ADOBE® INDESIGN® CS5

FEATURE DEVELOPMENT WITH SCRIPTING

Adobe

Adobe® InDesign® CS5 Feature Development with Scripting

Technical note #10122

Contents

Contents

Feature Development with Scripting

Scripting is often used to automate InDesign features. This document describes how to go beyond automation to developing new features for Adobe® InDesign® CS5. Improving this area was a major emphasis for the CS5 development cycle.

NOTE: This document describes XHTML Export, a feature development with ExtendScript, and FlexUIStroke, a user interface sample developed with ActionScript and Creative Suite SDK.

Scripting

A feature developed with ExtendScript, such as Export XHTML, takes advantage of some portion of the following InDesign capabilities and related technologies:

- JavaScript (ExtendScript) — InDesign supports JavaScript for cross-platform development. Adobe's implementation is called ExtendScript. JavaScript's cross-platform nature makes it more useful for feature development than the platform-specific scripting languages AppleScript and VBScript. You can use these languages when developing a feature, but you'd then have to write platform-specific versions of your scripts.
- Menu/action scripting — The InDesign scripting DOM includes access to its menu and action system. These are typically used to add menus and actions.
- ScriptUI — ScriptUI provides a mechanism to create user interfaces using ExtendScript. This is the method used by Export XHTML.
- Script events — Script events provide a way for scripting code to watch for changes in the application. The number and quality of events that can be observed were greatly improved in InDesign CS5. Most significantly, you can now observe changes in selection and in attributes of selected objects. This makes it possible to create high-quality panels that update automatically. This previously was possible only with a C++ plug-in.
- Custom script events — InDesign CS5 provides a fixed set of events. It's possible that you might need to observe some change that is not covered by InDesign events. CS5 makes it possible to add custom events with a relatively simple C++ plug-in. This allows you to provide the majority of your solution in scripting, even if the necessary events are not provided.
- Idle tasks — It is sometimes desirable to postpone certain operations until the application is idle. In the plug-in world, this is known as an idle task. InDesign CS5 provides the ability to implement idle tasks using scripting and attachable events.

User interfaces built with ActionScript typically take advantage of the following additional technologies.

- ActionScript — Adobe Flash-based technologies provide the most powerful and convenient way to create user interfaces for scripting-based solutions. The Creative Suite SDK provides ActionScript access to the suite application scripting DOMs.

- Creative Suite SDK — An environment and tool set for creating ActionScript Creative Suite extensions. The Eclipse-based environment is also known as Creative Suite Extension Builder (CS Extension Builder). CS Extension Builder provides a convenient way to create Flex- and ActionScript-based user interfaces for Creative Suite applications. It greatly simplifies creative containers (panels and dialogs) and menu items while providing ActionScript access to the InDesign scripting DOM.
- CSXS — The Creative Suite Extensibility infrastructure, accessible in ActionScript through CSXSLib. This component provides the container for Flash-based UIs.
- HBAPI — The High Bandwidth API is a suite-wide component that provides ActionScript access to the application scripting DOMs.
- CSAWLib — A library that provides ActionScript wrappers for the ExtendScript classes defined for each application.

Scripting versus C++

The capabilities provided by some InDesign Products SDK samples can be implemented with a script instead of a plug-in. For example, WriteFishPrice inserts tab-delimited text inside a text frame; this can be achieved more easily with a script that targets the current text selection. The TableBasics plug-in inserts a table in a text frame, which also can be automated easily via scripting. BasicTextAdornment, however, adds a new character-text attribute (kBscTAAttrBoss) to the InDesign model. It is not possible to implement such a feature using scripting.

Scripting is commonly used for controlling existing features. It is now especially useful for creating user interfaces using Flash-based technologies. The C++ SDK, on the other hand, is most commonly used for introducing features that can add to the InDesign data model, such as a new text attribute or a page-item adornment.

Scripting is far easier to understand and use than the C++ SDK. By using scripting, you can leverage well-designed APIs that are thoroughly tested in several InDesign code paths. Also, the scripting DOM is versioned, so a script written in one version usually is forward compatible in future releases and can be used without a major porting effort.

The advantages of developing features with scripting are as follows:

- Reduced development effort.
- Easier to debug and test.
- Cross-platform solution (one run-time environment targeting different platforms).
- Lower deployment cost.
- Higher reliability, as the scripting DOM is well tested.

The disadvantages of scripting-based solutions are as follows:

- It requires all features that it uses to be exposed to the scripting.
- Executing a script typically is slower than executing C++ code.

- Options for adding data to a document are limited to adding data to the script labels feature.
- Selection Suites are a C++-based solution. Script-based UI code will have to discover the selection.

Building blocks for using ExtendScript to implement a feature with scripting

In InDesign CS5, the Export as XHTML/Dreamweaver feature is implemented completely using ExtendScript. Export as XHTML/Dreamweaver is not distributed as a traditional InDesign plug-in; instead, it is installed as a folder containing several ExtendScript binaries, within the InDesign scripts folder located in <*InDesignInstallFolder*>/Scripts/export as XHTML. The source code for Export as XHTML is included in <*SDK*>/source/public/components/xhtmlexport. Export as XHTML is used throughout this document to illustrate what it takes to implement a new feature using ExtendScript.

Scripts folder

There are two scripts folders where the user can install scripts, so InDesign recognizes them as the scripts that you want to run with InDesign:

- The user's preferences folder. On Windows®, this is C:\Documents and Settings\<*username*>\Application Data\Adobe\InDesign\Version 7.0\<*locale*>\Scripts. On Mac OS®, it is <*username*>/Library/Preferences/Adobe InDesign/Version 7.0/<*locale*>/Scripts.
- The application's scripts folder. On Windows or Mac OS, this is <*InDesignInstallFolder*>/Scripts/

Having these two folders allows an administrator to install system-wide scripts and allows individual users, who might not have write access to the application folder, to install user-specific scripts.

