number of selections first with `app.document.selectionLength`, to avoid unnecessary access to a very large selection. This is much more efficient than accessing the collection, then checking its size. Similarly, use `app.document.visibleThumbnailsLength` to avoid accessing a very large array in `app.document.visibleThumbnails`.
- Use `app.document.getSelection()` to limit the collection to thumbnails for files of a specific type, or to collect only those thumbnails that are currently visible. This function takes an optional argument, a list of file extensions. For example:

  ```
  // Get the selected Thumbnail objects - only accept these file types
  var thumbs = app.document.getSelection("psd, jpg, png, tif, gif");
  ```

  By default, it matches all file types. If no thumbnails of the given type are selected, it returns those that are visible in the Content pane. See examples in Bridge SDK samples `SnpSaveAsJPEG.jsx` and `SnpRotateImage.jsx`.

- If you must iterate over many thumbnails, be careful to collect the `Thumbnail` objects first, then perform the iteration, rather than accessing the objects repeatedly.

  The correct style of thumbnail access is *hundreds of times faster* than the incorrect style. This example illustrates correct and incorrect styles of iteration.

  ```
  // correct: access thumbnails once, outside loop
  var theChildren = myThumb.children;
  for( var f = 0; f < theChildren.length; ++f ) {
     var child = theChildren[f];
  }
  // wrong: access thumbnails repeatedly, inside loop
  for( var f = 0; f < myThumb.children.length; ++f ) {
     var child = myThumb.children[f];
  }
  ```

Ensuring valid thumbnail data

When a script accesses the properties of a `Thumbnail` object, some properties of the object may not be immediately available. To ensure the object contains current data, set `app.synchronousMode` to `true` before accessing properties. If you do not do so, you may find that the values of `Thumbnail` properties are undefined, or not what you expect.

The default value of `app.synchronousMode` is `false`, for performance reasons; this is because thumbnails are accessed internally by Adobe Bridge much more frequently than by scripts. In your scripts, however, you should make a habit of setting it to `true`. It is automatically reset to `false` after scripts complete.

Metadata for thumbnails

A `Thumbnail` object is associated with a `Metadata` object, which allows you to access the embedded metadata for the associated file, such a copyright owner, author, or camera settings. Metadata is kept in the Adobe XMP format. It includes metadata defined in other formats, such as EXIF, in namespaces for those formats. For more information on the XMP metadata format, see the XMP Specification.

When a script needs to access the metadata through the `Thumbnail` object, it is important to make sure that the returned object contains the most current data. To ensure this, your script should set `app.synchronousMode` to `true` before attempting to retrieve values through `Thumbnail.metadata`, or else use `Thumbnail.synchronousMetadata`.

Keep in mind, however, that metadata access is a time-intensive operation. Do not do it unnecessarily, or as part of operations that occur very frequently, such as a `MenuItem.onDisplay` callback function.

Note: For metadata properties that are known date formats, the corresponding `Metadata` object property contains an ISO-8601 date string. For more information on date formats, see the XMP Specification.

You can extend the Adobe Bridge browser to display context-specific information indirectly associated with a selected thumbnail through its metadata. See 'Customizing the browser window' on page 17.

➤ Example code

The sample code distributed with the Adobe Bridge SDK includes these code examples that specifically demonstrate thumbnail and metadata access:

Thumbnail metadata access in *sdkInstall*/sdksamples/javascript/	
SnpInspectMetadata.jsx	Shows how to acquire metadata.
SnpModifyMetadata.jsx	Shows how to alter metadata on a selected file.

Embedding metadata in a script as XML

Adobe Bridge allows you to embed certain metadata in a script that describes the script file itself, using XML delimited by special tags within a comment block. Only Dublin Core properties can be embedded this way. The following Dublin Core properties are particularly useful for describing a JavaScript file:

dc:title	The display name for the script.
dc:description	A short description of the script.
dc:source	Where to get updates to the script.

For details of the Dublin Core metadata specification, see http://dublincore.org/documents/dces/.

Adobe Bridge uses the embedded metadata title and description (if it is less than 50 characters) in the Startup Scripts page of the Preferences dialog.

The tags **@@@START_XML@@@** and **@@@END_XML@@@** enclose a block of XML within a C-style comment block in a JavaScript script:

```
/*
@@@START_XML@@@
    XML block goes here
@@@END_XML@@@
*/
```

All XML in this block must be UTF-8 encoded:

```
/*
@@@START_XML@@@
<?xml version="1.0" encoding="UTF-8"?>
    ...
@@@END_XML@@@
*/
```

Within the XML block, use the `<ScriptInfo>` tag to describe each XMP metadata element. The XML may contain one or more `<ScriptInfo>` elements.

- A `<ScriptInfo>` element must specify the `xml:lang` attribute as an Adobe-supported language code.
- Within a `<ScriptInfo>` element block, you can specify Dublin Core metadata key-value pairs, each on a separate line. Specify each metadata property as an element whose name is the key and whose text content is the value:

  ```
  <dc:title>Adobe Flash CS3 Professional</dc:title>
  ```

Adobe Bridge's parser does not perform full XML parsing. It looks for the `<ScriptInfo>` block that matches the current locale, then searches from there for the relevant tags. For example, the following specifies two language versions of a description for the containing script:

```
/*
 @@@START_XML@@@
 <?xml version="1.0" encoding="UTF-8"?>
 <ScriptInfo xmlns:dc="http://purl.org/dc/elements/1.1/" xml:lang="en_US">
    <dc:title>Adobe Flash CS3 Professional</dc:title>
    <dc:description>This script enables other applications to communicate
        with Adobe Flash.<dc:description>
 </ScriptInfo>
 <ScriptInfo xmlns:dc="http://purl.org/dc/elements/1.1/" xml:lang="fr_FR">
    <dc:title>Adobe Flash CS3 Professional</dc:title>
    <dc:description>Ce script permet à d'autres applications de communiquer
        avec Adobe Flash.</dc:description>
 </ScriptInfo>
 @@@END_XML@@@
*/
```

Event Handling in Adobe Bridge

When a user takes certain actions in Adobe Bridge, such as copying a file, or creating a new browser window, Adobe Bridge generates a user-interaction event. User-interaction events include actions on thumbnails (such as selecting them), and also actions on the application (quitting) and on the browser window (such as activating it in the windowing system).

You can modify the way Adobe Bridge responds to these events by defining your own event handlers. Scripts can also generate events through function calls, that simulate user activity, such as `Thumbnail.open()`, or `Document.select()`.

Note: The event handling mechanism described in this chapter applies only to the Adobe Bridge objects. If your script defines its own user interface, events are handled differently, depending on what kind of object generated them:

- For events generated by ScriptUI objects (such as controls in the navigation bar), see the *JavaScript Tools Guide*.
- For events generated by menu elements, see 'Extending Adobe Bridge Menus' on page 44.
- Events generated by HTML controls in HTML navigation bars or dialogs are handled by their own HTML-defined handlers. These can access Adobe Bridge objects through a callback mechanism. See 'Displaying HTML in Adobe Bridge' on page 34.

Defining event handlers

An event-handler function takes one argument, an `Event` object. This object, which is passed to your registered handler when the event occurs, contains all of the context information about the event, such as which type of event occurred, the target object that generated it, and where that object was located within the browser window.

Your handler returns an object with a boolean `handled` property.

- When an event handler returns `{handled:true}`, Adobe Bridge does not look for any more handlers, nor does it execute the default handler.

- When an event handler returns `{handled:false}`, Adobe Bridge continues to look for registered handlers, and if no more script-defined handlers are registered, it executes the default handler. This is the default behavior if your handler does not return a value.

Using this mechanism, you can extend the default behavior of the Adobe Bridge objects. For example, when the user quits the Adobe Bridge application, your `destroy` or `close` event handler can take additional actions, such as cleaning up structures you have made, or displaying status information. To extend the default behavior, your handler returns the object `{handled:false}`.

In many cases, such as `Thumbnail` events, you can use the event handler to override the default behavior. You do this by returning the object `{handled:true}`, which prevents Adobe Bridge from executing the default handler.

For some events, such as `Document` events, you cannot override the default behavior of the event. Even if your handler returns `{handled:true}`, the default behavior still executes when your handler has finished. The `{handled:true}` return value does, however, prevent Adobe Bridge from executing any subsequent script-registered event handlers.

Registering event handlers

To register an event-handler function you have defined, create an `EventHandler` object and add it to the array `app.eventHandlers`. An `EventHandler` is a simple JavaScript object with a `handler` property that specifies the name of the event-handler function. There is no constructor, it is a simple script-defined object. For example:

```
var myEventHandler = { handler: doThisEvent };
app.eventHandlers.push (myEventHandler);
```

It is most efficient to write one handler that responds to many different events, rather than write one handler for each type of event. When an event occurs, Adobe Bridge iterates through the `app.eventHandlers` array, trying each handler in sequence, passing in the triggering event object. If one of the event handlers returns `{handled:true}` Adobe Bridge stops the iteration.

➤ **Example code**

The sample code distributed with the Adobe Bridge SDK includes a number of examples that use event handling in the course of demonstrating various other features. These code examples specifically demonstrate event handling:

Event handling examples in *`sdkInstall`*`/sdksamples/javascript/`	
`SnpDefineAppClosingHandler.jsx`	Shows how to create an event listener that responds to quitting from the Adobe Bridge application.
`SnpListenDocEvents.jsx`	Shows how to create separate event handlers for different events.

Communicating with Other Applications

The Adobe scripting environment provides an interapplication communication framework, a way for scripts to communicate with other Adobe applications, from Adobe Bridge or among themselves.

- A script can call certain basic functions exported by all JavaScript-enabled applications. For example, an Adobe Bridge script could ask the user to select an image file, then open that file in Photoshop® or

Illustrator® by calling the `photoshop.open` or `illustrator.open` function. These basic exported functions are called the *Cross DOM*.

- Individual applications export additional functions to make more complex functionality available to scripts. For example, an Adobe Bridge script can request a photo-merge operation in Photoshop by calling `photoshop.photomerge` with a set of selected image files. The set of functions available for each application varies widely.
- A messaging protocol based on the JavaScript `BridgeTalk` object provides a general and extensible framework for passing any kind of data between *messaging enabled* applications. Many Adobe applications are messaging enabled. You can send messages that contain JavaScript scripts. The target application can evaluate a script that it receives, and send results back in a response message.

For complete details, see the *JavaScript Tools Guide*.

➤ Example code

The sample code distributed with the Adobe Bridge SDK includes these code examples that specifically demonstrate interapplication messaging between Adobe Bridge and Photoshop:

Interapplication messaging examples in *`sdkInstall`*`/sdksamples/javascript/`

`SnpOpenInPhotoshop.jsx`	Shows how to call on another JavaScript-enabled application to open a file using the Cross DOM functionality.
`SnpSaveAsPNG.jsx`	Shows how to send multiple image files to Photoshop to be saved in a PNG format.
`SnpSendMessage.jsx`	Demonstrates interapplication communication using `BridgeTalk` messages, showing the order of arrival of messages and message responses.
`SnpSendCustomObject.jsx`	Shows how to pass a custom object from Adobe Bridge to Photoshop.
`SnpSendArray.jsx`	Shows how to pass an array from Photoshop to Adobe Bridge.
`SnpSendDOMObject.jsx`	Shows how to pass an object-model object from Photoshop to Adobe Bridge through the interapplication messaging framework.

3 Creating a User Interface for a Script

The Adobe Bridge scripting environment provides a number of options for interacting with users.

- You can define a response to a user's interaction with Adobe Bridge objects, such as thumbnails, through the event-handling mechanism, as discussed in Chapter 2, "Interacting with Adobe Bridge through Scripts. "
- This chapter discusses the various ways you can build a user interface into your scripts, either bringing up your own dialogs, or displaying UI controls in the navigation bar or Content pane.
- For more extensive customization of the browser window (adding menus and commands, tabbed palettes, and Inspector panels), see Chapter 4, "Customizing the Adobe Bridge Browser Window."

➤ Code examples for UI techniques

The sample code distributed with the Adobe Bridge SDK includes these code examples that specifically demonstrate various techniques for building a user interface for an Adobe Bridge script:

ScriptUI user interface examples in `*sdkInstall*/sdksamples/javascript/`	
`SnpAddScriptUINavBar.jsx`	Shows how to add a ScriptUI navigation bar to the Adobe Bridge browser window, and adds an event handler for changes in the selected thumbnail (file or folder).
`SnpCreateDialog.jsx`	Shows how to create and display a ScriptUI dialog with OK/ Cancel buttons and event listeners
HTML user interface examples in `*sdkInstall*/sdksamples/javascript/`	
`SnpAddHTMLNavBar.jsx`	Shows how to load an HTML page into the bottom navigation bar and execute a script embedded in the HTML page.
`SnpCreateHTMLDialog.jsx`	Shows how to create and display an Adobe Bridge dialog containing HTML components and use callbacks to pass simple data.
`SnpShowHTMLInContent.jsx`	Shows how to display a local HTML file in the Adobe Bridge Content pane.
`SnpPassObjectToHTML.jsx`	Shows how to pass complex values from callback functions in Adobe Bridge to a call made from an HTML page.
`SnpScheduleTask.jsx`	Shows how to schedule tasks that call JavaScript functions defined in an HTML page displayed in the Adobe Bridge Content pane.

