

CEP 8 Integration Cookbook

Table of Contents

- Overview
 - CEP Architecture
 - Host IDs for Applications
 - CEP Extensions
 - CEP Extension Folders
- Components
 - CEP Components for Native Integrators
 - Prerequisites
 - CEP PlugPlug Libraries (Mandatory)
 - CEP HTML Engine (Mandatory)
 - Vulcan 5 Message and Control Libraries (Mandatory)
 - PlugPlug External Object (Nice to Have)
 - CEP Demonstration Application
 - Signing Components
 - Other Components for Native Integrators
 - AMT Library
 - Scripting Engine Instance
 - CEP Components for Extension Developers
 - CEP JSLibrary
 - CEP ActionScript Library
 - CEP IMS Library
 - CEP Test Extensions
- Installation
 - Recommended Folder Structure
 - Tell CEP where to load its components
 - Install CEPHtmlEngine
 - Install CEP Extensions
- Integration
 - Extension Window Types
 - Special Notes about Window Types for HTML extension
 - Integrate Window->Extensions Menu
 - Window->Extensions menu items
 - CEP Events
 - Standard Events
 - Event after loading an extension
 - Event after unloading an extension
 - Types of unloading an HTML Extension
 - Integrate Capability Based Features
 - Capability based Feature Support
 - Current Limitation
 - Capability Based Approach
 - What to do if the host application supports a capability
 - What to do if host application does NOT support any capability
 - Capability EXTENDED_PANEL_MENU
 - What to do if host application supports this capability
 - Sample PlugPlugUpdateMenuItemFn Implementation on Windows
 - Sample PlugPlugSetMenuItemFn Implementation on Windows
 - Test this Capability
 - Capability EXTENDED_PANEL_ICONS
 - What point products MUST do, even if you do NOT support this capability.
 - What else to do if point products support this capability.
 - Test this capability
 - High DPI Panel Icons
 - DELEGATE_APE_ENGINE
 - SUPPORT_HTML_EXTENSIONS
 - DISABLE_FLASH_EXTENSIONS
 - Hosts supported by extension bundle
 - Supporting CEP Highbeam Data Collection
 - Supporting HTML Extension
 - Enabling SUPPORT_HTML_EXTENSIONS capability
 - Implementing PlugPlugGetPanelFn callback
 - Implementing PlugPlugGetCurrentImsUserIdFn callback
 - Implementing PlugPlugGetAppMainWindowFn callback (Windows only)
 - Supporting Hi-DPI display on Windows platform
 - Setting extension specific scale factor
 - Extension Signature Validation
 - Disabling Extension Signature Verification

- Native IMS APIs
 - PlugPlugIMSConnect
 - PlugPlugIMSConnectWithEndpoint
 - PlugPlugIMSFetchAccounts
 - PlugPlugIMSFetchAccessTokenWithStatus
 - PlugPlugIMSFetchContinueToken
 - PlugPlugShowAAM
 - PlugPlugIMSDisconnect
 - PlugPlugIMSSetProxyCredentials
 - PlugPlugIMSFetchAccessToken
 - PlugPlugIMSAttemptSSOJumpWorkflows
 - Trouble Shooting
 - Further Documentation
- Sending out notifications on panel fly-out menu open and close
- Drag and Drop
- Real Point Product Integration
- Set and Get the window title of the extension
- Remember Last Location and Size of Dialogs
- Customize Highlight Color
- Logging and Debugging
 - PlugPlug Logs
 - CEPHtmlEngine Logs
 - CEF Log
- FAQ, Useful Resources and Feedback
 - Test Automation
 - Logging Bugs
 - Useful Resources
 - Integration Dashboard
 - Cleanup Script
 - Past Editions of the Integration Cookbook
 - Feedback

This cookbook is a step by step guide to integrate CEP (formerly CSXS) Extensions/Services in your CC 2015 application.

Overview

CEP Architecture

Common Extensibility Platform CEP (formerly CSXS - Creative Suite Extensible Services) is a shared technology which provides a rich platform to create and run HTML5-based extensions across point products. Please read CEP Architecture document at [CEP 7 Architecture](#) .

Host IDs for Applications

Every application has a corresponding reserved Host ID. For a complete list of application Host IDs, see [CEP Host IDs for CC 2014](#)

CEP Extensions

CEP Extensions are HTML5-based extensions that extend the functionality of the host application that they run in. Extensions are loaded into applications through the [PlugPlug Library](#) architecture.

CEP extensions can be discovered/accessed from within the application depending on various criteria as described below:

- Window -> Extensions menu: Most extensions can be accessed from this menu.
- A customized menu location: A custom menu location is negotiated by the extension owner and the application team. One example of this extension is:
 - **Share On Behance**: This extension gets loaded from the 'File' menu.

CEP Extension Folders

CEP supports 3 types of extension folders.

- Product extension folder. Here is a suggestion, but each point product can decide where this folder should be.

- \${PP}/CEP/extensions (PPs may use different folder.)
- System extension folder
 - Win(x86): C:\Program Files\Common Files\Adobe\CEP\extensions
 - Win(x64): C:\Program Files (x86)\Common Files\Adobe\CEP\extensions, and C:\Program Files\Common Files\Adobe\CEP\extensions (since CEP 6.1)
 - Mac: /Library/Application Support/Adobe/CEP/extensions
- Per-user extension folder
 - Win: C:\Users\{USER}\AppData\Roaming\Adobe\CEP\extensions
 - Mac: ~/Library/Application Support/Adobe/CEP/extensions

How does CEP decide which extension to load?

- CEP searches these extension folders in sequence of product extension folder, system extension folder and per-user extension folder.
- Extensions without appropriate host application ID and version are filtered out.
- If two extensions have same extension bundle ID, the one with higher version is loaded.
- If two extensions have same extension bundle ID and same version, the one whose manifest file has latest modification date is loaded.
- If two extensions have same extension bundle ID, same version and same manifest modification date, the early-found one is loaded.

Extension Installation:

- Point product installers should install extensions to product extension folder.
- Extension Manager and Exchange Plugin in Thor should install extensions to system extension folder or per-user extension folder.

Note:

- Character '#' is not allowed in extension folder path on both Windows and Mac OSX, since CEF treats '#' as a delimiter.

Components

CEP Components for Native Integrators

Prerequisites

For CEP 7.0, all Windows machines using x64 components must have the VS 2015 C++ Redistributable x64 installed.

CEP PlugPlug Libraries (Mandatory)

PlugPlug are libraries which are mainly responsible for communication between the application and other CEP components.

For CEP HTML5, PlugPlug controls the life cycle CEP HTML engine, which runs HTML extensions. PlugPlug libraries are also responsible for displaying HTML extension UI as well as managing the communications between host application and HTML extension. There are two versions of PlugPlug library - PlugPlugOwl and PlugPlug, corresponding to OWL and Drover applications respectively. They are identical in most of the aspects, except for UI related code. Please choose the correct PlugPlug library base on the UI library your application uses.

PlugPlug is responsible for send/receive CSXS events between native APIs and HTML extensions within host application.

PlugPlug integrates Vulcan Message library and Vulcan Control library and expose their APIs as JavaScript to HTML extensions.

Codex

- Product: CEP
- Version: 8.0.0
- Component: PlugPlug/PlugPlugOwl

CEP HTML Engine (Mandatory)

CEP HTML Engine is a standalone executable responsible for rendering HTML extensions. It uses open source project CEF version 3 (<http://code.google.com/p/chromiumembedded/>). Since CEF has only 32bit version, CEP runs HTML engine is a separate process, in order to work together with both 32bit and 64bit creative desktop applications. PlugPlug launches a HTML engine process when an extension is loaded, and closes it when the extension is unloaded. PlugPlug uses IPC to communicate with HTML engine. HTML engine's main process is CEF's browser process, and it may have one or more Render and GPU process.

Node.js (<http://nodejs.org/>) had been integrated into CEP HTML engine, in order to provide richer JavaScript programming environment to HTML extension developers. All Node.js JavaScript modules can be used.

CEP Version	CEF Version	Node.js Version
-------------	-------------	-----------------

CEP 4.2	CEF 1453@r1339, Chromium 27.0.1453.110, Webkit revision 151310	0.8.22
CEP 5.0	CEF 1453@r1339, Chromium 27.0.1453.110, Webkit revision 151310	0.8.22
CEP 6.1	CEF 3 release branch 2272, Commit e8e1f98ee026a62778eb2269c8e883426db645ea Chromium 41.0.2272.104	IO.js 1.2.0 from nw.js 0.12.1 (io.js is a fork of Node.js.)

Codex

- Product: CEP
- Version: 8.0.0
- Component: CEPHtmlEngine

Vulcan 5 Message and Control Libraries (Mandatory)

Starting with CEP 5.0 February release, the Vulcan 5 Control and Messaging libraries must be integrated by integrating applications to satisfy the dependency needs of PlugPlug/PlugPlugOwl. For most of the cases, you don't need to import header files of Vulcan 5 libraries and link your application against the libraries. You just pick up the libraries in your installer. Of course, you can also directly integrate Vulcan 5 to send and receive Vulcan messages from your application's native code. For this case, you will need to import the header files and link against the libraries.

Vulcan Message Library enables communication between Creative desktop applications(or any application integrating the library and deploying the required infrastructure) and HTML 5 extensions. It launches Vulcan IPC Broker if it is not running, and connects to it in order to send/receive messages.

Vulcan Control Library can be used to control Creative desktop applications(check if the application is installed, whether it is running, launch the application, etc.). It uses adobe_caps library (dll or framework) to access PCD data, so adobe_caps must be integrated by integrating applications either.

Please note that the Vulcan 5 libraries do not have a Codex entry under product 'CEP'. They are just put alongside the PlugPlug/PlugPlugOwl libraries on p4://dub-ham.corp.adobe.com:1760 so as to simplify the integration process.

Codex

- Product: Vulcan
- Version: 5.0.0
- Sub-product:
 - MessageLib - Vulcan Message library
 - ControlLib - Vulcan Control library
- Find Vulcan Release Notes at: [Vulcan 5 Release Notes](#)

PlugPlug External Object (Nice to Have)

PlugPlug External Object is a library which provides APIs to call from ExtendScript to HTML extension JavaScript. The library should be put into the folder containing PlugPlug library, so that it can be loaded by ExtendScript library.

For more details, please refer to "Access HTML DOM from extend script" in CEP HTML Extension Cookbook.

Please note that this feature should only be integrated if your product supports ExtendScript.

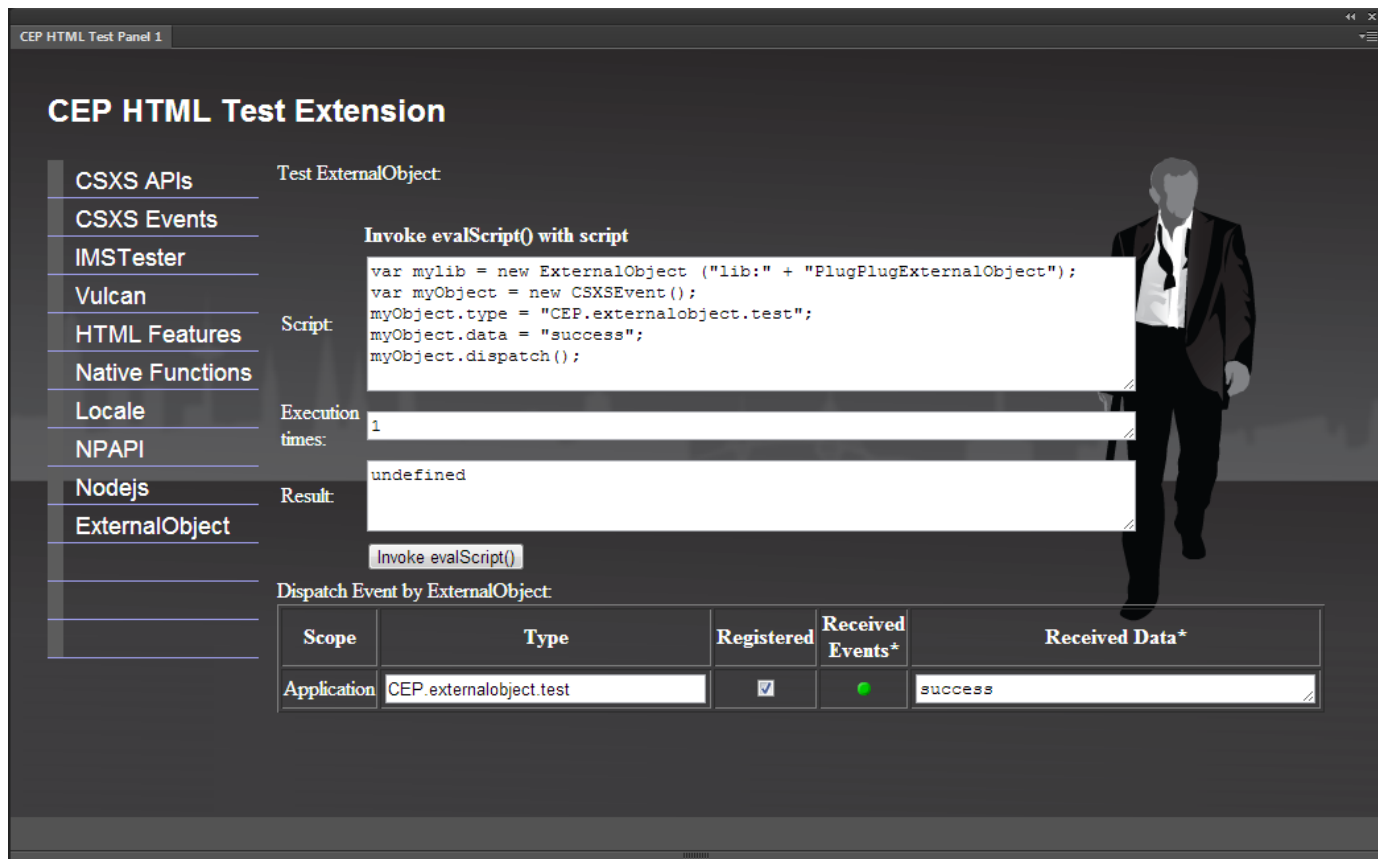
Codex

- Product: CEP
- Version: 8.0.0
- Component: PlugPlugExternalObject

Use CEP_HTML_Test_Extension to test this feature

Test Steps:

1. Open extension.
2. Click ExtenalObject Tab
3. Check "Registered".
4. Click "Invoke evalScript()" button.
5. If the green light is shown up, that means event is successfully sent from ExtentScript to Html extension. Please refer to the snapshot.