Inside the default Scripts folder in the application folder, there are folders called Export As XHTML, Scripts Panel, XML Rules, and so on. Scripts inside the Scripts Panel folder are displayed in InDesign's Scripts Panel, so users can run them from the InDesign user interface. The Export As XHTML folder is where the Export as XHTML feature's binaries are located. If you open the Export As XHTML folder, you will see a startup scripts folder, along with some files with the .jsxbin extension. The .jsxbin files are compiled JavaScript; they are in binary format so the source code is not exposed, which serves several purposes:

- Source code can be protected.
- Scripts will not be modified accidentally, which could cause features to behave incorrectly.

Any script located inside a folder named startup scripts that is under the application-specific or user-specific Scripts folder is executed when InDesign launches.

NOTE: Scripts located under a folder named Scripts Panel—even if they are in a folder named startup scripts—are ignored by the code that executes startup scripts.

NOTE: If a script inside the startup scripts folder is in binary format, it cannot use the #targetengine directive as discussed in "ExtendScript engines" on page 9.

The XHTMLExportMenuItemLoader.jsx script (inside Export As XHTML's startup scripts folder) serves only one purpose: Load and execute another script (in binary format), XHTMLExportMenuItem.jsxbin. XHTMLExportMenuItem.jsxbin has one main purpose, to install a menu and the menu's event handlers for the feature at startup. Table 1 is a brief overview of the Export As XHTML scripts folder.

TABLE 1 Three main JavaScript binary components for Export as XHTML

Name	Source-code path	Type	Purpose
XHTMLExport.jsxbin	<*SDK*>/source/public/components/xhtmlexport/XHTMLExport.jsx	Model	This script contains the main logic of iterating over the model and generating and saving XHTML. It also contains the model's implementation of XHTML Export Options and a stub implementation for a progress bar in case it is called, for example, from InDesign Server.
XHTMLExportMenuItem.jsxbin	<*SDK*>/source/public/components/xhtmlexport/XHTMLExportMenuItem.jsx	User interface	This script is executed automatically on launch. It loads the other two scripts as needed and installs the menu item along with the necessary event handlers. When the user chooses the menu item, it brings up the necessary user interface (using XHTMLExportUI.jsxbin) and triggers the export using XHTMLExport.jsxbin.
XHTMLExportUI.jsxbin	<*SDK*>/source/public/components/xhtmlexport/XHTMLExportUI.jsx	User interface	This script contains the strings (along with the localization mechanism), the XHTML Export dialog, and an implementation of a progress bar.

<*SDK*>/source/public/components/xhtmlexport/ also contains these folders:

- The include folder. The scripts inside this folder are included by the three main scripts in Table 1, and they are compiled into binary form along with the script that includes them.
- The resource folder. It contains localized string resource to be loaded by the three main scripts in Table 1. Localization is discussed in "Localization" on page 11.

If you are developing features using scripting, we encourage you to create a folder inside the Scripts folder and/or use the startup scripts folder to store files that you need to use at startup. Because a script in binary format cannot use the #targetengine directive, if you want to target a specific engine during startup, you need to make that script an uncompiled one, like XHTMLExportMenuItemLoader.jsx.

ExtendScript engines

InDesign has two types of ExtendScript engines. Each type of engine supports the same scripting DOM and other capabilities:

- The default engine, named "main," is created automatically and is reset after each time it executes a script.
- Persistent session engines, which exist until the application quits and are not reset, may be created at any time by running a script with a #targetengine directive. The engine will have whatever name is specified in the #targetengine directive. It retains objects and properties between scripts. This is important for scripts attached as function call-backs, such as event handlers, which must remain active after they are attached. It also is a requirement for scripts that display floating script user-interface panels, which may float around indefinitely during an entire user session. The engine is visible to the debugger.

To target a specific engine, use the #targetengine directive at the beginning of your script. For example, the following code executes the script in an engine named "mySession":

```
#targetengine "mySession"
```

NOTE: You may use "#targetengine main" to target the main engine; however, typically you do not need to do so, because scripts are run in the main engine by default.

If a #targetengine directive specifies an engine name that InDesign does not recognize, InDesign automatically creates a persistent engine with that name. This feature prevents conflicts caused by other scripts changing objects/values your script uses. To specify your own script engine, simply put "#targetengine <*your engine name*>" at the top of your script.

You also can create an ExtendScript engine via the C++ API (see the new IExtendScriptUtils interface). There are three customizable options: engine name, whether the engine is reset after every script, and whether the engine is visible to the debugger.

Loading external scripts

As discussed in "Scripts folder" on page 7, Export As XHTML creates its own script folder under InDesign's main Scripts folder, to organize its scripting files. There are two major reasons why Export As XHTML modularizes its scripts in this way:

- *Model/user-interface separation* — See "Model/user-interface separation" on page 16.
- *Loading of localization scripts* — See "Localization" on page 11.

ExtendScript has an #include feature that you can use to include an external JavaScript file, so the functions in the include file are available for the current script to use; however, you cannot use it to load a compiled binary script. If you want to distribute your JavaScript feature in binary format like Export As XHTML, you cannot use #include to load an external JavaScript file.

The recommended approach to loading (and executing) an external script is calling app.doScript(). The source for Export As XHTML also uses a function called loadScript(), defined as in Example 1.