User Interface Options for Scripts

If you want to display your own window or pane to the user, you can do so in several ways: by creating popup or persistent *dialogs*; by configuring and displaying predefined *navigation bars*; or by defining user-interface controls to be displayed in the Content pane, in response to selection of specially defined thumbnails.

You can define user-interface controls in any of these places in two ways:

- **ScriptUI Elements**: ScriptUI is a JavaScript module that defines windows and user-interface controls. You can create ScriptUI Dialogs boxes, or customized Tabbed palettes, and populate them with ScriptUI controls; or you can add ScriptUI controls to the existing Navigation bars. If you use ScriptUI controls, you can take advantage of the ExtendScript localization feature.

  For complete details about using ScriptUI, see the *JavaScript Tools Guide*.

- **HTML Pages**: An HTML page can contain standard HTML user-interface controls. You can display HTML pages in Navigation bars, in Adobe Bridge Dialogs boxes, in the Content pane, or in customized Tabbed palettes.

  You cannot use ExtendScript features or Adobe Bridge objects directly in an HTML page; for details, see 'Displaying HTML in Adobe Bridge' on page 34.

Your script-defined windows or panels must use one or the other of these methods. You cannot mix ScriptUI controls with HTML controls.

Navigation bars

Adobe Bridge provides two configurable navigation bars, one of which can be displayed at the top of the browser window (below the application navigation bar), and one at the bottom (above the status bar). There are two versions of each bar, for use with the two display modes of the Content pane. Access these existing `NavBar` object objects through the `Document` object' s properties.

- When the Content pane is displaying a web page (`Document.contentPaneMode="web"`), use these bars:

  ```
  topbar = app.document.navbars.web.top
  btmbar = app.document.navbars.web.bottom
  ```

- When the Content pane is displaying a folder's contents (`Document.contentPaneMode="filesystem"`), use these bars:

  ```
  topbar = app.document.navbars.filesystem.top
  btmbar = app.document.navbars.filesystem.bottom
  ```

The navigation bars are hidden by default. You can show and hide them by setting the `NavBar` object's `visible` property.

A navigation bar typically contains user-interface controls such as push buttons, radio buttons, scroll bars, list boxes, and so on. The `NavBar` objects are initially empty.

A navigation bar can display either ScriptUI user-interface controls that you add as children of the `NavBar` object, or an HTML page that you reference from the `NavBar` object. It cannot mix the two. In either case, you define the controls and program them to display information to or collect information from the user.

- Set the `NavBar.type` to `"scriptUI"` to display ScriptUI controls. See 'Displaying ScriptUI elements in a navigation bar' on page 33.
- Set the `NavBar.type` to `"html"` to display HTML controls. See 'Displaying HTML in a navigation bar' on page 37.

Dialogs boxes

A dialog box, like a navigation bar, can display either ScriptUI controls or HTML controls, but not both. In the case of dialogs, there are two different types of objects.

- Create a ScriptUI `Window` object to display ScriptUI controls. See 'Displaying ScriptUI dialogs' on page 33.
- Create an Adobe Bridge `Dialog` object to display HTML controls. See 'Displaying HTML in Adobe Bridge dialogs' on page 36.

Content pane

The Content pane display is determined by the *presentation mode* of the browser. Access the current presentation mode with `Document.presentationMode`, and set it, using `Document.setPresentationMode()`. The presentation mode determines how the Content pane interprets and displays the value of `Document.presentationPath`.

Mode	Path	Content pane display
`browser`	A path to a file system location or a location defined by a node-handling extension.	File and folder nodes, represented by `Thumbnail` objects. When a folder is selected in the Folders pane, you can access the current contents of the Content pane through `app.document.thumbnail.children[]`.
`html`	A URL for a local or remote HTML page.	A web page. You can define an HTML page containing user-interface controls.

Tabbed palettes

Your script can add palettes using the `TabbedPalette` object. A script-defined palette can display a user interface defined in ScriptUI or in HTML.

- To display HTML, specify the type `"web"`, and give a URL for the page to show. The URL is kept in the `TabbedPalette.url` property, and your scripts can modify it.
- To display a ScriptUI interface, specify the type `"script"`. In this case, the `TabbedPalette.content` property automatically contains a ScriptUI `Group` object. Use that object's `add()` method to add UI elements.

For complete information on custom palettes that your script defines with the `TabbedPalette` object, see Chapter 4, "Customizing the Adobe Bridge Browser Window.""

Displaying ScriptUI in Adobe Bridge

ScriptUI is a module that defines windows and user-interface controls. There are three ways to display ScriptUI elements:

- You can create an independent ScriptUI window, populate it with ScriptUI controls, and invoke it from your script using the window's `show` function. See 'Displaying ScriptUI dialogs' on page 33
- You can add ScriptUI controls to the existing Navigation bars, and display them by setting by setting the `NavBar` object's `visible` property to `true`.

- You can display ScriptUI controls in a script-defined tabbed palette; see Chapter 4, "Customizing the Adobe Bridge Browser Window."

Displaying ScriptUI dialogs

A script can define a window entirely in ScriptUI, by creating a `Window` object and populating it with ScriptUI controls using its `add` method.

You can invoke a ScriptUI window from a script as a *modal* or *modeless* dialog.

- A modal dialog retains the input focus, and does not allow the user to interact with any other windows in the application (in this case, the browser window) until the dialog is dismissed. The function that invokes it does not return until the dialog is dismissed.
- A modeless dialog does not keep the input focus. The user can interact with the browser window while the dialog is up. The function that invokes it returns immediately, leaving the dialog on screen until the user or script closes it.

In ScriptUI, a modal dialog is a window of type `dialog`, and a modeless dialog is a window of type `palette`. (Do not confuse this ScriptUI term with the Bridge tabbed palette, which is part of the browser and is represented by the `TabbedPalette` object.)

- Invoke a `dialog`-type window as a modal dialog using the `window` object's `show` function. In this case, the function does not return until the user dismisses the dialog, or you close it from a control's callback using the window's `hide` or `close` function. The `close` function allows you to return a value, which is passed to and returned from the call to `show`.
- Invoke a `palette`-type window as a modeless dialog using the `window` object's `show` function, which returns immediately, leaving the window on screen. The user can close the window using the OS-specific close icon on the frame, or you can close it from the script or a control's callback using the window's `hide` function.

The usage of the ScriptUI objects is discussed fully in the *JavaScript Tools Guide*.

Displaying ScriptUI elements in a navigation bar

To display ScriptUI controls, set the `type` property to `"scriptui"`, then use the `NavBar.add` method to add controls. This is the same as the ScriptUI `Window.add` method.

- For an example of this, see the script `SnpAddScriptUINavBar.jsx`, included in the Adobe Bridge SDK.
- For detailed information on using the ScriptUI objects, see the *JavaScript Tools Guide*.

Displaying ScriptUI elements in a custom palette

To display ScriptUI controls, set the `type` property to `"script"`. The `TabbedPalette.content` property automatically contains a ScriptUI `Group` object. Use that object's `add()` method to add UI elements.

- For an example of this, see the script `SnpCreateTabbedPaletteScriptUI.jsx`, included in the Adobe Bridge SDK.
- For detailed information on using the ScriptUI objects, see the *JavaScript Tools Guide*.
- For further details on creating customized tabbed palettes, see Chapter 4, "Customizing the Adobe Bridge Browser Window."

Displaying HTML in Adobe Bridge

There are four mechanisms you can use to display an HTML UI within Adobe Bridge:

- A top or bottom `NavBar` displays HTML when `navBar.file` is set to the path of the HTML file, and `navBar.type="html"`.
- A `Dialog` object always displays HTML UI controls (as opposed to a ScriptUI dialog object, which displays ScriptUI controls). You specify the HTML file to display as the argument when creating the Dialog object. For example:

  ```
  var myDialog = new Dialog("/C/BridgeScripts/HTML/dialogUI.html");
  ```

- When you set `Document.presentationPath` to the path of an HTML file, and `Document.presentationMode="html"`, the HTML page is displayed in the Content pane.
- You can display HTML controls in a script-defined tabbed palette; see Chapter 4, "Customizing the Adobe Bridge Browser Window."

In order to display the HTML, Adobe Bridge opens an embedded browser, which runs a standard JavaScript engine in a different process from the Adobe Bridge ExtendScript engine. The standard JavaScript engine can access only the standard HTML object model. A script on the HTML page cannot directly access the Adobe Bridge object model, or make use of ExtendScript features such as localization.

For a script in your UI page to communicate with the Adobe Bridge object model, the HTML JavaScript engine and the Adobe Bridge ExtendScript engine must exchange values via *remote calls*.

- For the JavaScript code to make remote calls to ExtendScript, you define *callback* functions on the Adobe Bridge object, and invoke them from the HTML page with the JavaScript `call` function. The callback functions access Adobe Bridge objects on the Adobe Bridge side and pass values back to the HTML page. See 'Defining callbacks for HTML scripts' on page 34.
- Your HTML page can define its own JavaScript functions in a script. For the Adobe Bridge side to use these functions, it must make a remote call using the Adobe Bridge object's `execJS` function. See 'Executing script functions defined on HTML UI pages' on page 35.

The three mechanisms for displaying an HTML UI differ slightly in the details of how you define and pass callbacks and invoke script-defined functions. This section provides examples for a web page displayed in the Content pane, in response to selecting a web-type thumbnail. Examples for navigation bars and dialogs are given with the discussions of those objects above.

When you make remote calls, you can pass simple values such as strings and numbers directly. However, in order to pass complex values such as objects and arrays, you must deconstruct them on the passing side using the JavaScript function `toSource`, and reconstruct them on the receiving side using the JavaScript function `eval`. Examples are given for callbacks; see 'Passing Complex Values in Remote Calls' on page 38. The embedded browser does not support `toSource`, so you cannot pass complex values from the HTML page back to the Adobe Bridge ExtendScript engine.

Defining callbacks for HTML scripts

If you want to make use of Adobe Bridge object values to dynamically alter the HTML controls as the user works with them, you must make calls back to the Adobe Bridge object model through a set of callbacks that you define. The exact way that you define and store the callbacks depends on which of the HTML mechanisms you are using:

- Defining callbacks for a dialog

  For a dialog, you define callback functions in a structure that you pass to the `Dialog.open` or `run` function when you invoke the dialog. The syntax for the callbacks argument is:

  ```
  {
   fn_name1: function( args ) { fn1_definition },
   fn_name2: function( args ) { fn2_definition }
  }
  ```

  The dialog's HTML page can invoke these functions using the JavaScript `call` method. See the 'Using callbacks in an HTML dialog' on page 37.

- Defining callbacks for the Content pane or navigation bar

  For HTML displayed in a navigation bar or in the Content pane, you define callback functions in the `jsFuncs` property of the appropriate object:

  - When a `Thumbnail` object displays an HTML page in the Content pane, the `Document.jsFuncs` property stores callback functions for that page. See the examples given below.
  - For a page displayed in a navigation bar, the callbacks are stored in the `NavBar.jsFuncs` property. For examples, see 'Displaying HTML in a navigation bar' on page 37.

From the HTML page, you can invoke your defined callback functions using the JavaScript `call` function. Typically, you will do this from a control's event handler, such as the `onClick` method for a button. For example, suppose one of your callbacks is defined as:

```
{ myCB: function(x) { return x > 0 } }
```

This defines a function named `myCB`. Within the HTML page's JavaScript, invoke the `myCB` Adobe Bridge object method as follows:

```
var positive = call("myCB", 29);
```

You must use the JavaScript `call` method to invoke callback functions. You cannot simply invoke them by name.

A callback function can access the Adobe Bridge object model and pass back a response, as shown in the examples. The callback functions can receive and return simple types directly, but must use `eval` to reconstruct complex types passed as arguments from the HTML side, and use `toSource` to serialize complex types that you wish to return. See 'Passing Complex Values in Remote Calls' on page 38.

Executing script functions defined on HTML UI pages

An HTML page that displays user-interface controls within Adobe Bridge can itself contain a script that defines functions. The `execJs` method (defined on the `Document`, `NavBar`, and `Dialog` objects) allows an Adobe Bridge script to invoke a JavaScript method defined in an HTML page.

- When a `Thumbnail` object displays an HTML UI in the Content pane, use the `Document.execJS()` method to execute functions defined in the script for that page. See the example below.
- For a page displayed in a navigation bar, use the `NavBar.execJS()` method to execute functions defined in the script for that page.
- For a modeless HTML dialog, use the `Dialog.execJS()` method to execute functions defined in the script for the dialog.

You should make sure that the HTML page which defines the remote function is actually loaded before you invoke the remote function with `execJS`.

Caution: You cannot call the `execJS` method from within a callback function. Doing so causes Adobe Bridge to hang. For an alternative, see 'Scheduling Tasks from Callbacks' on page 38.

The `execJS` method takes as its argument a string that contains the entire function call to be executed. For example, this JavaScript code packages a call to the function `updatePath`, defined in the HTML displayed by `myDialog`:

```
myDialog.execJS("updatePath('" + escape(tn.path) + "')");
```

In this case, it is passing a pathname that contains the backslash (\), which is an escape character. It uses the `escape` function to create the argument string, and on the HTML side, the `updatePath` function uses `unescape` to retrieve the path from the argument string:

```
<script> //define fns to be called from Bridge
  function updatePath(path) { window.path.value = unescape(path) };
</script>
```

The technique is exactly the same for a dialog as for a navigation bar, except for calling the function in the dialog object. See 'Calling functions defined in an HTML navigation bar' on page 37.