Below is the codex entry for test extension.

- Product: CEP
- Version: 8.0.0
- Component: CSXS Test Extensions

Point Product Integration Status

Product	Should integrate?	Integrated?	Works well?
AE	Yes	Yes	Win (Modal Dialog: Unable to execute script at line 0. After Effects error: Can not run a script while a modal dialog is waiting for reponse) Mac (Modal Dialog: Unable to execute script at line 0. After Effects error: Can not run a script while a modal dialog is waiting for response)
PR	Yes	Yes	Win Mac
PL	Yes	Yes	Win ExternalObject does not have a constructor Mac ExternalObject does not have a constructor
PS	Yes	Yes	Win Mac
AI	Yes	Yes	Win Mac
ID	Yes	Yes	
IC	Yes	No	
DW	No	No	

FL	No	No	
----	----	----	--

Notes:

- Those point product which do not support ExtendScript should not integrate PlugPlugExternalObject library.

CEP Demonstration Application

CEP Demonstration Application can be used as a model for integrating the various extensions the CC application. The source code of CEP Demonstration Application can be found here: <p4://dub-ham.corp.adobe.com:1760/releases/daily/csxs/5.0.0/demonstrationapplication/...>

Codex

- Product: CEP
- Version: 8.0.0
- Component: CSXS Demonstration Application

Signing Components

We have verified the CEP components work under Mac V2 signatures with the [Bast preflighting tool](#), here is the sample rule we used (to be modified to fit the application footprint):

```
staging.dmg/PlugPlugOwl.framework
staging.dmg/PlugPlugExternalObject.framework
staging.dmg/PlugPlug.framework
staging.dmg/CEPHtmlEngine.app/Contents/Frameworks/libplugin_carbon_interpose.dylib
staging.dmg/CEPHtmlEngine.app/Contents/Frameworks/Chromium Embedded Framework.framework/Libraries/libcef.dylib
staging.dmg/CEPHtmlEngine.app/Contents/Frameworks/Chromium Embedded Framework.framework/Libraries/ffmpegsumo.so
staging.dmg/CEPHtmlEngine.app/Contents/Frameworks/Chromium Embedded Framework.framework
staging.dmg/CEPHtmlEngine.app/Contents/Frameworks/CEPHtmlEngine Helper.app
staging.dmg/CEPHtmlEngine.app/Contents/Frameworks/CEPHtmlEngine Helper NP.app
staging.dmg/CEPHtmlEngine.app/Contents/Frameworks/CEPHtmlEngine Helper EH.app
staging.dmg/CEPHtmlEngine.app
```

Other Components for Native Integrators

AMT Library

AMT stands for Application Management Technologies. It is a unified architecture and a set of components for doing application management: installation, licensing, registration, updates, and so on. AMT should be integrated into the point product. The [CSXS-AMTLib Communication](#) wiki points to the changes in AMT for 5.5.

Scripting Engine Instance

You will need a scripting engine instance resident for the lifetime of your application. PlugPlug uses script fragments to implement the CEP Runtime so that all SWFs use a consistent naming scheme when querying for data. For example, each CEP extension has access to a core `getAppName():String` method. This method can be implemented in a scripting language that is supported by the host application.

CEP Components for Extension Developers

CEP JSLibrary

A set of JavaScript libraries that can be used by HTML extensions.

Codex

- Product: CEP
- Version: 5.0.0
- Component: JSLibrary

CEP ActionScript Library

The CEP Library is an ActionScript library that is compiled into each extension. Extensions use this library to make requests against the rest of the CEP Infrastructure.

Codex

- Product: CEP
- Version: 5.0.0
- Component: CSXS SWC

CEP IMS Library

CEP now delivers two libraries to help support Identity Management Service (IMS) related workflows:

- An Actionsript client library known as *CEPIMSLibrary* which enables CS extensions to integrate with IMS. Please refer to <http://sdk.eur.adobe.com/sdk/ims/asdoc/> for further information.
- A group of C APIs known as *IMS Native API* is available in *PlugPlug* to support integrating IMS related workflows in native applications. The functions they support are the same as the above *CEPIMSLibrary*. Please refer to <https://wiki.corp.adobe.com/display/csxs/IMS+Native+API> for further information.

For troubleshooting information please refer to: <https://wiki.corp.adobe.com/display/csxs/IMS+TroubleShooting>

Codex

- Product: CEP
- Version: 5.0.0
- Component: IMSLibrary SWC

CEP Test Extensions

A set of extensions that can be used to verify the integration with CEP.

Codex:

Product: CEP

Version: 5.0.0

Component: CSXS Test Extensions

Installation

Recommended Folder Structure

CEP does not deliver any STI starting with CEP 5.0. Integrators need to integrate CEP components as files in their installers. Here is recommended folder structure.

Windows:

```

<PP Folder>\CEP\html\... (CEP HTML engine folder)
<PP Folder>\CEP\html\CEPHtmlEngine.exe (CEP HTML engine)
<PP Folder>\CEP\extensions\... (All extensions installed by PP
installer)
<PP Folder>\CEP\extensions\AdobeExchange (Adobe Exchange extension)
<PP Folder>\PointProduct.exe
<PP Folder>\PlugPlug(Owl).dll (CEP PlugPlug or PlugPlugOwl
library)
<PP Folder>\PlugPlugExternalObject.dll (CEP PlugPlug External Object
library)
<PP Folder>\VulcanMessage5.dll (Vulcan Message library)
<PP Folder>\VulcanControl.dll (Vulcan Control library)

```

Mac:

```

<PP Folder>\PointProduct.app\Contents\Frameworks\PlugPlug(Owl).framework
(CEP PlugPlug(Owl) framework)
<PP
Folder>\PointProduct.app\Contents\Frameworks\PlugPlugExternalObject.framew
ork (CEP PlugPlug External Object framework)
<PP Folder>\PointProduct.app\Contents\Frameworks\VulcanMessage5.dylib
(Vulcan Message library)
<PP Folder>\PointProduct.app\Contents\Frameworks\VulcanControl.dylib
(Vulcan Control library)
<PP Folder>\PointProduct.app\Contents\MacOS\CEPHtmlEngine.app
(CEP HTML engine)
<PP Folder>\CEP\extensions\... (All
extensions installed by PP installer)
<PP Folder>\CEP\extensions\AdobeExchange (Adobe
Exchange extension)

```

Notes:

- The recommended folder structure is not mandatory. Integrators may choose their own folder structure if needed.
- Please do not mix CEP extensions with other plug-ins or panels. Instead, please create a separate extension folder in product folder. Since CEP searches manifest.xml in all its sub-folders, mixing CEP extensions with other plug-ins/panels will cause CEP take more time to initialize.

Tell CEP where to load its components

Several new members were added to struct PlugPlugHostData, enabling integrators to let CEP know where to locate certain components.


```

struct PlugPlugHostData
{
    ...
    /**
     * Path to the folder where CEP HTML engine is installed
     * \since 4.1
     */
    const char* hostCEPHtmlEngineDirectory;

    /**
     * Path to the folder where host extension is installed
     * \since 4.1
     */
    const char* hostExtensionDirectory;

    /**
     * Path to the folder where host StageManager is installed
     * \since 4.1
     */
    const char* hostStageManagerDirectory;
    ...

    /**
     * Path to the IMSLib.dll/dylib. Optional.
     * \since PLUGPLUG_API_VERSION_NUMBER_6
     *
     * The integrating application could use the AMTLib API
    AMTRetrieveLibraryPath
     * to retrieve the path to the IMSLib library in OOBEE's directory and
    pass the path
     * to CEP. If this field is NULL, PlugPlug will try to load the one
    bundled in the host
     * application.
     */
    const char* imsLibPath;
};

```

Not all these members are mandatory.

Struct member name	Description
hostCEPHtmlEngineDirectory	<p>MANDATORY if the integrating application supports HTML extension.</p> <p>Path to the directory where CEPHtmlEngine executable/bundle is installed, can be terminated with or without the slash. It is suggested that on Windows, you place the CEPHtmlEngine executable in a directory alongside your main executable. On Linux, place the CEPHtmlEngine bundle somewhere inside your main application bundle (otherwise, CEPHtmlEngine will not be found by LaunchPad).</p> <p>If you don't want HTML extension support, leave it NULL.</p>
hostExtensionDirectory	<p>MANDATORY if the integrating application wants to install extensions in its installation folder.</p> <p>Path to the directory where extensions are located in host application's installation folder, can be terminated with slash/backslash.</p>
hostStageManagerDirectory	Deprecated. Leave it NULL.

imsLibPath	<p>MANDATORY if the integrating application doesn't bundle IMSLib in its installer.</p> <p>If IMSLib is already bundled in the integrating application's installer, CEP will automatically search for the library & Otherwise, this member must be set so that CEP knows where to find the IMSLib library.</p> <p>It is suggested that you set this to the path to the system-wide IMSLib installed in OOB folder. To retrieve the p& OOB folder, refer to AMTLib API - https://wiki.corp.adobe.com/display/Argus/What%27s+new+in+AMTLib+7.0#b7.0-AMTLibAPItoretrieveIMSLibPath</p>
------------	---

Install CEPHtmlEngine

To support HTML extensions, you must include CEPHtmlEngine in your installer and have it installed in a directory in your installation folder, then let CEP know where to find it.

Install CEP Extensions

If you want to install any CEP extension (Adobe Exchange, Kuler, etc.) with your point product, please install to **product extension folder**. You need to unzip the extension ZXP file, sign by Bast, and include in your installer. You also have to let PlugPlug/PlugPlugOwl know where find those extensions.

Besides the point product extension folder, CEP also loads extensions from **system extension folder** and **per-user extension folder**. Globally installed extensions, such as the ones installed by Extension Manager, can go there. However, point products should NOT install extensions there.

See "CEP Extension Folders" for more details.

Integration

CEP Extensions/Services can be accessed in the host application from the Window->Extensions Menu. The procedure for integrating CEP with the application so the extensions can be accessed from this area is described in later sections.

Extension Window Types

The table below shows the types of window supported by CEP:

Window Type	Since CEP	HTML Extensions
Panel	beginning	Y
ModalDialog	beginning	Y
Modeless	beginning	Y
Custom	5.0	Y (used for Invisible HTML Extension)
Embedded	6.1	Y
Dashboard	6.1	Y (Owl applications only)

Special Notes about Window Types for HTML extension

To reduce integrator's efforts on integrating CEP, the creation, destruction of ModalDialog, Modeless and Custom windows are all handled by CEP. The behavior of these windows are not customizable. For Panel and Dashboard window, integrator needs to create a panel reference for each panel/dashboard HTML extension and pass the reference to CEP via the PlugPlugGetPanelFn callback function.

To support Invisible HTML extension, Custom window type must be supported.

Integrate Window->Extensions Menu

CEP needs a localized menu item "Window -> Extensions" as part of your application's main menu bar. This menu text should be part of your normal localization process. Initializing [PlugPlug Library](#) will supply your application with an ordered list (1 at a time) of menu names to insert under the "Extensions" item. These strings do not need to be part of your application's resources.

Everything that CSXS provides through the *PlugPlugSetMenu* *needs to be added to the Extensions menu. Do not filter out anything from the* *PlugPlugSetMenu* *call.*

Checklist for Integrating into the Window->Extensions menu.

After integrating the Window->Extensions menu, the following checklist can be used for maintaining consistency in the integration across the products:

Window->Extensions menu items

- Extension names to appear in alphabetical order
- A check mark should appear next to the extension's menu name when an extension is loaded and should be removed when the extension is unloaded

































CEP Events

Standard Events

For a list of events that are standardized over several point products and extensions, refer: [Guidance for CS5-CSXS Standardized Events](#)

Following table lists the standard events supported by Point Products now; please refer to [Support Event Notifications in CEP HTML Extensions](#) for more information:

( = supported,  = not supported)

Event Type	Event Scope	Description	Event Parameter	PS	ID	AI	FL	PR	PL
documentAfterActivate	APPLICATION	Event fired when a document has been activated (after new/open document; after document has retrieved focus).	URL to the active document. If the doc was not save, the NAME will be set instead of the URL.						
documentAfterDeactivate	APPLICATION	Event fired when the active document has been de-activated. (after document loses focus)	URL to the active document. If the doc was not save, the name will be set instead of the URL.						
applicationBeforeQuit	APPLICATION	Event fired when the application got the signal to start to terminate.	none						
applicationActivate	APPLICATION	Event fired when the Application got an "activation" event from the OS.	none					 on Mac;  on Windows	 on Mac;  on Windows
documentAfterSave	APPLICATION	Event fired after the document has been saved	URL to the saved document.						

