ADOBE® CREATIVE CLOUD® 2014

USING THE ADOBE EXTENSION SDK

Adobe

Using the Adobe Extension SDK

Adobe Systems Inc., 345 Park Avenue, San Jose, California 95110, USA.

1 Getting Started with the Adobe Extension SDK

The Adobe® Extension SDK is a set of libraries that make it possible to build HTML/JavaScript extensions for Creative Cloud® 2014 applications. Developers can include these libraries in their projects in order to create cross-application plug-ins and Add-ons.

CC 2014 supports the new HTML/JavaScript framework, which allows you to access the ExtendScript DOM of the host application directly. The extension is delivered as a set of HTML, JavaScript, and CSS files that run in an embedded browser in the host application.

The earlier extension model, based on Flash®/Flex®/ActionScript® and AIR® 2.0 API, is deprecated in Creative Cloud. Support has already been removed in the CC 2014 release of Photoshop®, and will be removed from other applications in upcoming releases. It is strongly recommended that you use the new model, and port existing Flash-based extensions to the new model. Flash-based extension will continue to run as before in versions CS5.x and CS6 of their host applications.

This document describes how to use the HTML/JavaScript model; for details of the previous model, and information on how to migrate existing extensions to the new model, see the documentation for the previous release.

About Adobe Application Extensions

This section provides an overview of the Adobe application extensibility technology, which provides a common infrastructure for development and deployment of extensions that work across a set of supported Adobe desktop applications. An Adobe Application Extension is a set of files that together extend the capabilities of one or more Adobe desktop applications. Developers can use extensions to add services and to integrate new features across the applications in the suite.

The Adobe Extension SDK provides developers with a consistent platform in which to develop and deploy extensions across the suite. Adobe Application Extensions run in much the same way in different Adobe desktop applications, providing users with a rich and uniform experience.

Adobe Application Extensions use HTML and JavaScript to create cross-platform user interfaces. Extensions have access to the host application's scripting interface, and can use the host's scripting DOM to interact with the application. ExtendScript is Adobe's extended version of ECMA JavaScript. The host applications that have ExtendScript DOMs are packaged with the ExtendScript Toolkit, which allows you to develop and debug ExtendScript code.

Tight integration with the desktop applications allows extensions to be controlled as if they were built into the host applications. For example, extensions are invoked from the application's menu and, depending on the type of extension, can be docked, undocked, and provide fly-out menus. Users can add or remove extensions quickly and easily to customize Adobe desktop applications to their needs.

Developer prerequisites

This document assumes that a developer of HTML extensions is familiar with HTML, CSS, JavaScript, and the DOMs of the host applications where your extensions will be running. Knowledge of jQuery and node.js can also be useful. See suggested "Learning resource links" on page 7.

If you want to develop *hybrid* extensions (that is, extensions that combine an HTML/JavaScript panel with a native plug-in for the host application) you should have a good understanding of C/C++ as well as the SDKs of the host applications you will be targeting; for more information, see Chapter 6, "Creating a Hybrid Extension."

Adobe application extensibility architecture

The Adobe application extensibility architecture is designed to make it easy to develop and deploy extensions. This section describes the components and explains how they work together to run extensions.

Adobe desktop applications that enable extensibility (such as Photoshop and Illustrator®) link to the extensibility architecture through a native library. This library performs the standard tasks involved in listing, invoking, and communicating with services, and in requesting defined actions that are executed in the host.

The integrated applications are made aware of the extensions (services or extended features) available to them by the Adobe Extension Manager. This key component in the extensibility infrastructure runs on the client machine along with the products, and provides a common way to manage extensions across the suite.

Underlying technologies

Extension technology is built on the Common Extensibility Platform (CEP). The current release is 5.0.

> **NOTE :** *CEP was formerly named Creative Suite Extensible Services, or CSXS, so you will sometimes see "csxs" in names in the API and file structure.*

CEP 5.0 uses CEF3, a multi-process implementation that uses asynchronous messaging to communicate between the main application process and one or more render processes (WebKit + V8 JavaScript engine). It uses the official Chromium Content API, thus giving performance similar to Google Chrome.

CEP 5.0 supports persistent cookies stored in the user's file system:

- In Windows: `C:\Users\<user>\AppData\Local\Temp\cep_cookies`
- In Mac OS X: `/Users/<user>/Library/Logs/CSXS/cep_cookies`

The CEP HTML Engine does not restrict the use of extension JavaScript libraries. As long as a library can be used in CEF Client or Chrome browser, it should be usable in CEP HTML Engine.

For a description of features supported by CEP, see this blog:

http://blogs.adobe.com/cssdk/2014/04/introducing-cep-5.html

Anatomy of an HTML/JavaScript extension

HTML5 extensions are packaged as ZXP files, in the same way as Flash-based extensions. Extension Manager CC 2014 supports the installation and management of all ZXP extensions. See Chapter 4, "Packaging and Signing your Extension for Deployment." You can distribute your ZXP files privately or through Adobe Exchange.

A deployed Adobe Application Extension has these components:

File or Folder	Description
`MyExtension.html`	The page that defines your extension UI. It typically contains JavaScript code that provides behavior and allows it to communicate with the host application and with the extensibility infrastructure (CEP). See the Chapter 2, "" for basic information on creating an extension project.
`CSInterface.js` `Vulcan.js`	You must include these CEP JavaScript libraries in the script on your HTML page in order to to access application and CEP information. The Vulcan library implements the IPC Toolkit, which allows you to send and receive messages between applications. (These correspond to the former Flex CSXS, CEP IMS, and Vulcan libraries.) ▶ The CEP JavaScript HTML engine provides access to the local file system and to native processes; see Chapter 8, "CEP Engine JavaScript Extension Reference." These functions are part of the JavaScript DOM, and can be used like any other built-in methods and functions. You do not need to include any special libraries.
`CSXS/manifest.xml`	The manifest, a configuration file that lists the host applications that can load the extension and the supported locales, so that the correct resources can be used. See Chapter 3, "Creating a Manifest File."
`icon_*.png`	Optional icons used to represent the extension when docked. You can provide icons for different states (normal, rollover, or disabled). For targets that support color themes, you can provide icons for different themes (light or dark). Specify these as part of the configuration.
`locale/*.*`	Optional folder containing localized string resources. A default localization file, messages.properties, stores key-value pairs that map UI strings to resources. Each specific locale folder contains a messages.properties file for that locale.

Extension management

CEP is integrated with Adobe desktop applications and determines what extensions should be loaded in an application, based on the information provided in each extension's manifest file. To specify or change this information, you edit the project properties. If you make changes to an extension that was previously loaded, you must restart the host application in order to load the updated version of the extension.

Users can install your packaged and signed Adobe Application Extension through the Extension Manager; see Chapter 4, "Packaging and Signing your Extension for Deployment." The Extension Manager installs all extensions in a common location, the `extensions/` folder, that all the Adobe desktop applications can access.

- The name of the CEP root folder (`<CEP_root>`) depends on the version; for Creative Cloud 2013, it is `CEPServiceManager4`. For Creative Cloud 2014, it is `CEP`.
- The exact location of the folder is platform-specific:

- In Windows:
  (Win32) `C:\Program Files\Common Files\Adobe\<CEP_root>\extensions\`
  (Win64) `C:\Program Files (x86)\Common Files\Adobe\<CEP_root>\extensions\`
- In Mac OS X: `/Library/Application Support/Adobe/<CEP_root>/extensions/`

Within the `extensions/` folder, extensions are organized by the assigned name (that is, the bundle identifier, not the display name that appears in the host application's **Window > Extensions** menu). You can remove an extension through the Extension Manager's UI.

About the Adobe Extension SDK

Extension developers should be familiar with HTML, JavaScript, and CSS; and have at least basic knowledge about Adobe Product Extensibility.

The Adobe Extension SDK includes the Common Extensibility Platform (CEP) library, which provides a set of core services that you can use to send events to other extensions, execute ExtendScript code, and discover information about the host application environment.

To create HTML/JavaScript extensions for CC 2014, you must use CEP 5.

- The CEP JavaScript HTML engine provides access to the local file system and to native processes. There is no need to include this library, as it is integrated into CEP 5. For complete details of the access functions, see Chapter 8, "CEP Engine JavaScript Extension Reference."
- Include these JavaScript libraries in your extension project:

CSInterface.js	Implements the control interface for extensions
Vulcan.js	Implements the IPC Toolkit for interapplication messaging.

Development environment requirements

A specialized development environment is not yet available for the Adobe Extension SDK. The SDK provides CEP libraries, documentation, and a command-line tool for packaging. You can use your preferred text editor to create the required configuration files, using the information provided in this document.

To create and run extensions that you create with the Adobe Extension SDK, you must have installed:

- Adobe Extension Manager CC 2014
- At least one of the Adobe Creative Cloud 2014 desktop applications that supports HTML/JavaScript extensions.
- Adobe ExtendScript Toolkit (installed with host applications that have an ExtendScript DOM)

Supported applications

The following Adobe desktop applications support CEP-based extensions and IPC interapplication messaging. The Creative Suite releases of these products support only Flash-based extensions. The

Creative Cloud releases support HTML5/JavaScript extensions; the Flash/ActionScript model is deprecated, and support is being removed.

Application	Host name
InCopy®	`AICY`
InDesign®	`IDSN`
Illustrator	`ILST`
Photoshop / Photoshop Extended (HTML extensions only in CC 2014 release)	`PHXS`
Prelude®	`PRLD`
Premiere® Pro	`PPRO`
Dreamweaver® (Flash extensions only)	`DRWV`
Flash® Pro (HTML extensions only)	`FLPR`
AfterEffects® (Not integrated with Extension Manager)	`AEFT`

Learning resource links

HTML5/ CSS3

w3.org	HTML/CSS Specifications
W3Schools.com	HTML5 Tutorials CSS Tutorials CSS 3 Tutorials
Lnda.com	HTML-Essential-Training HTML5 Structure Syntax and Semantics
Codecademy.com	Building blocks of web development with HTML and CSS
HTML5 Rocks Resources	

JavaScript

JavaScript Language Specification	
W3Schools.com	JavaScript Tutorials
Mozilla Network Developer (MND)	A Re-Introduction to JavaScript
Douglas Crockford On JavaScript	
Lynda.com	Introducing the JavaScript Language JavaScript-Essential-Training JavaScript and JSON JavaScript Templating
Codecademy.com	JavaScript Training

Query

jQuery.com	API Documentation
jQAPI - Alternative jQuery Documentation Browser	
learn.jquery.com	How jQuery Works
W3Schools.com	jQuery Tutorials
Learningjquery.com	Great Ways to Learn jQuery
Lynda.com	jQuery Essential Training

node.js

nodejs.org	API Manual & Documentation
Lynda.com	Node.js Essential Training
LeanPub.com	JavaScript and Node FUNdamentals
eBooks	Mastering node.js Mixu's Node book NodeBeginner.org Node: Up and Running

ExtendScript and Adobe SDKs

Adobe Scripting Center	Adobe InDesign Adobe Illustrator Adobe Photoshop Premiere Pro
Adobe Host Applications SDKs	Adobe InDesign Adobe Illustrator Adobe Photoshop Premiere Pro

C/C++

cplusplus.com	
Learncpp.com	
Lynda.com	C/C++ Essential Training
Books	C++ Primer Programming: Principles and Practice Using C++ Effective C++ Effective STL More Effective C++ Modern C++ Design

2 Running and Debugging your Extension

Once you have created the project you can run the extension within your chosen host application. Because it is still in development, your extension is unsigned, but if you set up the debug environment correctly, you can load and run it in the host application, and debug it in the brower using Remote Debugging.

When the environment is set up, you can deploy your extension to the common location so that the host application can automatically load it on launch. You can then invoke your extension and debug it in the browser.

Setting up the debug environment

Before you run for the first time, you must let the host application know that you are still in development, so that it won't expect your extension to be signed. You do this by setting the debug mode flag in a platform-specific configuration file, as described here.

CEP5 supports remote debugging of HTML extensions in browsers that support the Remote Debugging protocol, such as the latest versions of Google Chrome, Safari, Mozilla Firefox, and Opera. You must ensure that the host application and your browser are set up to use the same TCP port for debugging communication.

Setting the debug mode flag

To run an unsigned extension in the target host application (which you typically need to do for debugging) you must set an OS-specific flag. The location of this flag depends on which version of the host application you are targeting for your extension.

Editing the flag in Windows:

1. Choose **Run** from the Windows Start menu, and enter `regedit` to open the registry editor.
2. Navigate to the key
   `HKEY_CURRENT_USER\Software\Adobe\CSXS.5`
3. Choose **Edit > New > String Value**. Enter the **Name** key `PlayerDebugMode` , and set **Data** to 1 to enable debug mode.
4. Close the registry editor.

Editing the flag in Mac OS:

1. Navigate to the folder `<user>/Library/Preferences`
2. Find the property list (PLIST) file:
   `com.adobe.CSXS.5.plist`
3. Open this file with the XCode Property List editor, or the PlistBuddy command-line tool.

4. Change value for the key `PlayerDebugMode` to 1 to enable debug mode as described below, and save the file.

NOTES:

- If this file is read-only, you must add write permission for the user before you can update it. To do this, right click on the file and select **Get Info > Sharing & Permissions**.
- In OS X 10.9 Apple introduced a caching mechanism for property list files. This means that property modifications do not take effect immediately. To force you modification to take effect, open the Terminal application and enter this command:

```
sudo killall cfprefsd
```

Editing in XCode

Open Xcode (Normally found in the Applications folder) and choose **File > Open.** Locate the property list and click **Open**. If the `PlayerDebugMode` key already exists make sure its value is set to 1. If not:

- Hover over any entry and click Add (**+**) .
- Enter "PlayeDebugMode" as the key name.
- Set the key type to "String".
- Enter "1" as the value.

When the key has the correct value, save the file.