EXAMPLE 1 Export As XHTML's loadScript method

```
XHTMLExportMenuItem.loadScript = function(filename)
{
      return File(XHTMLExportMenuItem.scriptsFolder + '/' + filename );
}
```

The loadScript function simply returns a File object that contains the script. The application object's doScript method is then called (as shown in the XHTMLExportMenuItem.install() function) to execute the script.

Using a startup script to install a menu when InDesign launches

InDesign provides the ability to create new menu items and manipulate application-defined menu items via scripting. A menu in InDesign has a two-layer architecture, separating the underlying action and the displayed menu item. When a menu is invoked, the underlying action is executed. An action is an internal object that invokes a command or event. An action is not necessarily associated with a menu item. The scripting DOM mirrors the internal design; through scripting, you can access menus, menu items, and the underlying actions. You also can add or delete menus and menu items. A new menu item can be associated with an existing, application-defined action or a new, script-defined action. The behavior of a script-defined action is implemented via an attached script. Scripts also can be attached to execute before or after an action is invoked and before a menu or menu item is displayed.

NOTE: A script registered before an action can cancel the action's default behavior.

Although you can dynamically install a menu at run time, in most cases, menus/actions are created at startup. Export to XHTML installs its menu item during startup, so as soon as InDesign is launched, its menu item is available. As noted in "Scripts folder" on page 7, Export as XHTML has one startup script, which loads and executes another script in binary format, XHTMLExportMenuItem.jsxbin.

XHTMLExportMenuItem.jsx's main script contains only one line. It calls XHTMLExportMenuItem.install(), which is responsible for the following tasks:

1. Create a menu action and the action in the "KBSCE File menu" action area, which is defined in ActionDefs.h. The need to add an action to a specific action area is like defining an action through C++ API, where you must specify an action-area entry in the ActionDef resource.
2. Install event listeners for the new action. Table 2 provides more details about the event listener.
3. Install the menu item in the specific menu location, File > Export for > Dreamweaver...

TABLE 2 Event handlers for Export As XHTML action

Event	Handler	Description
afterInvoke	XHTMLExportMenuItem.cleanup	afterInvoke is a good place to clean up any unfinished business during onInvoke. For example, XHTMLExportMenuItem.cleanup makes sure the progress bar is closed, in case the user cancels the script.
beforeDisplay	XHTMLExportMenuItem.enableDisable	This handles the menu item's enable/disable states. Export As XHTML should be enabled only when there is a front document. XHTMLExportMenuItem.enableDisable uses the application document object count to modify the state of its action before the menu is displayed.
onInvoke	XHTMLExportMenuItem.exportSelectedItems	exportSelectedItems is called when the menu is invoked. It executes the Export as XHTML feature.

When XHTMLExportMenuItem.install adds a new action, the action name is localized, which is important in supporting your feature in different InDesign locales. Localization is discussed further in "Localization" on page 11.

Just like the C++ plug-in's typical action-component implementation, scripting allows you to listen to menu-action events in various stages when an action is invoked. An action object's addEventListener method is used to install the event and handler for the action. Table 2 shows the three events Export As XHTML listens to and handles.

Localization

To support scripting features in different InDesign locales, you must localize your user interface and even your feature. Specializing your feature to meet different locales' needs is beyond the scope of this article. In this section, we discuss how you can handle string localization through the InDesign scripting DOM and ExtendScript's localization objects.

Access to InDesign internal string tables

InDesign provides access to internal string-translation tables via the scripting DOM.

Format of key strings

To access internal string tables from the scripting DOM, key strings must be modified to include the prefix "$ID/." For example, if the key string appears as "my internal key string" in the internal string-translation tables, for scripting you would use "$ID/my internal key string."

Accessing key strings

If you have a translated string that is included in the internal InDesign string-translation tables for the current locale, you can access the associated key string(s) via the "find key strings" method on the application object. The return value is an array of strings, since there may be zero, one, or more keys that translate to the desired string. Example 2 shows sample uses.

EXAMPLE 2 Accessing InDesign internal strings

```
var keys = app.findKeyStrings( "Black" ) ; //Returns: $ID/Black
var keys = app.findKeyStrings( "Scripts" ) ; //Returns:
$ID/Script_Tree,$ID/Script_PanelName,$ID/KBSCE Scripts
menu,$ID/Scripts,$ID/ScriptsFolder
var keys = app.findKeyStrings( "None existing string" ) ; //Returns: empty array
```

Accessing translations

After you have a key string that is included in the internal InDesign string-translation tables, you can access the associated translation for the current locale by passing the key string in place of any other string, as you normally would do in the scripting DOM. Note, however, that for the translation to happen, the string must pass through the scripting-language client code inside InDesign. Example 3 shows how to access translated strings. The last alert in the example will not show the translated string, because the alert string is not passed through InDesign.

EXAMPLE 3 Accessing a translated string

```
alert( app.colors.add({name:"$ID/OutOfRangeError"}).name ) ; //Alert "Data is out
of range." since a color is created with the translated string
alert( app.colors.add({name:"OutOfRangeError"}).name ) ; //Alert "OutOfRangeError"
since a color is created with the un-prefixed ($ID) string
alert( app.paragraphStyles.add({name:"$ID/None existing string"}).name ) ; //Assert
"No translation of key 'None existing string' to English string", then alert "None
existing string" since a paragraph style is created with the un-translated, un-
prefixed string
alert( "$ID/OutOfRangeError" ) ; //Alert "$ID/OutOfRangeError" since the alert()
method is handled by the ExtendScript engine, not InDesign's scripting architecture
```

The new scripting API translateKeyString() of the application object also allows you to access an existing user-interface string by name in a locale-independent manner. For example:

```
alert( app.translateKeyString( "$ID/OutOfRangeError" ) ) ; //Alert "Date is out of
range."
```

ExtendScript localization objects

In addition to providing access to InDesign's internal string-translation table, ExtendScript supports localization objects. Localization objects essentially are an array of strings mapped to different locales. In Example 4, all localized strings are stored in the array variable CANCEL. When it is time to use the variable, localize() is used to make sure the proper localized string is put into the variables, based on the current locale of the host environment.