Event after loading an extension

After fully loading an html extension, PlugPlug dispatches an event.

- Event Type: csxs::event::EVENT_TYPE_CSXS_EXTENSION_LOADED ("com.adobe.csxs.events.ExtensionLoaded")
- Event Data: It contains a XML string.
 - The value of "Id" is the extension ID which is loaded.
 - The value of "Params" is the start up parameters string of an extension. It is always empty string at present.

```
<ExtensionLoadedEvent><Id>%@</Id><Params>%@</Params></ExtensionLoadedEvent>
```

Event after unloading an extension

PlugPlug dispatches an event after unloading an extension.

- Event Type: `csxs::event::EVENT_TYPE_CSXS_EXTENSION_UNLOADED` ("com.adobe.csxs.events.ExtensionUnloaded")
- Event Data: It contains a XML string.
 - The value of "Id" is the extension ID which is unloaded.
 - The value of "ClosingType" is from `PlugPlugCloseExtensionType` enum.
 - For modal and mode-less dialog, the type is accurate.
 - For panel, clicking 'X' button cannot be detected.
- Please note that this event is only available to native APIs so the closing HTML extension cannot listen to this event to trigger a cleanup before close.

```
<ExtensionUnloadedEvent><Id>%@</Id><ClosingType>%d</ClosingType></ExtensionUnloadedEvent>
```

Types of unloading an HTML Extension

(Since 5.2)

CEP 5.2 supports the close types of an extension that indicates how an extension is closed.

```
typedef enum PlugPlugCloseExtensionType
{
    PlugPlugCloseExtensionType_None = 0,
    PlugPlugCloseExtensionType_CloseWindow,    //!< close extension by
clicking 'X' button
    PlugPlugCloseExtensionType_CallJSApi,      //!< close extension by
calling CSInterface.closeExtension
    PlugPlugCloseExtensionType_KillHtmlEngine, //!< close extension by
killing CEPHtmlEngine process
    PlugPlugCloseExtensionType_lastValue = 0x7FFFFFFF
} PlugPlugCloseExtensionType;
```

Integrate Capability Based Features

Capability based Feature Support

In CEP version 3.0 (CS6), we introduced Capability Based [PlugPlug](#) Library APIs. This section provides instructions for the host application on how to support a capability based feature.

Current Limitation

The [PlugPlug](#) Library provides the ability to support application specific features from within the context of a CS extension. For example, a user selecting an application specific menu item to execute some code in an extension. The current architecture of [PlugPlug](#) Library is slightly monolithic. Introducing new features into this architecture is difficult. Previously, there was a desire to have all applications support all new features, failure to do so could result in a runtime error. With this approach, it was difficult to evolve the features provided by the core.

Capability Based Approach

CEP has introduced a capability based feature concept into [PlugPlug](#) Library and CSXS library. An application can choose to support or not support a capability based feature. Extension developers can test the application's capability for supporting a feature before using it, or catch `com.adobe.csxs.error.NotSupportedError` to know if a feature is supported.

The following explains the capability based concept by using the "extended panel menu" capability as an example.

What to do if the host application supports a capability

- Initialize PlugPlugHostData to all zero first.
- Set necessary members, especially capability callback functions.
- Return true for supported capabilities in callback function PlugPlugGetHostCapabilitiesFn.

```
// PlugPlugTypes.h
typedef PlugPlugErrorCode (*PlugPlugGetHostCapabilitiesFn)
    (PlugPlugHostCapabilities* capabilities);

// Your code
void YourHousingPlugin::InitHostData(PlugPlugHostData& hostData)
{
    memset(&hostData, 0, sizeof(PlugPlugHostData));
    hostData.apiVersionNumber = PLUGPLUG_API_VERSION_NUMBER;
    // Set necessary members in hostData
    hostData.fnGetHostCapabilities = GetHostCapabilitiesFn;
}

static PlugPlugErrorCode GetHostCapabilitiesFn(PlugPlugHostCapabilities*
capabilities)
{
    if (capabilities)
    {
        capabilities->EXTENDED_PANEL_MENU = true;
        // other capabilities are false
    }
    return PlugPlugErrorCode_success;
}
```

What to do if host application does NOT support any capability

Set capability struct and callback functions in PlugPlugHostData to all 0. Nothing else. This can be do by initializing PlugPlugHostData with all 0 before setting values to it. It is highly possible that this is what you have done before.

```
void YourHousingPlugin::InitHostData(PlugPlugHostData& hostData)
{
    memset(&hostData, 0, sizeof(PlugPlugHostData));
    hostData.apiVersionNumber = PLUGPLUG_API_VERSION_NUMBER;
    // Set necessary members in hostData
}
```

Capability EXTENDED_PANEL_MENU

(Available from CEP 5.2.0.8 release for HTML extension)

Extended Panel Menu capability includes

1. menu and sub menu
2. enable/disable, check/uncheck panel menu items
3. update (enable/disable, check/uncheck) panel menu items

What to do if host application supports this capability

- Capability Based [PlugPlug](#) Library APIs are declared in `PlugPlugTypes.h`. Each capability is declared as one struct and one or more callback functions.

```
/*!  
 \brief called to update extension panel flyout menu items.  
  
 \param extensionId Update menu item(s) of this extension  
 \param itemName     Update menu item(s) with this name  
 \param enabled      true - enable menu item; false - disable menu item  
 \param checked      true - check menu item; false - uncheck menu item  
  
 \return an error code:  
 - PlugPlugErrorCode_success Menu item was successfully updated.  
 - PlugPlugErrorCode_unknown Some error occurred.  
  
 \since 3.0  
 */  
typedef PlugPlugErrorCode (*PlugPlugUpdateMenuItemFn)  
    (const char* extensionId, const char* itemName, bool enabled, bool  
checked);  
  
/*!  
 \brief Capability of extended panel menu  
 \since 3.0  
 */  
struct PlugPlugCapabilityExtendedPanelMenu  
{  
    /**  
     * Update extension panel flyout menu item  
     */  
    PlugPlugUpdateMenuItemFn fnUpdateMenuItem;  
};
```

- Host application needs to implement `PlugPlugUpdateMenuItemFn`. Details of a sample implementation are provided later.
- Host application needs to register `PlugPlugUpdateMenuItemFn` implementation to `PlugPlugHostData` struct before calling `PlugPlugSetup`.

```

struct PlugPlugHostData
{
    //...

    /**
     * Capability of extended panel menu
     * \since 3.0
     */
    PlugPlugCapabilityExtendedPanelMenu capabilityExtendedPanelMenu;
};

void YourHousingPlugin::InitHostData(PlugPlugHostData& hostData)
{
    // ...

    // UpdateMenuItemFn is implementation of callback function
    PlugPlugUpdateMenuItemFn.
    hostData.capabilityExtendedPanelMenu.fnUpdateMenuItem =
    UpdateMenuItemFn;

    // Host application should initialise the capability struct before
    calling PlugPlugSetup.
}

```

- Host application also needs to support new functionalities in existing PlugPlugSetMenuFn callback function.
 - **enabled** and **checked** in PlugPlugMenuItem
 - **submenu** in PlugPlugMenuItem- it may point to a sub menu

```

/*!
\brief Menu items as used in the app top-level (sub-)menus and in panel
flyout menus.

```

There are a number of conventions used to communicate what kind of menu item something is and thus how to handle it:

- an item with a NULL menu ID is disabled; it cannot be chosen. There is no "enablement" functionality: if PlugPlug wants to enable it then it will remake the whole menu. (Note that menu items for panels can be disabled; this is how PlugPlug controls mutually-conflicting extensions.)
- if the item name is "---" (three hyphens), then it's actually a separator. The menu ID in this case will always be NULL.
- an item with a non-null extension ID is for a panel and is owned directly by the Housing Plug-in. If one of these is chosen, the Housing Plug-in should load (or reveal) the indicated extension in its new (or existing) panel, using standard application workspace management. (These are not typically used in flyouts, but they are not forbidden there.)
- an item with a null extension ID is owned by PlugPlug. If one of these is chosen, the HousingPlugin should make a PlugPlugMenuCall.
- an item with a non-NULL submenu item is actually a submenu entry. In this case the menu ID and extension ID will always be NULL; only the name matters. The submenu points to the list of items to fill the submenu.

```

*/
struct PlugPlugMenuItem
{
    void* menuId;           //!< unique menu ID, if NULL menu is disabled
    const char* nameUtf8;   //!< Item title, if "---" item is a separator
    char* extensionId;      //!< optional extension ID, used for panels
only
    struct PlugPlugMenuItem *submenu;  //!< if non-NULL, this is a submenu
    struct PlugPlugMenuItem *next;    //!< next menu item when used in a list
    bool enabled;                    //!< true - enable menu item; false - disable
menu item. Optional. Default value is true.
    bool checked;                   //!< true - check menu item; false - uncheck
menu item. Optional. Default value is false.
};

```

```

/*!
\brief Menu items typically come in lists.
*/
typedef struct PlugPlugMenuItem* PlugPlugMenuList;

typedef PlugPlugErrorCode (*PlugPlugSetMenuFn)
    (const char *inExtensionID,
     const char *inMenuPosition,
     const PlugPlugMenuList inMenu);

```

Sample PlugPlugUpdateMenuItemFn Implementation on Windows

The sample code below is taken from the implementation in the Reference Application.

- PlugPlugConnector.cpp

```
static PlugPlugErrorCode UpdateMenuItemFn(const char* extensionId, const
char* itemName, bool enabled, bool checked)
{
    return housingPlugin->UpdateMenuItem(extensionId, itemName, enabled,
checked);
}
```

- ExtensionManager.cpp

```
PlugPlugErrorCode ExtensionManager::UpdateMenuItem(
    const char* extensionId, const char* itemName, bool enabled, bool
checked)
{
    PlugPlugErrorCode eRetVal = PlugPlugErrorCode_unknown;

    if (extensionId && itemName)
    {
        // flyout-menu in extension
        Extension* extension = extensionsMap[ string(extensionId) ];
        if (extension != NULL)
        {
            const UIContainer* container = extension->GetUIContainer();
            if (container != NULL)
            {
                const_cast<UIContainer*>(container)->UpdateMenuItem(itemName, enabled,
checked);

                eRetVal = PlugPlugErrorCode_success;
            }
        }
    }

    return eRetVal;
}
```

- PanelWindow.cpp

```

void PanelWindow::UpdateMenuItem(const char* itemName, bool enabled, bool
checked)
{
    if (m_controlRef && itemName)
    {
        ::OWLMenuRef flyoutMenu(NULL);
        if (::OWLPaletteGetMenu(m_controlRef, &flyoutMenu) == kOWLErrorNone
&& flyoutMenu)
        {
            UpdateMenuItem(flyoutMenu, itemName, enabled, checked);
        }
    }
}

void PanelWindow::UpdateMenuItem(::OWLMenuRef menu, const char* itemName,
bool enabled, bool checked)
{
    IVCString* itemNameToFind = IVCString::UTF8String(itemName);

    int count = ::GetMenuItemCount(menu);
    const int BUFSIZE = 128;
    for (int i = 0; i < count; ++i)
    {
        // update menu item
        WCHAR currentItemNameBuf[BUFSIZE] = {0};
        if (::GetMenuString(menu, i, currentItemNameBuf, BUFSIZE,
MF_BYPOSITION) > 0)
        {
            IVCString* currentItemName =
IVCString::String(currentItemNameBuf);
            if (currentItemName->Equals(itemNameToFind))
            {
                ::EnableMenuItem(menu, i, MF_BYPOSITION | (enabled ?
MF_ENABLED : (MF_GRAYED | MF_DISABLED)));
                ::CheckMenuItem (menu, i, MF_BYPOSITION | (checked ?
MF_CHECKED : MF_UNCHECKED));
            }
        }

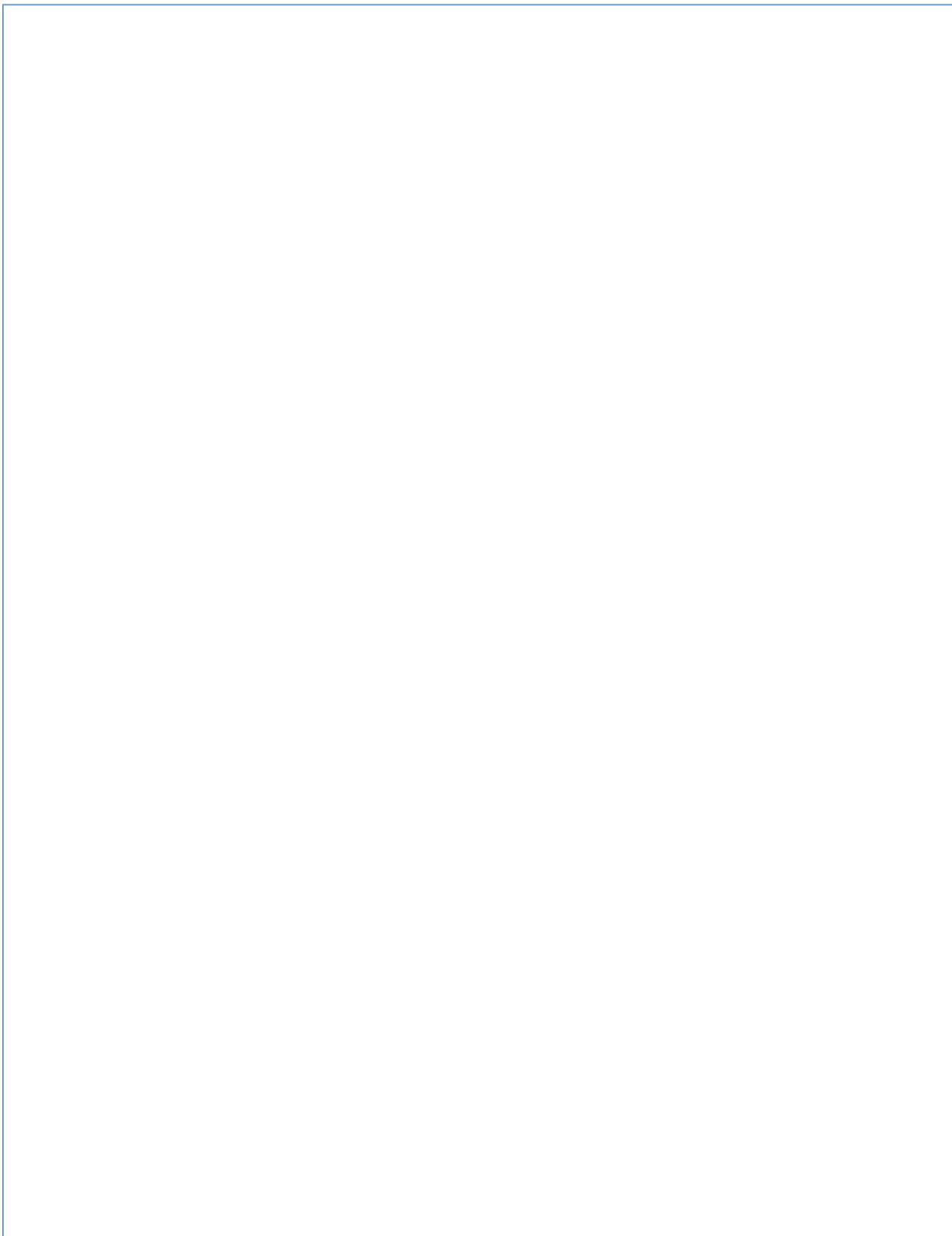
        // update submenu
        ::OWLMenuRef submenu = ::GetSubMenu(menu, i);
        if (submenu)
        {
            UpdateMenuItem(submenu, itemName, enabled, checked);
        }
    }
}

