USING THE ADOBE EXTENSION SDK

Using the Adobe Extension SDK

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

Contents

1 Getting Started with the Adobe Extension SDK

The Adobe® Extension SDK is a set of libraries that make it possible to build Flash®-based extensions for Creative Suite® 5.x and higher and Creative Cloud™ desktop applications, and HTML/JavaScript extensions for CC applications. Developers can include these libraries in their projects in order to create cross-application plug-ins that use either of these architectures.

There are currently two programming models for extensions, one based on Flash®/Flex®/ActionScript® and a new one based on HTML/JavaScript.

- Flash-based extensions use the Adobe Flex framework and AIR® 2.0 API, and access the document object model (scripting DOM) of the target applications through ActionScript objects. The extension is delivered as Flash executable (SWF) that runs in an embedded Flash Player in the host application.
- The new HTML/JavaScript framework allows you to access the ExtendScript DOM of the host application directly. The extension is delivered as a set of HTML pages that run in an embedded browser in the host application.

For compatibility with previous versions, you can still use the Adobe Flex® framework and AIR® 2.7 API, and access the document object model (scripting DOM) of the target applications through ActionScript® objects. However, the Flash/Flex model is deprecated in this release, and it is recommended that existing extensions be migrated to HTML/JavaScript for continued support in Creative Cloud applications.

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, see the documentation for the previous release.
- For information on porting an existing extension to the new model, see "Migrating to HTML/JavaScript" on page 11.

About Adobe Application Extensions

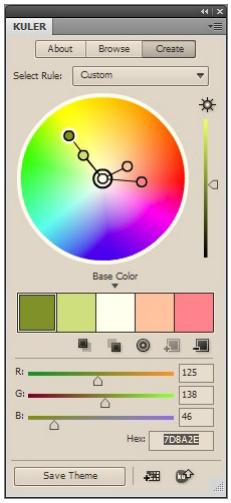
This section provides an overview of the Adobe application extensibility technology, which provides a common infrastructure for development and deployment of extensions that work across a set of supported Adobe desktop applications. An Adobe Application Extension is a set of files that together extend 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.



The Kuler panel, developed by Adobe and available in some products (CS5 and higher), is an example of a Adobe Application Extension.

Once available only as a web-hosted application for generating color themes, the Kuler extension makes the online Kuler service accessible within the suite products and allows users to access the color themes available in the web-hosted version. Kuler also integrates with the host application, allowing users to create themes that can be added, for example, to Photoshop® as a swatch.

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 Service 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

CEP 4.2 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 4.2 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.

Feature support

CEP 4.2 supports these features:

- Native file-handling through a JavaScript API
- External process management through a JavaScript API
- Complex graphics rendering, using Canvas and WebGL, and libraries that build on them
- Image manipulation using third-party JavaScript libraries such as `http://camanjs.com/`, `http://www.pixastic.com/lib/`, and many others
- Dialogs and message boxes; you can create a modal HTML extension and invoke it when needed, or use JavaScript `alert()` for debugging.
- HiDPI images (Retina display support in Mac OS)
- Video playback

Anatomy of an HTML/JavaScript extension

HTML5 extensions are packaged as ZXP files, in the same way as Flash-based extensions. Extension Manager CC supports the installation and management of all ZXP extensions. See Chapter 7, “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, and to send messages among extensions and applications. (These correspond to the former Flex CSXS, CEP IMS, and Vulcan libraries.) The JavaScript engine in CEP HTML engine had been extended to provide access to the local file system and to native processes; for complete details of the access functions, see . These extensions 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 5, “Creating a Manifest File."

File or Folder	Description
`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 7, "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 Adobe Service Manager root folder (`<ServiceMgr_root>`) depends on the version; for Creative Cloud, it is `CEPServiceManager4` .
- The exact location of the folder is platform-specific:
  - In Windows:
    (Win32) `C:\Program Files\Common Files\Adobe\<ServiceMgr_root>\extensions\`
    (Win64) `C:\Program Files (x86)\Common Files\Adobe\<ServiceMgr_root>\extensions\`
  - In Mac OS X: `/Library/Application Support/Adobe/<ServiceMgr_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

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, you must use version 4.2, implemented by these JavaScript libraries:

```
CSInterface.js
Vulcan.js
```

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.*

The JavaScript engine in the CEP HTML engine has been extended to provide access to the local file system and to native processes; for complete details of the access functions, see Chapter 10, "CEP Engine JavaScript Extension Reference."