Editing in PlistBuddy

Open the Terminal application (normally stored in the `/Applications/Utilities` folder), and enter this command to print the content of the property list file and check if the `PlayerDebugMode` key already exists:

```
/usr/libexec/PlistBuddy -c "print" ~/Library/Preferences/com.adobe.CSXS.5.plist
```

The output should look like this:

```
Dict {
    LogLevel = 1
}
```

- If the key already exists, use this command to set the flag:

```
/usr/libexec/PlistBuddy -c "set PlayerDebugMode 1"
    ~/Library/Preferences/com.adobe.CSXS.5.plist
```

- If not, use this command to create and set the flag:

```
/usr/libexec/PlistBuddy -c "add PlayerDebugMode String 1"
    ~/Library/Preferences/com.adobe.CSXS.5.plist
```

- To confirm that the entry has been added successfully, print the content of the property list file again. It should now have the new key:

```
Dict {
    PlayerDebugMode = 1
    LogLevel = 1
}
```

Setting up remote debugging

Debugging communication between the host application and the browser's development tools goes through a TCP port. You define which port to use in your extension's `.debug` configuration file. You must create this XML file in your extension project's root folder, and define the port number to use for debugging communication for each Adobe host application that your extension supports.

First, create an empty file name `.debug` in the extension's root folder according to your platform, then add the TCP port mapping.

Creating an empty file

- In Mac OS X

  Open the Terminal application, change the current working folder to the root folder of your extension, and use this command to create an empty file named ".debug":

  ```
  touch .debug
  ```

  Files whose names start with a dot are hidden. If you want to make this file visible in order to edit it, use this command, then restart Finder:

  ```
  defaults write com.apple.finder AppleShowAllFiles YES
  sudo killall Finder
  ```

  (To hide files again, use the same command with `NO` instead of `YES`.)

- In Windows

  In a command shell, change the current working folder to the root folder of your extension, and use this command to create an empty file named ".debug":

  ```
  copy con .debug
  ```

  Press CTRL Z to complete the file creation.

Set up port mapping

Open the `.debug` file with your preferred text editor, and add this XML code, with a `<Host>` element for each supported host application:

```
<?xml version="1.0" encoding="UTF-8"?>
<ExtensionList>
    <Extension Id="extensionID">
        <HostList>
            <Host Name="hostID" Port="portNum"/>
        </HostList>
    </Extension>
```

```
</ExtensionList>
```

`Extension > Id`	The unique ID of your extension, as set in the manifest. For example, `"com.myDns.myProductName.Panel1"`.
`Host > Name`	The host application ID. For example, the ID for Photoshop is `"PHSP"`. These host IDs are defined: After Effects `AEFT` Dreamweaver `DRWV` Flash Pro `FLPR` Illustrator `ILST` InCopy `AICY` InDesign `IDSN` Photoshop `PHSP` Photoshop Ext `PHXS` Prelude `PRLD` Premiere Pro `PPRO`
`Host > Port`	The TCP port number to use for debugging communication with the browser. For example, `"8000"`.

For example, for an extension that supports Photoshop Extended and InDesign, the XML would look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<ExtensionList>
   <Extension Id="com.myDns.myProductName.Panel1">
      <HostList>
         <Host Name="PHXS" Port="8000"/>
         <Host Name="IDSN" Port="8001"/>
      </HostList>
   </Extension>
</ExtensionList>
```

For a bundle of extensions, add an `<Extension>` element for each extension in the bundle.

Deploying an unsigned extension

Once the debugging environment and configuration is set up, you can run the extension in its host application and debug it in the browser.

To run and debug your extension in its target application, you must deploy it by copying the ZXP package to one of the deployment folders that CEP looks in to find available extensions.

- These are the system-wide deployment folders for all users:
  - In Windows:
    `C:\Program Files\Common Files\Adobe\CEP\extensions\`
  - In Mac OS:
    `/Library/Application Support/Adobe/CEP/extensions/`

- For a specific user, these are the default locations of the deployment folder:
  - In Windows:
    `C:\<username>\AppData\Roaming\Adobe\CEP\extensions\`
  - In Mac OS:
    `~/Library/Application Support/Adobe/CEP/extensions/`

On launch, an application searches for extensions first in the system folder, then the user's folder. If there is a conflict in extension IDs, the last one loaded is used. If the same extension is found in different locations, then if they have different bundle-ID versions, the latest version is used. If two extensions have the same ID and version, the one in the system folder is used.

Debugging your extension

You can only debug an unsigned extension that is running it the host application if you have correctly set up the debugging mode, as described in "Setting up the debug environment" on page 10.

To start a debugging session for a running host application that has loaded your extension:

- Open a browser.
- Go to `http://localhost:<portNum>` , using the port number you have assigned for the host application in the `.debug` file. For example, `http://localhost:8000`.
- Invoke your HTML extension.
  - When you start the host application, typically your extension's menu (as defined in the manifest file) appears in the **Window > Extensions** menu (or in the menu defined in the manifest for InDesign/InCopy). You invoke the extension's HTML UI in the browser by choosing the menu item.
  - Some host application support *invisible* extensions, which run in the background and have no UI. Such extensions are loaded on specific events, defined in the extension manifest's `<startOn>` element. To debug invisible extensions you must first invoke the event that triggers the extension load. The JavaScript code runs in the browser.
- Start debugging the extension using the browser's development tools. For information about using these tools, see your browser documentation. For example, http://developers.google.com/chrome-developer-tools/.

ExtendScript debugging and logging

During development, you can debug and test your ExtendScript code using ExtendScript Toolkit. Once it's intergrated into an HTML extension, however, the code is not available to the ExtendScript Toolkit. You can, however, build log messages into your ExtendScript code for when you are in debug mode, which allow you to monitor the ExtendScript behavior within an extension.

Your debug messages can be sent back to your extension's JavaScript component using the event notification system, or you can send messages directly to the browser development tool's console

Example: Event-based debugging

This method uses CEP event-based communication to send debug information from ExtendScript code to the HTML extension's JavaScript DOM. It makes use of an ExtendScript external shared library,

`PlugPlugExternalObject`. See more information about event handling and the library in Chapter 5, "Event Handling for Extensions," and Chapter 6, "Creating a Hybrid Extension."

Currently, the CC 2014 releases of Photoshop, Illustrator and Premiere Pro support `PlugPlugExternalObject`; support is planned for other CC applications.

The following example illustrates how to use ExtendScript for this debugging technique.

ExtendScript component

Add a JSX file containing similar code to your extension manifest:

```
<ScriptPath>./jsx/[scriptName].jsx</ScriptPath>
```

This script creates an instance of `PlugPlugExternalObject`, and uses its `getFileContents()` method to read and return the contents of a file. First, however, we define a function, `devToolsConsoleOut()`, to create and send a custom CSXS event. At each stage of the operation, we use this function to send information about possible problems to the HTML extension's JavaScript DOM.

```
DEBUG = 1;
try {
    var xLib = new ExternalObject("lib:\PlugPlugExternalObject");
}
catch(e) {
    alert(e);
}

function devToolsConsoleOut(in_message)
{
    if (DEBUG == 1) {
        var eventObj = new CSXSEvent();
        eventObj.type = "DevToolsConsoleEvent";
        eventObj.data = in_message;
        eventObj.dispatch();
    }
}

function getFileContents(in_path) {
    try {
        do {
            devToolsConsoleOut("Info: Entering getFileContents");
            var retVal = null;
            if (! in_path) {
                devToolsConsoleOut("Error: Argument is null");
                break;
            }
            if (typeof(in_path) != "string") {
                devToolsConsoleOut("Error: Argument is not the correct type");
                break;
            }
            if (in_path.length == 0) {
                devToolsConsoleOut("Error: Argument is an empty string");
                break;
            }
            var file = File(in_path)
            if (! file.exists) {
                devToolsConsoleOut("Error: Could not find file - " + file.fullName);
                break;
            }
            if (! file.open()) {
```

```
            devToolsConsoleOut("Error: Could not open file - " + file.fullName);
            break;
        }
        retVal = file.read();
        if (! file.close())
        {
            devToolsConsoleOut("Warning: Could not close file - " + file.fullName);
        }
        devToolsConsoleOut("Info: File read OK");
    }
    while (false);
}
catch(e) {
    devToolsConsoleOut(e);
}
devToolsConsoleOut("Info: Leaving getFileContents");
return retVal;
}
```

JavaScript component

In the JavaScript for your HTML extension, include code like this to call the ExtendScript function and receive the debugging messages that it sends from the ExtendScript engine.

This code registers a handler for the custom event type `DevToolsConsoleEvent`. It then defines a button click function that calls the ExtendScript file-read operation, using `csInterface.evalScript()`.

```
(function () {
    var csInterface = new CSInterface();

    function init() {

        themeManager.init();
        csInterface.addEventListener("DevToolsConsoleEvent", function(event){
            console.log(event.data);
        });

     $("#btn_readFile").click(function () {
            csInterface.evalScript("getFileContents(Folder.desktop+'/test.txt')",
                                   contentsCallBack);
     });
    }

    function contentsCallBack(in_contents) {
    // do something here with the contents of the file
    }
    init();

}());
```

Example: Console-based debugging

This approach works with all Creative Cloud applications that support ExtendScript APIs.

In this example, the ExtendScript code defines a callback function, `contentsCallBack`, and passes it in the call to `getFileContents()`. The callback logs information directly in the development-tool console.

ExtendScript component

```
DEBUG = 1;

function updateLog(in_log,in_msgType,in_msg)
{
    var retVal = in_log
    if (DEBUG == 1)
    {
        retVal[in_log.__count__+1] = {type: in_msgType.toLowerCase(), msg:in_msg}
    }
    return retVal;
}

function getFileContents(in_path) {
    try {
        do {
            var retVal = {fileContents: null,log: {}};
            updateLog(retVal.log,"Info","Entering getFileContents");
            if (! in_path) {
                updateLog(retVal.log,"Error","Argument is null");
                break;
            }
            if (typeof(in_path) != "string") {
                updateLog(log,"Error","Argument is not the correct type");
                break;
            }
            if (in_path.length == 0) {
                updateLog(retVal.log,"Error","Argument is an empty string");
                break;
            }
            var file = File(in_path)
            if (! file.exists) {
                updateLog(retVal.log,"Error","Could not find file: " + file.fullName);
                break;
            }
            if (! file.open()) {
                updateLog(retVal.log,"Error","Could not open file: " + file.fullName);
                break;
            }
            retVal.fileContents = escape(file.read());
            if (! file.close())
            {
                updateLog(retVal.log,"Warning","Could not close file: " + file.fullName);
            }
            updateLog(retVal.log,"Info","Read successfully file: " + file.fsName);
        }
        while (false);
    }
    catch(e) {
        updateLog(retVal.log,"Error","Line "+e.line+": "+e);
    }
    updateLog(retVal.log,"Info","Leaving getFileContents");
    return retVal.toSource()
}
```

JavaScript component

Below JavaScript code that could be used to control the UI of an HTML extension.

Pay special attention to:

1 - $("#btn_test").click method respondes to a "click" event in the UI (button) and triggers the ExtendScript function getFileContents

2 - contentsCallBack callback function that handles the return value of getFileContents and outputs

```
(function () {
    var csInterface = new CSInterface()

    function init() {

        themeManager.init();

     $("#btn_test").click(function () {

csInterface.evalScript("getFileContents(Folder.desktop+'/test.txt')",contentsCallBack
);
     });
    }

    function contentsCallBack(in_resultStr) {
        eval("resultObj="+in_resultStr);
        for (var propName in resultObj.log) {
            var logEntry = resultObj.log[propName]
            switch(logEntry.type.toLowerCase())
            {
                case "error":
                    console.error(logEntry.msg);
                    break;
                case "warning":
                    console.warning(logEntry.msg);
                    break;
                case "info":
                    console.info(logEntry.msg);
                    break;
                default:
                    console.log(logEntry.msg);
            }
        }
        console.log("File Contents:\n"+unescape(resultObj.fileContents));
    }
    init();
}());
```

System logs

Both CEP and the underlying Chromium Embedded Framework (CEF) keep logs that can help in debugging your Adobe extensions. The location of the log files is platform-specific:

- In Windows: `C: \Users\`*`USERNAME`*`\AppData\Local\Temp`
- In Mac OS X: `~/Library/Logs/CSXS`

The CEF engine log is written to the platform-specific location, in the file `cef_debug.log`. (You can create an alias for this file to access it more easily.) This log provides valuable information when an extension has

a problem at load time, before CEP and embedded logging begins.

CEP logging

Once an extension has been successfully loaded into a host applications, a log files with useful debug information is created. There is a separate log file for each of the applications that support CEP extensions. The log file names are based on the application name:

```
csxs<version>-<hostID>.log
```

For example, the InDesign the log file is named `csxs5-IDSN.log`.

You can set a log level in the CEP configuration that controls how much information is written to the log. To set the log level, change the `LogLevel` key in the platform-specific configuration file:

- In Windows: `regedit > HKEY_CURRENT_USER/Software/Adobe/CSXS.5`
- In Mac OS X: `~/Library/Preferences/com.adobe.CSXS.5.plist`

(See "Setting the debug mode flag" on page 10 for details of how to edit configuration keys.)

These log levels are defined:

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

3 Creating a Manifest File

Extensions created with the Adobe Extension SDK require a manifest file. The manifest is an XML file that describes the extension, tells the Extension Manager how to install it, and gives the author control over extension-specific options such as the extension life cycle, UI, and menus.

When you use the New Project Wizard in Extension Builder 3, the manifest is created for you. You can also create your own, or edit an existing manifest using the editor in Extension Builder 3 or any XML editor.

You must use version 4.0 or higher for HTML/JavaScript extensions. The complete schema for the XML, which you can use to validate the syntax, is included in the Adobe Extension SDK installation:

```
<Ext_SDK_root>/docs/ExtensionManifest-4.0.xsd
```

Adobe Extension SDK supports bundling multiple extensions into a single extension bundle, and the manifest schema reflects this structure.

In this section we look at a simple manifest file in detail, illustrating the usage of each XML tag with examples from the sample `manifest.xml` file included in the Adobe Extension SDK.

ExtensionManifest

The root element for an extension manifest XML file:

```
<ExtensionManifest
    Version="4.0"
    ExtensionBundleId="com.example.simple"
    ExtensionBundleVersion="1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    ...
</ExtensionManifest>
```

Attributes

The `ExtensionBundleId` attribute is an optional unique identifier for your extension bundle. Adobe recommends using a fully qualified namespace-like name such as `com.myCompany.extension`.

Allowed children

The possible child elements of `ExtensionManifest` are:

`Author`	Optional. The author of this extension bundle.
`Contact`	Optional. A contact for this extension bundle. ▶ Required attribute `mailto`.
`Legal`	Optional. A legal notice for this extension bundle. ▶ Optional attribute `href`.
`Abstract`	Optional. An abstract for this extension bundle. ▶ Optional attribute `href`.
`ExtensionList`	Contains a list of extensions defined in this bundle. See details below.

`ExecutionEnvironment`	Contains information about which host applications can run the extension under what conditions. See details below.
`DispatchInfoList`	Contains an `Extension` element for each of the listed extensions, each of which contains a `DispatchInfo` element. See details below.
`ExtensionData`	Optional. Contains arbitrary information about this extension. It can contain data of any type. ▶ Required attribute `Id` associates this data with an extension defined in the `ExtensionList`. ▶ Optional attribute `Host` associates the data with a specific host application. If you have provided localization resources (see Chapter 7, ""), you can use the `%key` syntax to localize values in the `ExtensionData` element. Because this section contains arbitrary information about the extension, you must localize the entire XML content of the element, and include all of the alternative XML files in your project: `<ExtensionData>%ExtensionData</ExtensionData>`

ExtensionList/Extension

An extension bundle can contain multiple extensions, each of which is implemented by a main HTML file. Each extension in the bundle must be listed here in its own `Extension` element, each with a unique extension identifier.