```

Sample PlugPlugSetMenuFn Implementation on Windows

The sample code below is taken from the implementation in the Reference Application.

- PlugPlugConnector.cpp has no change
- ExtensionManager.cpp has no change
- PanelWindow.cpp



```

void PanelWindow::AddMenuList(MenuList menuList)
{
    if (m_controlRef)
    {
        ::OWLMenuRef flyoutMenu = CreateMenu(menuList);
        ::OWLPaletteChangeMenu(m_controlRef, flyoutMenu, false);
    }
}

::OWLMenuRef PanelWindow::CreateMenu(MenuList menuList)
{
    ::OWLMenuRef menu = ::CreatePopupMenu();

    for (MenuItem* menuItem = menuList; menuItem; menuItem =
menuItem->next)
    {
        const char* const divider = "---";
        if (strncmp(menuItem->nameUtf8, divider, strlen(divider)) == 0)
        {
            // divider
            ::AppendMenuA(menu, MF_SEPARATOR, NULL, NULL);
        }
        else
        {
            // menu item
            UINT flags = (menuItem->enabled ? MF_ENABLED : (MF_GRAYED |
MF_DISABLED)) |
                        (menuItem->checked ? MF_CHECKED :
MF_UNCHECKED);
            UINT_PTR uIDNewItem = NULL;
            IVCString* menuStr = IVCString::UTF8String(menuItem->nameUtf8);
            if (menuItem->submenu)
            {
                // this menu item has submenu
                flags |= MF_POPUP;
                uIDNewItem = (UINT_PTR) CreateMenu(menuItem->submenu); //
submenu
            }
            else
            {
                // this menu item has no submenu
                flags |= MF_STRING;
                uIDNewItem = (UINT_PTR) menuItem->menuId; // menu
identifier
            }
            ::AppendMenu (menu, flags, uIDNewItem,
menuItem->CharactersPtr());
        }
    }

    return menu;
}

```

Test this Capability

You can use HTML Test Extension provided by CEP team to test your implementation for HTML extension.

- Open the extension in your applicatoin by Windows > Extensions > CEP HTML Test Panel 1
- Check host capabilities by CSXS APIs > Environment > Host Capabilities. If your application supports this capability, you can see EXTENDED_PANEL_MENU.
- Test panel menu by FlyoutMenu Tab. There is already a sample menu XML in Test Menu Structure.
 - Click "Set Panel Menu" and check if correct panel menu is created.
 - You can change menu XML and click "Set Panel Menu" again.
 - Click "Set null" and check if panel menu is cleared.

Capability EXTENDED_PANEL_ICONS

(Available since CEP 3.0)

With this capability, extension developers can use different panel icons for different host application theme colors. For example, use dark icons for dark theme color, and light icons for light theme color.

What point products **MUST** do, even if you do **NOT** support this capability.

The *PlugPlugFlashPanelItem* instance needs to be created in point product code, please use memset to initialize it to all zero. This is because new members can be added to PlugPlugFlashPanelItem. If you do not initialize new items to zero or set to meaningful value, a crash may occur in PlugPlug.

```
PlugPlugFlashPanelItem* tempItem = new PlugPlugFlashPanelItem;
memset(tempItem, 0, sizeof(PlugPlugFlashPanelItem));
// ... other code ...
```

Related APIs in PlugPlug.h.

```

struct PlugPlugFlashPanelItem
{
    const char *swfUTF8Path; /*!< location of the swf*/
    const char *scriptUTF8Path; /*!< location of the script file*/
    const char *iconUTF8PathNormal; /*!< window icon in the Normal State
for light theme */
    const char *iconUTF8PathRollOver; /*!< window icon in the Roll Over
State for light theme */
    const char *iconUTF8PathDisable; /*!< window icon in the Disabled State
(Not Used) */
    struct PlugPlugFlashPanelItem *next; /*!< next PlugPlugFlashPanelItem
in the list*/
    const char *iconUTF8PathDarkNormal; /*!< (since 3.0) window icon in
the Normal State for dark theme */
    const char *iconUTF8PathDarkRollOver; /*!< (since 3.0) window icon in
the Roll Over State for dark theme */
};

typedef struct PlugPlugFlashPanelItem* PlugPlugFlashPanelList;

PlugPlugDllExport PlugPlugErrorCode PlugPlugSetFlashPanelList
    (const char *inMenuPosition,
     PlugPlugFlashPanelList flashPanelList);

```

What else to do if point products support this capability.

- Return true for supported capabilities in callback function PlugPlugGetHostCapabilitiesFn.
- Handle new icon paths in callback function PlugPlugRegisterExtensionFn.

New icon path iconPathDarkNormal and iconPathDarkRollOver had been added to PlugPlugExtensionData.

```

// PlugPlugTypes.h

struct PlugPlugExtensionData
{
    PlugPlugExtensionPlayerType inPlayerType;
    const char *mainUTF8Path; /*!< location of the main content (e.g.
index.html / main.swf)*/
    const char *iconPathNormal; /*!< window icon in the Normal State for light
theme */
    const char *iconPathRollOver; /*!< window icon in the Roll Over State for
light theme */
    const char *iconPathDisable; /*!< window icon in the Disabled State (Not
Used) */
    PlugPlugWindowType windowType; /*!< type of window to create */
    PlugPlugUInt32 showWindowOnCreate; /*!< if non-zero, show the window
immediately, else keep it hidden until the extension requests that it be
shown */
    PlugPlugWindowGeometry *defaultGeometry; /*!< default creation geometry,
including positioning */
    PlugPlugWindowGeometry *minSize; /*!< minimum allowed size (no
positioning) */
    PlugPlugWindowGeometry *maxSize; /*!< maximum allowed size (no
positioning) */
    const char *iconPathDarkNormal; /*!< (since 3.0) window icon in the
Normal State for dark theme */
    const char *iconPathDarkRollOver; /*!< (since 3.0) window icon in the
Roll Over State for dark theme */
    const char *title; /*!< (since 4.0) the title of the window */
    const char *id; /*!< (since 4.0) the id of the extension */
    const char *bundleId; /*!< (since 5.2) the id of the extension's bundle */
    PlugPlugCloseExtensionType closeType; /*!< (since 5.2) Defines how to
close the extension */
    PlugPlugVersionType *version; /*!< (since 6.1) If there is no version info
in manifest.xml, the value is NULL. */
    const char *mainUTF8FolderPath; /*!< (since 6.1) location of the extension
root folder (e.g. C:\Program Files\Adobe\Adobe Photoshop CC
2015\Required\CEP\extensions\com.adobe.DesignLibraryPanel on Win)*/
    bool whiteListedExtension; /*!< (since 8.0) check if extension is
whiteListed for codesign verification*/
    unsigned int hostListSize; /* since 8.0 The size of hostlist specified
in this manifest */
    PlugPlugSupportedHost *hostList; /* since 8.0 The required hostlist
specified in this manifest */
};
typedef struct PlugPlugExtensionData PlugPlugExtensionData;

```

PlugPlugExtensionData is used in callback function PlugPlugRegisterExtensionFn. Application needs to handle these new icons when PlugPlug gives them.

```
// PlugPlugTypes.h
typedef PlugPlugErrorCode (*PlugPlugRegisterExtensionFn)
                          (const char* inExtensionID,
                           PlugPlugExtensionType inExtensionType,
                           PlugPlugExtensionData *inExtensionData);
```

Normal and RollOver icon should be used for light theme. DarkNormal and DarkRollOver icon should be added for dark theme. Disabled icon should not use. All these icons are optional.

Here are the rules to use these icons.

- If both Normal and DarkNormal are provided, Normal should be used for light theme, and DarkNormal should be used for dark theme.
- If only one of Normal or DarkNormal is provided, it should be used for all themes.
- If no icon is provided, host application should use a default icon.
- Host application can use RollOver and DarkRollOver for rollover state, or use its own way of rollover.

Here is a demonstration.

light theme	----->	dark theme
Normal		DarkNormal
Normal		
DarkNormal		
host application's default icon		

Test this capability

Integrators can use CS API Test extension (CS_API_Test_Extension-5.0.zxp) provided by CEP team to test their integration.

- Launch the extension by Windows > Extensions > CS API 1
- Environment > Host Capabilities, check if EXTENDED_PANEL_ICONS is supported.
- Change application theme and check if panel icon is changed accordingly.

High DPI Panel Icons

In high DPI display mode, panel extensions may want to use high DPI icons. Host applications should be able to find and load those icons. For example, an extension sets these icons in manifest.

```
<Icons>
<Icon Type="Normal">./images/IconLight.png</Icon>
<Icon Type="RollOver">./images/IconLight.png</Icon>
<Icon Type="DarkNormal">./images/IconDark.png</Icon>
<Icon Type="DarkRollOver">./images/IconDark.png</Icon>
</Icons>
```

Host applications should use

- IconLight.png and IconDark.png for normal display
- IconLight@2X.png and IconDark@2X.png for 200% high DPI display

@2X.ext is the industry standard. Please see more details on https://developer.apple.com/library/ios/qa/qa1686/_index.html.

DELEGATE_APE_ENGINE

Deprecated, set it to FALSE always.

SUPPORT_HTML_EXTENSIONS

Set this capability to true if host application wants to support HTML extension, false otherwise.

DISABLE_FLASH_EXTENSIONS

Deprecated, set it to TRUE always.

Hosts supported by extension bundle

PlugPlugExtensionData is used in callback function PlugPlugRegisterExtensionFn. Application needs to get the hostlist from extensiondata. This is the list specified in "/ExtensionManifest/ExecutionEnvironment/HostList" in manifest file

```
// PlugPlugTypes.h
typedef PlugPlugErrorCode (*PlugPlugRegisterExtensionFn)
                           (const char* inExtensionID,
                            PlugPlugExtensionType inExtensionType,
                            PlugPlugExtensionData *inExtensionData);

struct PlugPlugExtensionData {
    ...
    ...
    unsigned int hostListSize; /* since 8.0 The size of hostlist specified in
this manifest */
    PlugPlugSupportedHost *hostList; /* since 8.0 The required hostlist
specified in this manifest */
}
/*! \brief Store the supported host data from manifest, the service will
fill in this data. \Since 8.0 */
typedef struct
{
    char* title;
    char* versionRange;
} PlugPlugSupportedHost;
```

Supporting CEP Highbeam Data Collection

CEP integrates Headlights since CEP 4. It creates its own Headlights log session to log CEP data. When a host application is running, there might be two headlights sessions. One is for host application itself(host application integrates Headlights too), the other is for CEP component. However, things are changed when Headlights is upgraded to Highbeam. The default OptIn state was changed from "OptOut" to "OptIn". It means there are more concurrent Highbeam sessions. If CEP still uses its own session, it will double the session count of Highbeam, which is not acceptable for Highbeam backend. Highbeam suggests CEP share the Highbeam session of host application. CEP 7 has made changes to share the Highbeam session of host application.