EXAMPLE 4 ExtendScript localization object

```
var CANCEL = { en: 'Cancel', de: 'Abbrechen' };
var s = localize(CANCEL);
```

There was one problem with using the preceding approach for Export as XHTML: putting all languages in one file makes it hard for Adobe's internal localization team to manage different languages. Therefore, Export As XHTML adopts a slight variation of ExtendScript's localization objects: It uses localization objects with only English strings, then it dynamically loads and executes a locale-specific language script that adds the necessary properties. The XHTMLEx-

portMenuItem.install method in XHTMLExportMenuItem.jsx loads the localized string resource whenever necessary, as shown in Example 5.

EXAMPLE 5 Export as XHTML's localization approach

```
if($.locale != 'en_US') {
     // try to load localized strings
     var localizationScript =
XHTMLExportMenuItem.loadScript('Resources/XHTMLStrings-' + $.locale + '.jsxbin');
     if ( !localizationScript.exists )
     {
         localizationScript =
XHTMLExportMenuItem.loadScript('Resources/XHTMLStrings-' + $.locale + '.jsx');
     }
     if ( localizationScript.exists )
     {
         ...
     }
}
var actionname = localize(xhtmlExportStrings.HTMLACTIONNAME);
```

ExtendScript stores the current locale in the $.locale variable. This variable is updated whenever the locale of the hosting application changes. Example 5 checks whether the current locale is English; if not, it tries to load the localized strings list in the Resources folder. It uses the technique discussed in "Loading external scripts" on page 9 to load the script and make all strings in the localized resource file available to the current function. All the English strings are defined in XHTMLStrings-en_US.jsx inside the include folder and included in the beginning of XHTMLExport.jsx, which also is loaded by the XHTMLExportMenuItem.install.

Setting up scripting preferences

For many things in InDesign, you must temporarily change some preferences to achieve what you want. For example, to read the coordinates of a page item in a specific measurement unit, you must switch the view preferences. You may want to restore the preferences after you are done with your task. The CS5 version of Export As XHTML requires scripting DOM version 7.0. It also requires enableRedraw to be set to true, so the progress bar can be drawn correctly, and it needs to allow the user a full level of user interaction.

To set application preferences temporarily and then restore the old values, Export As XHTML implements a helper class, prefsContext, in XHTMLUtils. prefsContext is an object that manages the tasks of only changing those preferences that need to be changed and remembering what was changed and what were the old values. You simply pass a reference to the preferences object into its constructor and use its methods to change and restore the preferences.

Storing persistent data

JavaScript has built-in features for storing and retrieving data; that is, the tosource() and eval() functions. tosource() is a method for all built-in objects that returns a string representing the source of the object. eval() evaluates a string of JavaScript code. Example 6 shows how tosource()/eval() is used typically.

EXAMPLE 6 JavaScript's built-in mechanism for making objects persist

```
var obj = { prop: "value" };
var storedObj = obj.toSource();
// storedObj -> "({prop:"value"})"
var clone = eval(storedObj);
// clone.prop -> "value"
```

There is one major security concern with using the technique in Example 6: you end up saving a script in your document that you later load and execute. It would be possible to create a virus script that would procreate whenever you export as XHTML. To address this potential security risk, Export As XHTML uses E4X to save its data in XML format, then a string representation of the XML data is stored as a label (XHTMLExportOptions) in the document.

InDesign supports adding script labels to objects within a document. Each label essentially is a key-value pair.

According to Wikipedia, "ECMAScript for XML (E4X) is a programming language extension that adds native XML support to ECMAScript (ActionScript™, DMDScript, E4X, JavaScript, JScript)". For more information about E4X, see http://www.ecma-international.org/publications/standards/Ecma-357.htm. ExtendScript supports a subset of E4X.

To use E4X to store your object:

1. Create an XML class object that represents your DOM. For Export As XHTML, data is represented by the XHTMLExportOptions object. SOS.serialize() in SimpleObjectStore.jsx instantiates a new XML object, then it iterates through all properties in XHTMLExportOptions and stores the properties as elements and attributes in the XML object.
2. Serialize the XML object; that is., create a string representation of the XML. After you have an XML object, you can call the toXMLString() method of the XML object to serialize the XML object.
3. Save the serialized string as a label in the document. Use app.activeDocument.insertLabel() to insert a label that contains the serialized XML data in the current active document.

To use E4X to retrieve an object saved in a document:

1. Extract the serialized XML data from the saved label in the current document, using app.activeDocument.extractLabel().
2. Deserialize the saved label. Use the label extracted from the previous step to construct a new XML object.
3. Restore the properties of the object that represents your saved data from the XML object created in the previous step. For example, SOS.serialize() in SimpleObjectStore.jsx uses XHTMLExportOptions's property name as the corresponding XML attribute's name, so in SOS.deserialize(), it simply uses the same name to search the XML object for an attribute tag that matches the property name, then it restores the property value with the found attribute value.

For implementation details, see the Export as XHTML source code.