```
<ExtensionList>
    <Extension Id="com.example.simple.extension" Version="1.0" />
</ExtensionList>
```

Attributes The `Extension` tag takes two attributes:

`Id`	A unique identifier for the extension, unique within the entire CEP system. Adobe recommends using a reverse domain name. Other tags within the manifest use this id to reference this extension.
`Version`	Optional, a version identifier for this extension.

ExecutionEnvironment

The `ExecutionEnvironment` element contains information about which Adobe desktop applications will run the extension under what conditions.

This element must list each of the Adobe host applications targeted by your extension, the supported locales, and the runtime requirements. In this example, an extension that targets InDesign CC requires CSXS 4.2

```
<ExecutionEnvironment>

    <HostList>
       <Host Name="IDSN" Version="11" />
    </HostList>
```

```
        <LocaleList>
            <Locale Code="All" />
        </LocaleList>

        <RequiredRuntimeList>
            <RequiredRuntime Name="CSXS" Version="4.2" />
        </RequiredRuntimeList>

    </ExecutionEnvironment>
```

HostList/Host

The `HostList` element contains a list of `Host` elements for all supported hosts. Each `Host` tag specifies a supported Adobe desktop application.

Attributes The `Host` tag contains the following attributes:

`Name`	Required, the host name of the host application. See "Supported applications" on page 6.
`Version`	Required. The version or versions in which this extension will work. A single version number specifies the minimum supported version; the extension works in all versions greater than or equal to this version. Specify a version range using interval notation, a comma-separated minimum and maximum version number enclosed by inclusive, `[ ]`, or exclusive, `( )`, endpoint indicators. You can mix endpoint types. For example, to target InDesign 7 and all versions up but excluding version 10, use the string `"[7,10)"`. The entire element looks like this: `<Host Name="IDSN" Version="[7,10)" />`

LocaleList/Locale

CEP checks the License Locale of the host application against supported locales declared in an Extension's locale list to determine if the extension is loadable for the host application.

The `LocaleList` element contains a list of `Locale` elements for all supported locales. Each `Locale` tag contains the locale code for a supported language/locale, in the form *`xx_XX`*; for example, `en_US` or `ja_JP`. You can use the special value All to indicate that the extension supports all locales.

Use a single `Locale` element with the special value "All" to make your extension load in the host application regardless the language used:

```
<LocaleList>
    <Locale Code="All"/>
</LocaleList>
```

To restrict the locales your extension supports, create a `Locale` element for each language, whose value is a locale code. If the application locale does not match one of those specified, the application does not load the extension. For example, an extension with these settings loads when the application is running in US or British English:

```
<LocaleList>
    <Locale Code="en_US" />
    <Locale Code="en_GB" />
</LocaleList>
```

For information on how to localize your extension, see Chapter 7, "".

LOCALE SUPPORT NOTE: MENA ("Middle East and North Africa") support allows localization for Arabic, Hebrew, and North African French languages in the CS6 and CC versions of InDesign, Photoshop, Illustrator, DreamWeaver, and Acrobat. An appropriate language (such as standard French for North African French) is automatically substituted if the MENA language is not available and the substitute is.

RequiredRuntimeList/RequiredRuntime

The `RequiredRuntimes` element contains a list of `RequiredRuntime` elements for all required runtimes; that is, executables that must be available in order for the extension to run.

- For extensions that target CS6 or CC applications, use CSXS 4.0:

```
<RequiredRuntimeList>
   <RequiredRuntime Name="CSXS" Version="4.0" />
</RequiredRuntimeList>
```

- For extensions that target CC 2014 applications, use CSXS 5.0:

```
<RequiredRuntimeList>
   <RequiredRuntime Name="CSXS" Version="5.0" />
</RequiredRuntimeList>
```

DispatchInfoList/Extension/DispatchInfo

This section of the manifest determines the lifecycle and appearance of your extension. Each extension listed in the `ExtensionList` element must have a corresponding `Extension` element in the `DispatchInfoList`, containing a `DispatchInfo` element. The `Id` attribute in this `Extension` tag associates it with its corresponding tag in the `ExtensionList`.

```
<DispatchInfoList>
   <Extension Id="com.example.simple.extension">
      <DispatchInfo >
         ...
      </DispatchInfo>
   </Extension>
</DispatchInfoList>
```

The `DispatchInfo` element contains parameters that the application needs to run the extension. This includes information about the resources used by the extension, the lifecycle, and the UI configuration.

```
<DispatchInfo >
   <Resources>
      <MainPath>./Simple.html</MainPath>
   </Resources>

   <Lifecycle>
      <AutoVisible>true</AutoVisible>
      <StartOn>
         <Event>applicationActivate</Event>
      </StartOn>
   </Lifecycle>
```

```
            <Geometry>
                <Size>
                    <Height>500</Height>
                    <Width>400</Width>
                </Size>

                <MaxSize>
                    <Height>500</Height>
                    <Width>400</Width>
                </MaxSize>

                <MinSize>
                    <Height>500</Height>
                    <Width>400</Width>
                </MinSize>
            </Geometry>
        </UI>
    </DispatchInfo>
```

Attributes The `DispatchInfo` tag can have an optional attribute `Host`, in which case the parameters apply only to that host application. Specify the application using the Host name shown in "Supported applications" on page 6.

If a host is not specified, the element defines default values for all parameters that are not set in a host-specific `DispatchInfo` element.

Resources

The `Resources` element contains the paths to source files that are needed to run the extension. All paths are relative to the extension's root directory, and must use forward-slash delimiters. Typically contains these elements:

`MainPath`	Contains the path to the extension's home-page HTML file.
`ScriptPath`	Contains the path to the extension's script file, if any.

Lifecycle

The `Lifecycle` element specifies the behavior at startup and shutdown. It can contain these elements:

`AutoVisible`	Boolean, true to make the extension's UI visible automatically when launched.

`StartOn/Event`	A set of events that can start this extension. Use fully-qualified event identifiers; for example:
	
	`<Lifecycle>`
	`  <StartOn>`
	`  <Event>applicationActivate</Event>`
	`  </StartOn>`
	`</Lifecycle>`
	
	You can register for any of the CEP/CSXS standard events or any arbitrary `CSXSEvent` sent from a C++ plug-in. The standard events (which are not necessarily supported by all applications) are:
	
	▶ `documentAfterActivate`: When a document has been activated.
	
	▶ `documentAfterDeactivate`: When the active document has been deactivated.
	
	▶ `applicationActivate`: When the application gets an activation event from the OS.
	
	▶ `applicationBeforeQuit`: When the application is about to shut down.
	
	▶ `documentAfterSave`: After the document has been saved

UI

The `UI` element configures the appearance of the extension window. It can contain these elements:

`Type`	The type of the extension controls the kind of window that displays its UI. Value is one of:
	
	`Panel`
	`ModalDialog`
	`Modeless`

`Menu`	The label of the menu item for this extension in the host application's **Window > Extensions** menu. The value can be a localization key; see Chapter 7, "." If not included, no menu item is added for the extension, and you are responsible for starting it in response to some event, by providing a `Lifecycle/StartOn/Event` element.

`Geometry`	Specifies the preferred geometry of the extension window. The host application may not support all of these preferences, and the values can be overwritten for an AIR extension, using the AIR window API. The value can be a localization key; see "Localizing the extension's manifest file" on page 58. The example above shows the possible elements. If you provide a size element, both the width and height value must be provided.

`Icons/Icon`	The `Geometry` element can contain this list, which identifies icons used for the extension in the host application's UI; for example, when docking an extension of type `Panel`.
	
	Each `Icon` element contains the path to the icon file (relative to the extension's root directory), and the required attribute `Type`, which is one of:
	
	`Normal`
	`Disabled`
	`Rollover`
	
	The path value can be a localization key; see "Localizing the extension's manifest file" on page 58.

4 Packaging and Signing your Extension for Deployment

The Extension Manager package file which allows you to install the extension you are developing on machines other than the one you are currently using (across platforms), to share the extension with other users, and to distribute it to customers.

Extension Manager requires that the package be signed and timestamped. See "How signing works" on page 29.

The package format

An Extension Manager package uses the ZXP format. This is an archive file with the extension `.zxp`, which contains:

- A copy of the `CSXS` folder containing the `manifest.xml` file.
- A copy of the folder containing the extension-panel HTML file and any dependant files.
- A copy of any other optional resources used by the extension, such as icons and localization files. For a hybrid extension, it must include the resource files for the native plug-in or scripting component.
- A file named `mimetype`, generated by the packaging and signing process.

Creating the deployment package

Adobe provides a number of packaging tools that help you to configure and create the ZXP package for your HTML/JS extension.

- Extension Builder 3 and Configurator 4 include easy-to-use Packaging Wizards for packaging extensions that you create with those tools.
- If you have a producer account with Adobe Exchange, the Adobe Exchange Packager is available from http://www.adobeexchange.com/resources.
- For all types of extensions and plug-ins, you can use the lower-level `ZXPSignCmd`, a command-line tool available from Adobe Labs.

Using the CC Extension Signing Toolkit

Adobe provides a command-line tool, `ZXPSignCmd`, that you can use to package and sign extensions so they can be installed in Adobe desktop applications using Extension Manager. See the Adobe Extension SDK page to download the toolkit for your platform.

After testing your extension thoroughly, you must package and sign your extension so users can install it in their systems using Extension Manager. To prepare for this step, it is recommended that you copy all of the files in the Output folder for your extension to a staging folder for ease of packaging. Make sure the staging folder contains a subfolder named `CSXS/`, which contains the `manifest.xml` file:

```
<staging_folder>/CSXS/manifest.xml
```

You can add any extra resources to the root or to a folder within the root folder. Within the manifest file, references to these resources should use pathnames that are relative to the root. For example, if your main panel HTML file is located at `<staging folder>/Simple.html`, the path in the manifest should be specified as `./Simple.html`.

For a hybrid extension, you must package and sign the Adobe Extension SDK component separately, then take some additional steps to package that with the native plug-in or scripting component; see "Packaging a hybrid extension" on page 31.

Using ZXPSignCmd

You can use this tool to create a self-signed certificate, create a signed ZXP package, or verify an existing ZXP package.

- To create a signed package:

```
ZXPSignCmd -sign <inputDir> <outputZxp> <p12> <p12Password> [options]
```

inputDir	The path to the folder containing the source files to package.	
outputZxp	The path and file name for the ZXP package.	
p12	The signing certificate; see "How signing works" on page 29.	
p12Password	The password for the certificate.	
options	`-tsa <timestampURL>`	The timestamp server. For example: `https://timestamp.geotrust.com/tsa`

- To verify a ZXP package:

```
ZXPSignCmd -verify <zxp>|<extensionRootDir> [options]
```

zxp	The path and file name for the ZXP package.	
extensionRootDir	The path to the folder containing the deployed ZXP.	
options	`-certinfo`	If supplied, prints information about the certificate, including timestamp and revocation information.
	`-skipOnlineRevocationChecks`	If supplied, skips online checks for certificate revocation when `-certinfo` is set.
	`-addCerts <cert1> <cert2> ...`	If supplied, verifes the certificate chain and assesses whether the supplied DER-encoded certificates are included .

- To create a self-signed certificate:

```
ZXPSignCmd -selfSignedCert <countryCode> <stateOrProvince> <organization>
<commonName> <password> <outputPath.p12> [options]
```

countryCode *stateOrProvince* *organization* *commonName*	The certificate identifying information.	
password	The password for the new certificate.	
outputPath.p12	The path and file name for the new certificate.	
options	`-locality <code>`	If supplied, the locale code to associate with this certificate.
	`-orgUnit <name>`	If supplied, an organizational unit to associate with this certificate.
	`-email <addr>`	If supplied, an email address to associate with this certificate.
	`-validityDays <num>`	If supplied, a number of days from the current date-time that this certificate remains valid.

Example

If you already have a certificate, you can use that. Otherwise, begin by creating a self-signed certificate:

```
./ZXPSignCmd -selfSignedCert US NY MyCompany MyCommonName abc123 MyCert.p12
```

This generates a file named `MyCert.p12` in the current folder. You can use this certificate to sign your extension:

```
./ZXPSignCmd -sign myExtProject myExtension.zxp MyCert.p12 abc123
```

This generates the file `myExtension.zxp` in the current folder, adding these two files to the packaged and signed extension in the final ZXP archive:

- `mimetype`

  A file with the ASCII name of `mimetype` that holds the MIME type for the ZIP container (`application/vnd.adobe.air-ucf-package+zip`).

- `signatures.xml`

  A file in the `META-INF` directory at the root level of the container file system that holds digital signatures of the container and its contents.

How signing works

The signature verifies that the package has not been altered since its packaging. When the Extension Manager tries to install a package, it validates the package against the signature, and checks for a valid certificate. For some validation results, it prompts the user to decide whether to continue with the

installation. In addition, CEP checks for a valid certificate each time a host application tries to run an extension.

Certificates used to cryptographically sign documents or software commonly have an expiration duration between one and four years, and a certificate with a very long lifetime can be prohibitively expensive. If the certificate used to sign the extension has expired and it has no valid time stamp, the extension cannot be installed or loaded. There is no warning or notification to the user before the signature expires. To make your extension available to users again, you would have to repackage it with a new certificate.

A valid timestamp ensures that the certificate used to sign the extension was valid at the time of signing. For this reason, you should always add a time stamp to the signature when you package and sign your extension. A timestamp has the effect of extending the validity of the digital signature, as long as the certificate that you use to add the time stamp is valid at the moment the time stamp is added. You can use a self-signed certificate for adding the time stamp.

These are the possible validation results:

Signature	Signing certificate	Extension Manager action	CEP action
No signature	N/A	Shows error dialog and aborts installation	Extension does not run
Signature invalid	Any certificate	Shows error dialog and aborts installation	Extension does not run
Certificate used to sign has expired, and no time stamp	Any certificate	Shows error dialog and aborts installation	Extension does not run
Certificate used to sign has expired, but has a valid time stamp	Any certificate	Silently installs extension	Extension runs normally
Signature valid	Adobe certificate	Silently installs extension	Extension runs normally
	OS-trusted certificate	Silently installs extension	Extension runs normally
	other certificate	Prompts user for permission to continue the installation	Extension runs normally

To sign extensions, a code-signing certificate must satisfy these conditions:

- The root certificate of the code-signing certificate must be installed in the target operating system by default. This can vary with different variations of an operating system. For example, you may need to check that your root certificate is installed into all variations of Win XP, including home/professional, SP1, SP2, SP3, and so on.
- The issuing certificate authority (CA) of the code-signing certificate must permit you to use that certificate to sign extensions.

To make sure a code-signing certificate satisfies these conditions, check directly with the certificate authority that issues it.

The following CAs and code-signing certificates are recommended for signing extensions:

- GlobalSign
  - ObjectSign Code Signing Certificate
- Thawte
  - AIR Developer Certificate
  - Apple Developer Certificate
  - JavaSoft Developer Certificate
  - Microsoft Authenticode Certificate
- VeriSign
  - Adobe AIR Digital ID
  - Microsoft Authenticode Digital ID
  - Sun Java Signing Digital ID

Packaging a hybrid extension

For a hybrid extension:

- Package and sign the Adobe Extension SDK portion separately, as described in "Creating the deployment package" on page 27.
- Prepare the native plug-in or scripting component for packaging as described in the application-specific SDK.

When all of the components are ready:

1. Create a new staging folder.
2. Add the signed package for the Adobe Extension SDK extension component to the root of the staging folder.
3. Add the application-specific files to the staging folder in their platform-specific subfolders.
4. Add the MXI configuration file to the root of the staging folder; see "Configuring a hybrid extension" on page 32.

   For example, for a hybrid extension that includes a Adobe Extension SDK extension component is named MyExtension, and a C++ plug-in component named MyPlugin that has Mac OS and Windows versions:

   ```
   /staging
       /mac/MyPlugin.plugin
       /win32/MyPlugin.8li
       /win64/MyPlugin.8li
       /MyExtension.zxp
       /MyExtension.mxi
   ```

5. Run the `ZXPSignCmd` tool on the staging folder to bundle and sign its contents into a single ZXP archive.

Configuring a hybrid extension

Extension Manager requires an XML configuration file named *projectName*.MXI to correctly install the extension and all its components in the user's environment. You must create this MXI file and customize it to describe your desired configuration.

When you package your hybrid extension for deployment, the MXI file must be included alongside the packaged and signed Adobe Extension SDK extension component. See "Packaging a hybrid extension" on page 31. For more information about editing the MXI file, see the document *Packaging Extensions with Adobe Extension Manager* (http://www.adobe.com/go/em_file_format).

The MXI file looks like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<macromedia-extension name="com.example.myextension" requires-restart="true"
       version="1.0">

    <author name="Adobe Developer Technologies"/>
    <description><![CDATA[The description.]]></description>
    <license-agreement><![CDATA[Legal Text.]]></license-agreement>

    <products>
        <product familyname="Photoshop" maxversion="" primary="true" version="12.0"/>
    </products>

    <files>
        <file destination="" file-type="CSXS" products="" source="MyExtension.zxp"/>
        <!-- ADD APPLICATION SPECIFIC FILE HERE -->
    </files>

</macromedia-extension>
```

- The file includes the display strings that Extension Manager uses when the extension has been installed, such as the author and description; these can be copied from the ones in the manifest, if those are already set.
- The <files> set must include the <file> element for theAdobe Extension SDK extension component, of file-type "CSXS". In this case there is no need to indicate the destination; Extension Manager knows about the shared installation location used by Adobe extensions.
- You must add a <file> element for each resource file in the cs_resources/ folder. The Extension Manager copies only those files that are specified in the MXI file to the host application. Each application-specific <file> element must include the destination and platform attributes. For example:

```
<files>
    <file destination="" file-type="CSXS" products="" source="MyExtension.zxp"/>
    <file destination="$automate" platform="mac" products="Photoshop"
           source="cs_resources/mac/MyPlugin.plugin"/>
    <file destination="$automate" platform="win" products="Photoshop32"
           source="cs_resources/win32/MyPlugin.8li"/>
    <file destination="$automate" platform="win" products="Photoshop64"
           source="cs_resources/win64/MyPlugin.8li"/>
</files>
```

Installing a packaged and signed extension

Adobe Extension Manager, a tool that is included with all Adobe desktop applications (CS5 and higher), installs extensions that are properly packaged and signed. Adobe Extension Manager is installed at the same time as CS applications; you can launch it from the Start menu in Windows or the Applications folder in Mac OS.

Using Extension Manager

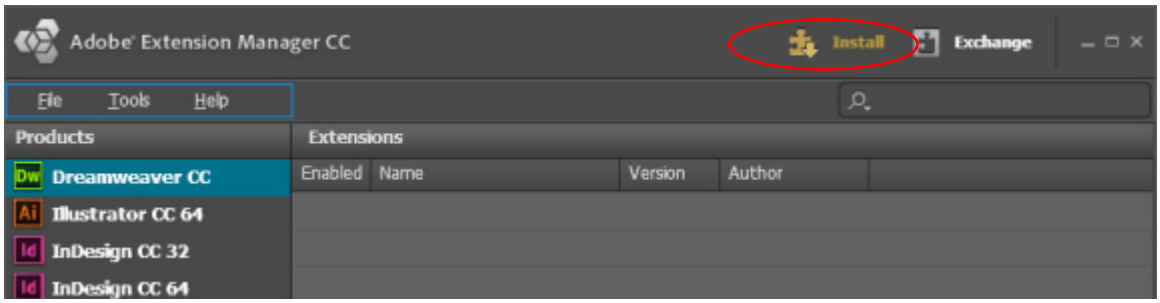
To install the signed ZXP file follow these steps:

1. Open Extension Manager and click **Install**.
2. Browse to the location where your ZXP file is saved, select it, and click **Open** to start the installation process.
3. Extension Manager attempts to validate the package against the signature. For some validation results, it prompts the user to decide whether to continue with the installation; for example, if it cannot verify the publisher, you can choose to install the extension anyway; see "How signing works" on page 29.
4. Once the installation has completed, check that your extension appears in all of the products that it supports.

Testing extension installation

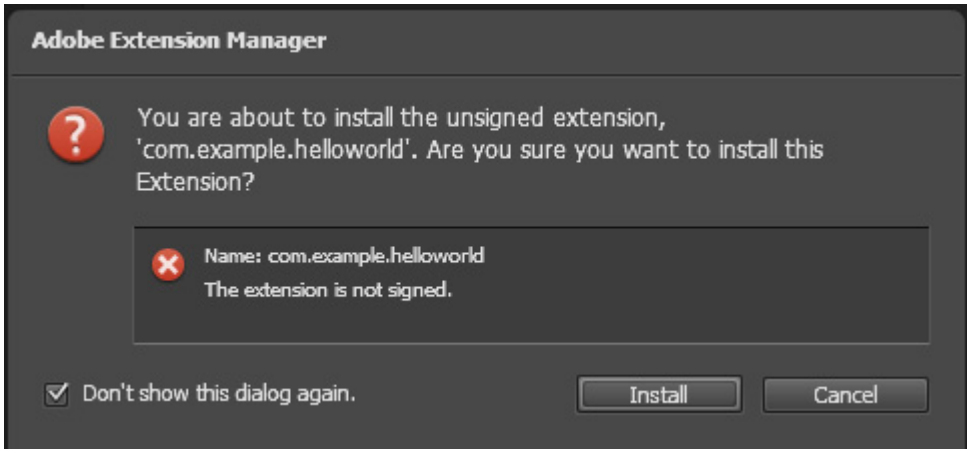
To test whether your package works properly, use Extension Manager to install your extension on your local versions of the Adobe desktop applications.

1. Open Extension Manager and click **Install**.



2. Browse to the location where your ZXP file is saved, select it, and click **Open** to start the installation process.

3. Extension Manager attempts to validate the package against the signature. For some validation results, it prompts the user to decide whether to continue with the installation; for example, if it cannot verify the publisher, the user can choose to install the extension anyway.

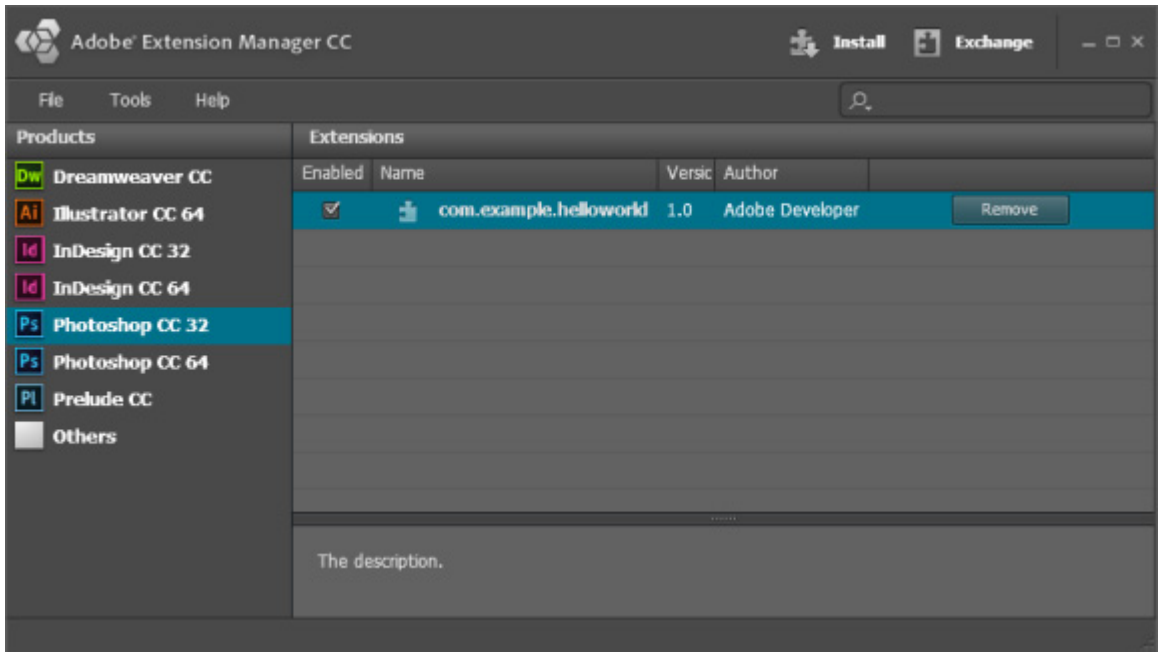


4. Once the installation has completed, check that your extension appears in all of the products that it supports.

Notice that the Extension Manager UI provides the user with information about an installed extension; this information derives from the project properties specified in the manifest. Depending on what you have specified, some of these fields might be blank:

Extension property	Comments
Name	This is the identifying name of the extension bundle, not the display name that appears in the Extensions menu of the host application.
Version	Larger version numbers indicate newer versions.
Author name	May be blank.
Description	May be blank. You can specify a descriptive string, which is simply displayed in the Description panel, or you can provide a URL, in which case the referenced page is shown in the Description panel.
Product	Your extension must support at least one host application for the extension to be installed successfully.

To update how this information is displayed for your extension in the Extension Manager UI, you must specify the corresponding values in your project's manifest.



Troubleshooting the installation

If your package fails to install properly:

- Verify that you have built your extension with the correct structure, and that your extension package contains the correct files in the correct locations.
- Verify that the package has not been modified since being properly signed.

Because the ZXP is an archive file, you can rename the package with the `.zip` extension to examine its contents and verify that it contains all needed files. If you change anything in it, however, the signature no longer matches the content, and the Extension Manager cannot load the package. If you need to make changes, you must create and sign a new package.

Running an extension

Once your extension has been successfully installed, you can test in any of the applications specified in your extension's manifest file. To run your extension, open the host application and choose your extension for the list in **Window > Extensions**. The name that appears in this menu is the one you specified in the manifest.

Here are some problems you might encounter when running an extension, and possible solutions. For further help, check the known problems section in the SDK's Readme file.

Extension does not appear in the application's Window > Extensions menu

Verify that the extension's `manifest.xml` file is set up correctly:

- Verify that the Host ID for your application is correct. Notice that the ID for Photoshop Extended (PHXS) is different from the ID for Photoshop (PHSP).

- Verify that the product locale matches the one listed in the manifest file, or that the locale is given as "All".
- Verify the path given in the Extension/DispatchInfo/MainPath element. The path must be relative to the extension's root folder.
- Verify that the extension has been successfully copied to the Adobe Service Manager's extensions folder. For more details, refer to "Deploying an unsigned extension" on page 13.

If the problem persists, check the host application's CEP log for possible errors; see "System logs" on page 18.

Removing an extension

You can use the Extension Manager to remove an extension.

1. Select the extension in the list of installed programs.
2. Choose **File > Remove Extension**.

The Extension Manager removes it both from the file system, and from the displayed list of currently installed extensions.