In PlugPlugHost, a new field called "highbeamSessionHandle" was added. Host application should assign its Highbeam session handle to it. If this field is not set, or it is set to NULL, CEP will not log any Highbeam data.

```

/*!
\brief The host environment data for PlugPlug startup.
At initialization, the app must supply all the environmental data PlugPlug
needs to do its work. This structure encapsulates that data.
*/
struct PlugPlugHostData
{
    .....

    /**
     * Please give the handle of Highbeam session. CEP will use this session
to log CEP Highbeam data.
     * \since PLUGPLUG_API_VERSION_NUMBER_14
     */
    void*    highbeamSessionHandle;

    /*@}*/
};

```

This handle is maintained by host application. CEP does not close it. CEP just uses this session to log its Highbeam data. **Host application should not close this session until it calls PlugPlugTerminate to terminate CEP.** As a result, CEP Highbeam data are written to the same Highbeam log file of host application. In order to avoid messing up the data, CEP puts all its data under category "CEP".

```

<NewGroup category="CEP" subcategory="HostEnvironment"
name="HostEnvironmentInfo" time="2015-12-05T00:08:11Z" group="9" doc="0"
seq="44" />
<DataRec time="2015-12-05T00:08:11Z" group="9" name="appName" data="PHXS"
seq="45" />
<DataRec time="2015-12-05T00:08:11Z" group="9" name="appVersion"
data="16.1.0" seq="46" />
<DataRec time="2015-12-05T00:08:11Z" group="9" name="locale" data="en_US"
seq="47" />
<DataRec time="2015-12-05T00:08:11Z" group="9" name="userInterfaceLocale"
data="en_US" seq="48" />
<DataRec time="2015-12-05T00:08:11Z" group="9" name="cepVersionNumber"
data="6.1.0.495" seq="49" />
<DataRec time="2015-12-05T00:08:11Z" group="9" name="cepApiVersion"
data="13" seq="50" />
<NewGroup category="CEP" subcategory="Extension" name="ExtensionInfo"
time="2015-12-05T00:08:12Z" group="10" doc="0" seq="51" />
<DataRec time="2015-12-05T00:08:12Z" group="10" name="id"
data="com.adobe.ccx.start" seq="52" />
<DataRec time="2015-12-05T00:08:12Z" group="10" name="name" data="Start"
seq="53" />
<DataRec time="2015-12-05T00:08:12Z" group="10" name="type" data="HTML"
seq="54" />
<DataRec time="2015-12-05T00:08:12Z" group="10" name="windowType"
data="Dashboard" seq="55" />
<DataRec time="2015-12-05T00:08:12Z" group="10" name="loadingTime"
data="313" seq="56" />
<DataRec time="2015-12-05T00:08:12Z" group="10" name="nodejsEnabled"
data="0" seq="57" />
<DataRec time="2015-12-05T00:08:12Z" group="10" name="mixedContext"
data="0" seq="58" />

```

Supporting HTML Extension

Here are the key factors for supporting HTML extensions.

Enabling `SUPPORT_HTML_EXTENSIONS` capability

See `SUPPORT_HTML_EXTENSIONS`.

Implementing `PlugPlugGetPanelFn` callback

CEP handled most of the stuff required by integrating HTML extension. However, there still are a few things to be handled by host application. See [here](#).

PlugPlugTypes.h

```
typedef PlugPlugErrorCode (*PlugPlugGetPanelFn)(const char* extensionId,
void** panel, PlugPlugHtmlViewCallbacks* viewCallbacks);
...
struct PlugPlugCapabilityDelegateEngine
{
    /**
     * Gets a panel window reference.
     */
    PlugPlugGetPanelFn fnGetPanel;

    /**
     * Engine allocated callback
     */
    PlugPlugEngineAllocatedFn fnEngineAllocated;

    /**
     * Engine disposed callback
     */
    PlugPlugEngineDisposedFn fnEngineDisposed;

    /**
     * The default unload policy for extensions created by PlugPlug.
     * @see PlugPlugExtensionUnloadPolicy
     */
    PlugPlugExtensionUnloadPolicy unloadPolicy;
};
```

The key is to implement the PlugPlugGetPanelFn callback. All other fields in the above struct are fine be be set to NULL or default value.

When CEP is about to load a panel HTML extension, CEP will call this callback with the ID of the extension to load, expecting below things to be returned from the host application

- A reference to a panel (OWL Palette, or Drover panel). The panel reference will be the parent view of HTML content.
 - For Dashboard extension, use below code to get a reference to a welcome screen panel.

```
OWLLeafControlRef welcomePanel = NULL;
OWLError errCode = OWLWorkspaceGetWelcomeScreen(&welcomePanel);
```

Don't forget to call this to enable welcome screen:

```
OWLWorkspaceWelcomeScreenChangeIsEnabled(true);
```

- A PlugPlugHtmlViewCallbacks struct that contains a set of optional callbacks (only one callback at the moment). These callbacks will be called when CEP wants to manipulate the panel for the given extension. For example, PlugPlugCloseHtmlViewFn will be called when the panel needs to close. But **DO NOT** call PlugPlugUnloadExtension in this function, this will lead to crash, because PlugPlug is unloading the same extension when calls PlugPlugCloseHtmlViewFn.

Implementing PlugPlugGetCurrentImsUserIdFn callback

Host application should implement callback **PlugPlugGetCurrentImsUserIdFn** to retrieve the GUID of the user that is currently logged in to the IMS system. CEP exposes a Javascript API **imsGetCurrentUserId** for extensions to retrieve current IMS user's GUID, which depends on host application implementing this callback. You may use AMTLib's **AMTRetrievePersonGUIDWithAuthSource** API ([AMTLib API Documentation](#)) to retrieve the current signed in user.

```
typedef PlugPlugErrorCode (*PlugPlugGetCurrentImsUserIdFn) (const char**
outResult);
...
PlugPlugGetCurrentImsUserIdFn fnGetCurrentImsUserId;
```

Implementing PlugPlugGetAppMainWindowFn callback (Windows only)

(Since 5.1)

To fix Watson [bug 3736982](#), CEP 5.1 added a callback function to get host application's main window, so that PlugPlug can set the main window as html extension's parent window. All host applications need to implement this callback in order to avoid the bug. This callback is only used on Windows platform.

```
/*!
 \brief This is for retrieving the main window handle of point product.
 Only used on Windows platform.
 <code>
 static PlugPlugErrorCode PlugPlugGetAppMainWindowFn(void** mainWindow)
 {
  *mainWindow = appMainWindow;
  return PlugPlugErrorCode_success;
 }
 </code>
 \param[out] mainWindow The main window handle.
 \return an error code:
 - PlugPlugErrorCode_success Successfully.
 - PlugPlugErrorCode_unknown Some error occurred.
 \since 5.1 (CC 2014)*/
typedef PLUGPLUG_CALLBACK_API(PlugPlugErrorCode,
PlugPlugGetAppMainWindowFn) (void** mainWindow);
```

Supporting Hi-DPI display on Windows platform

- CEP 5.2 introduces a new API `PlugPlugHostData::scaleFactor` that indicates the scale factor of point products on Windows platform. Point product needs to set this value when initializing PlugPlug. If PP dose not set this value, PlugPlug will calculate it according to system setting.

```

struct PlugPlugHostData
{
    .....
    /**
     * Please give the current scale factor on Windows platform, and set to
     * 0.0 on Mac platform.
     * \since PLUGPLUG_API_VERSION_NUMBER_10
     */
    float scaleFactor;
};

```

- If point product enables DPI Awareness, Html extension will be rendered in Hi-DPI mode automatically if extension is displayed in Hi-DPI device.
- PlugPlugExtensionData::minSize, PlugPlugExtensionData::maxSize and PlugPlugExtensionData::defaultGeometry are the original sizes which are from manifest.xml. Those are passed to PP when PlugPlug calls PlugPlugRegisterExtensionFn. Point products should handle them to adapt Hi-DPI display by multiplying the scaleFactor.

Setting extension specific scale factor

```

/*
\param[in] extensionid: The name of the extension.
\param[in] scalefactor: The scale factor to be set for given extension.
\return a true or false. A true return means a scale factor was set.
*/
typedef PLUGPLUG_CALLBACK_API(bool, PlugPlugSetExtensionScaleFactorFn)
(const char* extensionid, float* scalefactor);

```

Handling scale factor automatically by CEP - Set the scale factor to 0 using PlugplugSetup.

Global scale factor - This can be set by using old api PlugPlugSetScaleFactor. If you do not provide extension specific scalefactor then Global scale factor will be used. This is existing behaviour.

Extension specific scale Factor - Bind a function with fnGetExtensionScaleFactor field of PlugPlugHostData which will have two parameters extension id and scalefactor. In this callback, user has to set scale factor for given extension. This always overrides previous 2 options.

Binding function pointer to *PlugPlugSetExtensionScaleFactorFn* in point product

```

/* Define function to set extension scale factor
bool msetExtensionScaleFactor (const char *extension_id, float
*scalefactor) {
/* set the scale factor for extension with given extension_id here
* otherwise do not set any value
*/
}

/* binding function to a boost library */
boost::function<bool(const char*, float*)> s_setExtensionScaleFactor;
s_setExtensionScaleFactor = boost::bind(&msetExtensionScaleFactor, this,
_1, _2);
bool setExtensionScaleFactor (const char *extension_id, float *scalefactor)
{
return s_setExtensionScaleFactor(extension_id, scalefactor);
}

/*Binding above function to fnSetExtensionScaleFactor field of
PlugPlugHostData structure in PlugPlugSetup function.*/
PlugPlugHostData hostdata;
hostdata.fnSetExtensionScaleFactor = setExtensionScaleFactor;

```

ScaleFactor used by PlugPlug in different scenario

Extension Specific ScaleFactor	Global ScaleFactor	Action Taken
X	0	Automatic handling of ScaleFactor by CEP
0	0	ScaleFactor used is 1.0
C	0	ScaleFactor used is C
X	C	ScaleFactor used is C
0	C	ScaleFactor used is 1.0
C1	C2	ScaleFactor used is C1
X	X	Automatic handling of ScaleFactor by CEP
0	X	ScaleFactor used is 1.0
C	X	ScaleFactor used is C

X - Value not set

C - Some value greater than 0

Extension Signature Validation

Since CEP 6.1.0.138, a new flag named 'verifySignatureInEngine' is introduced in PlugPlugHostData to specify whether extension signature validation should be done in CEPHtmlEngine. This flag is for Html extensions only.

```

    /*!
    \brief The host environment data for PlugPlug startup.
    At initialization, the app must supply all the environmental data PlugPlug
    needs to do its work. This structure encapsulates that data.
    */
    struct PlugPlugHostData
    {
        /*!
        @name Host
        The application identification for the host in the manifest.
        @{
        */

        //.....

        /**
        * Please indicate if verify the signature of extensions on Html engine
        side.
        * \since PLUGPLUG_API_VERSION_NUMBER_13
        */
        bool    verifySignatureInEngine;

        /**
        * Please give callback function to get the verification result.
        * \since PLUGPLUG_API_VERSION_NUMBER_13
        */
        PlugPlugSetVerificationResultFn fnSetVerificationResult;

        /*@}*/
    };

    typedef struct PlugPlugHostData PlugPlugHostData;

```

When the flag is true, extension signature validation is done in CEPHtmlEngine instead of PlugPlug/PlugPlugOwl, reducing execution time of PlugPlugLoadExtension and PlugPlugLoadExtensionEx because multiple CEPHtmlEngine instances can verify the signatures of Html extensions concurrently. When the flag is false, the behavior is the same as before(PlugPlug/PlugPlugOwl will verify the signatures of Html extensions one by one).

When the flag is true, the result of PlugPlugLoadExtension and PlugPlugLoadExtensionEx is returned asynchronously via the 'fnSetVerificationResult' callback function, which is newly added to PlugPlugHostData along with the 'verifySignatureInEngine' flag. This callback function must be non-null when the 'verifySignatureInEngine' flag is true. When verification fails, host application should call PlugPlugUnloadExtension or PlugPlugUnloadExtensionEx to unload the extension. PlugPlugBelow is the definition of this callback.


```

/*
\brief This is the Housing Plug-in entry for getting the signature
verification result of the extension.
If host application sets verifySignatureInEngine of PlugPlugHostData to
true in PlugPlugSetup, html extension signature verification
will be in asynchronous mode. The verification result will be sent to host
application by this callback.
\param extensionId The extension id
\param verificationPassed true - Signature verification is successful.
false - Signature verification fails. Host application should call
PlugPlugUnloadExtension to release
the CEP resources allocated for this extension. For panel style extensions
(Panel, Embedded, Dashboard),
host application can release the window resources of this extension in this
callback.
\return a success or error code.
\since PLUGPLUG_API_VERSION_NUMBER_13, 6.1 (CC 2015)
*/
typedef PLUGPLUG_CALLBACK_API(PlugPlugErrorCode,
PlugPlugSetVerificationResultFn)(const char* extensionId, bool
verificationPassed);

```

Disabling Extension Signature Verification

Due to legal concerns, the flag below won't take effect in CEP 5.0 releases. Setting it to either true or false won't make any difference.

A new member in struct PlugPlugHostData can be used to bypass the signature verification for extensions.

```

struct PlugPlugHostData
{
    ...
    /**
     * Flag indicating whether extension signature verification is enabled
     or disabled.
     * \since PLUGPLUG_API_VERSION_NUMBER_8
     */
    bool disableSignatureVerification;
    /*@}*/
};

```

Setting this to 'true' will cause CEP not to do signature verification for all the extensions for your product. Use this at your own risk.