Displaying HTML in Adobe Bridge dialogs

The `Dialog` object represents a window that displays an HTML page. The window can be modal or modeless. A modal dialog prevents user interaction in other windows while it is open.

- Use the `Dialog.run()` function to open a modal dialog. This function does not return until the user dismisses the dialog. It then returns `true`.
- Use the `Dialog.open()` function to open a modal or modeless dialog. This function returns immediately, and the dialog remains on screen until the user or script dismisses it.

Both invocation functions takes as an argument a set of callback functions. These callbacks are used to respond to a dialog-closing event, and to provide the dialog's HTML JavaScript code with access to the Adobe Bridge object model (see 'Communicating with Adobe Bridge from dialog JavaScript' on page 36).

You can provide a special callback function named `doClose`. If provided, this is called automatically when the dialog closes in response to a user action (such as clicking the window's close icon, or clicking your button that sets `closing` to `true`). The `doClose` function takes no arguments and returns a Boolean value. The callback is *not* called when your script closes a modeless dialog using its `close` method.

Communicating with Adobe Bridge from dialog JavaScript

The HTML page displayed in a dialog runs its own JavaScript engine, which has access only to the HTML object model. If the page needs to exchange data with your Adobe Bridge objects, you must use the remote call mechanisms.

- You can define callbacks and pass them to the dialog as arguments to the `run()` or `open()` function that invokes it. You can call them remotely from the HTML page's JavaScript using the JavaScript `call` function. These callbacks provide access to the Adobe Bridge objects. See 'Using callbacks in an HTML dialog' on page 37.
- You can define JavaScript functions in the HTML page's script, and call them remotely from Adobe Bridge using the `Dialog.execJS()` function—as long as you do not call them directly from a

callback. See ‘Calling functions defined in an HTML dialog’ on page 37 and ‘Scheduling Tasks from Callbacks’ on page 38.

Simple values such as strings and numbers can simply be passed back and forth, as shown in the examples below. Complex value such as arrays or objects must be deconstructed and reconstructed, using `toSource` and `eval`. For details of how to pass objects in remote function calls, see ‘Passing Complex Values in Remote Calls’ on page 38.

Using callbacks in an HTML dialog

The callback functions that you define for a dialog are available to the code in the HTML page, which can invoke them using the `call` function. They run in Adobe Bridge’s JavaScript engine, and can use Adobe Bridge objects.

For example, suppose the `callback` argument that you pass to the `open()` function has the value:

```
{ isGreater: function(x) { return x > myDialog.height } }
```

A method in the HTML page (an event handler, for instance) can invoke the function and receive the result as follows:

```
var newHeightOK =  call("isGreater", 29);
```

Calling functions defined in an HTML dialog

If the HTML page displayed in the dialog defines any JavaScript functions of its own, Adobe Bridge can make remote calls to those functions using the `Dialog.execJS()` function. For a modeless dialog, this works exactly the same way as for a navigation bar or a page displayed in the Content pane. See ‘Calling functions defined in an HTML navigation bar’ on page 37.

Caution: You cannot call the `execJS` method from within a callback function. Doing so causes Adobe Bridge to hang. For an alternative, see ‘Scheduling Tasks from Callbacks’ on page 38.

Displaying HTML in a navigation bar

To display HTML controls, set, set the `type` property to `"html"`, and the `file` property to the HTML file that defines the page you want to display.

When a navigation bar displays HTML, it runs its own JavaScript engine, which has access only to the HTML object model. If the page needs to exchange data with your Adobe Bridge objects, you must use the remote call mechanisms.

- You can define and store callbacks in the `NavBar.jsFuncs` property and call them remotely from the HTML page’s JavaScript using the JavaScript `call` function. These callback provide access to the Adobe Bridge objects.
- You can define JavaScript functions in the HTML page’s script, and call them remotely from Adobe Bridge using the `NavBar.execJS()` function—as long as you do not call them directly from a callback. See ‘Calling functions defined in an HTML navigation bar’ on page 37 and ‘Scheduling Tasks from Callbacks’ on page 38.

Calling functions defined in an HTML navigation bar

If the HTML page displayed in a navigation bar defines any JavaScript functions of its own, Adobe Bridge can make remote calls to those functions using the `NavBar.execJS()` function, as shown in the Bridge

SDK example `SnpAddHTMLNavBar.jsx`. If an Adobe Bridge HTML dialog displays the HTML page, use the same technique, but call the `Dialog.execJS()` function.

Before you use `execJS` to call functions defined in the HTML JavaScript code, you need to make sure that the page is loaded, so that the functions are defined when you call them. You can use the HTML/JavaScript `onloaded` event defined on the `BODY` tag to invoke a callback (defined in the `NavBar`'s `jsFuncs` property). You can, for example, set a global variable, which the event handler can check before calling the remote functions.

Note: JavaScript uses the backslash (\) as the escape character, but the backslash is part of Windows platform path names. Therefore, in order to pass a path name value, a script must use the JavaScript `escape` function to encode the name it sends to HTML. On the HTML JavaScript side, `unescape` decodes the string so it is properly displayed in the UI with the backslash character.

Displaying HTML in a custom palette

To display HTML, specify the type `"web"`, and give a URL for the page to show. The URL is kept in the `TabbedPalette.url` property, and your scripts can modify it.

- For an example of this, see the script `SnpCreateTabbedPaletteHTML.jsx`, included in the Adobe Bridge SDK.
- For further details on creating customized tabbed palettes, see Chapter 4, "Customizing the Adobe Bridge Browser Window."

Passing Complex Values in Remote Calls

To exchange simple values such as strings and numbers between Adobe Bridge and an HTML UI page, you can simply pass arguments and return values of those types in your callback and `execJS` functions. However, complex values such as objects and arrays must be broken down and reconstructed on the other side. This is true for communication in both directions—callbacks from HTML to Adobe Bridge, and execution of HTML script functions by Adobe Bridge using `execJs`.

For a callback to receive an object as an argument, the calling function on the HTML side must serialize the object into a string, using `toSource`, and pass the serialized string. On the Adobe Bridge side, the callback function uses `eval` to reconstruct the object from the serialized string. Similarly, to pass an object back, the callback function must use `toSource` to serialize the object and return the serialized string. The receiving code on the HTML side must in turn reconstruct the object using `eval`.

Note: The embedded browser does not support the `toSource` method, so you cannot use this mechanism to pass complex values to and from HTML-page functions that you invoke using the `execJS` method. Pass only simple values from the HTML JavaScript engine to the Adobe Bridge ExtendScript engine.

Scheduling Tasks from Callbacks

You cannot call the `execJS` method from within a callback function (either stored in a `jsCallbacks` property or passed as an argument to the Dialog `open` or `run` method). This attempts to re-enter the JavaScript engine, which is already running and is not re-entrant. If you try to do this, Adobe Bridge will hang.

The alternative is to schedule a task, using the `app.scheduleTask()` function, from within the callback function. From the function associated with the task, you can call `execJS`. Because it is not executed until the callback returns, the task is free to make another remote call.

Note: If the scheduled script needs to load another script, do not use the JavaScript `eval()` function to do so. Instead use the ExtendScript `$.loadFile()` function; see the *JavaScript Tools Guide* for details.

The first argument to `scheduleTask` is a string containing a script—in this case, a call to the `execJS` function. For example:

```
var result = app.scheduleTask("myFn(3);", 10);
```

If the script itself contains any strings, those must be indicated by enclosed quotes. For example:

```
var result = app.scheduleTask("myFn('string argument');", 10);
```

If the enclosed string contains values derived from expressions, the script string must be concatenated, and can become quite complex:

```
var result = app.scheduleTask("myFn('" + escape(tn.path) + "')", 10);
```

The argument to `execJS` is also a string containing a script—in this case, a call to a function defined on an HTML page. When the arguments to that function are also strings, and those contain values derived from expressions, the resulting string is very complex.

The example below makes this string a little more manageable by breaking it down into modular pieces. First, it builds the argument string for the remote function:

```
var toRecordArg = "'For File: " + tn.metadata.FileName + "'";
```

It uses that to build the entire string for the remote function call, which is the argument to `execJS`:

```
var execFn = "recordData(" + toRecordArg + ")";
```

Building the string for the call to `execJS` requires an additional layer of embedded quotes, which is very difficult to achieve with only two types of quote character. To get around this, the example creates a variable for a string containing the double-quote character, and uses it to build the entire function call string, which is passed to `scheduleTask`:

```
var quote = '"';
app.scheduleTask("app.document.execJS(" + quote + execFn + quote + ")")
```

Note: For the complete script from which these code fragments are taken, see the Bridge SDK example `SnpScheduleTask.jsx`.

4 Customizing the Adobe Bridge Browser Window

The browser window has a set of default panes and palettes that the user can show or hide and drag to different positions. Your scripts can access the contents of these default panes to some extent, as described in 'Accessing the Adobe Bridge browser through scripts' on page 12.

In addition to the default palettes, however, there are two completely script-defined display areas that allow you much greater control:

- Use the `TabbedPalette` object to create entirely new palettes for display or user interface, defined by ScriptUI or HTML. See 'Creating a Customized Tabbed Palette' on page 40.
- Use the `InspectorPanel` object to create object-inspector panels, which provide additional information related to the currently selected thumbnail in the Content pane. See 'Script-Defined Inspector Panels' on page 41.

You can also customize the browser by adding your own menus, submenus, and commands to the default Adobe Bridge menu bar and menus. See 'Extending Adobe Bridge Menus' on page 44.

Creating a Customized Tabbed Palette

The default configuration of Adobe Bridge provides a number of tabbed palettes that the user can open and close, and resize or move into different combinations. They are arranged in three columns, and the user can drag any of the palettes into any of the columns, or any vertical position within a column.

Your script can add palettes using the `TabbedPalette` object. A script-defined palette can display a user interface defined in ScriptUI or in HTML.

- To display HTML, specify the type `"web"`, and give a URL for the page to show. The URL is kept in the `TabbedPalette.url` property, and your scripts can modify it.
- To display a ScriptUI interface, specify the type `"script"`. In this case, the `TabbedPalette.content` property automatically contains a ScriptUI `Group` object. Use that object's `add()` method to add UI elements. See the *JavaScript Tools Guide* for information on ScriptUI.

  You can specify an `onResize()` method for the `Group` object, which will be used to automatically resize the elements when the palette is resized. See the *JavaScript Tools Guide* for details.

You can add a palette to any existing browser, or use the document `create` event to add your palette to new browser windows; see 'Event Handling in Adobe Bridge' on page 27.

A script-defined palette is always added at the top of the leftmost column, and its name is automatically added to all relevant menus. You cannot specify where the palette goes, or move it programmatically. When it is shown, however, it can be dragged and dropped like the default palettes.

You can save a workspace with a script-defined palette that has been moved to a non-default position; however, before your script can reload the workspace successfully, it must recreate the palette objects and contents. See the example `SnpLoadSavedWorkspace.jsx`.

You can get a list of all defined palettes, including both default and script-defined ones, from `app.document.palettes`. You can show or hide any palette from a script by setting the `TabbedPalette.visible` property to `true` or `false`. To remove the palette permanently and destroy the object, use the `TabbedPalette.remove()` method.

➤ Code examples

The sample code distributed with the Adobe Bridge SDK includes these code examples that demonstrate how to define tabbed palettes:

Tabbed palette examples in *`sdkInstall`*`/sdksamples/javascript/`	
`SnpCreateTabbedPaletteScriptUI.jsx`	Shows how to create a tabbed palette containing ScriptUI components, in response to browser-creation event.
`SnpCreateWebTabbedPalette.jsx`	Shows how to create a tabbed palette containing HTML components, in response to browser-creation event.
`SnpLoadSavedWorkspace.jsx`	Shows how to defined a workspace that contains a script-defined palette, save the workspace, and reload it after recreating the palette.

Script-Defined Inspector Panels

An *object inspector* is a script-defined panel which provides context for the selected thumbnail by displaying information related to that node. As a user browses through thumbnails in the Content pane, each new selected thumbnail becomes the inspection focus of the panel. The Inspector panel can show simple text or `Thumbnail` property values, or more indirectly derived node information.

There is no default content in an inspector panel; the information that it shows and the way it retrieves that information from the focus thumbnail is entirely script defined. A browser can display one inspector panel, represented by an `InspectorPanel` object; the panel can contain multiple subpanels, represented by subtypes of the `Panelette` class. The type of node-related information you can display in the subpanels is extremely flexible. You can specify information that is directly or indirectly linked to the focus thumbnail through layers of pointers in metadata, or you can calculate display values from the linked data.

Like the `TabbedPalette`, the panel is placed by default in the upper left palette position. After its initial, automatic placement, the user can open and close it and drag it to other palette locations, like any other palette.

The `InspectorPanel` object acts as a container for one or more subpanels, represented by types of `Panelette`. The different types of subpanels display, in various ways, information or other nodes that are related to the inspected node through the `Thumbnail`, `Metadata` or `Infoset` objects.