Remote debugging

CEP 5.0 supports remote debugging for HTML/JavaScript extensions using the Chrome debugger.

To use this method, you must specify debug ports in a mapping file in your extension's root folder. You can then open the debug port for the host application from a Chrome browser and use the Chrome debugging tools. For example, if you have specified the debug port 8088 for Photoshop, and you open your extension in Photoshop, open the Chrome browser and go to `http://localhost:8088`.

To specify the debug ports, create a special file named ".debug" and place it in your extension's root folder; for example, `MyExtension\.debug`. This is a special file name in both Windows and Mac OS, and you must use the command line to create it:

- In Windows, use `copy con .debug` and CTRL Z to create the empty file.
- In Mac OS, use `touch .debug` to create the empty file.

Edit this file to include valid remote debug ports for all applications you wish to debug, in the Extension Manifest XML format for `<Host>` specifications. Valid `Port` values are in the range 1024 to 65534.

For example, for a bundle that includes four extensions, the file might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<ExtensionList>
    <Extension Id="com.adobe.CEPHTMLTEST.Panel1">
        <HostList>
            <Host Name="PHXS" Port="8000"/>
            <Host Name="IDSN" Port="8001"/>
            <Host Name="AICY" Port="8002"/>
            <Host Name="ILST" Port="8003"/>
            <Host Name="PPRO" Port="8004"/>
            <Host Name="PRLD" Port="8005"/>
            <Host Name="FLPR" Port="8006"/>
        </HostList>
```

```
    </Extension>
    <Extension Id="com.adobe.CEPHTMLTEST.Panel2">
        <HostList>
            <Host Name="PHXS" Port="8100"/>
            <Host Name="IDSN" Port="8101"/>
            <Host Name="AICY" Port="8102"/>
            <Host Name="ILST" Port="8103"/>
            <Host Name="PPRO" Port="8104"/>
            <Host Name="PRLD" Port="8105"/>
            <Host Name="FLPR" Port="8106"/>
        </HostList>
    </Extension>
    <Extension Id="com.adobe.CEPHTMLTEST.ModalDialog">
        <HostList>
            <Host Name="PHXS" Port="8200"/>
            <Host Name="IDSN" Port="8201"/>
            <Host Name="AICY" Port="8202"/>
            <Host Name="ILST" Port="8203"/>
            <Host Name="PPRO" Port="8204"/>
            <Host Name="PRLD" Port="8205"/>
            <Host Name="FLPR" Port="8206"/>
        </HostList>
    </Extension>
    <Extension Id="com.adobe.CEPHTMLTEST.Modeless">
        <HostList>
            <Host Name="PHXS" Port="8300"/>
            <Host Name="IDSN" Port="8301"/>
            <Host Name="AICY" Port="8302"/>
            <Host Name="ILST" Port="8303"/>
            <Host Name="PPRO" Port="8304"/>
            <Host Name="PRLD" Port="8305"/>
            <Host Name="FLPR" Port="8306"/>
        </HostList>
    </Extension>
</ExtensionList>
```

5 Event Handling for Extensions

Event handling must take into account the various environment in which events are generated and received. Extensions that need to exchange information can be running in the same host or in different hosts. If an extension needs to exchange information with its own host application, it could be communicating with the host's ExtendScript interface, or with its native C/C++ interface.

In all frameworks, the basic procedure is similar: in the receiving extension, define a handler callback function or method for the event or message type of interest, and register your handler in an *event* or *message listener*. You can then add code to the sending extension to create and dispatch events or messages of the types that you define.

For communication among extensions:

- CEP supports sending and receiving various types of events within an extension, and among extensions running in the same host application. Extensions can communicate with each other through the CEP event framework. See "CEP event handling" on page 38.
- To communicate with extensions running in different host applications, you must create and send messages through the Interapplication Messaging framework, implemented by the Vulcan API. See "IPC message handling" on page 40.

  COMPATIBILITY NOTE: *The CEP* `GLOBAL` *scope is no longer supported; you must use Vulcan messaging for interapplication communication.*

To communicate with the plug-in API defined for your host application, we provide special-purpose libraries.

- The JavaScript `PlugPlugExternalObject` allows you to call your host's ExtendScript API. See "Event passing between CEP and ExtendScript" on page 41

  An extension that combines native code with an HTML/JavaScript interface is known as a *hybrid*. See Chapter 6, "Creating a Hybrid Extension."

CEP event handling

Even ts are represented in the CEP library by the `CSEvent` class. Instances of this class have a `type` property that reflects the kind of event that has occurred, and a `data` property that allows you to send arbitrary data, including JavaScript object.

Event instances are passed to extensions in notifications. Use JavaScript methods in the CEP library to register event handlers for specific, predefined types of CEP events. See "CEP host application events" on page 42; additional event types are defined by specific host applications. You can also define your own event types, and can create and dispatch events of those types.

The `csInterface.addEventListener()` method registers a handler for a CEP event of a given type. The method supports both *named* and *anonymous* (in-line) event-handler callback functions, and shown in this code snippet:

```
// Create your local CSInterface instance
  var csInterface = new CSInterface();
```

```
// Create a named event handler callback function
function myEventHandler(event)
{
  console.log("type=" + event.type + ", data=" + event.data);
}
// Register the named event handler
CSInterface.addEventListener("cep.sender.event.message", myEventHandler);

// Register an anonymous (in-line) event handler
// (the second argument is the callback function definition)
csInterface.addEventListener("cep.sender.event.message",
        function (event) {
           console.log("type=" + event.type + ", data=" + event.data);}
  )
```

To remove a registered handler for a given event type, call `csInterface.removeEventListener()`.

Sending CEP events

You can create a `CSEvent` object and dispatch it using `CSInterface.dispatchEvent()`.

In your event-handler callback, you can access the properties of the event object. For example, this anonymous handler function retrieves the event type and event data:

```
csInterface.addEventListener("cep.sender.event.message", function (event)
  { console.log("type=" + event.type + ", data=" + event.data); }
  ); // Anonymous function is the second parameter
```

You can pass JavaScript objects as *`Event.data`*. For example:

```
var csInterface = new CSInterface();
csInterface.addEventListener("cep.sender.event.message", function (event)
   {
     var obj = event.data;
     console.log("type=" + event.type + ", data.property1=" + obj.p
   }
); // Anonymous handler function expects data to be an object
```

Here are some examples of different ways to create and dispatch events in JavaScript:

```
// Create an event of a given type, set the data, and send
var csInterface = new CSInterface();
var event = new CSEvent("cep.sender.event.message", "APPLICATION");
event.data = "This is a test!";

csInterface.dispatchEvent(event);

// Create an event, set all properties, and send
var event = new CSEvent(); // create empty event

event.type = "cep.sender.event.message";
event.scope = "APPLICATION"; // only this scope is defined
event.data = "This is a test";

csInterface.dispatchEvent(event);

// Send an object as event data
var event = new CSEvent("cep.sender.event.message", "APPLICATION");
var obj = new Object();
obj.a = "a";
```

```
obj.b = "b";
event.data = obj;
cSInterface.dispatchEvent(event);
```

Before you send events, you might want to make sure the intended target extension has been loaded. You can use the `requestOpenExtension()` method to force a load, then delay sending the event until the load operation has completed by using `setTimeout()`.

For example, if you are sending from a extension named Sender, and handling the event in an extension named Receiver:

```
var csInterface = new CSInterface()
csInterface.requestOpenExtension("Receiver")
setTimeout( function()
    {
        var event = new CSEvent("cep.sender.event.message");
        event.scope = "APPLICATION";
        event.data = "Test message."
        csInterface.dispatchEvent(event);
    },
    400);
```

IPC message handling

If you wish to exchange messages with extensions running in different host applications, the extensions must include the `Vulcan.js` library, which implements the IPC Toolkit for inter-application communication. This library defines the `VulcanInterface` class with a singleton instance.

- Use the `VulcanInterface.addEventListener()` and `VulcanInterface.removeEventListener()` methods to register and unregister your event handlers for IPC messages. The handler function can be a method in an object, or a top-level function.
- Use `VulcanInterface.dispatchEvent()` to send messages.

IPC messages are represented by the `VulcanMessage` type. A message contains a *payload*, which is an arbitrary data string; it can be a JavaScript object. The `VulcanInterface` class defines `setPayload()` and `getPayload()` methods that you use to create and retrieve the message data.

To define a message type, concatenate a descriptive string to the constant `VulcanMessage.TYPE_PREFIX`, whose value, `"vulcan.SuiteMessage."`, identifies the object as an interapplication message. When you have created a message object of this type, use `setPayload()` to add the data content, and `dispatchEvent()` to send it. For example:

```
var interAppMessage = new VulcanMessage (VulcanMessage.TYPE_PREFIX + "myMsgType");
interAppMessage.setPayload("This is the message body");

VulcanInterface.dispatchMessage(interAppMessage);
```

In the receiving extension, register a handler for the message type you have defined. You handler callback method or function take a single argument, the `VulcanMessage` object, and calls `getPayload()` to extract the data. For example:

```
VulcanInterface.addMessageListener (
    VulcanMessage.TYPE_PREFIX + "myMsgType",
    function(message) {
        console.log(VulcanInterface.getPayload(message));}
);
```

Before sending a message, you can verify whether the target application has been installed and is running, and launch an installed application if needed. Use these methods:

```
VulcanInterface.isAppInstalled()
VulcanInterface.isAppRunning()
VulcanInterface.launchApp()
```

The application specifier that you pass to these function is a lower-case app name, with an optional version number:

```
appname[-version]
```

For example:

```
indesign
indesign-10
indesign-10.064
illustrator-18
photoshop
```

If you omit the specific version number, the latest version is assumed.

Call `VulcanInterface.getTargetSpecifiers()` to find application specifiers that are available in your environement.

Event passing between CEP and ExtendScript

A number of desktop applications have ExtendScript APIs, which use a different JavaScript engine from the one that your extension code uses. To send events between the two engines, you use the ExtendScript `ExternalObject`. The `PlugPlugExternalObject` library defines the ExtendScript class `CSEvent`, which allows your ExtendScript code to create and dispatch CEP events.

This library is currently integrated with Photoshop, Illustrator and Premiere Pro CC 2014. It will be added to other applications; for now, obtain the shared library from the Adobe Extension SDK distribution for use with other applications, such as InDesign/InCopy CC 2014.

To receive a CEP event from ExtendScript in your HTML extension, set up an event listener for the message type you define:

```
var csInterface = new CSInterface()
csInterface.addEventListener("cep.extendscript.event.message",
                 function() {
                    console.log(event.data);}
    );
```

In your ExtendScript code for an application that integrates the library, simply load the `PlugPlugExternalObject` shared library. This is done once, preferably at start of your script.

```
try {
    var xLib = new ExternalObject("lib:\PlugPlugExternalObject");
    }
catch(e) { alert(e); }
```

You can then use the library functions to create and dispatch a CEP event:

```
function dispatchCepEvent(in_eventType, in_message) {
    if (xlib) {
        var eventObj = new CSXSEvent();
        eventObj.type = in_eventType;
```

```
        eventObj.data = in_message;
        eventObj.dispatch();
        }
    }

dispatchCepEvent("cep.extendscript.event.message","Update the UI");
```

For an application such as InDesign CC 2014 that does not yet integrate the shared library, you must indicate the path to where you have installed it. For example, this code assumes you have installed the `PlugPlugExternalObject` shared library in the same folder as your script:

```
try {
    var xLib = null;
    var ppLibFile =
        File(File($.fileName).parent).fullName+"/PlugPlugExternalObject");
    if (ppLibFile.exists) {
        var xLib = new ExternalObject("lib:"+ ppLibFile.fullName);
        }
    else {
        throw new Error("Can't find PlugPlugExternalObject: "
                        +ppLibFile.fullName+,$.fileName,$.line);
    }
}
catch(e) { alert(e); }
```

Handling host application events

Some applications that have ExtendScript interfaces have their own ExtendScript event systems, and define application events of many types. To fullly integrate your extension with the application, you can define CEP event listeners that listen for and respond to these application-specific events.

CEP host application events

These basic application event types are defined and supported by Creative Cloud desktop applications. Additional application-specific event types are defined by some applications.

Event type	Sent after	Supported in hosts
`documentAfterActivate`	Document has been activated (new document created, existing document opened, or open document got focus)	InDesign/InCopy Illustrator
`documentAfterDeactivate`	Active document has lost focus	Photoshop InDesign/InCopy Illustrator
`dcoumentAfterSave`	Document has been saved	Photoshop InDesign/InCopy

Event type	Sent after	Supported in hosts
`applicationBeforeQuit`	Host gets signal to begin termination	InDesign/InCopy
`applicationActivate`	Host gets activation event from operating system	Photoshop InDesign/InCopy Illustrator Mac OS only: Premiere Pro Prelude

CEP application event parameters

The event object passed in an event notification contains these parameters, set by the sending host application:

Parameter	Value

`type`	`documentAfterActivate`
	`documentAfterDeactivate`
	`dcoumentAfterSave`
	`applicationBeforeQuit`
	`applicationActivate`

`EventScope`	`APPLICATION`
`appId`	The host application name; see "Supported applications" on page 6.
`extensionId`	null

`data`	The event payload.
	▶ For application events, this is null.
	▶ For document events, an XML string in this format:
	`<eventType>`
	`  <url> URLToFileOnDisk </url>`
	`  <name> fileName </name>`
	`</eventType>`
	For new, unsaved files, the URL element is empty.

Handling Photoshop ExtendScript events

You can register callback functions for Photoshop events by dispatching a CEP event of type `"com.adobe.PhotoshopRegisterEvent"`. The event must include the unique string identifier (type ID) for the Photoshop event you are registering. See Photoshop documentation for a full list of Photoshop events and their unique IDs.

For example:

```
var event = new CSEvent("com.adobe.PhotoshopRegisterEvent", "APPLICATION");
event.extensionId = csInterface.getExtensionID();
event.appId = csInterface.getApplicationID();
```

```
event.data = "1131180832" //Close document event

csInterface.dispatchEvent(event);
```

In your extension, register a handler for the `PhotoshopCallback` event:

```
csInterface.addEventListener("PhotoshopCallback", callback)
```

The following example assumes that you have a simple HTML extension with a button that performs an operation on a Photoshop document. The code shows how to enable or disable the button based on the existence of an active document.

- The `registerPhotoshopEvent()` function registers handler for the Photoshop document events "Open" and "Close".
- The csInterface.addEventListener() call registers a callback for the `PhotoshopCallback` event that is triggered each time the user opens or closes a document in Photoshop.