Native IMS APIs

The PlugPlug library provides APIs to support the following AAM/IMS workflows:

- Retrieval of all accounts that have signed in on the current device
- Retrieval of an *access token* for a particular user and service
- Retrieval of a *continue token* for a particular user and service.
- Setting of proxy credentials
- Launching of Adobe Application Manager (AAM) to enable a user to log in with their Adobe ID
- Supporting Single Sign On workflows from Desktop to Web

PlugPlugIMSConnect

The first call should be PlugPlugIMSConnect to establish a connection with IMSLib. You can create a session for the lifetime of your application or alternatively create a session per a task.

Please note that ever PlugPlugIMSConnect request must have a corresponding PlugPlugIMSDisconnect request.

PlugPlugIMSConnect API

```

/ * !
\ brief Retrieves an Identity Management Service(IMS) reference so the
native client can call any IMS API. This function is not thread safe.
\ param[in]  inCallerID      Caller ID.
\ param[out] outPlugPlugIMSRef An IMS reference returned from IMS. The
memory of returned PlugPlugIMSRef is controlled by PlugPlug.
                                DO NOT delete it. If user wants to keep the
IMS reference string, please copy it.
\ return a success or error code.
* /
PLUGPLUG_API(PlugPlugIMSErrorCode) PlugPlugIMSConnect(const char*
inCallerID, PlugPlugIMSRef* outPlugPlugIMSRef);

```

The following is an example of calling the PlugPlugIMSConnect API. Once you should receive a PlugPlugIMSErrorCode_SUCCESS result you can start to complete any of the IMS workflows stated above.

Connecting to the IMS Native APIs

```

try{      PlugPlugIMSRef ref = NULL;
          PlugPlugIMSErrorCode result = PlugPlugIMSConnect("MyCallerID",
&ref);    if (PlugPlugIMSErrorCode_SUCCESS == result)      {          //
Should copy the string.      std::string
refStr(ref);    }      else      {          // Connect
failed.    } } catch(PlugPlugException& e) {          // An error occurred.
This is more than likely due to invalid parameters being
passed.      PlugPlugErrorCode errorCode = e.GetErrorCode(); }

```

[Corresponding IMSLib API](#)

PlugPlugIMSConnectWithEndpoint

Please note that ever PlugPlugIMSConnectWithEndpoint request must have a corresponding PlugPlugIMSDisconnect request.

PlugPlugIMSConnectWithEndpoint API

```
/*!
\brief Retrieves an Identity Management Service(IMS) reference so the
native client can call any IMS API. This function is not thread safe.
\param[in] inCallerID Caller ID.
\param[in] inIMSEndPoint The IMS URL which needs to be passed in the
format like "ims-nal.adobelogin.com".
\
If inIMSEndPoint is set to empty string or
null, IMSLib will use production endpoint of IMS.
\param[out] outPlugPlugIMSRef An IMS reference returned from IMS. The
memory of returned PlugPlugIMSRef is controlled by PlugPlug.
DO NOT delete it. If user wants to keep the IMS reference string,
please copy it.
\return a success or error code.
*/
PLUGPLUG_API(PlugPlugIMSErrorCode) PlugPlugIMSConnectWithEndpoint(const
char* inCallerID, const char* inIMSEndPoint, PlugPlugIMSRef*
outPlugPlugIMSRef);
```

The following is an example of calling the PlugPlugIMSConnectWithEndpoint API. Once you should receive a PlugPlugIMSErrorCode_SUCCESS result you can start to complete any of the IMS workflows stated above.

Connecting to the IMS Native APIs

```
try{
    PlugPlugIMSRef ref = NULL;
    PlugPlugIMSErrorCode result =
PlugPlugIMSConnectWithEndpoint("MyCallerID", "EndPointName", &ref);
    if (PlugPlugIMSErrorCode_SUCCESS == result)
    {
        // Should copy the string.
        std::string refStr(ref);
    }
    else
    {
        // Connect failed.
    }
}
catch(PlugPlugException& e)
{
    // An error occurred. This is more than likely due to invalid
parameters being passed.
    PlugPlugErrorCode errorCode = e.GetErrorCode();
}
```

Corresponding IMSLib API

PlugPlugIMSFetchAccounts

The next step is usually to try and retrieve the active user(s) on the device. There are a number of APIs available for this including using [AMTLib's AMTRetrievePersonGUIDWithAuthSource](#) through our [CEP callback](#) however for this example we are going to use PlugPlugIMSFetchAccounts.

This call and others take the client ID for your extension as a parameter.

- If the response is empty, use the Adobe Application Manager to prompt the user to log in, then attempt to fetch accounts again before proceeding.
- If the response contains a list of accounts, you can prompt the user to select the account with which they would like to connect.
- If one is available, you can select the default user. This is the most recent active user of your extension on the current device.

PlugPlugIMSFetchAccounts

```
\brief Retrieves user account information from the Identity Management
Service(IMS). This function is not thread safe.
\param[in]  inCallerID      Caller ID.
\param[in]  inPlugPlugIMSRef An IMS reference returned from the
PlugPlugIMSConnect call.
\param[in]  inClientID      The unique identifier of the client.
\param[out] outAccounts     The non NULL char pointer. To this pointer
IMSLib will allocate NULL terminated XML string, which constitutes of user
data.
                                The memory of returned string is controlled
by PlugPlug. DO NOT delete it.
                                If user wants to keep the account information
string, please copy it.
\return a success or error code.
*/
PLUGPLUG_API(PlugPlugIMSErrorCode) PlugPlugIMSFetchAccounts(const char*
inCallerID, PlugPlugIMSRef inPlugPlugIMSRef, const char* inClientID, const
char** outAccounts);
```

The following is an example of using the PlugPlugIMSFetchAccounts API.

PlugPlugIMSFetchAccounts Code Sample

```
try
{
    const char* accounts = NULL;
    PlugPlugIMSErrorCode result = PlugPlugIMSFetchAccounts("MyCallerID",
myIMSRef, "MyClientID", &accounts);
    if (PlugPlugIMSErrorCode_SUCCESS == result)
    {
        // User should copy the returned accountns string. This string is a
XML format string.
        std::string accountsStr(accounts);
        // You can now use the XML API to traverse the string. If you want
to use the actively signed in user
        // then look for the default="true" attribute. However please be aware
that if the user has not signed in
        // with the Client ID passed in the PlugPlugIMSFetchAccounts then this
attribute may not be found.
        // If this happens you have two options:
        // 1) Take the first user on the list. Particularly useful if only one
user is returned.
        // 2) Show AAM and prompt the user to login. See PlugPlugShowAAM API for
more information.
    }
}
```

```

        else
        {
            // Fetch failed.
        }
    }
    catch(PlugPlugException& e)
    {
        // An error occurred if pass the invalid parameters.
    }

    /*
    The accounts should be like the following structure:
    <UserAccounts>
        <UserData default="true">
            <UserID></UserID>
            <Name></Name>
            <FirstName></FirstName>
            <LastName></LastName>
            <Email></Email>
            <CountryCode></CountryCode>
            <SAOList>
                <SAOData id="">
                    <ServiceCode></ServiceCode>
                    <ServiceStatus></ServiceStatus>
                    <ServiceLevel></ServiceLevel>
                </SAOData>
            </SAOList>
        </UserData>
    </UserAccounts>

```

* /

Corresponding IMSLib API

PlugPlugIMSFetchAccessTokenWithStatus

Once you have found the active user on the device you will want to retrieve an access token to authorize access to your service. This is achieved using the PlugPlugIMSFetchAccessTokenWithStatus API.

PlugPlugIMSFetchAccessToken

```

/ * !
\ brief Fetches an Access Token from Identity Management Service(IMS) for a
given user and service.
This function is asynchronous, and will call inCallback function when
PlugPlug gets Access Token data.
Please note the the inCallback function is not guaranteed to be called on
the main thread.
\ param[in] inCallerID           Caller ID.
\ param[in] inPlugPlugIMSRef     An IMS reference returned from the
PlugPlugIMSConnect call.
\ param[in] inClientID           The unique identifier of the client.
\ param[in] inClientSecret       The secret code of the client.
\ param[in] inUserAccountGUID    The user account GUID against which access
token needs to be generated.
\ param[in] inCallback           A callback function which is called when
PlugPlug gets the Access Token data.
                                The callback function is not guaranteed to
be called on main thread.
\ param[in] inServiceAccountGUID The service account GUID of the user
account. This will not be used currently.
\ param[in] inScope              The comma delimited scopes for which the
refresh token was requested.
\ return a success or error code.
*/
PLUGPLUG_API(PlugPlugIMSErrorCode)
PlugPlugIMSFetchAccessTokenWithStatus(const char* inCallerID,

PlugPlugIMSRef inPlugPlugIMSRef,

                                const
char* inClientID,

                                const
char* inClientSecret,

                                const
char* inUserAccountGUID,

::csxs::event::IMSCallbackWithStatus inCallback,

                                const
char* inServiceAccountGUID = 0,

                                const
char* inScope = "openid,AdobeID");

```

inCallerID, inPlugPlugIMSRef, inClientID, inClientSecret, inUserAccountGUID and inCallback are all required parameters. An initial PlugPlugIMSErrorCode will be returned by the API to inform the user whether the request for an access token has been successful. However this API is asynchronous so the actual access token is returned to the provided Callback.

*Please note that the parameter **inCallback** of function **PlugPlugIMSFetchAccessToken** is not guaranteed to be called on the main thread.*

The following is an example of using the PlugPlugIMSFetchAccessTokenWithStatus API. If you are returned PlugPlugIMSErrorCode_SUCCESS then you can extract the access token and use it to connect to your service, If you are returned an error then please refer to the ['What errors should I care about'](#) table to help you determine your next set of actions.

PlugPlugIMSFetchAccessTokenWithStatus Code Sample

```
// Define a callback function
// The parameter "details" of callback function is a JSON string that
contains the detailed access token information.
void FetchAccessTokenCallback(int status, const char* const details)
{
    // process the returned JSON string here.
}

// elsewhere in the code...
try
{
    // The "MyUserID" is the entityRef returned by
    PlugPlugIMSFetchAccounts e.g. 123456780@AdobeID
    PlugPlugIMSErrorCode result =
    PlugPlugIMSFetchAccessToken("MyCallerID", myIMSRef, "MyClientID",
    "MyClientSecret", "MyUserID", FetchAccessTokenCallback);
    if (PlugPlugIMSErrorCode_SUCCESS != result)
    {
        // An error occurred when attempting to fetch an access token.
    }
}
catch(PlugPlugException& e)
{
    // An error occurred if pass the invalid parameters.
}
```

[Corresponding IMSLib API](#)

PlugPlugIMSFetchContinueToken

PlugPlugIMSFetchContinueToken

```
/*!
\brief Fetches a continue Token from Identity Management Service(IMS) for a
given user and service.
This function is asynchronous, and will call inCallback function when
PlugPlug gets Continue Token data.
Please note the inCallback function is not guaranteed to be called on the
main thread.
\param[in] inCallerID          Caller ID.
\param[in] inPlugPlugIMSRef    An IMS reference returned from the
PlugPlugIMSConnect call.
\param[in] inBearerToken       The access token issued by IMS.
\param[in] inClientID          The unique identifier of the client.
\param[in] inCallback          A callback function which is called when a
continue token is received.
\param[in] inredirectUri       (optional) The URL to jump to. It should be
URL encoded.
\param[in] inScope             (optional) The comma delimited scopes for
which the refresh token was requested.
\param[in] inresponseType      (optional) The IMS integration type the
target has.
\param[in] inlocale            (optional) The locale in the format
language_country.
*/
PLUGPLUG_API(PlugPlugIMSErrorCode) PlugPlugIMSFetchContinueToken(const
char* inCallerID,
                        PlugPlugIMSRef inPlugPlugIMSRef,
                        const char* inBearerToken,
                        const char* inClientID,
                        ::csxs::event::IMSCallbackWithStatus inCallback,
                        const char* inredirectUri = "",
                        const char* inScope = "",
                        const char* inresponseType = "",
                        const char* inlocale = "en_US");
```

An initial PlugPlugIMSErrorCode will be returned by the API to inform the user whether the request for an continue token has been successful. However this API is asynchronous so the actual continue token is returned to the provided Callback.

*Please note that the parameter **inCallback** of function **PlugPlugIMSFetchContinueToken** is not guaranteed to be called on the main thread.*

The following is an example of using the PlugPlugIMSFetchContinueToken API. If you are returned PlugPlugIMSErrorCode_SUCCESS then you can extract the continue token and use it to connect to your service, If you are returned an error then please refer to the ['What errors should I care about'](#) table to help you determine your next set of actions.

PlugPlugIMSFetchContinueToken Code Sample

```
// Define a callback function
// The parameter "details" of callback function is a JSON string that
contains the detailed continue token information.
void FetchContinueTokenCallback(int status, const char* const details)
{
    // process the returned JSON string here.
}

// elsewhere in the code...
try
{
    PlugPlugIMSErrorCode result =
PlugPlugIMSFetchContinueToken("MyCallerID", myIMSRef, "MyAccessToken",
    "MyClientID", FetchContinueTokenCallback, "RedirectUri", "Scope",
    "ResponseType", "Locale");
    if (PlugPlugIMSErrorCode_SUCCESS != result)
    {
        // An error occurred when attempting to fetch an continue token.
    }
}
catch(PlugPlugException& e)
{
    // An error occurred if pass the invalid parameters.
}
```

[Corresponding IMSLib API](#)

PlugPlugShowAAM

Finally, if all the above workflows result in errors then more than likely you will be looking to launch AAM. This API will launch AAM and prompt the user to login and accept/confirm any user account requirements such as DOB entry, TOU acceptance or setting of additional profile settings. These requirements are all set through IMS when you [configure your Client ID](#).