You define the display format, and generate dynamic values using Adobe Bridge- or script-defined properties of the inspected thumbnail. You can access embedded metadata for a file through the `Metadata` object in the `Thumbnail.metadata` property.

The type of `Panelette` are:

- `TextPanelette`: Displays a simple block of static or dynamic text.
- `IconListPanelette`: Displays two or three columns. The first contains an icon, and the others contain static or dynamic text.
- `ThumbnailPanelette`: Displays resizeable thumbnail icons, plus a set of text items for each thumbnail.

Text in any of these subpanels can be specified with literal strings, or derived dynamically at display time from the inspected thumbnail, its metadata, and its related `Infoset` properties; or values can be otherwise calculated using JavaScript.

Dynamic text is specified using special *panelette markup elements*, indicated by double-brackets. See 'Specifying string values in an inspector panel' on page 42.

Creating and displaying inspector panels

To create an inspector panel, create the `InspectorPanel` object and its subpanel objects, using the subclasses of the `Panelette` base class. Add each subpanel to the panel, then add your Inspector panel to the set of available panels for Adobe Bridge:

```
myPanel = new InspectorPanel("MyPanel", "My Panel");
myTextSubpanel = new TextPanelette ("MyText", "More about your selection",
  "[[this]]", [["key1: ", "value 1"]["key2: ", "value 2"]]);

myPanel.registerPanelette(myTextSubpanel);
app.registerInspectorPanel(myPanel);
```

Registered panels appear in the various view menus, and the user can choose whether and where to display them. To turn the display of registered panels on or off programmatically in a particular browser window, use `Document.displayInspectorView`. For example, to show the Inspector panels in the current browser window:

```
app.document.displayInspectorView = true;
```

You can access the currently registered subpanels for a particular panel with `InspectorPanel.panelettes`, and remove them with `InspectorPanel.unregisterPanelette()`.

Similarly, you can access all currently registered Inspector panels with `app.inspectorPanels`, and remove them with `app.unregisterInspectorPanel()`.

➤ Code examples

The sample code distributed with the Adobe Bridge SDK includes these code examples that demonstrate how to define object-inspection panels:

Inspector panel examples in *`sdkInstall`*`/sdksamples/javascript/`	
`SnpCreateTextInspectorPanel.jsx`	Shows how to display static and dynamic text in an Inspector panel.
`SnpCreateIconInspectorPanel.jsx`	Shows how to display text with icons in an Inspector panel.
`SnpCreateThumbInspectorPanel.jsx`	Shows how to display thumbnails with associated text in an Inspector panel.

Specifying string values in an inspector panel

A display string in a `TextPanelette` or `ThumbnailPanelette` is specified as a key-value pair, an array with two string elements:

```
[ key_string, value_string ]
```

The key string is displayed in bold text on the left of a text field in the subpanel, and the value string is displayed in plain text on the right.

These strings, and all other display strings in an Inspector panel, including panel-title and tab-title strings, can be specified as literal text, or can combine literal text with markup elements that calculate or retrieve values dynamically and concatenate them into display strings.

Panelette markup elements for dynamic text

You can specify dynamic or calculated string content to be displayed in the subpanels, or in the title string of the panel (`InspectorPanel.displayTitle`) or subpanels (`Panelette.displayTitle`). To specify these special string values, you use *panelette markup elements*.

Markup elements are enclosed by double brackets:

```
[[markupElement]]
```

The special markup element `[[this]]` is a variable that refers to the currently selected thumbnail. For example, here the variable is used to derive the *`thumbnails`* parameter for the subpanel constructor, which identifies the subpanel's focus object or objects:

```
var tp = new TextPanelette("SnpCreateTextInspectorPanel Text Panelette",
  "Information about thumbnails", "[[this]]", keyValuePairs);
```

When specifying text in a subpanel, markup can indicate:

- **Dynamic text**: Dynamic text values are retrieved from the thumbnail's associated node data. To insert a dynamic value retrieved from node data, use a markup element that identifies the `ExtensionHandler`, `Infoset`, and member element:

  ```
  [[extensionName.infosetName.elementName]]
  ```

- **JavaScript**: Values can be retrieved or calculated at display time using JavaScript. To specify a dynamically calculated value, embed JavaScript within the content string, using this tag:

  ```
  [[javascript:code]]
  ```

  A function in this context is not allowed to block; if it takes more than 10 milliseconds, the display string is converted to an error string.

  Within the context of the markup tag, you can refer to the currently selected `Thumbnail` object using a special variable `inspectorThumbnail`. This is useful for accessing embedded file metadata. For example:

  ```
  [[javascript:"Name: " + inspectorThumbnail.name]]
  [[javascript:"Author: " + inspectorThumbnail.metadata.author]]
  ```

- **Hyperlinks**: To insert a hypertext link element in a string, use the format:

  ```
  [[Link text][URL]]
  ```

  For example:

  ```
  [[Click here][http://www.myURL.com]]
  ```

Markup examples

Display strings can be constructed from literal text, markup elements, or a combination. For example:

```
["Checkin: ","[[versioncue.versionData.checkinDate]]"]
```

The key field here is a string literal. The value field displays the string retrieved from the indicated node data field: "3/10/2007 12:32 pm". You can also combine dynamic and literal text in a single field. For example:

```
["State: ", "Checked in at" + "[[versioncue.versionData.version]]"]
```

Here, the value field concatenates the string literal to the retrieved string: "Checked in at 3/10/2007 12:32 pm".

To retrieve values from application, browser window, or thumbnail properties, access them using JavaScript. You can also use JavaScript to calculate a value at display time. For example, this element concatenates a string literal to a string retrieved from a property:

```
[[javascript:"Siblings: " + app.document.thumbnail.children.length]]
```

Extending Adobe Bridge Menus

The `MenuElement` class is used to represent Adobe Bridge application menu bars, their menus and submenus, and individual items or commands. Each Adobe Bridge host application creates `MenuElement` instances for each of the existing menu elements, and you can create additional instances to extend the existing menus.

Each `MenuElement` object has a unique identifier. Existing menu elements that can be extended have predefined identifiers, listed in the *Adobe Bridge JavaScript Reference*. Not all existing menu elements can be extended.

- A script can execute an Adobe Bridge menu command using `app.document.chooseMenuItem(`*`menuId`*`)`.
- To add a menu, submenu, or command, create a `MenuElement` instance, providing the identifier of an existing element, and a new identifier for the new item and a localized display string. The new item is placed relative to the existing item.

The menu, submenu, and command identifier names do not necessarily match the display names. Menu identifiers are case sensitive. They are not displayed and are never localized. When a script creates a new menu or command, you should assign a descriptive unique identifier. If you do not, one is generated using an alphanumeric value in the form "JavaScriptCommand*0..n*".

The display text of a new menu element can be localized by specifying it with the global `localize` function, provided by ExtendScript; see the *JavaScript Tools Guide*.

Menu separators are not independent elements, but can be inserted before or after an element that you add to a menu. Specify a separator on creation of the menu element, by putting a dash character (-) at the beginning or end of the location string.

➤ Code examples

The sample code distributed with the Adobe Bridge SDK includes these code examples that demonstrate how to extend Adobe Bridge menus:

Menu modification examples in *`sdkInstall`*`/sdksamples/javascript/`	
`SnpAddMenuItem.jsx`	Shows how to add a new menu element to Adobe Bridge.
`SnpAddContextMenuItem.jsx`	Shows how to add a context-sensitive menu item when a folder or file is selected.

5 Extending Adobe Bridge Node-Handling Behavior

This chapter is intended for advanced users, tool developers who want to create plugin-type extensions to basic Adobe Bridge functionality, in order to integrate Adobe Bridge with other applications and systems.

The basic behavior of a thumbnail icon is determined by a *node handler*. A node handler determines how Adobe Bridge performs navigation for each thumbnail—that is, exactly what happens internally when a user opens, closes, clicks, double-clicks, or otherwise interacts with the `Thumbnail` object in an Adobe Bridge pane.

The core node handler provides the default behavior, such as displaying the contents of a folder, or displaying a web page (see 'Thumbnails as node references' on page 23). Other Adobe applications that are integrated with Adobe Bridge, such as Version Cue and Stock Photos, define their own node handlers and node types.

You can extend the node handling behavior to handle new types of nodes that you define. To do this, you define a node-handling *extension*, using the `ExtensionHandler` and `ExtensionModel` objects.

- 'Creating Script-Defined Node-Handling Extensions' on page 45
- 'Defining Long-Running Operations' on page 56
- 'Defining Node Data Sets' on page 54
- 'Defining Node Searches and Filters' on page 61

Creating Script-Defined Node-Handling Extensions

The `ExtensionHandler` object, together with an associated `ExtensionModel` object, defines the properties and callback methods needed to extend the Adobe Bridge node model. To define your own type of node, you create and register an `ExtensionHandler` object, implementing all of the required callback methods; these include a `makeModel()` method that creates the associated `ExtensionModel` object. See 'Defining an ExtensionHandler' on page 49 and 'Defining an ExtensionModel' on page 50.

A *prefix* for the Adobe Bridge node URI identifies node types and associates them with their handlers. For example, the VersionCue handler handles nodes with the prefix "vc:". Your handler can register one or more prefixes for the node types it handles; see 'Registering your node-handling extension' on page 46.

An extension defines one and only one `ExtensionHandler` object. Adobe Bridge uses that object's `makeModel()` method to create an `ExtensionModel` object every time it needs to display a handled node.

Your `ExtensionHandler` object must implement all of the node-handling behavior that you need for your own node types, either in its own methods or in the methods defined for the `ExtensionModel` object it creates.

- Some handler and model methods simply perform the desired action and return a value. For potentially time-intensive or resource-intensive file-system operations, the model method does not perform the action directly. Instead, it defines and returns an `Operator` object that can perform the action, blocking the main thread but allowing for a progress report. When the model method returns, Adobe Bridge uses the `Operator` object to initiate the action at an appropriate point. See 'Defining Long-Running Operations' on page 56.

- Node handlers can associate their nodes with private data, represented by the `Infoset` object. Adobe Bridge defines core data sets that all handlers must support, and you can also define your own node data for your node types. Scripts can access the private node data through the `Thumbnail` object for a node. See 'Accessing the node-handling model and data' on page 48.
- You can define your own searching and filtering criteria for your node types. See 'Defining Node Searches and Filters' on page 61.

Registering your node-handling extension

Register your script-defined `ExtensionHandler` with `app.registerExtension()`. You can access the global list of all registered extensions through the property `app.extensions`.

To associate your node type with your script-defined extension handler, you register a URI *prefix*, using `app.registerPrefix()`. The prefix is prepended to a pathname or URL with a colon separator:

```
myPrefix:myPath/myFile.ext
```

You can register any number of prefixes to identify your handled nodes. All nodes created with a given prefix are managed by the handler that registers that prefix. For example, the `versioncue` extension handler manages nodes whose URIs begin with the string "`vc:`" or "`bridge:versioncue:`".

➤ Code example

The sample code distributed with the Adobe Bridge SDK includes these code examples that demonstrate how to define node-handling extensions:

Node-handling extension examples in `*sdkInstall*/sdksamples/javascript/`

`BasicExtensionHandler.jsx`	Shows how to create a basic node-handler, defining a minimal set of handler and model methods.
`CustomInfosetExtensionHandler.jsx`	Extends the basic example to define node information for the defined node type.
`CustomSearchExtensionHandler.jsx`	Extends the basic and infoset examples to implement a search among handled nodes, using the defined node information.

Installing a node-handling extension

A node-handling extension, like other JavaScript scripts, can run from any location. However, when you are ready to deploy it, you will want it loaded automatically on startup of the application. Startup scripts are automatically loaded from the `Bridge CS3 Extensions` folder, a fixed location shared by all users.

For deployment, use the startup script location to install your extension definition files and all their resources, including JavaScript files, libraries, string files, images, and so on.

- In Windows®, the location is:

  ```
  %CommonProgramFiles%\Adobe\Bridge CS3 Extensions\
  ```

- In Mac OS®, the location is:

  ```
  /Library/Application Support/Adobe/Bridge CS3 Extensions/
  ```

Group all files related to your extension in an appropriately named subfolder. The Workflow Automation Scripts, for example, are installed in a subfolder named `Adobe Workflow Automation`. Your installer should create these folders, including the `Bridge CS3 Extensions` folder if necessary, and install the files as appropriate. Your installer does not need to know where Adobe Bridge is installed.

Installation structure

The subfolder for your extension should have this structure:

```
Bridge CS3 Extensions/
     My Extension/
        My Extension.jsx
        Read Me.txt
        manifest.xml
        Resources/
           Images/
              myext.png
           Scripts/
              001_myext.jsx
              002_myext.jsx
           Plugins/
              myext.dll
```

Your extension's main folder should contain a single JSX file, your main script, whose name is the name of your extension (the same as your folder). This file should do the minimal amount of work to install your extension on Adobe Bridge startup. It is best to make your extension as unobtrusive as possible for users that may not need it. Your main script should load other scripts as necessary on demand; for example, when the user invokes your extension in Adobe Bridge.

