SnippetRunner Cookbook

1/27/21 Adobe, Inc.

Contents

1 Snippet Runner Cookbook

Introduction

The SnippetRunner architecture consists of these major components:

- A back-end server that provides the basic functionality, which includes a parameter input mechanism, debug support, and exception handling. For the Acrobat SDK, the back-end server is the SnippetRunnerServer Acrobat plug-in. For the PDF Library, it is the SnippetRunner application.
- The SnippetRunner Common Interface, which acts as a client to the back-end server and provides a unified cross-platform user interface to use and extend SnippetRunner functionality.

Installing and running SnippetRunner

SnippetRunner consists of a set of files, organized into folders in the Samples\SnippetRunner\ directory of the Acrobat SDK or PDF Library SDK. The directory includes the following files:

- Source code files for the SnippetRunner server. These files are located in the Sources\platform\ directory. For each platform, there is a project file you can use to compile SnippetRunner along with the SDK header files. It consists of a .vcproj file for Microsoft® Windows and an .xcodeproj file for Mac.
- SnippetRunner environment and utility files. These files are located in the Sources\platform\Acrobat (for the Acrobat SDK only) and Sources\platform\Shared directories.
- Individual code snippets. Each of these is intended to demonstrate one or more APIs. Each snippet exists as a single, separate file within the Sources\snippets\Acrobat (for the Acrobat SDK only) or Sources\snippets\Shared directory, and is included in the SnippetRunner project. If you build your own snippets (see Writing Snippets), you can add them to the SnippetRunner project and rebuild the project.
- External files required by snippets. These are in the ExampleFiles directory and can be sample files for input or resources for user interface components.

Before using the SnippetRunner, you need to build it in the appropriate manner for your platform. For the Acrobat SDK, after the SnippetRunner project is built, the SnippetRunner Server plug-in must be installed in the appropriate plug_ins directory so that it will be loaded by Acrobat when it is launched. It will be copied automatically after being built if the `AcroSDKPIDir` environment variable is defined, or it may be copied manually. For the PDF Library, the SnippetRunnerServer application will be executed when the Common Interface is launched (see Starting the Common Interface for the PDF Library).

SnippetRunner Common Interface

The SnippetRunner Common Interface extends the usability and functionality of SnippetRunner, and it provides uniformity and availability across platforms. It works as a front end for communicating commands triggered by user interactions to the SnippetRunner serverand intepreting feedback from the server as a result of executing commands. See Using the Common Interface for information about its features and usage.

The Common Interface was first introduced as a Java application with the Acrobat 7.0 SDK for Linux (on which Adobe Reader was available), and was subsequently shipped with Acrobat 7.0 and PDF Library 7.0 SDKs.

The initial releases (versions 1.0 and 1.5) of the Common Interface were Java applications that required a platform-specific Java Virtual Machine (JVM) to be properly installed and accessible. The recommended JVM was the Java Runtime Environment (JRE) available for download from Sun Microsystems, Inc.

The SnippetRunner Common Interface that was shipped with the Acrobat 8.0 SDK was a rewrite of the previous version with Adobe Flex™ 2 technology, taking advantage of the high-performance Adobe Flash® 9.0 player engine to achieve cross-platform availability while eliminating the need for a Java VM on your system. As modern web sites make substantial use of Flash technology for their site content, it is very likely that you already have the Flash player plug-in installed for your browser.

In the Acrobat 9.0 SDK, the Common Interface went a step further by taking advantage of the Adobe Integrated Runtime (AIR), which enables convenient and configuration of the Common Interface and eliminates the need to use a browser, while preserving the familiar user interface in a desktop application.

Installing the Common Interface

For the Acrobat SDK, the Common Interface is delivered as an Adobe AIR application. See http://www.adobe.com/go/air to download and install the free Adobe AIR runtime. Once the runtime has been installed, double-click the CommonInterfaceAIR.air file located in your Acrobat SDK installation.

For the PDF Library SDK, you will continue using the Java-based Common Interface, so make sure you have the Java VM properly installed. If you already have a JDK installed, you do not need a separate JVM for the SnippetRunner Common Interface. If you are specifically installing one for this application, the J2SE 1.4.2 (or 1.5) JRE from Sun Microsystems is recommended.

Starting the SnippetRunner and the Common Interface

The SnippetRunner back-end server must be started prior to starting the Common Interface front end to ensure the establishment of socket communication channels.

When running snippets as plug-ins, you must launch Acrobat or Adobe Reader before starting the Common Interface. As long as you have copied the SnippetRunnerServer plug-in file to the plug_ins directory of Acrobat or Adobe Reader, the SnippetRunner server loads on application launch and starts listening on a port designated for socket connection requests.