Unlike the other APIs, this will call into AMTLib and NOT IMSLib.

PlugPlugShowAAM

```
/*!
 \brief Launches the Adobe Application Manager (AAM) AIR application.
 If an error occurs when fetching an access token, this allows the user to
 log in with their Adobe ID credentials, or accept new terms of use.
 This function is asynchronous, and will call inCallback function when
 PlugPlug receives the event.
 \param[in] inCallerID          Caller ID.
 \param[in] inClientID          The unique identifier of the client.
 \param[in] inClientSecret      The secret code of the client.
 \param[in] inRedirectUri       The redirect URL.
 \param[in] inCallback          A callback function which is called when it
 receives event.
 \param[in] inUserAccountGUID   The user account GUID against which access
 token needs to be generated.
 \param[in] inServiceAccountGUID The service account GUID of the user
 account. This will not be used currently.
 \param[in] inScope             The comma delimited scopes for which the
 refresh token was requested.
 \return a success or error code.
 */
PLUGPLUG_API(PlugPlugErrorCode) PlugPlugShowAAM(const char* inCallerID,
                                                const char*
inClientID,
                                                const char*
inClientSecret,
                                                const char*
inRedirectUri,
::csxs::event::AAMCallback inCallback,
                                                const char*
inUserAccountGUID = 0,
                                                const char*
inServiceAccountGUID = 0,
                                                const char* inScope =
"openid,AdobeID");
```

Please note that *inCallerID* is a required parameter and should be unique as it identifies the calling client so you can determine if the callback was triggered by your request e.g. ignore sign-in/out notifications when you did not make the original request to ShowAAM.

PlugPlugShowAAM Code Sample

```
// define the callback function.
void ShowAAMCallback(const ::csxs::event::AAMIMSStatusEvent* const event)
{
    // process the event here, the event contains authorization status and
    callerID information.
    if ("USER_AUTH_SUCCESS" != event->status)
    {
        // process authorization failure.
    }
}

try
{
    PlugPlugErrorCode result = PlugPlugShowAAM("MyCallerID", "MyClientID",
    "MyClientSecret", "RedirectUri", ShowAAMCallback);
    if (PlugPlugErrorCode_success != result)
    {
        // An error occurred when calling ShowAAM.
    }
}
catch(PlugPlugException& e)
{
    // An error occurred when calling ShowAAM or pass the invalid
    parameters.
    PlugPlugErrorCode errorCode = e.GetErrorCode();
}
```

AMTLib API: <https://wiki.corp.adobe.com/display/csxs/CSXS+and+AMTLib+communication>.

PlugPlugIMSDisconnect

When shutting down the client/completing task requirements, please call PlugPlugIMSDisconnect to end your IMS session.

PlugPlugIMSDisconnect

```
/*!
\brief Disconnects from Identity Management Service(IMS) and disposes the
IMS reference.
\param[in] inCallerID      Caller ID.
\param[in] inPlugPlugIMSRef An IMS reference returned from the
PlugPlugIMSConnect call.
\return a success or error code.
*/
PLUGPLUG_API(PlugPlugIMSErrorCode) PlugPlugIMSDisconnect(const char*
inCallerID, PlugPlugIMSRef inPlugPlugIMSRef);
```

PlugPlugIMSDisconnect Code Sample

```
try
{
    PlugPlugIMSErrorCode result = PlugPlugIMSDisconnect("MyCallerID",
myIMSRef);
    if (PlugPlugIMSErrorCode_SUCCESS == result)
    {
        // Disconnect successful.
    }
    else
    {
        // Disconnect failed.
    }
}
catch(PlugPlugException& e)
{
    // An error occurred if pass the invalid parameters.
}
```

[Corresponding IMSLib API](#)

PlugPlugIMSSetProxyCredentials

If the error returned by your call to `PlugPlugIMSFetchAccessToken` is `PlugPlugIMSErrorCode_ERROR_AUTH_PROXY_REQUIRED`, this means that the user is protected by an authenticated proxy server. In this case, you must prompt the user to enter their proxy credentials (which can differ from their Adobe ID credentials). You must provide UI to do this; you cannot do it through Adobe Application Manager.

Please note that IMSLib currently does not support double byte chars in the username and password. This requirement is tracked with Watson bug #3922360.

PlugPlugIMSFetchAccessToken

deprecated in CEP 5.1

PlugPlugIMSAttemptSSOJumpWorkflows

Since CEP 6.1

CEP have introduced a helper function to provide support in implementing Single Sign On (SSO) workflows for clients to their web services. This API has combined the features that were available in `AgoraLib` to provide a generic API that can be used by all clients.

PlugPlugIMSAttemptSSOJumpWorkflows

```
/*!
\brief A helper function to simplify supporting Single-Sign-On (SSO)
workflows in services with the Identity Management Service (IMS).
This function is asynchronous, and will call inCallback function when
PlugPlug completes the SSO workflows. Please note the inCallback function
is not guaranteed to be
called on the main thread.
There are three different levels of customization:
The most basic is to pass true for openBrowser and include a valid URL. CEP
```

will open the URL in the default browser without attempting any SSO workflows.

By supplying valid entries for all the other parameters but passing false for openBrowser, CEP will complete the SSO workflows but will return the generated URL to the

client so that they can complete the last step. The data returned to the client will be the same as the response currently returned in API

PlugPlugIMSFatchContinueToken.

The most comprehensive workflow is to include all parameters and pass true to OpenBrowser. CEP will complete the SSO workflows and open the generated URL in the default

browser so that the user is automatically signed into the web service.

inCallback will return a JSON string with the format

{"status":"0","details":{}} where the details could contain the results of: FetchAccessToken request if attempting this step in the process fails.

FetchContinueToken request if this step failed or you passed false to openBrowser

A status of PlugPlugIMSErrorCode_IMSMANAGER_SUCCESS_BROWSER_OPENED if you passed true for openBrowser

\param[in] inCallerID Caller ID

\param[in] inPlugPlugIMSRef An IMS reference returned from the PlugPlugIMSConnect call.

\param[in] inOpenBrowser Boolean value to determine if the SSO token should be opened in the browser by CEP.

\param[in] inURL The URL to jump to. It should be URL encoded.

\param[in] inClientID The unique identifier of the client as supplied by IMS

\param[in] inClientSecret The secret code of the client.

\param[in] inScope The comma delimited scopes for which the refresh token was requested, for example 'openid,AdobeID,browserlab'.

\param[in] inUserAccountGUID The user account GUID which is used in the SSO workflows.

\param[in] inTargetClientId The client id needed for the destination.

\param[in] inTargetScope A comma delimited list of services for which the refresh token should be authorized to use to ensure the user can successful sign-in

to the service.

\param[in] inTargetResponseType The IMS integration type the target has.

\param[in] inTargetLocale The locale in the format language_country, for example "en_US".

\param[in] inCallback A callback function which is called when the SSO workflows are complete

\return a success or error code.

\since 6.1

*/

PLUGPLUG_API(PlugPlugIMSErrorCode) PlugPlugIMSAttemptSSOJumpWorkflows(const char* inCallerID,

PlugPlugIMSRef inPlugPlugIMSRef,

bool

```
inOpenBrowser,  
  
const char* inURL,  
  
const char* inClientID,  
  
const char* inClientSecret,  
  
const char* inScope,  
  
const char* inUserAccountGUID,  
  
const char* inTargetClientId,  
  
const char* inTargetScope,  
  
const char* inTargetResponseType,
```

```
const char* inTargetLocale,  
  
::csxs::event::IMSCallbackWithStatus inCallback);
```

We have tried to incorporate all the workflows that a possible client would like to support:

If you would like to open a URL in the browser without SSO workflows then use:

Without SSO

```
// Define a callback function  
// The parameter "details" of callback function is an string  
void PlugPlugIMSAttemptSSOJumpWorkflowsCallback(int status, const char*  
const details)  
{  
    // check status is  
    PlugPlugIMSErrorCode_IMSMANAGER_SUCCESS_BROWSER_OPENED  
}  
  
// elsewhere in the code...  
  
try  
{  
    PlugPlugIMSErrorCode result =  
    PlugPlugIMSAttemptSSOJumpWorkflows("MyCallerID", NULL, true,  
    "http://www.adobe.com", "", "", "", "", "", "", "", "", "",  
    PlugPlugIMSAttemptSSOJumpWorkflowsCallback);  
    if (PlugPlugIMSErrorCode_SUCCESS == result)  
    {  
        // Attempting to complete SSO workflows has been requested.  
    }  
    else  
    {  
        // Failed to trigger SSO workflows  
    }  
}  
catch(PlugPlugException& e)  
{  
    // An error occurred if pass the invalid parameters.  
}
```

If you would like to support SSO workflows but would like to manage the launching of the URL then use:

Supporting SSO workflows without browser launch

```
// Define a callback function
// The parameter "details" of callback function is a JSON payload contain
the Jump URL
void PlugPlugIMSAttemptSSOJumpWorkflowsCallback(int status, const char*
const details)
{
    // check status is PlugPlugIMSErrorCode_SUCCESS and details is
populated
}

// elsewhere in the code...

try
{
    PlugPlugIMSErrorCode result =
PlugPlugIMSAttemptSSOJumpWorkflows("MyCallerID", MyIMSRef, false,
"http://www.adobe.com", "MyClientID", "MyClientSecret", "openid,AdobeID",
"ABC123@AdobeID", "TheJumpClientID", "openid,AdobeID,behance", "", "en_US",
PlugPlugIMSAttemptSSOJumpWorkflowsCallback);
    if (PlugPlugIMSErrorCode_SUCCESS == result)
    {
        // Attempting to complete SSO workflows has been requested.
    }
    else
    {
        // Failed to trigger SSO workflows
    }
}
catch(PlugPlugException& e)
{
    // An error occurred if pass the invalid parameters.
}
```

If you would like to support SSO workflows and have CEP open the URL in the default browser then use:

Support SSO workflows and getting CEP to launch browser

```
// Define a callback function
// The parameter "details" of callback function will contain the Jump URL
// as a JSON payload. this can be used as reference but not used unless the
// launching of the browser failed as the URL can only be used once.
void PlugPlugIMSAttemptSSOJumpWorkflowsCallback(int status, const char*
const details)
{
    // check status is
    PlugPlugIMSErrorCode_IMSMANAGER_SUCCESS_BROWSER_OPENED otherwise extract
    the URL from the details JSON payload.
}

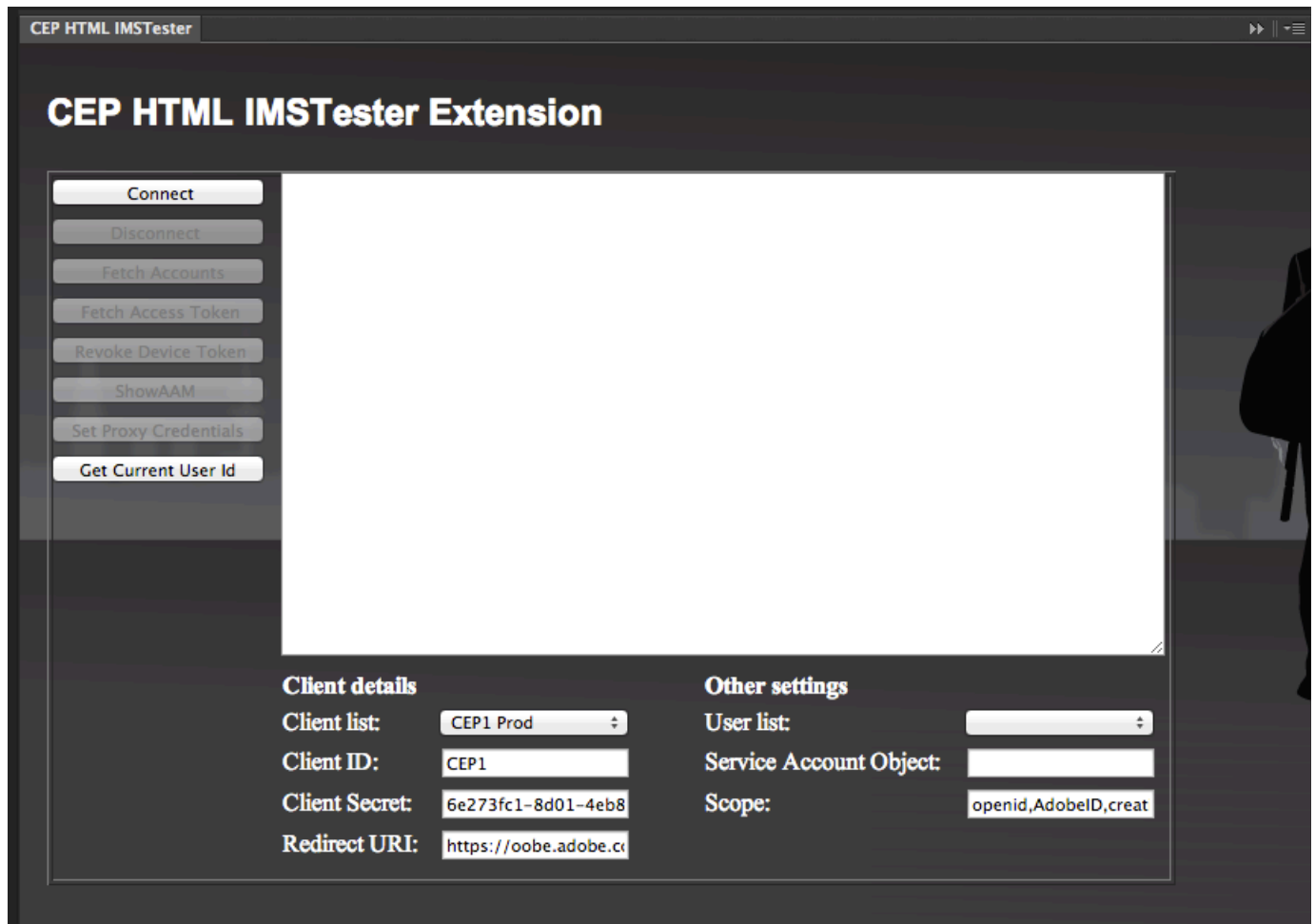
// elsewhere in the code...

try
{
    PlugPlugIMSErrorCode result =
    PlugPlugIMSAttemptSSOJumpWorkflows("MyCallerID", MyIMSRef, true,
    "http://www.adobe.com", "MyClientID", "MyClientSecret", "openid,AdobeID",
    "ABC123@AdobeID", "TheJumpClientID", "openid,AdobeID,behance", "", "en_US",
    PlugPlugIMSAttemptSSOJumpWorkflowsCallback);
    if (PlugPlugIMSErrorCode_SUCCESS == result)
    {
        // Attempting to complete SSO workflows has been requested.
    }
    else
    {
        // Failed to trigger SSO workflows
    }
}
catch(PlugPlugException& e)
{
    // An error occurred if pass the invalid parameters.
}
```

Trouble Shooting

If you are experience any issues with the IMS workflows either through the native APIs or through a HTML extension, please try CEP IMSTester extension. Add the IMS details that have been provided to you by IMS. If it does not work in the test extension then you know that the issue could be in CEP, IMSLib or IMS. If it does work in this extension then you know that it is an issue with your code.