User interface

InDesign integrates the ExtendScript user-interface library, called ScriptUI, which also is supported in other Adobe Creative Suite® applications. ScriptUI enables the creation of dialogs and floating panels that are children of InDesign application windows. It supports most standard, platform-widget types. Windows created with ScriptUI are not native InDesign user interface, because they do not contain native InDesign user-interface widgets; they use platform user-interface widgets. However, they interact with other InDesign windows as if they were owned by the application. Widgets interact with each other and with the InDesign scripting DOM via scripts attached as event handlers. Dialog widgets can be laid out using a string-based resource and/or dynamically created at run-time via scripting.

NOTE: The Dialog object in the scripting DOM uses the native InDesign user interface.

Export As XHTML uses the Window class in ExtendScript to create its export-options dialog. It uses a three-stage process to bring up the dialog as users see it in InDesign CS5:

1. It passes a string-based resource (XHTMLExportDialog.dlgResource) into the Window class constructor during Window object instantiation. The string resource specifies the initial layout of the dialog.
2. It handles things that cannot be done in the resource string; for example, it populates pop-up menus. Also, it installs event handlers for the widgets, to dynamically handle widget state changes.
3. It initializes the dialog widgets with the export-options data stored in the document.

There are other user-interface elements that you might want to implement; for example, a progress bar for long operations. Export As XHTML implements a general-purpose progress bar that can be reused; for details, see ProgressBar.jsx. *Adobe InDesign CS5 Scripting Guide* also has a progress-bar sample using ScriptUI technology.

Responding to events

It is possible to automatically trigger scripts or script functions when certain changes are made to the application or to a document. This mechanism is similar to the responder pattern used in the InDesign C++ SDK. The scripting DOM for InDesign event handling is based on the W3C DOM for Level 2 Events Specification. A script can be attached as an external file or in JavaScript as a function callback. A script attached to a document event like open is triggered by user actions or a script.

Scripts can add and remove event listeners by calling the addEventListener and removeEventListener scripting methods. These methods are supported on a number of objects, but most commonly will be called on the application or document objects. Because events are not persistent, events need to be registered each time the feature is added or enabled. This commonly is done with a startup script or in response to a menu item or action.

One very relevant use of event listeners comes in implementing panels. A panel needs to respond to changes in the model (document and application persistent data) and selection. For example, the stroke weight panel displays a different value based on what's selected. It also

updates any time the stroke weight of the selected item is changed. This can happen via a UI operation, script, or plug-in code. Export As XHTML doesn't contain such a panel. There is, however, support for this in the scripting DOM. For an example, see the FlexUIStroke sample in the InDesign CS5 Products SDK.

To listen for events, a panel implementation should use event listeners. The following example demonstrates the code that would be used to add and remove event listener functions. It would be desirable for an implementation to call addMyEventListeners when the panel is activated, and removeMyEventListeners when the panel is closed. This results in the onSelChg and OnSelAttrChg implementations being called when the selection changes or a property of the selection changes. These two methods would then need to contain panel-specific code to deal with changes in the selection.

```
#targetengine "session"
// Add Selection Events
function addMyEventListners()
{
      app.addEventListener("afterSelectionChanged", onSelChg, false );
      app.addEventListener("afterSelectionAttributeChanged", onSelAttrChg, false );
}
function removeMyEventListerns()
{
      app.removeEventListener("afterSelectionChanged", onSelChg, false);
      app.removeEventListener("afterSelectionAttributeChanged", onSelAttrChg,false);
}
// The Selection Changed
function onSelChg (myEvent){
      //... Respond to the change in the seleciton
}
// Some attribute (or property) of the selected object or objects changed.
function onSelAttrChg(myEvent){

      //... Respond to a possible change in the selected object
}
```

NOTE: Scripts that use handler functions (script files) must use #targetengine "session". If the script is run using #targetengine "main" (the default), the function will not be available when the event occurs, and the script will generate an error.

Events are covered in more detail in the "Events" chapter in *Adobe InDesign CS5 Scripting Guide*.

Model/user-interface separation

Separating the user interface and the model can make your scripting plug-in functionality available on InDesign Server, just like a C++ plug-in. One of the design goals for Export As XHTML is to use it within an ID Server workflow. Thus, we separated the user interface from the underlying functionality, which also helps automate testing. As discussed in "Scripts folder" on page 7, Export As XHTML consists of three main ExtendScript binaries.

There is an includes folder inside the Export As XHTML's source folder in <*SDK*>/source/public/components/xhtmlexport. You do not see this folder in the feature's script folder, because scripts inside the includes folder are "included" in one of the three main JavaScript files listed

in Table 1. ExtendScript supports an #include statement that can be used to split a function among multiple JavaScript source files. The #include statement is very useful for structuring source code (for example, model/view separation and having all strings in one file for easy localization). The #include statement also provides an easy way to reuse code.

To support progress bars in a server environment, Export As XHTML has two versions of the progress bar, one of which does not do anything that is used in the server. The same approach can be used in other model/user-interface separation cases.

Script optimization

The ExtendScript Toolkit has a built-in profiling capability through the IDE's Profile menu. It is useful for tightening loops and spotting CPU-intensive code lines. Once source code is profiled, the ExtendScript Toolkit shows the result in a color-coded bar, which makes it easy to spot bottlenecks in your program. For more information about using the profiling feature of the ExtendScript Toolkit, see *JavaScript Tools Guide*.

Compile a script into binary format

To compile a script into binary format, simply open the script in the ExtendScript ToolKit and choose File > Export As Binary.... to save the .jsx script to a file with a .jsxbin extension.

Tips and hints

This section provides script-development tips and tricks that were learned during the development of Export As XHTML.

Development techniques

Use object-oriented techniques

Export As XHTML source code was designed using object-oriented techniques. In most cases, classes that hold attributes and methods were implemented.

