

ADOBE® INDESIGN® CS4



FEATURE DEVELOPMENT WITH SCRIPTING



© 2008 Adobe Systems Incorporated. All rights reserved.

Feature Development with Scripting

Technical note #10122

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Creative Suite, and InDesign are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. All other trademarks are the property of their respective owners.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA. Notice to U.S. Government End Users. The Software and Documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.



Contents

What is a scripting plug-in?	5
Scripting versus C++.	5
Building blocks for using ExtendScript to implement a scripting plug-in.	6
Scripts folder.	7
ExtendScripts engines	8
Loading external scripts.	9
Using a start-up script to install a menu when InDesign launches	11
Localization.	12
Setting up scripting preferences	14
Storing persistent data	14
User interface	16
Responding to events	16
Model/user-interface separation	16
Script optimization	17
Compile a script into binary format.	17
Lessons learned.	17
Development techniques.	17
Performance techniques	20
Frequently asked questions	21
Is it possible to have a mixed plug-in, with new user-interface items in a script and old user-interface items in C++?	21
Can script-based floating panels be 100% equivalent to C++-based ones?	21
Can an ExtendScript-based (ScriptUI) panel react to the current selection?	21
Can you add a panel to InDesign's Preferences dialog using ExtendScript?	21
What about database connectivity from ExtendScript?	21
Resources.	22

Feature Development with Scripting

This document describes how to develop Adobe® InDesign® features using scripting.

What is a scripting plug-in?

A *scripting plug-in* is a plug-in implemented entirely (or partially) via scripting instead of using the InDesign C++ SDK. InDesign CS4 supports the following features to make scripting plug-ins possible:

- *A cross-platform script language* — InDesign has supported JavaScript since the CS version. In addition, InDesign supports the platform-specific scripting languages AppleScript and VBScript, which also may be used to create a scripting plug-in.
- *A way to create a user interface that can interact with InDesign* — InDesign provides the ability to create user-interface elements, like menus and dialogs, via scripting. Instead of using C++ code to drive the logic behind these user-interface elements, the end user or developer can use scripts. Using scripts, you can replace, but not modify, InDesign's existing menus and dialogs.
- *A way to observe what is happening in InDesign and attach scripting code that executes appropriately* — Attachability allows scripts to execute automatically when something in InDesign happens. You can think of it as a scripting-observer mechanism. Users already can execute scripts from the script palette. Attachability allows them to have scripts run when they do certain things; for example, every time a document is opened or closed.

Scripting versus C++

Some SDK samples can be implemented with a script. For example, WriteFishPrice inserts tab-delimited text inside a text frame; this can be done easily with a script targeting the current text selection. The TableBasics plug-in insert a table in a a text frame, which also can be automated easily via scripting. BasicTextAdornment, however, adds a new character-text attribute (kBscTAAAttrBoss) to the InDesign model. To achieve that, the plug-in implements an IAttrReport interface and provides an implementation for ITextAdornment, so the attribute is drawn on the document text. It is not possible to implement such a feature using only a script.

Scripting is best for automating existing features that are exposed in the InDesign scripting DOM. The C++ SDK is best for introducing feature that can modify the InDesign data model, like a new text attribute or page item.

This somewhat over-simplifies the choice of scripting versus C++ SDK. There are things only the C++ SDK can do; for example, complex user-interface functions like drag and drop, complex event handlers, adding a selection suite, and adding a tool to the tool bar.

Almost all InDesign functions are exposed in the InDesign scripting DOM, which is easier to understand and use than the C++ SDK. By using the scripting DOM, 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 scripting plug-ins are as follows:

- Reduced development effort, because scripts are faster to prototype and easier to implement.
- 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 plug-ins are as follows:

- You can only use features that are exposed to the InDesign DOM.
- Executing a script typically is slower than executing C++ code.
- There is no dialog observer scheme as in the C++ SDK. You can observe a widget's value in ScriptUI and then, for example, update another widget's state. Mainly you can react to the widget's state changes (e.g. onClick). A C++ observer is more flexible; you can do things also based on message, protocol, etc.
- You cannot introduce new data to the InDesign model; however, you can store information in a document using script labels.

This document focuses on the scripting-only approach. There is an SDK sample, FlexUIBasicScriptUI, which shows the scripting-only approach at its minimalist. The sample shows how to implement FlexUI in InDesign using Flash player in a ScriptUI palette window. This approach is discussed here, using the example from the CS4 Export as XHTML feature (also available from the SDK). For a scripting/C++ hybrid approach, see the SDK sample, FlexUIStroke, which uses a combination of Flex/ActionScript and C++. It uses FlexBuilder to build its user interface to be used in InDesign, and it implements a selection observer in C++ to watch for selection change, to update its Flex user interface. For implementation details, see the samples' design documents available in the SDK. Also, there is a chapter discussing the hybrid approach in *Adobe InDesign CS4 Solutions*.