```
(function () {
    var csInterface = new CSInterface()
    function init() {
        themeManager.init();
    // set the button state when the panel/dialog/window is opened the first time

        csInterface.evalScript("return (app.documents.length > 0)", setButtonState)

        function registerPhotoshopEvent(in_eventId) {
            var event = new CSEvent("com.adobe.PhotoshopRegisterEvent",
                                    "APPLICATION");
            event.extensionId = csInterface.getExtensionID();
            event.appId = csInterface.getApplicationID();
            event.data = in_eventId
            csInterface.dispatchEvent(event);
            }

        csInterface.addEventListener("PhotoshopCallback" , function(event) {
            csInterface.evalScript("return (app.documents.length > 0)",
                                    setButtonState)
            });

        function setButtonState(in_msg) {
            var disabled = (in_msg.data != "true");
            if ($('#btn_applyFilter').prop("disabled") != disabled) {
                $('#btn_applyFilter').prop("disabled",disabled);
                }
        };

    $('#btn_applyFilter').click(function() {
            // perform your Photoshop operation here
        });

    var closeEventid = "1131180832"
    var openEventId = "1332768288"
    registerPhotoshopEvent(closeEventid);
    registerPhotoshopEvent(openEventId);
}

init();
}());
```

Handling InDesign ExtendScript events

The InDesign ExtendScript DOM defines a wide variety of application-specific events.

Here is an example that assumes that you have a simple HTML extension with a button that performs an operation over selected text in an InDesign document. It shows how to enable or disable the button based on the user selection.

The following code is the content of an ExtendScript JSX file, which you would typically add to your extension manifest as:

```
<ScriptPath> ./jsx/[scriptName].jsx </ScriptPath>
```

In this script:

- The `app.addEventListener()` call registers an interest in the event `"afterSelectionChanged"`.
- The callback function `hasSelectedText()` checks if the user has selected text in the current document, and then dispatches the CEP event `"indesign.cep.event.hasSelectedText"`, which is handled by the HTML extension.

```
try {
    var xLib = null;
    var plugExternalLib =
        File(File($.fileName).parent).fullName+"/PlugPlugExternalObject");

    if (plugExternalLib.exists) {
        var xLib = new ExternalObject("lib:"+ plugExternalLib.fullName);
        }

    else {
        throw new Error("Can't find PlugPlugExternalObject: "
                        +plugExternalLib.fullName,$.fileName,$.line);
    }

    app.addEventListener("afterSelectionChanged",hasSelectedText,false);
    }
catch(e) { alert(e); }

function dispatchCepEvent(in_eventType,in_message) {
    if (xlib) {
        var eventObj = new CSXSEvent();
        eventObj.type = in_eventType;
        eventObj.data = in_message;
        eventObj.dispatch();
        }
    }

// Checks if there is any text selected
function hasSelectedText(in_event) {
    try {
        do {
            var retVal = false;
            if (! in_event) { break; }

            if (! (in_event.target instanceof Document ||
                   in_event.target instanceof LayoutWindow)) {break; }

            if (in_event.target.selection.length != 1) { break; }
```

```
            var selection = in_event.target.selection[0];
            if (! (selection.parent instanceof Story)) { break; }
            retVal = (selection.characters.length > 0)
        } while (false);

        dispatchCepEvent("indesign.cep.event.hasSelectedText",retVal);
    }
    catch(e) { alert(e.line + ": " + e) }
}
```

The following is an example of JavaScript code that could be used to control the UI of an HTML extension.

- The call to `csInterface.addEventListener()` registers a listener for the event `"indesign.cep.event.hasSelectedText"`.
- Theevent handler callback `setButtonState()` shows how to retrieve the data from the event object.

```
(function () {
    var csInterface = new CSInterface()

    function init() {
        themeManager.init();
        csInterface.addEventListener("indesign.cep.event.hasSelectedText",
                                     setButtonState)

    function setButtonState(in_event) {
        var disabled = (in_event.data != "true")
        if ($('#btn_applyFormat').prop("disabled") != disabled) {
            $('#btn_applyFormat').prop("disabled",disabled);
            }
        };
    }
    init();
}());
```

6 Creating a Hybrid Extension

A hybrid extension is a package that combines an Adobe extension with an application-specific extension or plug-in that uses the native C/C++ or scripting API. This allows you to build extensions with rich interfaces and still take advantage of the extended native API for the host application.

You must package the several components of a hybrid extension into a ZXP package. The Extension Manager installs the package on the user's machine as a single extension; it looks the same as any other extension to the end user.

As an extension developer, you can choose to use application-specific C/C++ plug-ins or scripting extensions to extend Adobe desktop applications, in addition to your HTML/JavaScript extension component. You might want to do this, for example, when:

- You have legacy code that you still want to support.
- The feature you are developing requires a capability supported by the native scripting or C/C++ API layer, that is not accessible via your Adobe extension; for example, some applications allow you to create custom menus using C++ extensibility.
- You have CPU-intensive tasks to perform that are more suited to C++ than to JavaScript.

Writing hybrid extensions

If you are already familiar with writing Adobe extensions and native application extensions (for example, a Photoshop or InDesign C++ extension, or a Dreamweaver JavaScript extension) there is little more you need to learn. The two parts of a hybrid extension are implemented as standalone components.

- Create the Adobe Application Extension using the Adobe Extension SDK.
- Create your C/C++ or scripting API plug-in using the application-specific SDK and recommended tools. If you have never built a native plug-in for your host application, check the application-specific SDKs for details; see Adobe Developer Connections.

The only thing you need to do is package them together so that they can be deployed in the user's environment as a single extension.

Testing a hybrid extension

During development, test the components of your hybrid extension separately.

- Launch and debug the Adobe Extension SDK component as described in Chapter 2, "Running and Debugging your Extension."
- Install the application-specific plug-in or extension in the host as instructed in the application-specific SDK. Debug it using the recommended development tools, such as XCode or Visual Studio.

To install the plug-in component, copy the files to the Plug-ins or Extensions folder, or point the host application to your plug-in build folder. For example, InDesign looks for its plug-ins in:

```
<InDesign installation location>/Plug-ins/
```

For details of how to package your hybrid extension for deployment, see "Packaging a hybrid extension" on page 31.

Communicating between components

Adobe offers the `PlugPlug` library for communication between C/C++ and JavaScript. You can include these libraries directly in Photoshop and InDesign native plug-ins.

The `PlugPlug` library exposes these functions to the C++ plug-in:

```
PlugPlugLoadExtension()
PlugPlugUnloadExtension()
PlugPlugDispatchEvent()
PluPlugAddEventListener()
PlugPlugRemoveEventListener()
```

For ease of use, InDesign and Illustrator SDKs wrap these functions around classes. Photoshop and Premiere Pro do not directly expose the `PlugPlug` functions, but you can easily do so, as shown in the samples that are provided with the Adobe Extension SDK.

InDesign hybrid extensions

There are two basic techniques for implementing two-way communication between an HTML extension and a native InDesign plug-in:

- Add an ExtendScript scripting interface to your plug-in, and call `evalScript()` in the HTML extension to run the ExtendScript functions that you define:

  ```
  new CSInterface().evalScript("app.speak");
  ```

- Send CEP events between the native plug-in and the HTML extension. This is easier, and is recommended.

This example of sending and receiving CEP events in a native plug-in is based on the sample project ScriptingComms, which is included in the InDesign SDK.

```
#include "ICSXSPlugPlugEventHandler.h"
#include "adobe/unicode.hpp" // for adobe::to_utf8
#include "FileTypeRegistry.h"

InterfacePtr plugPlug(GetExecutionContextSession(), UseDefaultIID());

PMString csxsEventStr;
csxsEventStr.SetCString("Hello from CPP!");
PMString eventData(csxsEventStr);
std::string csxsEventUtf8;
adobe::to_utf8(csxsEventStr.begin(), csxsEventStr.end(),
               std::back_inserter(csxsEventUtf8));

ICSXSPlugPlugEventHandler::CSXSEvent responseEvent;
responseEvent.type = "com.adobe.indesign.scriptingcomms.html";
responseEvent.scope = ICSXSPlugPlugEventHandler::kEventScope_Application;
responseEvent.extensionId = nil;
responseEvent.data = csxsEventUtf8.c_str();

if (LocaleSetting::GetLocale().IsProductFS(kInDesignProductFS)) {
    // as defined in ICSXSPlugPlugEventHandler.h
```

```
        responseEvent.appId = kIDEnigmaCode;
        }
    else {
        // as defined in ICSXSPlugPlugEventHandler.h
        responseEvent.appId = kICEnigmaCode;
        }
    plugPlug->DispatchPlugPlugEvent(&responseEvent);
```

This adds an event listener for CSXS events. A good place to create such a message listener is in a startup/shutdown service, to ensure that the listener is created on InDesign startup.

```
InterfacePtr plugPlug(GetExecutionContextSession(), UseDefaultIID());
plugPlug->AddPlugPlugEventListener("com.adobe.indesign.scriptingcomms.cpp",
                                   &MessageEventListener, nil);
```

This callback method handles a received message by displaying an alert dialog ,and then sends a message back to the HTML Extension.

```
MessageEventListener(const ICSXSPlugPlugEventHandler::CSXSEvent* const csxsEvent,
                     void* const context) {
    // Display received data in alert dialog..
    PMString receivedData = PMString("Message received: ");
    receivedData.Append(csxsEvent->data);
    receivedData.SetTranslatable(kFalse);
    CAlert::InformationAlert(receivedData);
    // Send a message back to the HTML extension..
    InterfacePtr<ICSXSPlugPlugEventHandler> plugPlug(GetExecutionContextSession(),
                                                     UseDefaultIID());
    PMString csxsEventStr;
    csxsEventStr.SetCString("Hello from CPP!");
    PMString eventData(csxsEventStr);
    std::string csxsEventUtf8;
    adobe::to_utf8(csxsEventStr.begin(), csxsEventStr.end(),
    std::back_inserter(csxsEventUtf8));
    ICSXSPlugPlugEventHandler::CSXSEvent responseEvent;
    responseEvent.type = "com.adobe.indesign.scriptingcomms.html";
    responseEvent.scope = ICSXSPlugPlugEventHandler::kEventScope_Application;
    responseEvent.extensionId = nil;
    responseEvent.data = csxsEventUtf8.c_str();
    if (LocaleSetting::GetLocale().IsProductFS(kInDesignProductFS)) {
        responseEvent.appId = kIDEnigmaCode;
        }
    else {
        responseEvent.appId = kICEnigmaCode;
    }
    plugPlug->DispatchPlugPlugEvent(&responseEvent);
}
```

Sending and receiving events from JavaScript

This JavaScript code dispatches a CEP event to native event listener:

```
    var csInterface = new CSInterface()
    var event = new CSEvent("com.adobe.indesign.scriptingcomms.cpp")
    event.scope = "APPLICATION";
    event.appId = csInterface.getApplicationId()
    event.extensionId = "com.adobe.indesign.scriptingcomms.html";
    event.data = "Hello from HTML!";
    csInterface.dispatchEvent(event);
```

This adds an event listener for an InDesign event:

```
csInterface.addEventListener("com.adobe.indesign.scriptingcomms.html",
    function(event) {
        alert("Message received: " + event.data);
        }
);
```

Illustrator hybrid extensions

Illustrator SDK defines two classes to manage the communication between plug-ins and HTML extensions:

- The `SDKPlugPlug` class loads and unloads the `PlugPlug` library, and exposes these functions:

  ```
  PlugPlugLoadExtension()
  PlugPlugUnloadExtension()
  PlugPlugDispatchEvent()
  PluPlugAddEventListener()
  PlugPlugRemoveEventListener()
  ```

- The `HtmlUIController` class contains these virtual functions:

  ```
  LoadExtension()
  UnloadExtension()
  RegisterCSXSEventListeners()
  RemoveEventListeners()
  ```

  `RegisterCSXSEventListeners()` and `RemoveEventListeners()` are purely virtual functions, and must be implemented in a derived class.

The source and binary files that define the functions are part of the Illustrator SDK:

`Illustrator_SDK_root``/samplecode/common/includes`
`Illustrator_SDK_root``/samplecode/common/source`

This example code shows how a plug-in can receive an event from an HTML extension and then send an event straight back to the extension. The call to `RegisterCSXSEventListeners()` adds an event listener and `SendMessageToHtml()` constructs and dispatches a CEP Event.

The code snippet is assumed to be part of a subclass called `PanelController`, derived from `HtmlUIController`, which implements the virtual functions `RegisterCSXSEventListeners()` and `RemoveEventListeners()`.

```
#define EXTENSION_ID "comms"
static const char* EVENT_FROM_HTML = "com.adobe.illustrator.event.fromHTML";
static const char* EVENT_FROM_AISDK = "com.adobe.illustrator.event.fromAISDK";
static const char* ILST_APP = "ILST";

PanelController::PanelController()
:HtmlUIController(EXTENSION_ID)
{}
```

```
csxs::event::EventErrorCode PanelController::RegisterCSXSEventListeners() {
    csxs::event::EventErrorCode result = csxs::event::kEventErrorCode_Success;
    do {
        result = htmlPPLib.AddEventListener(EVENT_FROM_HTML,handleMessage,this);
        if (result != csxs::event::kEventErrorCode_Success) { break; }
        }
    while (false);
    return result;
    }

csxs::event::EventErrorCode PanelController::RemoveEventListeners() {
    csxs::event::EventErrorCode result = csxs::event::kEventErrorCode_Success;
    do {
        result = htmlPPLib.RemoveEventListener(EVENT_FROM_HTML,handleMessage,this);
        if (result != csxs::event::kEventErrorCode_Success) { break; }
        }
    while (false);
    return result;
    }

static void handleMessage(const csxs::event::Event* const eventParam,
                          void* const context) {
    sAIUser->MessageAlert(ai::UnicodeString(eventParam->data));
    AppContext appContext(gPlugin->GetPluginRef());
    PanelController* panelController = (PanelController*) context;
    panelController->SendMessageToHtml();
    }

ASErr PanelController::SendMessageToHtml() {
    AIErr error = kNoErr;
    std::string msgStr("Hello back, from the AI SDK");
    csxs::event::Event event = {EVENT_FROM_AISDK,
                                csxs::event::kEventScope_Application,
                                ILST_APP,
                                NULL,
                                msgStr.c_str()};
    htmlPPLib.DispatchEvent(&event);
    return error;
    }
```

You can only register event listeners after you have been notified that the setup of the `PlugPlug` library has been completed. Your `Plugin` class should add a notifier on `StartupPlugin`, so that it receives this notification:

```
error = sAINotifier->AddNotifier(fPluginRef, kSimplePluginName,
                                 kAICSXSPlugPlugSetupCompleteNotifier,
                                 &fPlugPlugSetupCompleteNotifier);
```

- Define `fPlugPlugSetupCompleteNotifier(AINotifierHandle)` as a private member of the `PanelController` class.