Use global variables/namespaces

In the current design, all scripts that install menus need to share the same scripting engine. This means they also share their global variables. Best practice for JavaScript development is to use namespaces to encapsulate global variables and functions.

Use undefined instead of nil or “”

When checking missing function parameters, arrays elements, or variables, use undefined, as shown in the following snippet:

```
ProgressBar.prototype.newSection = function(numSteps, title, fractionOfParentStep)
```

```
{
      if (fractionOfParentStep == undefined)
```

InDesign errors out when asking for an nonexisting property

You cannot do the following:

```
var footnotes = story.footnotes;
if (footnotes != undefined) {
```

Instead, you need to do this:

```
if( 'footnotes' in story) {
footnotes = story.footnotes
```

InDesign's collection objects are not completely compatible with JavaScript arrays

InDesign collection objects offer features that JavaScript arrays do not have, such as itemByRange() and nextItem(). If you want to read the items from a collection into an array, use everyItem() or iterate; however, you will lose the capabilities of the collection objects.

Error handling

JavaScript's built-in exception handling, such as try...catch blocks, works very well. For examples of try...catch blocks, see the Export As XHTML source code. For more information on error handling, see *Adobe InDesign CS5 Scripting Guide*.

Differentiating between InDesign's feature sets

Check for app.featureSet. It returns a FeatureSetOptions enum that contains either FeatureSetOptions.roman for the Roman feature set or FeatureSetOptions.japanese for the Japanese feature set.

DOM versioning

As discussed in "Setting up scripting preferences" on page 13, the CS5 version of Export As XHTML requires DOM version 7.0. The DOM version is available from the script-preferences property, app.scriptPreferences.version.

Persistent data versioning

You can version your own saved data, but be careful not to confuse your persistent-data version with the DOM version. Versioning your own persistent data makes it easier to maintain compatibility for your feature among different releases. For example, in different versions of your software, you might have saved different sets of data. If you versioned the saved data in your code, you can provide conversion code to deal with compatibility issues. Export As XHTML stores its saved-data version as a property, currVersion, in the XHTMLExportOptions class. In XHTMLExportOptions.restore(), where saved data is restored, the version is checked to ensure that the proper options are restored or a new default set of options is used if the saved version is too old.

Edit-compile-run

"Loading external scripts" on page 9 discusses the benefits of modularizing your scripts and loading the script module as needed dynamically. One benefit of this approach is quick development time. At startup, InDesign loads only the script that installs the menu, and this part of the script probably is easy enough that you do not have to debug it much. For the rest of the scripts, you can always edit and compile, then return to InDesign and execute the already loaded menu, which dynamically loads the newly modified modules so you can check the correctness of the new implementation.

Debugging modular scripts

ExtendScript ToolKit's debugging feature does not work with binary scripts or dynamically loaded scripts. There is no easy way to deal with this limitation. Usually, it is necessary to write test code within a module boundary. Also, the "divide and conquer" technique always is an effective way of debugging; that is, comment out different blocks of code to narrow your investigation until you isolate the problematic code.

NOTE: The earlier version of the ExtendScript ToolKit supported a "#show include" directive to help debug-included scripts. The latest version of the ExtendScript ToolKit has built-in support for include-file debugging; thus, the "#show include" directive is deprecated.

Mixing and matching JavaScript and C++

Sometimes, you may want to use C++; for example, for performance considerations, if you are adding a feature that a script cannot achieve, or you simply want to reuse existing features that you implemented in C++. To achieve this, expose your C++ features in the scripting DOM; then you can call those features from within your script. For information on how to make your C++ plug-in scriptable, see the "Scriptable Plug-in Fundamentals" chapter of *Adobe InDesign CS5 Products Programming Guide*.

To run a script from C++, use IScriptUtils::DispatchScriptRunner. Alternately, you can use the lower-level APIs as shown in the following snippet to access the IScriptRunner::RunFile. (IScriptRunner is aggregated on kJavaScriptMgrBoss.)

EXAMPLE 7 Low-level script-runner call

```
// assume scriptFile is an IDFile representing the script file to run
InterfacePtr<IScriptRunner> scriptRunner(Utils<IScriptUtils>()-
>QueryScriptRunner(scriptFile));
if (scriptRunner)
{
 ScriptRecordData arguments;
 ScriptData resultData;
 PMString errorString;
 const bool16 showErrorAlert = kTrue;
 const bool16 invokeDebugger = kFalse;
 scriptRunner->RunFile(scriptFile, arguments, resultData, errorString,
showErrorAlert, invokeDebugger);
}
```

Performance techniques

Minimize access to InDesign's DOM

Querying the InDesign DOM may be the main performance bottleneck for your script. A considerable amount of time typically is spent resolving object references, because InDesign does not hand out pointers to objects but rather uses references that need to be resolved every time they are used. Here are some techniques to alleviate this problem:

- Reduce the number of calls to the scripting DOM.
- Store and reuse resolved references in variables wherever possible.
- Use everyItem() to fetch and cache data of a collection object all at once, instead of querying the properties with separate calls.

Fast array look-ups with object properties

JavaScript objects essentially are associative arrays; there is a built-in hashing function for properties on an object. Any JavaScript array can use other objects as keys to look for value. That is, the property:

```
myArray.one
```

is the same as:

```
myArray['one']
```

Combining this capability with the "for (var i in object)" statement, which goes through each element of an associative array, you can write efficient code for fast array look-ups.

Concatenating large strings is slow

JavaScript's String class-concatenation methods, such as += operator, can be very slow, especially with large strings. Try to minimize the number of concatenations and the size of the strings that are be concatenated. One common method to reduce String class overhead is to write your own string-buffer class to gain a performance boost; this uses the Array object's join method to "concatenate" all the elements of an array into one string.