Building blocks for using ExtendScript to implement a scripting plug-in

In InDesign CS4, 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 InDesign scripts folder located in *<InDesign application>/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

Starting in InDesign CS3, there are two scripts folders (as shown below) where the user can install scripts, so InDesign recognizes them as the scripts you want to run with InDesign:

1. The user's preferences folder. On Windows®, this is `C:\Documents and Settings\<user>\Application Data\Adobe\InDesign\Version #.#\<local>\Scripts`. On Mac OS®, it is `<user's home folder>/Library/Preferences/Adobe InDesign/Version #.#/\<local>/Scripts`.
2. Windows or Mac OS: `<InDesign Application folder>/Scripts/`

Having these two folders allows an administrator to install system-wide scripts and allows individual users, who may 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,” etc. 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; it is 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, causing 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 start-up scripts.

NOTE: If a script inside the “startup scripts” folder is in binary format, it cannot use the `#targetengine` directive as discussed in [“ExtendScripts engines” on page 8](#).

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`. In Export as XHTML's case, `XHTMLExportMenuItem.jsxbin` has one main purpose, to install a menu and the menu's event handlers for the feature at start-up. [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 gets executed automatically on launch. It loads the other two scripts on an as-needed basis 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 has an 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.

<SDK>/source/public/components/xhtmlexport/ also has a 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 12](#).

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 you need to use at start-up. Since a script in binary format cannot use the #targetengine directive, if you want to target a specific engine during start-up, you need to make that script an uncompiled one, like XHTMLExportMenuItemLoader.jsx.

ExtendScripts engines

There are two types of ExtendScript engines in InDesign. Each type of engine supports the same scripting DOM and other capabilities, with one exception:

- The default engine, named “main,” is created automatically and is reset after each time it executes a script.
- “Persistent” engines, which 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-back, like 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. Session engines are created on the fly via the `#targetengine` directive in a script.

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, since scripts are run in the main engine by default.

If a `#targetengine` directive specifies an engine name InDesign does not recognize, InDesign automatically creates an engine with that name. The engine is a non-resettable engine that exists until the application quits, and it is visible to the debugger. 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 gets 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 12.

ExtendScript has an `#include` feature you can use to include an external JavaScript file, so the functions in the include file are available for the locale 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 CS3 and CS4 versions of Export as XHTML use different approaches to dynamically load an external script. These are described below

CS3 approach

The solution that CS3 Export as XHTML uses is to load an external file into the local variable of a JavaScript File class type and call `eval()` on the variable.

In CS3, Export as XHTML’s source has a function called `loadScript()`, which is defined like [Example 1](#).

EXAMPLE 1 CS3 Export As XHTML's loadscript method

```
XHTMLExportMenuItem.loadScript = function(filename) {
    var file = File(XHTMLExportMenuItem.scriptsPath + filename );
    var script = undefined;
    if (file.exists) {
        file.open();
        script = file.read();
        file.close();
    }
    // we return the script rather than calling eval() right here
    // because the results of eval() are only valid within the
    // scope of the function that calls eval()
    return script;
} //XHTMLExportMenuItem.loadScript
```

Loading the script in this context simply means reading the contents of the script file into a local variable.

Objects/classes defined with `eval()` are scoped within the current method. Passing that variable to `eval()` executes that script. This means any globally defined functions and variables are now defined in the function that does the call to `eval()`. For example, suppose you have a script file with the following two lines of scripts:

```
var foo = "bar"; function hello() { alert("Hello"); }
```

When this script is loaded using `File.read`, you will have a variable that contains the string “`var foo = “bar”; function hello() { alert(“Hello”); }`” When you pass that variable to `eval()`, you have a global variable named `foo` with a value of “bar,” and I have a global function named `hello` which you can call. Since `eval` is scoped within the current calling function, `ExportAsXHTML` does not call `eval()` within the `loadscript()` method; instead, it lets the function that needs to use the external script to call `eval()` right after the `loadscript()` call.

Using the technique demonstrated here, you can separate your scripts according to their functional purposes and load any of them only when needed. This also promote code re-use.

NOTE: `ExtendScript` offers a function, `$.evalFile (file [, timeoutInMsecs])`, which loads and evaluates a file with an optional timeout in milliseconds. The results of the `eval` is the result of the method. This means something like the following:

```
// load and run myFile.jsx with a timeout of 10 seconds
result = $.evalFile ('myFile.jsx', 10000);
// load and run myFile.jsx with no timeout
result = $.evalFile ('myFile.jsx');
// If the internal eval() of the file's contents return a result, that result is
returned.
```

CS4 approach

In CS3, scripts executed via `DoScript` are executed in the main engine unless they contained a `#targetengine` directive. Unfortunately, binary scripts may not use `#targetengine`; therefore, the CS3 Export as XHTML had to dynamically load the script and call `eval` to make sure the child scripts were executed in the correct engine.

This problem is fixed in CS4. Scripts executed via `DoScript` are now executed in the same engine as the parent script. Since it is much more straightforward to use `DoScript`, that is the

recommended approach to load (and execute) an external script. The source for CS4 Export as XHTML also has a function called `loadScript()`, defined as in Example 2.