- Codex Product: CEP
- Sub Product: CSXS Test Extensions
- HTML Extension: CEP_HTML_IMSTester_Extension.zxp



These might be useful either.

- PDApp.log
- Charles wiretraces
- IMS_TroubleShooting

Further Documentation

[CEP IMS Library](#)

[IMS Home](#)

Historic reference: [IMS native APIs](#)

Sending out notifications on panel fly-out menu open and close

There are chances where HTML extensions want to get notified when their fly-out menus are opened and closed. To support this, host applications need to invoke to PlugPlug APIs, which were introduced in CEP 6, when panel fly-out menu is opened and closed.

```
PLUGPLUG_API(PlugPlugErrorCode) PlugPlugFlyoutMenuOpened(const char*
inExtensionID);
PLUGPLUG_API(PlugPlugErrorCode) PlugPlugFlyoutMenuClosed(const char*
inExtensionID);
```

Drag and Drop

For the current status of Drag and Drop support, please see [The Status of Drag and Drop in CEP](#).

Real Point Product Integration

Referring to a real point product instead of CEP demo app is immensely beneficial. To refer to code of Owl point products which integrate PlugPlugOwl library, please get access to Photoshop or Illustrator.

Set and Get the window title of the extension

(Since 6.1)

CEP 6.1 introduces two APIs to set and get the title of the extension window. Those functions work with modal and modeless extensions in all Adobe products, and panel extensions in Photoshop, InDesign, InCopy, Illustrator and Animate (Flash Pro).

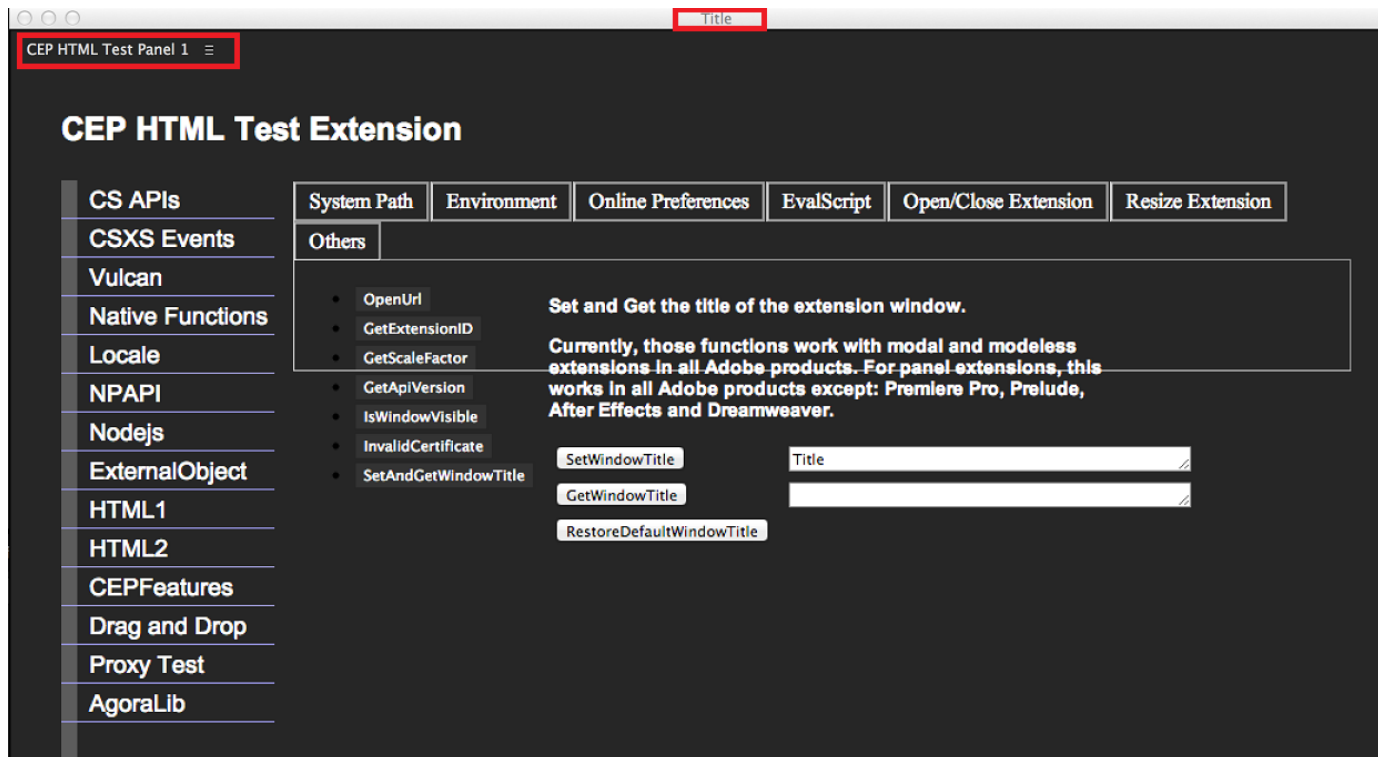
```
CSInterface.prototype.setWindowTitle = function(title)
{
    window.__adobe_cep__.invokeSync("setWindowTitle", title);
};

CSInterface.prototype.getWindowTitle = function()
{
    return window.__adobe_cep__.invokeSync("getWindowTitle", "");
};
```

CEP can handle setting and getting the panel extension window title in Photoshop, InDesign, InCopy, Illustrator and Animate (Flash Pro). Other host applications need implement those callbacks.

```
PlugPlugSetWindowTitleFn fnSetWindowTitle;
PlugPlugGetWindowTitleFn fnGetWindowTitle;
```

When you implement the callbacks, please set and get the title of the extension (left top corner of the below snapshot) rather than the title of the window itself (the top center of the below snapshot).



Remember Last Location and Size of Dialogs

CEP provides a new API to load extension with initial options so that host app can pass the geometry of dialog while loading.

```
PLUGPLUG_API(PlugPlugErrorCode) PlugPlugLoadExtensionEx(const char*
inExtensionID, PlugPlugExtensionInitOptions* options);
```

Host app should provide a new callback for CEP to call, so that CEP can pass the dialog geometry to host app. Host app is supposed to store it in preferences.

```
typedef PLUGPLUG_CALLBACK_API(PlugPlugErrorCode,
PlugPlugSaveWindowGeometryFn)(const char* extensionId,
PlugPlugWindowGeometry *geometry);
```

Customize Highlight Color

By default, CEP gets the highlight color from the operating system. CEP 7.0.0.67 supports host application to customize the highlight color.

- Host application needs to set the flag `customizeHighlightColor` to true and provide the highlight color in struct `PlugPlugSkinInfo` when initializing `PlugPlug`.

```

struct PlugPlugHostData
{
    .....
    /**
     * Flag indicating whether the host application wants to customize the
     highlight color for the extension.
     * \since PLUGPLUG_API_VERSION_NUMBER_15
     *
     * If it is true, panelHighlightColor in PlugPlugSkinInfo needs to be
     provided in API PlugPlugSetup and PlugPlugNotifyStateChange.
     * Otherwise, the highlight color is retrieved from OS.
     */
    bool customizeHighlightColor;
};

struct PlugPlugSkinInfo
{
    .....
    PlugPlugUIColor* panelHighlightColor; //!< highlight color for windows
    since PLUGPLUG_API_VERSION_NUMBER_15. It is controlled by the flag
    customizeHighlightColor in PlugPlugHostData.
};

```

- When the highlight color is changed, host application needs to use API PlugPlugNotifyStateChange to notify the color change.
- HTML extension needs to retrieve the latest color information when it receives the event "com.adobe.csxs.events.ThemeColorChanged".

Logging and Debugging

PlugPlug Logs

Log files with useful debug information are created for each of the applications supporting CEP extensions. The platform-specific locations for the log files are as follows:

- Win: C:\Users\USERNAME\AppData\Local\Temp
- Mac: /Users/USERNAME/Library/Logs/CSXS

These files are generated with the following naming conventions.

- CEP 4.0 - 6.0 releases: csxs<versionNumber>-<HostID>.log. For example, PlugPlug in Illustrator generates log file csxs6-ILST.log.
- CEP 6.1 and later releases: CEP<versionNumber>-<HostID>.log. For example, PlugPlug in Illustrator generates log file CEP6-ILST.log.

Logging levels can be modified as per the following levels:

- 0 - Off (No logs are generated)
- 1 - Error (the default logging value)
- 2 - Warn
- 3 - Info
- 4 - Debug
- 5 - Trace
- 6 - All

The **LogLevel** key can be updated at the following location (The application should be restarted for the log level changes to take effect):

- Win: regedit > HKEY_CURRENT_USER/Software/Adobe/CSXS.6
- Mac: /Users/USERNAME/Library/Preferences/com.adobe.CSXS.6.plist

For example of Mac, in the terminal do:

```
defaults write com.adobe.CSXS.6 LogLevel 6
```

CEPHtmlEngine Logs

In CEP 6.1 and later releases, CEPHtmlEngine generates logs. Each CEPHtmlEngine instance usually generate two log files, one for browser process, the other for renderer process.

These files are generated with the following naming conventions.

- Browser process: CEPHtmlEngine<versionNumber>-<HostID>-<HostVersion>-<ExtensionID>.log
- Renderer process: CEPHtmlEngine<versionNumber>-<HostID>-<HostVersion>-<ExtensionID>-renderer.log

For example,

- CEPHtmlEngine6-PHXS-16.0.0-com.adobe.DesignLibraries.angular.log
- CEPHtmlEngine6-PHXS-16.0.0-com.adobe.DesignLibraries.angular-renderer.log

They are also controlled by PlugPlug log level.

CEF Log

In CEP 4.0 - 6.0, the Chromium Embedded Framework (CEF) in CEPHtmlEngine generates log.

- Win: C:\Users\USERNAME\AppData\Local\Temp\cef_debug.log
- Mac: /Users/USERNAME/Library/Logs/CSXS/cef_debug.log

In CEP 6.1 and later releases, this log is merged into CEPHtmlEngine log.

FAQ, Useful Resources and Feedback

Test Automation

Adobe Illustrator team had successfully implemented CEP extension test automation by Selenium (<http://www.seleniumhq.org/>). Here is the link to recording of how they did automation of CEP based panel in Illustrator. This recording also has discussion on how other teams (InDesign, Dreamweaver) handled the same.

- Recording URL: <https://my.adobeconnect.com/p3xtd6mc0u6/>
- Please use the pass code to access it: cepdemo
- Contact: E Ramalingam <eramalin@adobe.com>

Logging Bugs

Please file bugs in JIRA by following [CEP Feature and Bug Reporting Workflow in JIRA](#).

Useful Resources

Questions? Talk to the CEP Integration mail list (<DL-CEP-Integrators@adobe.com>)

Integration Dashboard

To get to know the latest and compatible versions of the different components involved, refer the [Integration Dashboard](#)

Cleanup Script

You may use this script to clean up install records of any installations of CC. For more information on CC cleaner tool, refer the following Twiki page: <https://wiki.corp.adobe.com/display/Suites/CS+Installer+Clean+Up+Scripts>.

Past Editions of the Integration Cookbook

- [CEP 7 Integration Cookbook](#)
- [CEP 6 Integration Cookbook](#)
- [CEP 5.0 HTML Extension Cookbook](#)
- [CEP 4.0 Integration Cookbook](#)
- [CEP 3.0 Integration Cookbook](#)
- [CSXS 2.5 Integration Cookbook](#)
- [CSXS 2.0 Integration Cookbook](#)
- [CSXS 1.0 Integration Cookbook](#)

Feedback

And, finally don't hesitate to let us know if something should be in this document and isn't (or better yet, just change this wiki now. Add, fix or include the step for what you think is missing and we will fill in the details.)