```
SErr SimplePlugin::Notify(AINotifierMessage* message) {
    ASErr error = kNoErr;
    if (message->notifier == fPlugPlugSetupCompleteNotifier) {
        // fPanelController is an instance of PanelController
        if (fPanelController != NULL)
        fPanelController->RegisterCSXSEventListeners();
    }
return error;
}
```

To remove event listeners on `::ShutdownPlugin`:

```
ASErr SimplePlugin::ShutdownPlugin(SPInterfaceMessage *message) {
    if (fPanelController != NULL) {
        fPanelController->RemoveEventListeners();
        delete fPanelController;
        fPanelController = NULL;
        Plugin::LockPlugin(false);
    }
    return kNoErr;
}
```

Sending and receiving CEP events from JavaScript

This JavaScript code dispatches a CEP event to native event listener:

```
var csInterface = new CSInterface()
var event = new CSEvent("com.adobe.illustrator.event.fromHTML")
event.scope = "APPLICATION";
event.appId = csInterface.getApplicationId()
event.extensionId = "com.adobe.illustrator.simpleUI";
event.data = "Hello from HTML!";

csInterface.dispatchEvent(event);
```

To add an event listener for an Illustrator event:

```
csInterface.addEventListener("com.adobe.illustrator.event.fromAISDK",
       function(event) {
          alert("Message received: " + event.data);
          }
    );
```

Photoshop hybrid extensions

Although the `PlugPlug` library is included in Adobe Photoshop, the application does not currently expose the functions. As a workaround we have modified the `SDKPlugPlug` class from the Illustrator SDK for use with your Photoshop plug-ins. Download the `SDKPlugPlug` class for Adobe Photoshop from [link not defined, waiting for decision].

The `SDKPlugPlug` class loads and unloads the `PlugPlug` library and exposes the following functions:

```
PlugPlugLoadExtension()
PlugPlugUnloadExtension()
PlugPlugDispatchEvent()
PluPlugAddEventListener()
PlugPlugRemoveEventListener()
```

These code snippets illustrate an automation plug-in that implements an event listener (`fSDKPlugPlug->AddEventListener`) for a CEP event coming from an HTML extension, and dispatches a CEP event (`HandleMessage`) back to the HTML extension. This code forces the extension dialog to be displayed (`fSDKPlugPlug->LoadExtension`).

```
#define EXTENSION_ID "com.adobe.photoshop.simpleUI"
static const char* EVENT_FROM_HTML = "com.adobe.photoshop.event.fromHTML";
static const char* EVENT_FROM_PSSDK = "com.adobe.photoshop.event.fromPsSDK";
static const char* PS_APP_ID = "PHXP";
```

```
SDKPlugPlug* fSDKPlugPlug = NULL;
DLLExport SPAPI SPErr PluginMain(const char* caller,
                                 const char* selector,
                                 const void* data ) {
    SPErr error = kSPNoError;
    SPMessageData* msg = (SPMessageData *) data;
    SPBasicSuite* sSPBasic = msg->basic;

    if (sSPBasic->IsEqual(caller, kSPInterfaceCaller)) {
        if (sSPBasic->IsEqual(selector, kSPInterfaceStartupSelector))
            error = StartupPlugin();
        else if (sSPBasic->IsEqual(selector, kSPInterfaceShutdownSelector))
            ShutdownPlugin();
        }

    else if (sSPBasic->IsEqual(caller, kPSPhotoshopCaller)) {
        if (sSPBasic->IsEqual(selector, kPSDoIt))
            error = DoIt(msg, sSPBasic);
        }
    return error;
    }

    SPErr StartupPlugin(void) {
        SPErr error = kSPNoError;
        if (fSDKPlugPlug == NULL) {
            fSDKPlugPlug = new SDKPlugPlug();
            error = fSDKPlugPlug->Load();
            }
        return error;
    }

    SPErr ShutdownPlugin(void) {
        SPErr error = kSPNoError;
        if (fSDKPlugPlug != NULL) {
            fSDKPlugPlug->RemoveEventListener(EVENT_FROM_HTML,HandleMessage,NULL);
            delete fSDKPlugPlug;
            fSDKPlugPlug = NULL;
            }
        return error;
    }

    static SPErr DoIt (SPMessageData* sSPMsg, SPBasicSuite* sSPBasic) {
        SPErr error = kSPNoError;
        error = sSPBasic->AcquireSuite(kPSActionControlSuite,
                                       kPSActionControlSuiteVersion,
                                       (const void**) &sPSActionControl);
        if (! error) {
            if (fSDKPlugPlug != NULL) {
                fSDKPlugPlug->AddEventListener(EVENT_FROM_HTML,HandleMessage,NULL);
                fSDKPlugPlug->LoadExtension(EXTENSION_ID);
                }
            }
        return error;
    }
```

```
static void HandleMessage(const csxs::event::Event* const eventParam,
                          void* const context) {
    std::string msgStr("Hello from Photoshop the cpp SDK");
    csxs::event::Event event = {EVENT_FROM_PSSDK,
                                csxs::event::kEventScope_Application,
                                PS_APP_ID,
                                NULL,
                                msgStr.c_str()};
    fSDKPlugPlug->DispatchEvent(&event);
}
```

You must plan where to load and unload the `PlugPlug` library. In this code snippet we assume that the library has been added to the plug-in's PiPL resource with the messages `startupRequired` and `shutdownRequired`, so that the plug-in is notified and can load the `PlugPlug` library on startup and unload it on shutdown.

```
resource 'PiPL' ( 16000, "SimpleExtension", purgeable) {
    {
        ...
        Messages {
            startupRequired,
            doesNotPurgeCache,
            shutdownRequired,
            acceptProperty
            },
        },
    }
```

Sending and receiving CEP events from JavaScript

This JavaScript code dispatches a CEP event to native event listener:

```
var csInterface = new CSInterface()
var event = new CSEvent("com.adobe.photoshop.event.fromHTML")
event.scope = "APPLICATION";
event.appId = csInterface.getApplicationId()
event.extensionId = "com.adobe.photoshop.simpleUI";
event.data = "Hello from HTML!";

csInterface.dispatchEvent(event);
```

To add an event listener for a Photoshop event:

```
csInterface.addEventListener("com.adobe.photoshop.event.fromPsSDK",
                 function(event) {
                    alert("Message received: " + event.data);
                    }
    );
```

Premiere Pro hybrid extensions

Although the `PlugPlug` library is included in Adobe Premier Pro, the application does not currently expose the functions. As a workaround we have modified the `SDKPlugPlug` class from the Illustrator SDK for use with your Premier Pro plug-ins. Download the `SDKPlugPlug` class for Adobe Premiere Pro from [link not defined, waiting for decision].

The `SDKPlugPlug` class loads and unloads the `PlugPlug` library and exposes the following functions:

```
PlugPlugLoadExtension()
PlugPlugUnloadExtension()
PlugPlugDispatchEvent()
PluPlugAddEventListener()
PlugPlugRemoveEventListener()
```

This code snippet illustrates a Device plug-in, based on the Sample project "Device" that is part of the Premiere SDK (`Premiere Pro SDK/Examples/Projects/Device`). The code shows how to implement an event listener (`fSDKPlugPlug->AddEventListener`) for a CEP event coming from an HTML extension, and dispatch a CEP event (`HandleMessage`) back to the HTML extension. This code forces the extension dialog to be displayed (`fSDKPlugPlug->LoadExtension`).

The selector `dsSetup` is sent when the plug-in's device is selected (that is, the user chooses the device in **Preferences > Devices Controls > Options**). We use this selector to load the `PlugPlug` library, add listeners and display the extension dialog.

When the selector `dsCleanup` is sent, we remove our listeners and delete the `SDKPlugPlug` class.

```
#define EXTENSION_ID "com.adobe.premiere.simpleUI"
static const char* EVENT_FROM_HTML = "com.adobe.premiere.event.fromHTML";
static const char* EVENT_FROM_PRSDK = "com.adobe.premiere.event.fromPrSDK";
static const char* PPRO_APP_ID = "PPRO";

SDKPlugPlug* fSDKPlugPlug = NULL;

PREMPLUGENTRY DllExport
xDevice (short selector, DeviceHand theData) {
    ....
    switch (selector) {
        case dsInit:
            resultS = dmNoError;
            ...
            break;

        case dsRestart:
            ...
            break;

        case dsSetup:
            resultS = dmNoError;
            ...
            if (fSDKPlugPlug == NULL) {
                fSDKPlugPlug = new SDKPlugPlug();
                fSDKPlugPlug->Load();
                }
            fSDKPlugPlug->AddEventListener(EVENT_FROM_HTML,HandleMessage,NULL);
            fSDKPlugPlug->LoadExtension(EXTENSION_ID);
            break;
```

```
    case dsCleanup:
        resultS = dmNoError;
        lDataH = reinterpret_cast<LocalHand>((*theData)->deviceData);
        // Here is where we dispose of anything we allocated
        // and close drivers that we opened
        if (lDataH) {
            (*theData)->piSuites->memFuncs->disposeHandle (
               reinterpret_cast<PrMemoryHandle>(lDataH));
            (*theData)->deviceData = 0;
            }
        if (fSDKPlugPlug != NULL) {
            fSDKPlugPlug->RemoveEventListener(BUTTON_PRESSED,HandleMessage,NULL);
            delete fSDKPlugPlug;
            fSDKPlugPlug = NULL;
            }
        break;
    }
...
}

static void HandleMessage(const csxs::event::Event*
                          const eventParam,
                          void* const context)
{
    std::string msgStr("Hello from Premiere the cpp SDK");
    csxs::event::Event event = {EVENT_FROM_PRSDK,
                                csxs::event::kEventScope_Application,
                                PPRO_APP_ID,
                                NULL,
                                msgStr.c_str()};
    fSDKPlugPlug->DispatchEvent(&event);
}
```

7 Localizing an Extension

In order to localize your extension, you must ceate resource files for your project. Your localized string resources can be used in both the HTML components that make up your UI, and in a number of places in the manifest.

- Provide the localized string resources for each supported locale as part of your extension, using the folder structure and naming conventions described in "Localization resources" below.
- To localize elements of your HTML interface, you must first initialize the JavaScript ResourceBundle object with the current locale, then use that object in your JavaScript code to retrieve the strings for the current locale. See "Localizing the extension's UI" on page 58.
- You can also localize strings, such as your product description, that are taken from your manifest and displayed in Adobe Extension Manager. See "Localizing the extension's manifest file" on page 58.

Localization resources

You must provide a resource file for each supported locale, in the proper location in your root extension folder, using this naming convention for both the folders and files. Each file defines a set of string in the form of key-value pairs, so that your JavaScript code and the Adobe Extension Manager can access each string by its key.

Define your localization string resources in a set of files that contain key/value pairs in UTF-8 format. Name each such file "`messages.properties`", and store it in a locale-specific subfolder of a folder called "`locale`" in the root folder of your project.

For example:

```
#locale/es_ES/messages.properties
    menuTitle=Mi extension
    buttonLable=Mi boton
    ...
```

If you have decided that your extension should run in all languages and you do not have specific support for a locale, the resources in the default file are used. The application looks for a properties file at the top level of the `locale/` folder to use as the default resource file.

```
#locale/messages.properties
    menuTitle=My extension
    buttonLabel=My button
    ...
```

If the application UI locale exactly matches one of the locale-specific folders, those resources are used in your extension interface. The match must be exact; for instance, if you have resources for `fr_FR` but the application locale is `fr_CA`, the default properties are used.

You must copy the `locale/` folder and its contents into the project's Output folder before you attempt to run or debug the extension.

To make a localized string available to HTML controls, use this special format in the locale resource file:

```
    keyName.value=string value
```

For example:

```
buttonLabel.value=My button
```

This allows you to reference the string from acustom attribute, `data-locale`, which is available for elements that normally have a string value in the `value` attribute. See example below.

Localizing the extension's UI

You must make the localization resources available as part of initializing your extension during load.

Call the JavaScript method `CSInterface.initResourceBundle()` in your extension's initialization routine in order to initialize the locale resources.

```
var csInterface = new CSInterface();
csInterface .initResourceBundle();
```

At run time, the extension infrastructure loads the resources that match the locale used in the host application, or the default `messages.properties` file if no matching folder is found.

Your JavaScript code can use the `ResourceBundle` object to access the localized strings for the current locale. For example, this simple code snippet accesses the localized string associated with the key "`menuTitle`".