Using regular expressions

JavaScript supports Perl-style regular expressions, which can be very useful for string manipulation such as complex string replacements.

Building blocks for using ActionScript to implement user interfaces

It is easier than ever to create Flash/ActionScript-based panels and dialogs for a number of Creative Suite applications, including InDesign. The building blocks for using ActionScript and other Flash-based technologies such as Flex are primarily covered in the Creative Suite

SDK. This section highlights the FlexUIStroke panel (which was reimplemented using the Creative Suite SDK) and other points of interest to the InDesign developer.

Creative Suite SDK

The Creative Suite SDK allows developers to create extensions for several of the Creative Suite applications. Such an extension can be either a panel or dialog. This provides a convenient way to implement most of the InDesign user interfaces required by third-part components.

Communicating with InDesign

CS Extension Builder project's properties can be set to target one or more of a number of supported Creative Suite applications. Targeting InDesign automatically imports the CSAWLib wrapper library that allows you to call InDesign's ExtendScript DOM from ActionScript. This includes excellent type-ahead support in Eclipse, making this a very comfortable environment for coding and debugging.

Working with ExtendScript

Where possible, convert your ExtendScript to ActionScript. The syntax is similar and the strongly typed development environment will be helpful. It may be necessary to make calls between ExtendScript and ActionScript.

Calling ExtendScript from ActionScript

An extension can include an ExtendScript component. A path to the file is specified in the extension's manifest.xml file. In the development environment, you will find the manifest.xml file in the .staged-extension/CSXS directory. When deployed, this appears in the CSXS directory at the root of the extension. The code is executed in a scripting target engine that is unique to the extension. You can call functions declared in the script filing using the CSXSInterface instance and evalScript() as follows:

```
CSXSInterface.getInstance().evalScript("foo()");
```

Calling ActionScript from ExtendScript

To call from ExtendScript into ActionScript, you need to call through a reference to an ActionScript class instance. This reference needs to be passed to the ExtendScript code but it can't be passed through a CSXSInterface and evalScript(). For example, to call a function called registerFlashExtension() in the ExtendScript file, and pass a reference to the ExtendScript code, use HostObject as follows:

```
_extendScriptInterface = HostObject.getRoot("com.adobe.indesign.MYEXTENSIONID");

if(_extendScriptInterface != null)
      _extendScriptInterface.registerFlashExtension(this);
```

Handling InDesign events

CSXS includes a number of Suite-wide events, but these events are not as extensive as InDesign's ExtendScript events. InDesign ExtendScript events include events for selection changes, idle tasks, placing files, updating links, and more. These are important events for writing meaningful InDesign panels.

Although the methods to register event listeners are exposed in ActionScript, they don't currently do anything; registering ActionScript event listeners is not supported. The workaround is to register and implement event handlers in ExtendScript and call back into the ActionScript-based extension when the events are triggered. This relies on the techniques for calling between ExtendScript and ActionScript and is demonstrated in the FlexUIStroke sample. The ExtendScript code will be something like the following, where _flashExtensionInterface is a reference to the extension's ActionScript class and updatePanel() is an ActionScript method.

```
 function updateNow()
{
      _flashExtensionInterface.updatePanel();
}

function addEventListeners()
{
      app.addEventListener(Event.AFTER_SELECTION_CHANGED, updateNow, false);
      app.addEventListener(Event.AFTER_SELECTION_ATTRIBUTE_CHANGED, updateNow,
      false);
      app.addEventListener(Event.AFTER_CONTEXT_CHANGED, updateNow, false);
}
```

Event listeners need to be added when the panel is presented to the user, and removed when it is hidden. This code needs to be called from the applicationComplete and removedFromStage events in the panel's mxml file. This is demonstrated in the main.mxml of the FlexUIStroke sample.

Working with selection

Another part of writing meaningful panels can require working with InDesign's selection model. There are no selection suites in scripting like there are in C++. To initialize an extension's UI based on the selection, you must write the appropriate code to determine what is selected and what should be presented to the user. Similarly, you must write code that sets the properties on the selected items. There are a number of scenarios to keep in mind, and each is demonstrated in the FlexUIStroke sample.

InDesign includes a selection object. Acquiring the selection in ActionScript and ExtendScript is roughly equivalent. The application object includes a selection property.

```
var selections:Object = InDesign.app.selection;
```

If there is no selection, selections.length is zero. When that is the case, the application panels and dialogs set application and/or document defaults. An extension may need to behave similarly. If there is no selection and no open document, InDesign UIs act on the application defaults. If there is a document open, InDesign UIs can act on either the application or docu-

ment defaults depending on the property. Certain settings (typically related to viewing options) exist only in the application defaults.

If there is an open document with a selection, you might encounter the following objects in the selection's objects:

- PageItem — It's possible to select one or more page items. This is equivalent to a layout selection in C++. In this case, there will be one or more PageItem objects in the selection.
- InsertionPoint — A selection can include exactly one text insertion point. This is equivalent to a flashing I-beam cursor in a single text frame. This is represented by the InsertionPoint object. If there is an InsertionPoint in the selection, there will be only one InsertionPoint and it will be the only thing in the selection; there cannot be a mix of other selection types.
- Text — It's possible to select a range of text in a single text frame or across linked frames. Such a range of text is represented by a single Text object.
- Table — An entire table selected is represented by a Table object.
- Cell — One or more selected table cells are represented by a Cell object.

Overriding default menu placement

NOTE: Overriding the default menu position is supported by InDesign, but not yet by other Creative Suite applications.