- Include a `Read Me` file next your main script, with a brief description of the purpose of your extension and the version number. You can also include, for example, a link to a web site with more information.
- The optional `manifest.xml` contains the extension name, version, and localized description, in the following format:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
     <bridge_extension>
        <name>My Extension</name>
        <version>1.0</version>
        <description locale="en_US">My Extension is amazing!</description>
        <description locale="fr">My Extension est fantastique!</description>
     </bridge_extension>
  ```

  The `<name>` element is required; other elements are optional. The description, if supplied, should be short, no more than two or three sentences.

- The `Resources` folder should contain all support files, such as other scripts, image files, libraries, and so on. You can organize the content as you wish.

  You can include shared libraries, compiled for a specific platform, as resources. Load them into JavaScript as external objects. For example:

  ```
  Folder.current = File ($.fileName).parent;
  new ExternalObject("Plugins/myext");
  ```

  See the `ExternalObject` documentation in the *JavaScript Tools Guide*.

Note: The folder and file names must not be localized, except the `Read Me` file, if you wish. If you do localize this name, be aware that in Mac OS X, some Vietnamese characters trigger a corruption warning from Disk First Aid.

Extension workspaces

Your extension can define a workspace to be automatically loaded and added to the Workspace menu. Workspaces are defined as XML files the `.workspace` extension; for an example of the syntax, save a workspace and examine the resulting file in the user location (see 'The Adobe Bridge browser window' on page 11). For your workspace to be automatically loaded on startup, save it in these locations:

- In Windows, the user workspace folder is:

  ```
  %CommonProgramFiles%\Adobe\Bridge CS3 Extensions\Workspaces\
  ```

- In Mac OS, the user workspace folder is:

  ```
  /Library/Application Support/Adobe/Bridge CS3 Extensions/Workspaces/
  ```

Shared startup scripts

You may need to have a script loaded by other Adobe applications, for example to add a "Go To" command. On startup, Adobe Bridge executes all JSX files that it finds in the installation startup folder. For each platform, there is a startup folder shared by all Adobe applications that support JavaScript.

- In Windows®, the installation startup folder is:

  ```
  %CommonProgramFiles%\Adobe\Startup Scripts CS3\
  ```

- In Mac OS®, the installation startup folder is:

  ```
  /Library/Application Support/Adobe/Startup Scripts CS3/
  ```

If your script is in the shared startup folder, it is executed by all Adobe JavaScript-enabled applications at startup; use the application-specific folder if possible. See the *JavaScript Tools Guide* for details of targeting a startup script to a specific application.

It is recommended that you name the startup script for your extension using the name and version number of the extension, in order to avoid collision with other scripts.

Accessing the node-handling model and data

When Adobe Bridge needs to create a thumbnail of a type that is managed by your handler, it uses the handler's `makeModel()` method to create an instance of your `ExtensionModel` that implements the set of methods it will need to manage your node type (see 'Defining an ExtensionModel' on page 50).

Your model allows you to create and update a set of script-defined properties in the `Thumbnail` objects for your nodes. Data managed by each model is kept in `Infoset` objects. When you define an `Infoset` member, you can access it as a `Thumbnail` property.

The `Thumbnail` object has a property with the same name as each extension handler that manages it. To access a script-defined property value in a `Thumbnail` object, use this format:

```
thumbnailName.handlerName.infosetName.memberName
```

For example, suppose the `myHandler` extension handler is associated with an `Infoset` named `mySet`, which contains a member named `color`. To access this value in `Thumbnail` object `t1`, use:

```
currentColor = t1.myHandler.mySet.color;
```

Adobe Bridge defines a core set of data in predefined `Infoset` objects, and a core set of node-handling methods. Your handler must support core data and methods, but can add new data sets and methods. See 'Defining Node Data Sets' on page 54.

Defining an ExtensionHandler

Your `ExtensionHandler` instance must implement all of the methods described here. Handler methods can be immediate or long-running:

- Immediate handler operations simply perform an operation and return when it is done. These functions must not take a significant amount of time; if they are slow, they will negatively affect Adobe Bridge browsing performance.
- Long-running handler operations create and return `Operator` objects to perform time-intensive file-system operations that block the main thread. Adobe Bridge view code or your display code passes the object to `app.enqueueOperation()` to initiate the action when appropriate.

Immediate handler operations

Method	Return type
getBridgeUriForPath (**path** : String) Returns a unique Bridge URI for the supplied path, parsing it as needed. The URI must begin with a registered prefix for this handler, but can otherwise include any path string. If the path cannot be parsed, return "undefined".	String
getBridgeUriForSearch (**scope** : Thumbnail, **specification** : SearchSpecification) Performs the specified search within the container node specified by scope, and returns the Bridge URI for a container node that contains the matching nodes. Called when the user initiates a search in a handled node.	String
makeModel (**path** : String) Creates a model instance that implements this extension for a specific thumbnail. See 'Defining an ExtensionModel' on page 50. Called when Adobe Bridge needs to display a node managed by this handler.	`ExtensionModel`

Long-running handler operations

Implement these functions to create `Operator` objects which can perform the desired operation, and if needed, provide Adobe Bridge with information about the status and progress. See Defining Long-Running Operations.

Method	Return type
acquirePhysicalFile (**sources** : Array of Thumbnail, **timeoutInMs*** : Number, **showUi*** : Boolean, **message*** : String, **recursionOption*** : String) Creates and returns an operator that acquires actual file data for a set of placeholder nodes.	`Operator`
duplicate (**sources** : Array of Thumbnail, **timeoutInMs*** : Number, **showUi*** : Boolean, **message*** : String) Creates and returns an operator that duplicates a set of nodes.	`Operator`

Method	Return type

moveToTrash
`( `**sources**` : Array of Thumbnail, `**timeoutInMs**`* : Number,`
`  `**showUi**`* : Boolean, `**message**`* : String )`
Creates and returns an operator that deletes a set of nodes, marking the associated files for deletion on disc by moving them to the system trash or recycle bin.
	`Operator`

setXmp
`( `**targets**` : Array of Thumbnail, `**xmpPackets**` : Array of String,`
`  `**timeoutInMs**`* : Number, `**showUi**`* : Boolean, `**message**`* : String )`
Creates and returns an operator that embeds XMP file metadata packets in a set of files.
	`Operator`

Defining an ExtensionModel

The `ExtensionModel` that your handler creates implements the actual node-handling methods that perform operations on a selected thumbnail. Model methods can override or extend the default node handling behavior.

These model methods must be implemented:

`initialize()`
`terminate()`
`exists()`
`authenticate()`
	Basic node creation and access.

`refreshInfoset()`
`getCacheStatus()`
	Association and update of script-defined `Thumbnail` properties, defined by `Infoset` objects

A variety of other node operations are scriptable, but you only need to define the ones of interest to you. These can include, for example:

- Node labeling, sorting, and searching behavior.
- Procedures to copy, move, or duplicate nodes.
- Procedures to acquire or create files associated with your nodes.

Your extension must inform Adobe Bridge of which of the optional methods your handler supports by setting the corresponding values in the Core node data sets `item` and `itemContent`. Your handler's `refreshInfoset()` method should do this. For example, if you implement the `createNewContainer()` method, set the corresponding capability value:

```
...
  myModel.refreshInfoset = function( infosetName ) {
    if( infosetName == "item"  ) {
      this.privateData.cacheElement.item.canCreateNewContainer = true;
    }
...
```

Model methods can be immediate or long-running:

- Immediate model operations simply perform an operation and return when it is done.
- Long-running model operations create and return `Operator` objects to perform time-intensive file-system operations that block the main thread. Adobe Bridge view code or your display code passes the object to `app.enqueueOperation()` to initiate the action when appropriate.

Immediate model operations

Method	Return type
`addToDrag` `(` **`pointerToOsDragObject`** `: Number)` Adds this model object to the platform-specific drag object.	Boolean
`authenticate` `( )` Required. Handles any required authentication for this node.	`undefined`
`cancelRefresh` `(` **`infosetName`** `: String )` Cancels a background refresh task started by a call to `refreshInfoset()`.	`undefined`
`createNewContainer` `(` **`name`** `: String)` Creates a new container node in this container node, and returns its URI string. If this node is not a container, does nothing.	String
`exists` `( )` Required. Reports whether this node is valid according to this model.	Boolean

`getCacheStatus`
`(` **`infoset`** `: Infoset,` **`cookie`** `: String )`
Required. Reports the cache status of a node data set for this node. See 'Managing the data cache' on page 54.
	String

`getFileUrl` `( )` Retrieves a file URL for this node.	String
`getDisplayName` `( )` Retrieves a display name for this node.	String
`getFilterCriteria` `( )` Creates the full set of filter criteria that can be applied to this container node. These filters appear in the Filter palette when Adobe Bridge displays the contents of this container.	Array of `FilterDescription`
`getParent``()` Returns the Bridge URI of the parent node of this node.	String
`getSearchDefinition` `( )` Creates a search definition with which to populate the Find dialog when it is invoked for this node.	`SearchDefinition`
`getSearchDetails``( )` Retrieves or recreates the search specification and target node that were used to create this search-result container node, when it was created by the `ExtensionHandler.getBridgeURIForSearch()` method.	`SearchDetails`
`getSortCriteria` `( )` Creates the full set of sorting criteria for member nodes of this container node. Can construct an entirely new list of criteria, or retrieve the default set from `app.defaultSortCriteria` and modify or append criteria, or return the set unchanged.	Array of `SortCriteria`

Method	Return type
`initialize` `()` Required. A constructor for the model instance for this node. Creates any necessary support data structures and stores them in this object. This is the equivalent of a constructor for this model instance. Called by handler's `makeModel()` when Adobe Bridge needs to display a handled node.	`undefined`
`needAuthentication` `( )` Reports whether this node requires authentication.	Boolean

`refreshInfoset`
`(` **`infosetName`** `: String,`
`priority``* : "low" | "high",`
`cost``* : guaranteedLowCost | lowCostEvenIfFail |`
`lowCostEvenIfLowQuality | unlimited ,`
`pageNumber``* : Number )`
Required. Starts a background task with the specified priority and processing cost, to update the data in an node data set for this node. Called when any related `Infoset` or `CacheData` values change.
The `pageNumber` parameter is used for nodes that represent multi-page documents; for other node types, it is ignored.
Within this method, access each data element in the stored data cache, using this format (assuming you have stored the cache reference in the `privateData` property):
`this.privateData.cacheElement.`*`setName`*`.`*`memberName`*
● The operation must set the appropriate core `item` and `itemContent` capability values to reflect which optional model methods are supported by this handler. See 'Defining an ExtensionModel' on page 50 and 'Core node data' on page 55.
● If the node is a container, the operation must add its child nodes to the the core `children` data set.
● The operation must set the cache status. See 'Managing the data cache' on page 54.
	`undefined`

`registerInterest` `(` **`cacheElement`** `: CacheElement )` Required. Notifies this model object of the cache that contains the model itself and all its associated data. Your implementation must store the cache object, and use it to access the node data. Typically, you store it in the model's `privateData` property. Adobe Bridge instantiates the cache element and passes the object to this method whenever it displays a handled node.	`undefined`
`setName` `(` **`newName`** `: String )` Sets the file name of this node. Changes the base name and extension, but does not affect the path name. Returns the new URI for the node.	String
`terminate` `( )` Required. A destructor for the model instance. Cleans up any private data created by the initialization.	`undefined`

Method	Return type
unregisterInterest (**cacheElement** : CacheElement) Required. Removes the association between this model and the cache element that contains it. Your implementation must remove the stored reference to the cache object, typically in the model's `privateData` property.	`undefined`
verifyExternalChanges ( ) Called when the user attempts to view data in this model's `children` core data set, and its cache status is good. The model can decide whether to force a refresh or not.	`undefined`
wouldAcceptDrop (**type** : "copy" | "move" , **sources** : Array of String, **osDragRef** : Number) Reports whether this node can accept a drop of a specific set of nodes in a drag-and-drop operation of a particular type. Adobe Bridge passes a pointer to a platform-specific structure containing the nodes to be dropped. This is the same set of nodes as specified in the sources array, but a handler might prefer this format. Your implementation must return the type string if the drop of that type can be accepted, or `false` if it cannot.	String

Long-running model operations

Implement these methods to create `Operator` objects which can perform the desired operation, and if needed, provide Adobe Bridge with information about the status and progress. See 'Defining Long-Running Operations' on page 56.

All of these methods are optional.

Method	Return type
copyFrom (**sources** : Array of Thumbnail, **timeoutInMs*** : Number, **showUi*** : String, **message*** : String, **newNames*** : Array of String) Creates and returns an `Operator` that copies a set of nodes, allowing rename.	`Operator`
eject (**sources** : Array of Thumbnail, **timeoutInMs*** : Number, **showUi*** : String, **message*** : String) Creates and returns an `Operator` that unmounts a path.	`Operator`
moveFrom (**sources** : Array of Thumbnail, **timeoutInMs*** : Number, **showUi*** : String, **message*** : String, **newNames*** : Array of String) Creates and returns an `Operator` that moves a set of nodes, allowing rename.	`Operator`
resolveLink (**sources** : Array of Thumbnail, **timeoutInMs*** : Number, **showUi*** : String, **message*** : String) Creates and returns an `Operator` that resolves the link path for this node; for example, by mounting a volume.	`Operator`

Defining Node Data Sets

Your script can define node data using the `Infoset` object. Data kept in `Infoset` objects is specific to Adobe Bridge. It can be any information related to nodes, as defined by Adobe Bridge itself or by an Adobe Bridge script. By creating and registering an `Infoset`, you add a set of script-defined properties to the `Thumbnail` object for any handled node.

Note: Node data differs from metadata. Metadata is associated with the file that a node represents, external to Adobe Bridge. It is represented by Adobe Bridge in the `Metadata` object. It is stored in XMP format, which includes other standard metadata formats such EXIF.

The `Infoset` mechanism is closely integrated with Adobe Bridge node handling, and can be built into your own node-handling extensions by declaring an `Infoset` object associated with your `ExtensionHandler`, using `app.registerInfoset()`. The `Infoset` object is added to the list in the `ExtensionHandler.infosets` property, and the `Infoset.extension` property contains a reference back to the handler name.

An `Infoset` is a named set of data elements. The name of the set becomes a property of the `ExtensionModel` for its handler, whose name is a property of each handled `Thumbnail`. Thus, you can access the set through the `Thumbnail` object. For example, to access an `Infoset` object named `myInfo` in thumbnail `t1`, where the `myInfo` set is managed by `myExtension`, use:

```
mySet = t1.myExtension.myInfo;
```

Each member element of the set has a name and data type, defined by an `InfosetMemberDescription`. The member name becomes a property of the containing `Infoset`, and you can access the data value, of the corresponding type, through that property.

For example, to access a `color` value in `myInfo` in thumbnail `t1`, where the `myInfo` set is managed by `myExtension`, use:

```
colorValue = t1.myExtension.myInfo.color;
```

➤ Code example

The sample code distributed with the Adobe Bridge SDK includes this code example that demonstrates how to define node information sets:

Node data set example in *sdkInstall*`/sdksamples/javascript/`	
`CustomInfosetExtensionHandler.jsx`	Extends the basic example to define node information for the defined node type.

Managing the data cache

Adobe Bridge tracks all node data for a given node using a *data cache*, represented by the `CacheElement` object. When Adobe Bridge needs to display a node, it instantiates this class with the current complete set of `Infoset` objects, each of which is associated with a `CacheData` object that reflects its *cache status*. The status determines whether any data associated with a set has changed, requiring that set to be updated or *refreshed*.

Within the model methods that you implement, you must use the cache to access all Adobe Bridge-defined and script-defined `Thumbnail` properties.

When Adobe Bridge needs to display a handled node, it calls the handler's `makeModel()` method to create a new `ExtensionModel` object; it stores the model object itself in a `CacheElement`. It then passes this `CacheElement` object to the model method `registerInterest()`, so that the model object itself knows what cache it belongs to.

Your model must implement the `registerInterest()` method to store the cache with the model object, so that your model methods can access the data. In particular, the model's `refreshInfoset()` method must use the stored cache object to update the associated data.

Typically, you store the cache in the model's `privateData` property. For example, to store the cache reference in the model (and remove the reference when the node is no longer displayed):

```
// associate this node with the node data cache
myModel.registerInterest = function( cacheElement ) {
  this.privateData.cacheElement = cacheElement;
}
// dissociate this node from the node data cache
myModel.unregisterInterest = function() {
  this.privateData.cacheElement = undefined;
}
```

Your model methods can then access your data through the cache:

```
this.privateData.cacheElement.myInfoset.myInfosetMember
```

Adobe Bridge calls the model's `refreshInfoset()` method whenever it needs to access the data (or the cache status of data) associated with the node. Your implementation should update the cache status, as well as all other relevant data. For example:

```
    myModel.refreshInfoset = function( infosetName ) {
...   // update the cache status
      if( infosetName == "immediate"  ) {
        this.privateData.cacheElement.immediate.cacheData.status = "good";
      }
...
```

Core node data

Adobe Bridge defines a core set of data in predefined `Infoset` objects. All node-handling extensions must support the core data values that Adobe Bridge uses for its default node handling. For example, a container node must set the core data value `immediate.isContainer` to `true` in order for Adobe Bridge to treat the node as a container.

Core data values that a handler needs to set are generally in the `immediate`, `item`, and `itemContent` sets. These sets include flags for handler capabilities, which you must set if your handler supports a capability. For example, if your handler supports node searches, it should set `item.canSearch`.

A container must store its child nodes in the core set `children`, using the `Infoset.addChild()` function.

These values are available to scripts through the `Thumbnail.core` property; for example:

```
myThumbSize = myThumb.core.immediate.size
```

Within a model method, they are available through the `CacheElement` object, typically stored in `this.privateData.cacheElement`. For example:

```
   myModel.refreshInfoset = function( infosetName ) {
      // retrieve the cache
      thisCache = this.privateData.cacheElement;
...   // update the cache status
      if( infosetName == "immediate"  ) {
         thisCache.immediate.cacheData.status = "good";
      }
...
```

Note: Refreshing of infosets occurs frequently and needs to be very fast. For time-intensive operations, it is better to schedule a task; see 'Scheduling Tasks from Callbacks' on page 38 and the Bridge SDK example `SnpScheduleTask.jsx`.

The *Adobe Bridge JavaScript Reference* provides a complete list of core data `Infoset` and member names, with the description of the `Infoset` object.

Defining Long-Running Operations

For operations that are time- or resource-intensive, such as those that involve file-system access, the override method you define for your model should create and return an `Operator` object.

The `Operator` class itself is a template. You define one of the two types of `Operator` to perform the desired operation, and to provide Adobe Bridge with feedback on the status and progress of the operation. An operation can be one of these types, as specified in `Operator.operatorType`:

- `modal`: Performs a task that blocks the main thread and provides its own user interface. This type of operation is encapsulated in an instance of `ModalOperator`.
- `progress`: Adobe Bridge provides a progress bar, and you can schedule a task to periodically update the progress value. This type of operation is encapsulated in an instance of `ProgressOperator`. Depending on the extension implementation, it can be modal, or can spawn a thread to perform a background task, and return to the main thread.

Your node-handling `ExtensionModel` method, such as `copyTo()` or `moveTo()`, creates and returns an `Operator` object that implements the desired operation. You (or Adobe Bridge) can then pass the object to `app.enqueueOperation()`.

When it is ready, Adobe Bridge calls the `Operator.start()` method of a queued operation to begin execution.

- For a `modal` operation, the `start()` method returns when the operation is completed.
- For a `progress` operation, the `start()` method returns immediately and Adobe Bridge displays the Progress bar, then resumes activity on the main thread. If the user clicks **Cancel** in the Progress bar, Adobe Bridge halts the operation. Otherwise, the operation thread must notify Adobe Bridge of status changes using `app.operationChanged()`. Adobe Bridge then queries the `Operator` object to update the UI appropriately.

Implementing an operation

For a `ModalOperator` object, you must define a `start()` method that executes the entire operation, providing any user interface you require.

To create a `ProgressOperator` object, you must define all of the following methods:

Method	Return type
`getConflictInfo` `( )` Returns a description of a file-system conflict that prevents the operation from being performed on the current thumbnail. This string should be suitable for display in a conflict-resolution dialog.	String

`getOperationStatus` `( )`
Returns the current status of the operation with respect to the current source thumbnail.
	String, one of:
	`incomplete`
	`inCancellation`
	`inConflict`
	`inError`
	`succeeded`
	`cancelled`
	`failed`

`getPercentageComplete` `( )` Returns the percentage of the operation that has currently been completed. Used in displaying the Progress dialog.	Number [0..100]
`getProcessedNodeCount` `( )` Returns the number of source nodes that have been processed so far.	Number

`getProcessingStatus` `( )`
Returns the current overall status of the operation with respect to Adobe Bridge; that is, whether the operation has begun, is still going on, has been paused by the user, or has finished.
	String, one of:
	`notStarted`
	`inProgress`
	`awaitingResume`
	`completed`

`getProgressMessage` `( )` Returns a message suitable for display in the Progress dialog.	String
`getTotalBytesTransferred` `( )` Returns the current number of bytes that have been transferred to the target in the course of this operation.	Number
`getTotalNodeCount` `( )` Returns the total number of source nodes to be operated on.	Number

`getType` `( )`
Returns the subclass type of this operator.
	String, one of:
	`modal`
	`progress`

`resume` `( )` Sets up the operator to resume execution after it has been interrupted.	Boolean, `true` if the operation can be resumed.

`resolveConflict`
`( `**`method`**` : "noOverride" | "abort" | "override" |`
`"overrideConditionally",`
`policy`` : "applyForOneConflictOnly" | "applyToAllConflicts" )`
Implements a conflict resolution selected by the user in the Adobe Bridge conflict-resolution dialog. See 'Resolving conflicts' on page 59.
	`undefined`

Method	Return type
start ( ) Begins execution of the operation after Adobe Bridge enqueues it, following a call to `app.enqueueOperation()`.	`undefined`
stop ( ) Halts execution of the operation when the operator is stopped by user interaction in the Progress dialog.	`undefined`

Monitoring operation progress and status

For a `ProgressOperator`, Adobe Bridge automatically displays a Progress dialog (unless you specifically disable UI). In the course of performing the operation, you can update the `Operator` object with the current status, and inform Adobe Bridge using `app.operationChanged()`. You can simply make this call as needed, or, for example, use `app.scheduleTask()` to perform the update periodically.

When Adobe Bridge is informed of a change, it gathers information about the change by checking properties and calling developer-defined methods in the `Operator` object. It then updates the Progress dialog, and, if necessary, invokes the conflict-resolution dialog; see 'Resolving conflicts' on page 59.

Many of the `Operator` properties and methods, such as `getPercentageComplete()`, report the current progress in various ways. There are also two general types of status to maintain, the *operation* status and the *processing* status.

- An operation status describes the result of the attempt to perform the specific action (copying, for example) on the current source thumbnail. Adobe Bridge uses the developer-defined method to update this value during the processing of each source thumbnail.

  The developer-defined `Operator.getOperationStatus()` method must return one of these status strings:

`incomplete`	The operation is still in progress for the current thumbnail.
`inCancellation`	`Operator.cancelRequested` is set to `true` when the user requests cancellation through the Progress dialog.

`inConflict`	Set `Operator.conflictType` to identify the type of conflict:
	`userConfirmationRequired`
	`fatal`
	Set `Operator.conflictMessage` to specify a string to display in the conflict-resolution dialog. Use the following string values to identify predefined Adobe Bridge error messages:
	`readOnlyFile`
	`readOnlyFileExists`
	`targetFolderExists`
	`fileExists`
	`sameFile`
	`sameFolder`
	`moveToChild`
	`sourceNotAvailable`
	`storageFull`
	`sourceAccessDenied`
	`targetAccessDenied`
	`unknown`

`inError`	Set `Operator.errorTarget` to the source `Thumbnail` for which the error occurred.
`succeeded`	The operation succeeded for the current source thumbnail.
`cancelled`	The operation was cancelled by the user in the Progress dialog.
`failed`	The operation failed for the current source thumbnail.

- Processing status describes the current overall status of the operation thread with respect to Adobe Bridge; that is, whether the operation has begun, is still going on, has been paused by the user, or has finished.

  The developer-defined `Operator.getProcessingStatus()` method must return one of these status strings:

`notStarted`	Adobe Bridge has not yet called the `start()` method of this `Operation`.
`inProgress`	Adobe Bridge has called the `start()` method of this `Operation` to enqueue the thread.
`awaitingResume`	Adobe Bridge has called the `start()` method of this `Operation`, but has later called the `stop()` method to interrupt processing, typically because the operation has encountered an error condition. The operation can be resumed by calling the `resume()` method.
`completed`	The thread has been terminated, because the operation has completed or timed out, or because it has been cancelled by the user or aborted because of a conflict.

Resolving conflicts

File-system conflicts or errors can prevent the operation from being carried out. Some conflicts are correctable or continuable, others are fatal. If a conflict is not fatal, it can be retried or ignored, after which the operation can attempt to continue.

For a `ModalOperator`, you must supply any user interface you need for error handling and conflict resolution.

For a `ProgressOperator`, your operation can set the `operationStatus` and `conflictMessage`, then call `app.operationChanged()` to notify Adobe Bridge, which provides an interface for error and conflict handling:

- When the file system reports a fatal error that your `Operator` code cannot handle, your handler should set `Operator.operationStatus` to `inError`. Adobe Bridge checks the `Operator.conflictMessage` value, displays an appropriate error dialog, and offers the user the opportunity to halt execution of your operation.
- If a file-system conflict occurs during the execution of your operation should set the `Operator.operationStatus` to `inConflict`. Adobe Bridge then displays the conflict-resolution dialog. This gives the user an opportunity to correct the problem and instruct your handler to retry, or instruct your handler to skip the current action and attempt to continue the operation, or simply to quit the operation.
  - If the user cancels from the dialog, Adobe Bridge halts execution of the operation.
  - If the user chooses to continue, Adobe Bridge passes the user’s choices to your `Operator.resolveConflict()` method. Your implementation of this method must resolve the conflict as directed.

Your `resolveConflict()` method can handle a conflict using the following basic resolution methods., which correspond to choices in the conflict-resolution dialog:

`noOverride`	Do not perform the current action, but continue with the operation. Corresponds to **Skip** in the conflict-resolution dialog.
`override`	Make another attempt to perform the current action. Corresponds to **Replace** in the conflict-resolution dialog.
`overrideConditionally`	Use an extension-defined default style of resolving the conflict. Corresponds to **Auto-resolve** in the conflict-resolution dialog.

The user can also choose a policy by checking or unchecking **Apply to all** in the conflict-resolution dialog. The policy parameter passed to your `Operator.resolveConflict()` method is one of these constants, which correspond to the checked state:

`applyForOneConflictOnly`	Resolve as specified for the current action, but request user input again if the same type of conflict occurs again.
`applyToAllConflicts`	Resolve as specified for the current action, then resolve with this method again if the same type of conflict occurs again.

The constant values passed to your `resolveConflict()` method are also the allowed values of `Operator.resolveMethod` and `Operator.resolvePolicy` properties; however, Adobe Bridge does not check the values of these properties. They are entirely for your own use, in implementing a `ModalOperator` or an operator that suppresses the Adobe Bridge-supplied interface.

Defining Node Searches and Filters

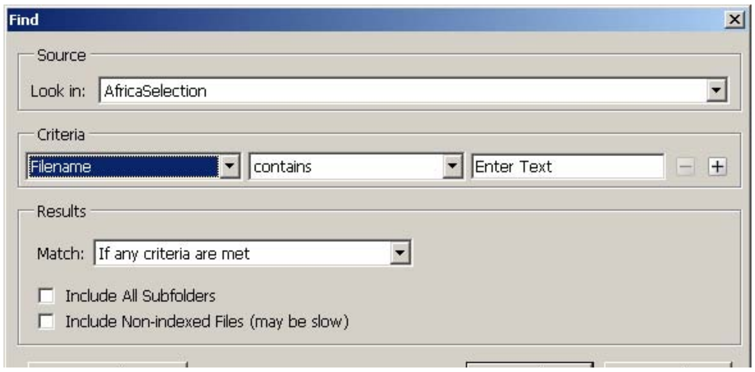
Adobe Bridge allows a user to search for nodes that match a set of criteria that the user selects from the Find dialog. The user can also select filters from the filter panel, which has the same effect as searching for nodes that match the filter criteria.

Your scripts can interact with both of these searching mechanisms. If you define your own node types, you can customize the available search or filter parameters to include node attributes that you have defined.

- Your extension can define a set of search objects and methods that the Find dialog can use to search among nodes that are handled by that extension. See ‘Handling interactive node searches’ on page 61.
- You can customize the Filter palette by adding filters for your own node types. See ‘Adding filters’ on page 63.

Handling interactive node searches

When the user invokes the Find dialog for a handled container node, the dialog calls the model method `getSearchDefinition()` to retrieve the handler-defined `SearchDefinition` object, and uses this object to populate the drop-down lists in the dialog fields.



Look in: AfricaSelection

Filename | contains | Enter Text

Match: If any criteria are met

☐ Include All Subfolders
☐ Include Non-indexed Files (may be slow)

When the user makes selections in the dialog and clicks **Find**, the dialog creates a `SearchSpecification` object from the user selections, and passes it, along with the target node (the node that was selected when the dialog was invoked), to the model’s `getBridgeUriForSearch()` method. The target node tells the function where to search from, and the `SearchSpecification` tells the function what to search for. The search returns a result node, which is a container for all nodes that match the search criteria. Adobe Bridge then displays the contained nodes in the Content pane.

To support searching with the Find dialog for your own node types, you must implement these two methods for your node handler.

➤ Code examples

The sample code distributed with the Adobe Bridge SDK includes this code example that demonstrates how to define node searches:

Node search examples in *`sdkInstall`*`/sdksamples/`	
`CustomSearchExtensionHandler.jsx`	Extends the basic and infoset examples to implement a search among handled nodes, using the defined node information.

➤ **Node search workflow**

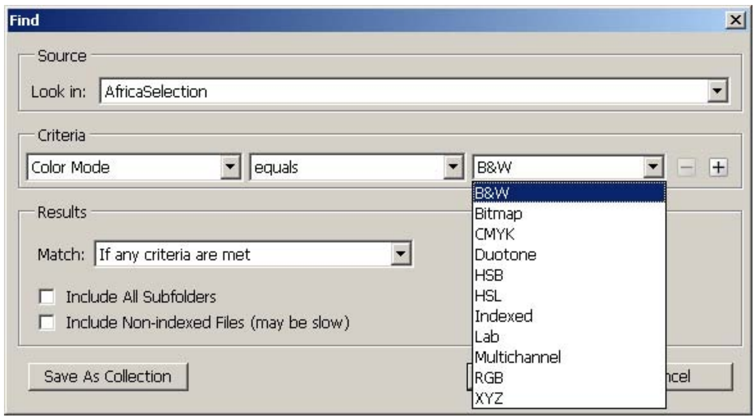
1. User invokes the Find dialog for a handled node.
2. If the selected node is handled by a node-handling extension, the dialog calls the node handler's model method `getSearchDefinition()` to retrieve the extension-defined `SearchDefinition`.
3. Adobe Bridge populates the Find dialog fields according to the `SearchDefinition`.
4. User makes selections in the dialog and clicks **Find**.
5. Adobe Bridge creates a `SearchSpecification` object from user selections in the dialog.
6. Adobe Bridge executes the search query, which calls the node handler's `getBridgeUriForSearch()`. It passes this function a target node (where to search from, the originally selected node) and the `SearchSpecification` object (what to search for).
7. The `getBridgeUriForSearch()` implementation uses the target node and the `SearchSpecification` to build up a container of child nodes that match the search criteria. It returns the URI for the container node.
8. Adobe Bridge displays the search results in the Content pane. For each handled node, it calls the node handler's `makeModel()` method, and associates the resulting `ExtensionModel` object with the `Thumbnail` object for that node.

Implementing getSearchDefinition()

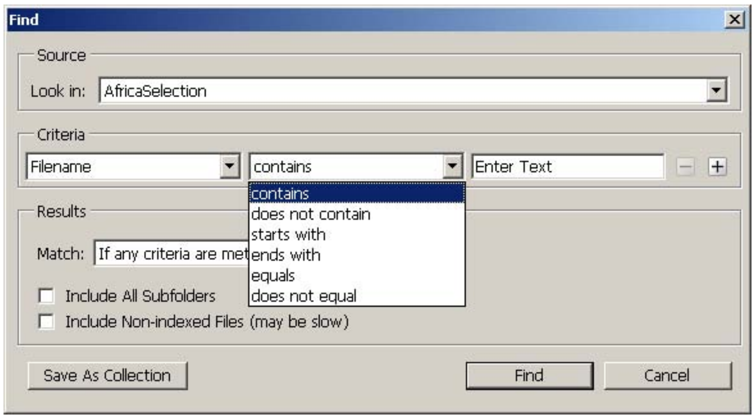
Your implementation of `ExtensionModel.getSearchDefinition()` creates and returns a `SearchDefinition` object, which defines search parameters within your node type. The definition provides a set of possible search criteria, defined by `SearchCriteria` objects, which identify node attributes of interest and the values to match against.

Each `SearchCriteria` object corresponds to a line in the Criteria section of the Find dialog. The object specifies:

- A *search field*, which appears in the drop-down list on the left side.
- A value type or closed list of *operands*, used on the right side. For a simple value type, the string "Enter text" appears in the right-side field; for a date, it shows the expected format. If you specify a list of operands (defined by the `Operand` object), they are use to populate the drop-down list.


Find
Source
Look in: AfricaSelection
Criteria
Color Mode | equals | B&W
B&W
Bitmap
CMYK
Duotone
HSB
HSL
Indexed
Lab
Multichannel
RGB
XYZ
Results
Match: If any criteria are met
☐ Include All Subfolders
☐ Include Non-indexed Files (may be slow)

- Optionally, a subset of the predefined *operators* to exclude. By default, all operators are available. If you specify an exclusion list, those operators are not available for the search field. Operators are identified by the string that appears in the middle drop-down list.



Your returned `SearchDefinition` object can also include:

- An optional set of scope modifiers (such as the default option to search subfolders), which you can use to limit or expand the basic scope of the search. A scope modifier is encapsulated in a `Scope` object, which simply identifies a modification. The nature of the modification is determined entirely by your search implementation.
- An optional set of limits to the number of results, and, if results are limited, a set of possible *rank fields*. Adobe Bridge sorts result nodes by the value of the rank field, and returns no more than the maximum number of result nodes with the highest rank values. When the result is displayed, the view sorts the nodes again using its sorting criteria. The attribute name and display name of a rank field are encapsulated in a `Rank` object.

Implementing getBridgeUriForSearch()

The method `ExtensionHandler.getBridgeUriForSearch()` takes a target node and `SearchSpecification` object. Your implementation must perform the specified search. It must collect matching nodes into a container node, and return the Bridge URI of that container node.

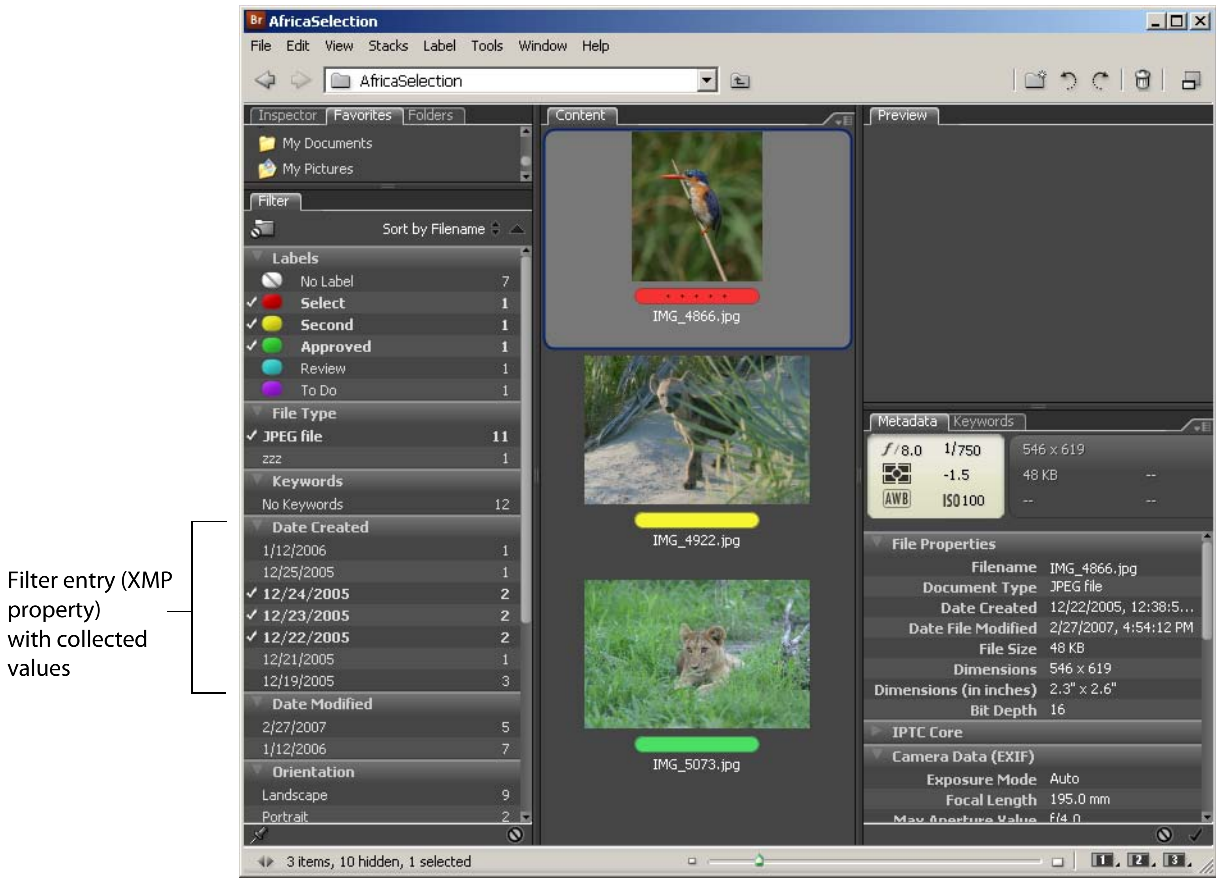
Adobe Bridge can query the result node of a search to retrieve the `SearchSpecification` object used to create it. It does so using the method `getSearchSpecification()` that you define for your `ExtensionModel`. You might want to save the `SearchSpecification` object passed to `getBridgeUriForSearch()` such that the model method can later retrieve it. Alternatively, you can implement the `getSearchSpecification()` method to recreate the specification by some other means.

Similarly, the model for the returned search-result node must implement the `getSearchDetails()` method to retrieve the search target node and search criteria used to create the result.

Adding filters

Filters allows the user to organize and filter the display of thumbnails in the Content pan. When the user selects on or more filter values from the Filter palette, Adobe Bridge applies that filter to children of the current container node whenever it needs to display that container's contents. It shows only those nodes whose attribute values match the selection—that is, those nodes that match the filtering criteria. The selected filters also control the population of any menus that list children of the container.

The `FilterDescription` object provides programmatic control and customization of the Filter palette. Each `FilterDescription` object corresponds to one filter entry in the Filter palette.



When displaying a container node handled by a node-handling extension, Adobe Bridge builds the list of filters by calling the developer-defined `getFilterCriteria()` method of the node's `ExtensionModel`. Your implementation of this method can use the `FilterDescription` object to define new filters. In this way, you can allow the user to filter on the values of properties that you have defined for you own node types.

A list of all filters that are available for the currently displayed container node is kept in the `Document.filters` property. This list includes filters you have added with the `getFilterCriteria()` method.

Implementing the getFilterCriteria() method

Your implementation of `ExtensionModel.getFilterCriteria()` must create and return an array of `FilterDescription` objects. The Filter palette is initially populated with a set of default file filters; your `getFilterCriteria()` method can retrieve this list from `app.defaultFilterCriteria`, and modify it or add to it.

A `FilterDescription` associates a filter name with a node attribute, which is typically an XMP metadata property. For your own node types, you can specify an `Infoset` member as the filtering attribute. For example:

```
myFilter = new FilterDescription("myfiltername", "My Filter",
                                  "MyHandler.MyInfoset.MyInfosetMember", false);
```

Beneath each filter attribute, the Filter tab lists all values that occur for that attribute in all child nodes in the current scope, along with the number of nodes found with each value. It does not normally list nodes that have no value for a particular attribute. You can choose to allow nodes that have no value for the attribute by adding the empty string to the values list in the `FilterDescription.closedValuesList` property.

6 Porting Guide

This chapter summarizes changes between this release and the previous release of Adobe Bridge, to aid you in porting applications to this release.

New Features in Adobe Bridge CS3

Two major features are new in Bridge:

- Node-handling customization through the new `ExtensionHandler` and `ExtensionModel` objects.
- Browser customization through script-defined palettes created with the `TabbedPalette` object, and the Inspector, which shows context-specific information for the selected thumbnail.

Node handling customization

The browser node mechanism that underlies the `Thumbnail` object has been completely reworked. The internal Adobe Bridge node structure is now referenced through a unique resource identifier called the Bridge URI, a filesystem path or URL preceded by an identifying prefix: for example, `mynode:path/file.ext`. Each prefix is registered with an `ExtensionHandler`, which defines how nodes of that type should be handled.

The behavior of a node handler is implemented through the `ExtensionModel`, whose properties and methods are provided by the developer. Whenever Adobe Bridge displays a handled node, it calls the methods of the associated model to perform operations such as file manipulation and node searches.

A node handler associates type-specific node data with the `Thumbnail` object using the new `Infoset` object. This has the effect of adding developer-defined properties to the `Thumbnail` object for the handled node. Scripts can use these properties to define display filters and searches, and to display context for selected nodes.

Browser customization

- All tabbed palettes can be shown or hidden, or moved to any location by the user. The visibility is now scriptable; individual tabbed palettes are represented by the `TabbedPalette` object, and are available to scripts through the `app.document.palettes` property. You can now add your own script-defined palettes using the `TabbedPalette` object, with contents defined in ScriptUI or HTML. ScriptUI can now display Flash animation.
- The new object-inspector palette extends the Content pane by showing specific information directly or indirectly related to the selected thumbnail. It is completely script-defined; there is no default Inspector.

  Inspector panels are defined by the `InspectorPanel`, and each panel contains subpanels defined by `Panelette` objects. The subpanels content can be specified in a flexible format that allows you to access embedded metadata through the `Metadata` object, and node data associated with the thumbnail through its node handler and `Infoset` objects.
- Adobe Bridge CS3 provides a Filter palette that allows users to select criteria by which to filter the thumbnail display. The set of filter criteria that populates this palette is extensible for script-defined node types, using the `FilterDescription` object.

Changes and Deprecations in Adobe Bridge CS3

- The browse-scheme mechanism has been completely replaced by the node-handling `ExtensionHandler/ExtensionModel` mechanism. The following methods and properties are removed or deprecated:
  - `app.browseTo()` and `app.registerBrowseScheme()` methods
  - `Thumbnail.displayMode`, `displayPath`, and `path` properties
  - `BrowseSchemeEvent` object
  - `ThumbnailIterator` object
- `Document.bottomNavbar` and `topNavbar` are deprecated in favor of `Document.navbars`.
- `Document.content` and `contentPaneMode` properties are deprecated in favor of `Document.presentationPath` and `presentationMode.`
- `app.preflightFiles()` has been deprecated in favor of `app.acquirePhysicalFiles()`.
- For the `Event` object, the thumbnail `select` and `deselect` event types have been replaced with two new `document` events, `selectionsChanged` and `selectionsChanging`.
- The Preferences dialog has changed, and this is reflected in the properties and methods of the `Preferences` object.
- Menus have changed, resulting in new and changed menu identifiers; see the `MenuElement` object in the *Bridge JavaScript Reference*.

Object model additions

The following new objects have been defined:

General	Node-handling	Node search and sort
`BitmapData`	`ExtensionHandler`	`SearchSpecification`
`Color`	`ExtensionModel`	`SearchDefinition`
		`SearchCriteria`
`TabbedPalette`	`CacheElement`	`SearchCondition`
	`CacheData`	
`InspectorPanel`		`Operand`
`Panelette`	`Infoset`	`Rank`
`IconListPanelette`	`InfosetMemberDescription`	`Scope`
`TextPanelette`		`SearchDetails`
`ThumbnailPanelette`	`Operator`	
	`ModalOperator`	`FilterDescription`
	`ProgressOperator`	`SortCriterion`

Index

A

application object
  about 15
  registering event handlers 28
applications, communicating 28
arrays, passing between engines 38

B

basic node model 15
bibliography 7
Bridge URI 23
browser window
  about 9
  area descriptions 12
  as document object 14
  components 11
  CS3 changes 66
  customizing 17, 40
  extending behavior 21
  extending the Content pane 41
  panes 16
browse-scheme mechanism 67

C

cache management 54
callbacks
  Content pane 35
  defining for HTML scripts 34
  dialogs 35
  executing 36
  navigation bar 35
  passing complex values 38
  scheduling tasks 38
  using in HTML dialogs 37
code examples, *See* example code
commands, adding to menus 44
communication framework, interapplication 28
conflict resolution 59
Content pane
  about 13
  adding an inspector panel 41
  defining callbacks 35
  determining display 32
context menus 14
conventions, typographic 7
core Infosets 55
CS3 features 66

D

data
  associating with nodes 54
  cache status 54
  ensuring validity 25
  managing cache 54
  node handling data 48
  passing between engines 34, 38
  refreshed 54
date formats 25
dialogs
  about 19, 32
  closing 36
  defining callbacks 35
  displaying HTML controls 36
  displaying ScriptUI 33
  executing callbacks and remote functions 36
  HTML 19
  modal and modeless 20
  modal and modelessl 36
  ScriptUI 19
  using callbacks 37
document object model (DOM) 14
documents
  about 10, 14
  reference materials 7, 9
  thumbnails 15
Dublin Core properties 26
dynamic text values, inserting 43

E

embedded browser for displaying HTML 34
embedded metadata 25
engines, JavaScript 34
error handling 59
event handling
  Bridge events 23
  defining handlers 27
  example code 28
  registering handlers 28
  user-interaction events 16, 27
events, version changes 67
example code
  event handling 28
  inspector panels 42
  interapplication communication 29
  locations 9
  markup elements 43
  menu elements 44
  metadata access 26
  node filters 54
  node searches 61
  node-handling extensions 46
  tabbed palettes 41
  UI techniques 30
ExtendScript
  communicating with users 19

executing scripts 10
ExtensionHandler object
about 45
defining 49
ExtensionModel object
about 45
accessing through thumbnail 48
defining 50
extensions
about 45
defining node data 54
defining searches in 61
installation structure 47
installing 46
node-handling 21
registering 46
script-defined 45
workspaces 48

F

Favorites pane
about 12, 16
illustration 11
Filter pane
about 13
applying filters 63
illustration 11
filters
adding 63
defining 61
Find dialog, extending 61
flyout menus 14
Folders pane
about 12, 16
illustration 11
fonts used in this guide 7
formats for metadata dates 25

G

getBridgeUriForSearch() method 63
getFilterCriteria() method 64
getSearchDefinition() method 62
global information 15

H

handlers
about 45
core data capabilities 55
default behavior 23
defining 27
immediate vs. long-running operations 49
registering 28
HTML
defining callbacks 34
defining user interfaces 19
dialog box controls 32
dialogs 19
display mechanisms 34
displaying controls in navigation bars 37
displaying controls in tabbed palettes 38
displaying in dialogs 36
executing script functions 35
navigation bar controls 31
user interface controls 31
hyperlinks, inserting 43

I

immediate handler operations
about 49
method descriptions 49
immediate model operation methods 51
Infoset object
accessing through thumbnail 48
defining node data 54
predefined sets 55
inspector panels
about 14
creating and displaying 42
script-defined 41
specifying string values 42
InspectorPanel object 41
interapplication communication framework 28

J

JavaScript
additional resources 7
engines for web browser and Bridge 34
panelette markup elements 43
standards information URL 8

K

Keywords pane
about 13
illustration 11

L

long-running handler operations
about 49
method descriptions 49
long-running model operations
about 50
method descriptions 53

M

Mac OS
extensions folder 46
startup folder 10, 48
user workspace folder 11
workspace folder 48
main window, *See* browser window
markup elements, panelettes 43
menubar
about 14
extending 44
menus

about 20
context and flyout 14
element objects 44
extending 44
version changes 67
metadata
display 13
embedding as XML 26
example code 26
files associated with thumbnails 25
node handling 21
pane illustration 11
vs. node data 54
methods, version changes 67
modal dialogs
about 20
displaying 33
displaying HTML 36
ModalOperator object, methods 56
modeless dialogs
about 20
displaying 33
displaying HTML 36
models
extension 50
immediate operations 51
managing data cache 54
node handling 48

N
navigation bars
about 14, 19, 31
configuring 19
defining callbacks 35
displaying HTML controls 37
displaying ScriptUI 33
executing remote functions 37
showing and hiding 31
navigation nodes 23
node data
core 55
defining 46, 54
managing cache 54
sets 54
vs. metadata 54
node handling
about 45
accessing model and 48
associating private data 46
core data values 55
creating script-defined extensions 45
customization 66
extending behavior 45
extension models 50
extensions 21
installing extensions 46
long-running operations 56
registering extensions 46
searches 61
nodes
inspection 41
navigation 23
object model 15
references 23
thumbnails 15

O
object model
about 10, 14
application and document 15
thumbnails 15
version changes 67
objects
extending default behavior 28
inspector 41
menus and commands 44
passing between engines 38
version changes 67
operations
defining long-running 56
handler 49
implementing 56
model 50
progress and status 58
resolving conflicts 59
Operator object 56

P
palettes
custom tabbed 14
defining 17
listing 40
modeless dialogs 20
removing 40
script-defined 40
tabbed 40
panelettes
markup elements 43
string values 42
types 41
panels, panelette types 41
panes
browser window 16
customizing 40
defining 17
performance
node-handling operations 56
thumbnail access 24
plug-in extensions, creating 45
porting guide 66
preferences, application 16
prefixes
about 45
registering 46
presentation mode, Content pane display 32
Preview pane
about 13
illustration 11

ProgressOperator object methods 57
properties, associating with nodes 54

R
reference materials
  code examples and tools 9
  more information 7
references, nodes 23
release notes 66
remote functions, executing
  dialogs 36
  HTML 35
  navigation bars 37

S
scripting
  additional resources 7
  interactions 16
  overview 9
  user interaction 30
scripts
  access to Bridge window 11
  accessing browser window 12
  accessing preferences 16
  adding palettes 40
  adding user interfaces 17
  communicating with users 19
  creating user interfaces 30
  defining callbacks 34
  embedding metadata 26
  executing 10, 35
  search mechanisms 61
  shared startup 48
  startup 10
  user interface options 30
ScriptUI
  dialog box controls 32
  dialogs 19, 33
  displaying elements 32
  navigation bar controls 31
  navigation bars 33
  tabbed palettes 33
  user-interface controls 31
searches
  defining 61
  interactive 61
  specifying parameters 62
  workflow 62
startup scripts
  locations 10, 48
  shared 48
status
  in browser window 14
  monitoring for operations 58
string values
  panelette markup elements 43
  specifying 42
subpanels
  markup elements 43
  string values 42
  types 41

T
tabbed palettes
  about 14
  creating 40
  displaying HTML controls 38
  displaying ScriptUI 33
TabbedPalette object 40
tabs, adding to browser 40
tasks, scheduling 38
thumbnails
  about 12
  accessing 24
  basic behavior 23
  ensuring valid data 25
  example code 26
  in documents 15
  inspection 41
  metadata 25
  performance considerations 24
typographic conventions 7

U
user interfaces
  creating for scripts 30
  customizing 17
  script options 30
  ScriptUI elements 32
  using HTML 19
user-interaction events 16, 27

V
version differences in CS3 66
view mode controls 14
view, Inspector 41

W
web browsers, embedded 34
window, *See* browser window
Windows
  extensions folder 46
  startup folder 10, 48
  user workspace folder 11
  workspace folder 48
workflow, node searches 62
workspaces
  extensions 48
  folder locations 11

X
XML, embedding metadata with 26
XMP metadata 25