When running snippets using the PDF Library SDK, you can start up the Common Interface (see Starting the Common Interface for the PDF Library). This automatically starts the PDF Library SnippetRunner application prior to attempting to establish a socket communication channel, so no manual invocation is required.

Starting the Common Interface for the PDF Library

This section describes how to start the Java version of the Common Interface to be used with the PDF Library SDK.

You may load the Common Interface as a standalone Java application or as a Java applet. The following sections describe the procedures for each of these cases. The Common Interface is packaged as a Java Archive (JAR) file containing Java byte code (class files) and associated resources. This file is CommonInterface.jar in the `SnippetRunner` folder. The JAR format provides for resource integrity and

also allows the content of the archive to be digitally signed, which is required so that the Common Interface can be certified to run as an applet by a web browser's JVM.

Creating the configuration file

The first time the Common Interface is invoked, a configuration file (pdflsdk.config) is automatically generated and stored in the user's home directory.

To ensure that the contents of this file are created accurately, you must launch the Common Interface from its installed location in the SnippetRunner folder so that a *base* directory (a platform-specific absolute path to the SnippetRunner folder) can be properly written to the file. This path is used by the Common Interface to locate reference files and snippet source code at run time. Once this configuration file is written, the Common Interface can be invoked from any folder location.

If a path other than that of the SnippetRunner folder is written to the configuration file, the Common Interface will not function properly. If this occurs, delete the configuration file and restart the Common Interface from its base directory.

Running as a standalone Java application

This section explains how you can run the Common Interface as a standalone Java application.

To run the client as a standalone Java application:

1. From a terminal or console window, switch to the directory where the CommonInterface.jar file resides (the SnippetRunner directory).
2. Execute the following command: `console>java -jar -cp . CommonInterface.jar`

On Windows and Mac OS, you can also double-click the CommonInterface.jar icon to launch the Common Interface. (This procedure assumes that the JAR file extension has not been associated with other applications after your JDK/JRE installation.) The SnippetRunner Common Interface should begin running very soon thereafter.

Running as a Java applet

To run the Common Interface client as a Java applet, you must digitally sign the JAR file before loading the client into your web browser. This requires the `keytool` and `jarsigner` command-line utilities from the J2SE Development Kit (or equivalent).

To sign the CommonInterface.jar file for running as an applet:

1. Generate a public/private key pair and the self-signed certificate.
2. Sign the JAR file with the private key.
3. Load the signed client into your browser.

To generate a public/private key pair and the self-signed certificate, execute a command similar to this from a console using the `keytool` utility:

`console>keytool -genkey -alias` *`EntryAlias`* `-keypass` *`EntryPassword`*

where *`EntryAlias`* is a name you want to assign for this key pair entry in the keystore and *`EntryPassword`* is a password required to guard against that key entry.

You will be prompted for the keystore password of your choice and some information to incorporate into the self-signed certificate. The newly generated public/private key pair and the self-signed certificate will be saved in the keystore file in the default location. Refer to the JDK Security Tools documentation for details on the keystore.

To sign the JAR file with the private key, issue a command similar to the following from a console using the `jarsigner` utility:

```
console>jarsigner -storepass StorePass -keypass KeyPass CommonInterface.jar
EntryAlias
```

where *`StorePass`* is the keystore password assigned while creating the public/private key pair entry in the previous step. *`KeyPass`* is the key pair entry password assigned in the previous step. *`EntryAlias`* is the name assigned to the key pair entry in the previous step.

This completes the applet signing process in preparation for the Common Interface to be run as an applet by the JVM plug-in of the platform browser. The signed JAR file contains a copy of the certificate from the keystore for the public key corresponding to the private key used to sign the JAR file.

Now that you have signed the JAR file, load the provided HTML page CommonInterface.html (in the SnippetRunner folder) into your default browser. This page is an applet starter page that marks up the applet properties.

Accept the certificate to allow the browser JVM plug-in to execute the applet byte code. The Common Interface should begin running soon thereafter, and you can interact with it within the boundary of the browser window.

Known issues

The following issues pertain to the Java-based Common Interface.

- The socket communication between the SnippetRunner server and client may be lost during the operating system's sleep mode. To re-establish communication, restart the Common Interface.
- If you start the Acrobat process by loading a PDF document into the web browser plug-in, the Common Interface socket communication will carry on with that process, which is most likely not expected.
- Currently, only Internet Explorer supports resizable content. All other browsers render the Common Interface at a fixed dimension.
- On Mac OS X, if you minimize the Common Interface window to the Dock and then bring it back into view, the Common Interface window might not be properly repainted. Resize the window to force the GUI to refresh.
- On Windows, an Acrobat dialog box triggered as a result of a command issued by the Common Interface may not come up to the top of all open windows, in which case the Common Interface may seem frozen. Open Acrobat to dismiss the dialog box.