By default, CSXS supports a single main menu item, which appears in the application's Window > Extensions menu. Extension developers may prefer a different menu location. InDesign supports a Menu element and Placement attribute in the manifest.xml that allows an extension to override the default menu position. For example:

```
<Menu Placement="'Main:&amp;Window',600.0,'KBSCE Window menu'">FlexUIStroke</Menu>
```

The Placement attribute needs to be created with care. See the FlexuUIStroke code for a more precise example (without a line break).

- Strings must be enclosed in single quotes.
- Commas (without white space) are used to separate fields.

Menu path

The menu path format is similar to the format used by MenuDef ODFRC resources (see also IMenuManager.h).

- Menu path components must either be localizable key strings or be prefixed with the kDontTranslateChar (.).
- The accelerator key must be escaped using & in the menu path. To include an actual ampersand character in a menu path component or menu item, use a double ampersand (&&).
- If the first character in the menu path is a hyphen, InDesign inserts a menu separator before the extension's menu item. Similarly, if the last character is a hyphen, InDesign inserts a separator afterwards. Both may be specified.

- Menu paths must exactly match the key string for existing menus. The easiest way to determine these strings is to dump the existing menus in the debug build using Test > UI > Actions > Dump MenuMgr info(all). This uses TRACE commands, so you must first set the location of trace output. For example, you can set it to trace text into an open copy of Notebook. Search for a string such as "Main:&Window" to find menu items that belong to the Window menu.

Menu position

The menu position value is identical to the one used by MenuDef ODFRC resources (see also IMenuManager.h).

- To sort a menu item alphabetically, use the same menu position as the other menu item(s).
- Refer to AdobeMenuPositions.h for predefined menu positions of other menu items.

ActionArea

The action area format is identical to the one used by ActionDef ODFRC resources (see also IActionManager.h).

- The action area string must be a localizable key string.
- Refer to ActionDefs.h for predefined action areas used by other menu items.

Debugging

Debugging extensions requires you to set a flag on the system that will create the Flash player with debugging enabled. You also may need to set the location of the InDesign executable. These settings are covered in the Creative Suite SDK documentation.

Frequently asked questions

Is it possible to have a mixed plug-in, with new user-interface items in a script and old user-interface items in C++?

Each user-interface object, like a dialog, needs to be one or the other; otherwise, yes.

Can script-based floating panels be 100% equivalent to native panels?

ScriptUI panels do not behave like native panels. Their container is fundamentally different. It cannot be docked and undocked, and it cannot have a fly-out menu. ScriptUI panels also cannot be included in a panel workspace.

CS Extension Builder panels are native panels containing a Flash player widget. They can be docked and undocked, provide fly-out menus, and be included in a panel workspace.

Can an ExtendScript or ActionScript based panel react to the current selection?

Yes. InDesign CS5 adds the afterSelectionChanged and afterSelectionAttributeChanged attributes that provide notification when selection, or some attribute of selection, changes. These events are covered in the "Events" chapter of the *Adobe InDesign CS5 Scripting Guide*.

NOTE: As discussed in "Handling InDesign events" on page 24, there is a limitation in ActionScript. The current support provides no way to register an ActionScript event handler. To work around this, events can be handled in a small bit of ExtendScript and forwarded to ActionScript. This is relatively simple and is demonstrated in the FlexUIStroke sample.

Can you add a panel to InDesign's Preferences dialog using ExtendScript or ActionScript?

No, not in InDesign CS5. You can, however, add a pane containing an OWL Flash Player Widget to the dialog using ODFRC resources, then implement it using Flash. The "Flash/FlexUI" chapter in *Adobe InDesign CS5 Solutions* describes how to implement a Flash player widget using Flash. You also can add your own preferences dialog and preferences menu item next to the regular preferences menu item.

Can you access the file system and other local and external resources from ExtendScript and ActionScript?

ExtendScript provides access to Adobe BridgeTalk-aware applications and the file system. It also includes a full socket implementation. For more information, see the *JavaScript Tools Guide*.

CS Extension Builder extensions can access resources through numerous ActionScript APIs. Extension UIs are executed using an Adobe AIR runtime so the file system on the local machine is accessible using Adobe AIR APIs if the panel loads the SWF from the local file system. Loading a remote SWF places the code in a security sandbox that prevents access to the local file system.

Resources

Adobe Creative Suite CS5 comes with guides and tools mentioned in this article, including the following:

- *Adobe Creative Suite 5: JavaScript Tool Guide* — This is the official ExtendScript ToolKit guide. It provides detailed information about ExtendScript ToolKit features, including the IDE and profiling features. It also has chapters dedicated to user-interface tools; specifically,

how to create a user interface using ScriptUI. There is a chapter about the unique ExtendScript features that are not in normal JavaScript, like the Dollar ($) object.

- ExtendScript ToolKit — This is the tool that you may want to use to develop your ExtendScript project. It is an ExtendScript IDE, scripting-dictionary viewer, and profiling tool, and it compiles ExtendScript into a binary format.
- *Adobe InDesign CS5 Scripting Guide* and *Adobe InDesign CS5 Scripting Tutorial* — These provide a lot of good information and script samples showing how to script via the InDesign scripting DOM.
- "Scriptable Plug-in Fundamentals" chapter of *Adobe InDesign CS5 Products Programming Guide* — This chapter shows how to make your C++ feature scriptable. It is useful if you are developing mix-in style plug-ins, as mentioned in this document. It also lists the SDK samples that have scripting support.
- Creative Suite SDK — CS Extension Builder includes documentation and samples that cover creating Creative Suite extensions.