Development environment requirements

The development environment for the Adobe Extension SDK is Extension Builder 3 Preview 3, which is available as a free download from http://labs.adobe.com/technologies/extensionbuilder3/.

HTML5/JavaScript extensions are supported only in the Creative Cloud release. To use the Adobe Extension SDK to create HTML5/JavaScript extensions, you must have:

- Adobe Extension Manager CC
- At least one of the Adobe Creative Cloud 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 Adobe Extension SDK extensions. The Creative Suite releases of these products support only Flash-based extensions; the Creative Cloud releases generally support HTML5/JavaScript extensions.

It is possible to build an extension that works in all of the Adobe desktop applications; for instance, one that connects to an Adobe LiveCycle server for workflow information.

Application	Host name
InCopy®	AICY
InDesign®	IDSN
Illustrator	ILST
Photoshop / Photoshop Extended	PHXS
Prelude®	PRLD
Premiere® Pro	PPRO
Dreamweaver® (Flash extensions only)	DRWV
Flash® Pro (HTML extensions only)	FLPR

Setting up the environment

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

Adobe Extension Builder 3 Preview 3 is an Eclipse-based tool that helps web developers create extensions for Adobe Creative Cloud Applications using modern HTML5. Extension Builder 3 Preview is free to download.

To begin using Adobe Extension Builder 3 to create an Adobe Extension SDK project, download it from:

http://labs.adobe.com/technologies/extensionbuilder3/

Follow these steps to get started with Extension Builder 3:

1. Download and install Eclipse 3.6 or later from: www.eclipse.org.
2. Download Extension Builder 3 Preview 3 from:

   http://labs.adobe.com/downloads/extensionbuilder3.html
3. Launch Eclipse and go to **Help > Install New Software**.
4. In the Add Site dialog, click **Archive** and browse to the Adobe Extension Builder 3 Preview ZIP file downloaded from the Adobe web site.
5. Click **OK** to confirm the changes.
6. Select all the Adobe Extension Builder 3 components by selecting the top item under Name. Click **Next** to confirm the selection.
7. In the Install Details review dialog, click **Next** to confirm that you want to install all the components listed.
8. Accept the terms of license agreements and click **Finish**.
9. Re-start Eclipse as recommended.
10. Follow the operation instructions in the Extension Builder 3 release notes:

    http://labsdownload.adobe.com/pub/labs/extensionbuilder3/extensionbuilder_releasenotes_080113.pdf

You are now ready to make your first Adobe Extension SDK project.

You can ask questions or share your feedback in the Extension Builder 3 forum.:

http://forums.adobe.com/community/labs/extensionbuilder3/

Note that your submission of comments, ideas, feature requests and techniques on this and other Adobe maintained forums, as well as Adobe's right to use such materials, is governed by the Adobe.com Terms of Use.

Migrating to HTML/JavaScript

Starting with CEP 4 for the Creative Cloud release cycle, you can and should use the HTML/JavaScript technology to develop extensions, rather than the previous Flash/ActionScript model. CEP 4.x supports both Flash and HTML5 extensions. An Adobe desktop application that has integrated CEP4 can run either kind of extension.

Flash/Flex/AIR extensions run in APE (Adobe Player for Embedded). This product is deprecated, with end-of-life scheduled for May 31st, 2014. This means that developers must migrate their extensions to HTML5 if they want to continue to support them in Creative Cloud (CC) applications. Flash-based extensions will continue to run as before in versions CS5.x and CS6 of their host applications.

CEP 4.2 does not support the NPAPI plug-in, which means that you cannot embed Flash or Java Applets in your extension.

ExtendScript continues to be supported.

- If your UI is written entirely in ExtendScript, it will not be affected by the deprecation of APE.
- If your ExtendScript code uses APE to run a SWF file, you will have to migrate it to an HTML/JavaScript UI that runs in an embedded browser.

There are some changes in how host application and operating system events are exchanged with extensions; for details of the current event-handling model, see Chapter 9, “CEP Events and Event Handling in JavaScript."

For details of how to access CEP functionality for managing extensions and interacting with the file system, see Chapter 10, “CEP Engine JavaScript Extension Reference."

2 Creating a Basic HTML/JS Extension

This HelloWorld sample demonstrates the basic structure and components of an HTML/JS extension by creating one with only a text editor and host application. Many of these tasks are automated when you use the tools in Adobe Extension Builder 3 Preview .