Using the Common Interface

With the Common Interface, you can perform the following tasks:

- See the collection of available code snippets sorted by categories
- Get snippet information
- Execute snippets

- Examine the output generated as a result of a snippet execution
- Browse snippet source code

The Common Interface has four panes:

- The Snippet Collection pane (upper left)
- The Snippet Description pane (lower left)
- The Source/Reference Browser pane (upper right)
- The Snippet Output pane (lower right)

You can resize, maximize, or minimize the main window as you would with any application. You can adjust the relative sizes of the individual panes by dragging the pane dividers.

The Snippet Collection pane groups available snippets into a folder hierarchy for ease of access. These categories are defined in the snippets' registration macros (see Writing Snippets).

You can navigate the hierarchy by means of mouse or keyboard.

- Use the Up/Down arrow keys to move up and down the list.
- Use the Left/Right arrow keys or click the triangles to expand or collapse a folder.

Whenever you hover (AIR version) or select (Java version) a snippet name, its description appears in the Snippet Description pane.

To browse the source code of a snippet, click on a snippet node and select Browse the source code from the context menu. The source code appears in a tabbed window in the Source Browser pane.

To execute a snippet, click (AIR version) or right-click (Java version) on the snippet name and select Execute This Snippet.

Note: For the PDF Library SDK only, you can also right-click anywhere in the pane and select Open a New Document, which allows opening a document for use by a snippet. At the bottom of the pane is a document status area that shows the file name of the current open document. A plus sign (+) in the brackets indicates that the document has been modified.

The Source/Reference Browser pane provides a tabbed interface to allow switching between the Reference view and the snippet source code views.

- The Source view displays the code for a selected snippet. You can switch between multiple snippets by clicking the tabs at the top of the window. Within this view, you can right-click to change the size of the text being displayed. You can close the specific source view by clicking the X in the upper-left corner (AIR version) or lower-right corner (Java version).
- The Reference view displays SnippetRunner Cookbook documentation. You can navigate this document by means of mouse or keyboard. In addition, you can navigate between views of this document by right-clicking to access the Back and Forward commands in the context-sensitive menu.

2 Writing Snippets

A code snippet must contain at least the following:

- A single main function that acts as its entry point. Snippets may contain additional functions as needed.
- A macro call that binds the snippet to the SnippetRunner environment.

An example of a single-function snippet that is included with the SnippetRunner project, SimpleSnip.cpp, is shown below. The code for this snippet also includes comments (not shown here) that provide useful development hints.

```
#include "SnippetRunner.h"
#include "SnippetRunnerUtils.h"

void SimpleSnip()
{
   Console::displayString("This is a simple snippet.");
   Console::displayString("Simple snippet executed\n");
}

SNIPRUN_REGISTER(SimpleSnip, "Simple Framework", "SimpleSnip creates a
framework for a snippet.")
```

This snippet has a single function, SimpleSnip, which writes two messages to the output pane of the Common Interface (see Using the Common Interface) using the utility function `Console::displayString`. This function enables you to perform memory dumps and view strings (`char *`), `ASText` objects, and Cos objects. (See the source files SnippetRunnerUtils.cpp and Console.cpp for details.)

Because the `SimpleSnip` function requires no parameters, it uses the macro call `SNIPRUN_REGISTER` to bind to the SnippetRunner environment. (See Passing parameters to snippets for other possibilities.) This macro requires three parameters:

- The name of the function that is the entry point for the snippet (the same as the snippet's file name).
- A string indicating where the snippet's node is to appear in the Snippet Collection pane of the Common Interface user interface. If the snippet location is not at the root level of the hierarchy, the string specifies the path to the snippet, with folder names separated by colons. For example, the GetFontInfoSnip snippet would specify: "PD Layer:Fonts:Get Font Info".

  Note: A snippet's node name is limited to 49 characters.
- A string of descriptive text to be presented in the Snippet Description pane.

SimpleSnip is a *synchronous* or "one-shot" snippet, meaning that it executes and then terminates. See Toggling behavior and asynchronous snippets for other possibilities.

Passing parameters to snippets

You can pass parameters to a snippet's main function. To enable this mechanism, use the `SNIPRUN_REGISTER_WITH_DIALOG` macro in your snippet to bind the snippet to SnippetRunner. This call takes an extra parameter (beyond the three required by `SNIPRUN_REGISTER`), which is a single string representing default parameter(s) separated by spaces.