```
var cs = new CSInterface();
// Initialize for the current locale of the host application.
var resourceBundle = cs.initResourceBundle();

// Access a localized string by its key in your JavaScript code
<script type="text/javascript">
    document.write(resourceBundle.menuTitle);
</script>
```

To use a localized string in an HTML control, use the custom attribute, `data-locale`, which is available for elements that normally have a string value in the `value` attribute.

- This attribute must reference a string that is defined in the resource file using the special format *`keyName`*`.value=`*`string value`*.
- Supply the `data-locale="`*`keyName`*`"` attribute instead of the `value` attribute for the control.

For example, suppose you have defined this localized string resource:

```
submitButton.value=Submit
```

This HTML element retrieves the localized string and displays in an HTML input control:

```
<input type="submit" value="" data-locale="submitButton"/>
```

Localizing the extension's manifest file

If you have provided localization resources, you can localize values within a manifest's `DispatchInfo/UI` element by replacing the value with a `messages.properties` key, preceded by the percent symbol. For example:

```
<Menu>%menuTitle</Menu>
```

When your extension runs, the application looks for this key in the locale-specific `messages.properties` file, and uses the value to display the menu item.

You can use this mechanism to localize other information in the manifest file. For example, to have locale-dependent default extension geometry, or to load a different icon:

```
<Menu>%menuTitle</Menu>
<Geometry>
    <Size>
        <Height>%height</Height>
        <Width>%width</Width>
    </Size>
</Geometry>

<Icons>
    <Icon Type="Normal">%icon</Icon>
    <Icon Type="RollOver">%icon</Icon>
</Icons>
```

8 CEP Engine JavaScript Extension Reference

CEP (formerly CSXS) Extensions extend the functionality of the host application that they run in. Extensions are loaded into applications through the PlugPlug Library architecture. Starting from version 4.0, CEP supports the use of HTML/JavaScript technology to develop extensions.

In order to use the file I/O functionality provided by the CEP engine in an HTML5/JavaScript application extension, Adobe provides a JavaScript bridge to the native C++ CEP engine:

`CEPEngine_extension.js`

It is not necessary to include this library in your extension project; it is integrated into CEP 5. The engine defines:

- "Extension control functions" on page 60
- "File I/O functions" on page 65

Extension control functions

These functions allow you to start, query, and terminate extensions. The functions are presented here in alphabetical order.

CreateProcess()	Runs the executable file for an extension in a new process.
GetWorkingDirectory	Retrieves the working directory of an extension process.
IsRunning()	Reports whether an extension process is currently running.
OnQuit()	Registers an on-quit callback handler method for an extension process.
RegisterExtensionUnloadCallback()	Registers a callback function for extension unload.
SetPanelFlyoutMenu()	Creates a new flyout menu for an extension panel.
SetupStdErrHandler()	Registers up a standard-error handler for an extension process.
SetupStdOutHandler()	Registers a standard-output handler for an extension process.
Terminate()	Terminates an extension process.
UpdatePanelFlyoutMenu()	Adds or modfiies a menu item in the flyout menu for the extension panel.
WaitFor()	Waits for an extension process to quit.
WriteStdIn()	Writes data to the standard input of an extension process.

CreateProcess()

Runs the executable file for an extension in a new process.

`CreateProcess(`*`args`*`)`

`args`	Array of String	The path to the executable, followed by the arguments to that executable.

RETURNS: An object with these properties:

- `data`: The process ID (pid) of the new process (an integer), or -1 on error.
- `err`: The status of the operation, one of:
  `NO_ERROR`
  `ERR_UNKNOWN`
  `ERR_INVALID_PARAMS`
  `ERR_EXCEED_MAX_NUM_PROCESS`
  `ERR_NOT_FOUND`
  `ERR_NOT_FILE`

GetWorkingDirectory

Retrieves the working directory of an extension process.

`GetWorkingDirectory(`*`pid`*`)`

`pid`	Number	The process ID of the extension, as returned by CreateProcess().

RETURNS: An object with these properties:

- `data`: The path of the working directory.
- `err`: The status of the operation, one of:
  `NO_ERROR`
  `ERR_UNKNOWN`
  `ERR_INVALID_PARAMS`
  `ERR_INVALID_PROCESS_ID`

IsRunning()

Reports whether an extension process is currently running.

`IsRunning(`*`pid`*`)`

`pid`	Number	The process ID of the extension, as returned by CreateProcess().

RETURNS: An object with these properties:

- `data`: True if the process is running, false if not.
- `err`: The status of the operation, one of:
  `NO_ERROR`

ERR_UNKNOWN
ERR_INVALID_PARAMS
ERR_INVALID_PROCESS_ID

OnQuit()

Registers an on-quit callback handler method for an extension process.

`OnQuit(pid, callback)`

pid	Number	The process ID of the extension, as returned by CreateProcess().
callback	Function	The handler function for the on-quit callback.

RETURNS: An object with these properties:

- `err`: The status of the operation, one of:
  NO_ERROR
  ERR_UNKNOWN
  ERR_INVALID_PARAMS
  ERR_INVALID_PROCESS_ID

RegisterExtensionUnloadCallback()

Registers a callback function for extension unload. If called more than once, the last callback that is successfully registered is used.

`RegisterExtensionUnloadCallback (callback)`

callback	Function	The handler function for the extension-unload callback.

RETURNS: An object with these properties:

- `err`: The status of the operation, one of:
  NO_ERROR
  ERR_INVALID_PARAMS

SetPanelFlyoutMenu()

Creates a new flyout menu for an extension panel. You must register a handler for the `flyoutMenuClicked` event to respond to events in this menu. The `Event.data` is an object with attributes `menuId` and `menuName`.

`SetPanelFlyoutMenu(menu)`

menu	String	An XML string that defines the menu.

Here is an example of XML that defines a menu:

```
<Menu>
  <MenuItem Id="menuItemId1" Label="TestExample1" Enabled="true" Checked="false"/>
  <MenuItem Label="TestExample2">
     <MenuItem Label="TestExample2-1" >
```

```
        <MenuItem Label="TestExample2-1-1" Enabled="false" Checked="true"/>
      </MenuItem>
      <MenuItem Label="TestExample2-2" Enabled="true" Checked="true"/>
    </MenuItem>
    <MenuItem Label="---" />
    <MenuItem Label="TestExample3" Enabled="false" Checked="false"/>
  </Menu>
```

SetupStdErrHandler()

Registers a standard-error handler for an extension process.

`SetupStdErrHandler(pid, callback)`

pid	Number	The process ID of the extension, as returned by CreateProcess().
callback	Function	The handler function for the standard-error callback.

RETURNS: An object with these properties:

- `err`: The status of the operation, one of:
  `NO_ERROR`
  `ERR_UNKNOWN`
  `ERR_INVALID_PARAMS`
  `ERR_INVALID_PROCESS_ID`

SetupStdOutHandler()

Registers a standard-output handler for an extension process.

`SetupStdOutHandler(pid, callback)`

pid	Number	The process ID of the extension, as returned by CreateProcess().
callback	Function	The handler function for the standard-output callback.

RETURNS: An object with these properties:

- `err`: The status of the operation, one of:
  `NO_ERROR`
  `ERR_UNKNOWN`
  `ERR_INVALID_PARAMS`
  `ERR_INVALID_PROCESS_ID`

Terminate()

Terminates an extension process.

`Terminate(pid)`

pid	Number	The process ID of the extension, as returned by CreateProcess().

RETURNS: An object with these properties:

- `err`: The status of the operation, one of:
  - `NO_ERROR`
  - `ERR_UNKNOWN`
  - `ERR_INVALID_PARAMS`
  - `ERR_INVALID_PROCESS_ID`

UpdatePanelFlyoutMenu()

Adds or modfiies a menu item in the flyout menu for the extension panel. Menu items can contain sub-items.

`UpdatePanelFlyoutMenu(menuItemLabel, enabled, checked)`

menuItemLabel	String	An XML string that defines a new menu item. See example for SetPanelFlyoutMenu()
enabled	Boolean	True for the new menu item to be enabled, false for it to be disabled (grayed out).
checked	Boolean	True for the new menu item to be displayed with a platform-specific selection indicator.

WaitFor()

Waits for an extension process to quit.

`WaitFor(pid)`

pid	Number	The process ID of the extension, as returned by CreateProcess().

RETURNS: An object with these properties:

- `err`: The status of the operation, one of:
  - `NO_ERROR`
  - `ERR_UNKNOWN`
  - `ERR_INVALID_PARAMS`
  - `ERR_INVALID_PROCESS_ID`

WriteStdIn()

Writes data to the standard input of an extension process.

`SetupStdOutHandler(pid, callback)`

pid	Number	The process ID of the extension, as returned by CreateProcess().
data	String	The data to write.

RETURNS: An object with these properties:

- `err`: The status of the operation, one of:
  - `NO_ERROR`
  - `ERR_UNKNOWN`

```
ERR_INVALID_PARAMS
ERR_INVALID_PROCESS_ID
```

File I/O functions

These file I/O functions are defined as covers for the native-code versions. The functions are presented here in alphabetical order.

NOTE: *Currently, all native file I/O functions are synchronous; aynchronous file I/O is planned.*

DeleteFileOrDirectory()	Deletes a file or folder.
IsDirectory()	Reports whether an item in the file sytem is a file or folder.
MakeDir()	Creates a new folder.
OpenURLInDefaultBrowser()	Opens a page in the default system browser.
ReadDir()	Reads the contents of a folder.
ReadFile()	Reads the entire contents of a file.
Rename()	Renames a file or folder.
SetPosixPermissions()	Sets permissions for a file or folder.
ShowOpenDialog()	Displays the platform-specific File Open dialog, allowing the user to select files or folders.
WriteFile()	Writes data to a file, replacing the file if it already exists.

DeleteFileOrDirectory()

Deletes a file or folder.

`DeleteFileOrDirectory(`*`path`*`)`

path	String	The path to the file or folder.

RETURNS: An object with these properties:

- `err`: The status of the operation, one of:
  ```
  NO_ERROR
  ERR_UNKNOWN
  ERR_INVALID_PARAMS
  ERR_NOT_FOUND
  ERR_NOT_FILE
  ```

IsDirectory()

Reports whether an item in the file system is a file or folder.

`IsDirectory(path)`

path	String	The path to the file or folder.

RETURNS: An object with these properties:

- `data`: An object with these properties:
  - `isFile`: (Boolean) True if the item is a file.
  - `isDirectory`: (Boolean) True if the item is a folder.
  - `mtime`: (DateTime) The modification timestamp of the item.
- `err`: The status of the operation, one of:
  `NO_ERROR`
  `ERR_UNKNOWN`
  `ERR_INVALID_PARAMS`
  `ERR_NOT_FOUND`

MakeDir()

Creates a new folder.

`MakeDir(path)`

path	String	The path to the new folder.

RETURNS: An object with these properties:

- `err`: The status of the operation, one of:
  `NO_ERROR`
  `ERR_UNKNOWN`
  `ERR_INVALID_PARAMS`

OpenURLInDefaultBrowser()

Opens a page in the default system browser.

`OpenURLInDefaultBrowser(url)`

url	String	The URL of the page to open.

RETURNS: An object with these properties:

- `err`: The status of the operation, one of:
  `NO_ERROR`
  `ERR_UNKNOWN`
  `ERR_INVALID_PARAMS`

ReadDir()

Reads the contents of a folder.

`ReadDir(path)`

path	String	The path to the folder.

RETURNS: An object with these properties:

- `data`: An array of the names of the contained files (excluding `"."` and `".."`).
- `err`: The status of the operation, one of:
  `NO_ERROR`
  `ERR_UNKNOWN`
  `ERR_INVALID_PARAMS`
  `ERR_NOT_FOUND`
  `ERR_CANT_READ`

ReadFile()

Reads the entire contents of a file.

`ReadFile(path, encoding)`

path	String	The path to the file.
encoding	String	Optional. The encoding of the contents of the file, one of: `UTF8` (default) `Base64`

RETURNS: An object with these properties:

- `data`: The file contents.
- `err`: The status of the operation, one of:
  `NO_ERROR`
  `ERR_UNKNOWN`
  `ERR_INVALID_PARAMS`
  `ERR_NOT_FOUND`
  `ERR_CANT_READ`
  `ERR_UNSUPPORTED_ENCODING`

Encoding conversion

These utility functions are provided for conversion of encoding types:

`utf8_to_b64` (`str`)
`b64_to_utf8` (*`base64str`*)
`binary_to_b64` (*`binary`*)
`b64_to_binary` (*`base64str`*)
`ascii_to_b64` (*`ascii`*)
`b64_to_ascii` (*`base64str`*)

Rename()

Renames a file or folder. If a file or folder with the new name already exists, reports an error and does not perform the rename operation.

`Rensme(`*`oldPath, newPath`*`)`

oldPath	String	The original path to the file or folder.
newPath	String	The new path to the file or folder.

RETURNS: An object with these properties:

- `err`: The status of the operation, one of:
  ```
  NO_ERROR
  ERR_UNKNOWN
  ERR_INVALID_PARAMS
  ERR_NOT_FOUND
  ERR_FILE_EXISTS
  ```

SetPosixPermissions()

Sets permissions for a file or folder.

`SetPosixPermissions(`*`path, mode`*`)`

path	String	The path to the file.
mode	String	The new permissions, in numeric format. For example, `"0777"`.

RETURNS: An object with these properties:

- `err`: The status of the operation, one of:
  ```
  NO_ERROR
  ERR_UNKNOWN
  ERR_INVALID_PARAMS
  ERR_CANT_WRITE
  ```

ShowOpenDialog()

Displays the platform-specific File Open dialog, allowing the user to select files or folders.

`ShowOpenDialog(`*`allowMultipleSelection, chooseDirectory, title, initialPath, fileTypes`*`)`

allowMultipleSelection	Boolean	When true, multiple files/folders can be selected.
chooseDirectory	Boolean	When true, only folders can be selected. When false, only files can be selected.
title	String	Title of the Open dialog. Can be a ZString for localization.

initialPath	String	Initial path to display in the dialog. Pass `NULL` or `""` (the empty string) to display the last path chosen.
fileTypes	Array of String	The file extensions (without the dot) for the types of files that can be selected. Ignored when `chooseDirectory=true`.

RETURNS: An object with these properties:

- `data`: An array of the names of the selected files.
- `err`: The status of the operation, one of:
  `NO_ERROR`
  `ERR_INVALID_PARAMS`

WriteFile()

Writes data to a file, replacing the file if it already exists.

`WriteFile(`*path, data, encoding*`)`

path	String	The path to the file.
data	String	The data to write.
encoding	String	Optional. The encoding of the data, one of: `UTF8` (default) `Base64`

RETURNS: An object with these properties:

- `err`: The status of the operation, one of:
  `NO_ERROR`
  `ERR_UNKNOWN`
  `ERR_INVALID_PARAMS`
  `ERR_UNSUPPORTED_ENCODING`
  `ERR_CANT_WRITE`
  `ERR_OUT_OF_SPACE`