EXAMPLE 2 CS4 Export As XHTML's loadscript method

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

Note that the CS4 version of `loadScript` does not read the open and read the script as in CS3. When CS4 Export as XHTML needs to access the external script, it simply calls the `loadScript` method to return a `File` object that contains the script, then it calls the application object's `doScript` method (as shown in the `XHTMLExportMenuItem.install()` function) to execute the script.

Using a start-up script to install a menu when InDesign launches

InDesign CS3 introduced the ability to create new menu items and manipulate application-defined menu items via scripting. A menu in InDesign is architected as a two-layer design, 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, scripting-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 may cancel the action's default behavior.

Although you can dynamically install a menu at run time, in most cases, menus/actions are created at start-up. Export to XHTML installs its menu item during start-up, 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 start-up 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 to 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 > Cross Media Export > XHTML/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 gets 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 12](#).

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 locale's 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 CS4 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 3](#) shows sample uses.

EXAMPLE 3 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

Once 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 4](#) 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 4 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 5](#), 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 5 ExtendScript localization object

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

There was one problem with using the above 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 `XHTMLExport-`

MenuItem.install method in XHTMLExportMenuItem.jsx loads the localized string resource whenever necessary, as shown in [Example 6](#).

EXAMPLE 6 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 6](#) 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 CS4 version of Export as XHTML requires scripting DOM version 6.0. 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 7](#) shows how tosource()/eval() is used typically.

EXAMPLE 7 JavaScript's built-in mechanism for persisting objects

```
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 7](#): 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, follow these steps:

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; i.e., create a string representation of the XML. Once 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, follow these steps:

1. Extract the serialized XML data from the saved label in the current document, using app.activeDocument.extractLabel().
2. De-serialize 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 CS4 integrates the ExtendScript user-interface library, called ScriptUI, which also is supported in other Adobe Creative Suite® 4 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 CS4:

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 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 re-used; for details, see `ProgressBar.jsx`. *Adobe InDesign CS4 Scripting Guide* also has a progress-bar sample using ScriptUI technology.

Responding to events

It is possible to automatically trigger scripts from a document event. This mechanism is similar to the document-responder feature 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 be attached to events using the `EventListener` scripting object. An `EventListener` is a handler registered on an object in the scripting DOM that is to be triggered when a specific event occurs on the target object or its descendent. For details, see the “Events” chapter in *Adobe InDesign CS4 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/xhtmllexport`. 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 up a function among multiple JavaScript source files. The `#include` statement is very useful for structuring source code (e.g., model/view separation and having all strings in one file for easy localization). The `#include` statement also provides an easy way for code re-use.

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.

Lessons learned

In this section, we discuss lessons 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 a non-existing 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 100% compatible with JavaScript arrays

InDesign collection objects offer features that JavaScript arrays do not have, like `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 features.

Error handling

JavaScript's built-in exception handling, like `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 CS4 Scripting Guide*.

Differentiating between the 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 14](#), the CS4 version of `Export as XHTML` requires DOM version 6.0. 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

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 start-up, 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; i.e, comment out different blocks of code to narrow down 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 & 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 re-use existing features you implemented in C++. The way to achieve this is to 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 CS4 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 8 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, since InDesign does not hand out pointers to objects but rather references that need to get 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 re-use 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, like += operator, can be very slow, especially with large strings. Try to minimize the number of concatenations and the size of the strings that are 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 like complex string replacements.

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 C++-based ones?

No. There is much more you can do with observers in C++. Also, not all widgets are supported in a panel created via scripting; for example, the tree-view widget.

Can an ExtendScript-based (ScriptUI) panel react to the current selection?

There are only a few events that can update a ScriptUI panel, and changing selection is not one of them. There are three ways a ScriptUI panel can get updated:

- *Idle handler* — This works only while a script is running. A script can send an idle event to a ScriptUI pane and have it respond to the event and update. This probably will get used primarily for filling in progress indicators in a progress bar as a longer script runs.
- *Any event in InDesign that can invoke a script* — This includes start-up scripts, some document events (save, open, close, export, etc.), menu events, and just running scripts from the script panel.
- *ScriptUI events* — If you interact with the panel in any way (like moving it, resizing it, and clicking a button), a script can be fired to update the panel contents.

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

Not in InDesign CS4. You could, 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 CS4 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.

What about database connectivity from ExtendScript?

There is nothing direct. ExtendScript can talk only to Adobe BridgeTalk-aware applications. It can read and write files. Also, ExtendScript has a full socket implementation. For more information, see *JavaScript Tools Guide*.

Resources

Adobe Creative Suite CS4 comes with guides and tool mentioned in this article, including the following:

- *Adobe Creative Suite 3: 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 not in normal JavaScript, like the Dollar (\$) object.
- ExtendScript ToolKit — This is the tool 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 CS4 Scripting Guide* and *Adobe InDesign CS4 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 CS4 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.