When a snippet implemented with `SNIPRUN_REGISTER_WITH_DIALOG` is invoked, a dialog box with the snippet's descriptive text appears that includes a text edit box pre-populated with the default values set by the fourth parameter.

Note: If necessary, this dialog box can be suppressed using the utility calls `turnParamDialogOff` (`/On`) and `toggleParamDialog` (see SnippetRunnerUtils.cpp).

An example of the use of this macro and its resulting values is in the TextChangeColour snippet, whose macro call is written as follows:

```
SNIPRUN_REGISTER_WITH_DIALOG(TextChangeColourSnip, "PDE Layer:
   Text:Change colour","Shows how to change the colour of text
   in a document","0 0 65000")
```

When this snippet is invoked, a dialog box displays "Shows how to change colour of text in a document" and the parameter text edit box is pre-populated with the default values: "0 0 65000". For this example, the parameters are meant to represent RGB color values. You can edit the text in the dialog box to change the values of the parameters.

To access the parameters passed in through the dialog box, use the `ParamsManager` class. This class (see ParamManager.cpp) provides a set of methods that allow you to obtain the input parameters as integer, string, hex, and fixed data types. (To provide support for other data types, you must extend the `ParamsManager` class.)

For example, the `TextChangeColour` function is defined with a single parameter of type `ParamManager *`, to provide storage for the snippet's parameters:

```
void TextChangeColourSnip(ParamManager * thePM)
```

The following code in the `TextChangeColourSnip` function converts the input parameter string to three separate RGB values of type integer:

```
ASInt32 red,green,blue;

thePM->getNextParamAsInt(red);
thePM->getNextParamAsInt(green);
thePM->getNextParamAsInt(blue);
```

Toggling behavior and asynchronous snippets

Note: This section applies to Acrobat plug-in snippets only; not to the PDF Library SDK.

SnippetRunner provides utility methods for toggling behavior. For example, FormCalculationsSnip turns on and off the ability to perform form calculations. It uses the `toggleSnippetCheck` method (see SnippetRunnerUtils.cpp) to turn the state on if it was previously off, and vice versa.

Other snippets that toggle behavior include AVPageViewToggleWireframeDrawingSnip, and AVAppShowAnnotProperties.

Some snippets define and register callbacks in the same manner as plug-ins. (See the Acrobat and PDF Library API Reference and Developing Plug-ins and Applications for information regarding `ASCallback` objects, `ASCallbackCreateProto` and `ASCallbackDestroy`). Specifically, to register a snippet for a notification, use `AVAppRegisterNotification` and provide a callback function with the appropriate arguments. To register your snippet for a specific event, such as `IdleProc`, `PageViewDrawing`, `PageViewClicks` or `PageViewAdjustCursor`, use the related `AVAppRegister`*`XXX`* method. You can toggle a snippet to Off by checking for its On state and unregistering via the complementary `AVAppUnregister`*`XXX`* method.

Such snippets can be *asynchronous* in the sense that they register a callback whose output (or other result) does not appear until a particular event occurs. Snippets that register for notifications include: `OptContNotificationTracerSnip`, `AVAppFrontDocChangeNotSnip`, `AVAppRegisterForPageViewDrawingSnip`, `PDDocDidDeletePagesNotSnip` and `IdleProcSnip`.

Handling exceptions

The SnippetRunner provides an exception handler that reports the name of the snippet that caused an exception. Synchronous snippets require no special considerations with regard to exception handling within the SnippetRunner environment.

However, if you write a snippet containing a callback that is called asynchronously, the callback function should include its own exception handlers to trap and handle various exceptions. When an exception occurs, your exception handler can perform any necessary cleanup, such as releasing memory. The core API provides the following macros for handling errors: `DURING`, `HANDLER`, `END_HANDLER` and `E_RETURN`. If methods in your snippet code could return an error code or `NULL` if something goes wrong, you can call the `ASRaise` method, which generates an exception.

Handling documents

SnippetRunner provides a C++ class, `CDocument`, that handles getting documents in the SnippetRunner environment. (See CDocument.h and CDocument.cpp.)

To use this class, a document must be open. You declare a `CDocument` object and then cast it to the type you need. For example:

```
CDocument document;
AVDocGetFoo((AVDoc)document);
```

Supported cast types are:

`AVDoc` — The front document (does not apply to PDF Library)

`PDDoc` — The front `AVDoc` in Acrobat or the document that is open in PDF Library

`CosDoc` — Derived from the current `PDDoc`

`AVPageView` — The current page view in the front document (does not apply to PDF Library)

`PDPage` — The page associated with the current page view in Acrobat (or the page that has been set in PDF Library). It defaults to the first page.

The destructor method of `CDocument` is called when the snippet returns. Therefore, you do not need to write code to release or destroy these objects.