File structure

The first two files that we create for our extension, `index.html` and `manifest.xml`, must be arranged under a root folder whose name is the extension name. The manifest must be in a folder named `CSXS`:

```
HelloWorld/
    index.html
    CSXS/
        manifest.xml
```

- The manifest describes the contents and configuration of the extension; Adobe Extension Manager and host applications use this file to find and load extensions that are intended for particular hosts.
- The main page, `index.html`, defines the extension's UI and contains the initial script that defines its behavior. When your extension runs, this page is displayed in an embedded browser in the host application. Extensions are hosted in an embedded version of Chromium; that means that nearly everything you can do in Google Chrome, you can also do in an extension.

Creating the manifest

For now, just copy this XML in a text editor to create basic metadata for our extension. The elements are described in detail in Chapter 5, "Creating a Manifest File.'

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ExtensionManifest ExtensionBundleId="com.example.helloworld"
ExtensionBundleName="Hello world"
    ExtensionBundleVersion="1.0"
    Version="4.0">

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

    <ExecutionEnvironment>
        <HostList>
            <Host Name="PHXS" Version="[14.0,14.9]"/>
        </HostList>
        <LocaleList>
            <Locale Code="All"/>
        </LocaleList>
        <RequiredRuntimeList>
            <RequiredRuntime Name="CSXS" Version="4.0"/>
        </RequiredRuntimeList>
    </ExecutionEnvironment>

    <DispatchInfoList>
        <Extension Id="com.example.helloworld.extension">
```

```
            <DispatchInfo>
                <Resources>
                    <MainPath>./index.html</MainPath>
                </Resources>
                <UI>
                    <Type>Panel</Type>
                    <Menu>Hello world</Menu>
                    <Geometry>
                        <Size>
                            <Height>400</Height>
                            <Width>400</Width>
                        </Size>
                    </Geometry>
                </UI>
            </DispatchInfo>
        </Extension>
    </DispatchInfoList>
</ExtensionManifest>
```

This manifest describes an extension that is meant to run in Photoshop CC, where it is invoked with a menu item labeled "Hello world" in the **Window > Extensions** menu.

Creating the main page and script

In `index.html`, let's create a simple UI:

```
<!doctype html>
<html>
    <body>
        <button id="btn">Click me</button>
    </body>
</html>
```

This is actually enough to load and launch the extension in Photoshop, once the environment is prepared and the extension folder is copied into the host's `extensions` folder. Of course we will have to add some JavaScript code to give the button some behavior. We will do this by adding a code file, `main.js`, and referencing it from a script in the HTML page:

```
<!doctype html>
<html>
    <body>
        <button id="btn">Click me</button>
    </body>
    <script src='js/main.js'></script>
</html>
```

Create the code file in the *<ExtRoot>*`/js/` folder. For now, we will put just one function in it, to bring up the Chrome debugger. This function is defined by the CEP library that we included, and only works in extensions:

```
window.adobe_cep.showDevTools();
```

Now the extension's root folder looks like this:

```
HelloWorld/
    index.html
    CSXS/
        manifest.xml
    js/
        main.js
```

Preparing the environment

Applications normally cannot load an extension unless it is cryptographically signed. However, during development we want to be able to quickly test an extension without having to sign it. To run an extension that is in development and still unsigned, we have to set a debug flag in the operating system. You only have to do this once. For instructions, see "Setting the OS debug mode" on page 25

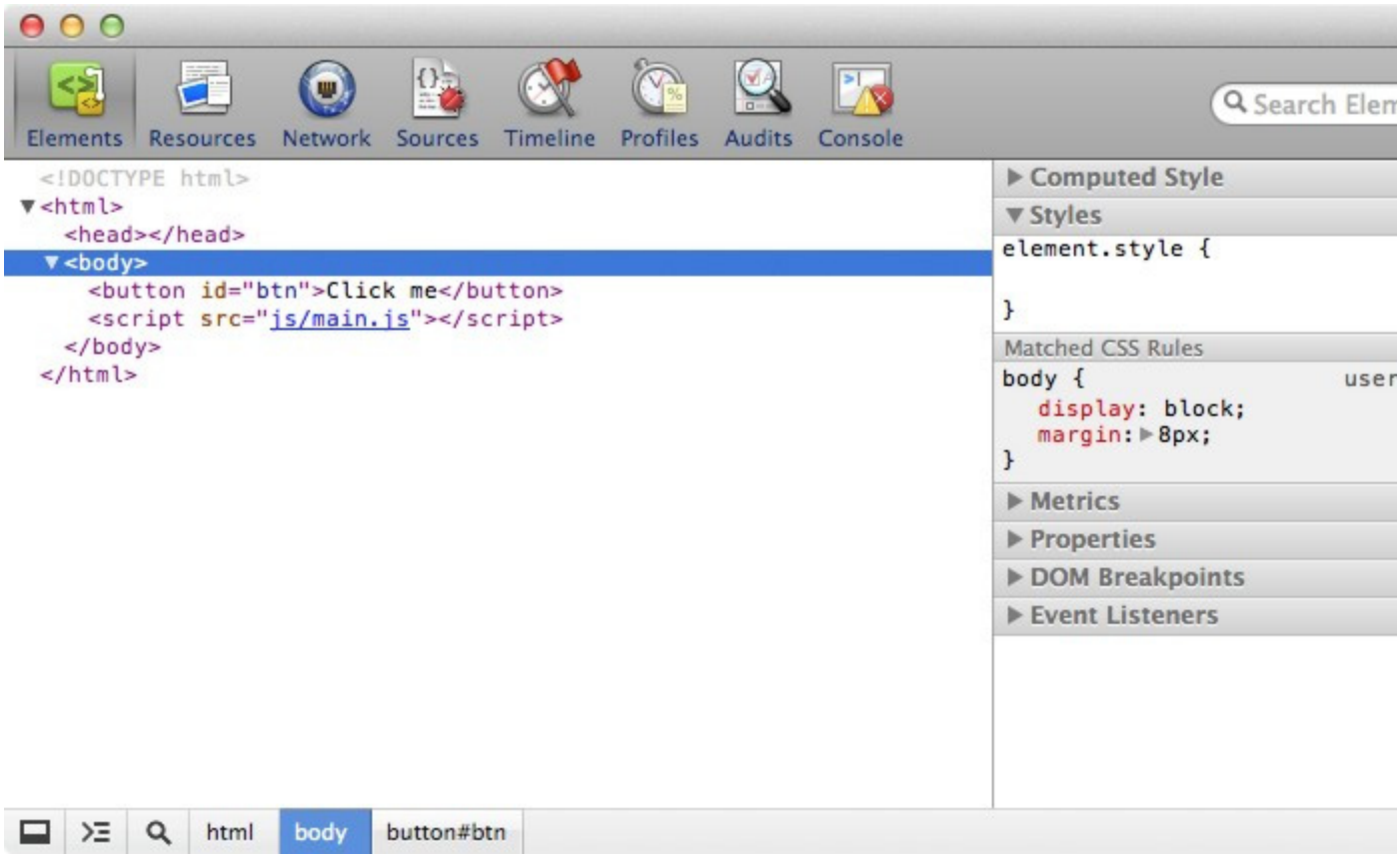
Once the debug flag is set, we just have to place the root folder in the shared `extensions/` folder. Any extension found in that folder is automatically loaded into supported host applications, and the configured menu item is added to the host's **Window > Extensions** menu.

- In Mac OS, copy the extension into:
  `~/Library/Application Support/Adobe/CEPServiceManager4/extensions`
- In Windows, copy the extension into:
  `%APPDATA%\Adobe\CEPServiceManager4\extensions`

Running the extension

Open Photoshop CC and choose **Window > Extensions > Hello World**. You should see a window like this:

When you click the button, the code you added to the script invokes the Chrome debugger.



Interacting with the host application

Typically, you will want your extension to interact with the host application. To accomplish this, you call methods defined in the CEP library `CSInterface.js`.

This library provides a number of useful functions, from retrieving the ID of the current host application to programmatically closing your extension. For this demo, we're going to use it to make calls against the host application's scripting DOM.

First, we need to add the library to our main page, `index.html` ,and copy it into the extension's `js/` folder:

```
<!doctype html>
<html>
    <body>
        <button id="btn">Click me</button>
    </body>
    <script src="js/CSInterface-4.0.0.js"></script>
    <script src='js/main.js'></script>
</html>
```

Next, we're going to add some ExtendScript that will create a new document using Photoshop's scripting DOM. Create a new file called `ps.jsx` with this code:

```
function addDocument() {
    app.documents.add();
}
```

Add this new file to a folder called `host` in your extension. Now the extension's root folder looks like this:

```
HelloWorld/
    index.html
    CSXS/
        manifest.xml
    js/
        CSInterface-4.0.0.js
        main.js
    host/
        ps.jsx
```

Finally, to make sure the new script is loaded along with the extension, we have to add a reference to it in the manifest. Edit `manifest.xml` to add this element:

```
    <Resources>
        <MainPath>./index.html</MainPath>
        <ScriptPath>./host/ps.jsx</ScriptPath>
    </Resources>
```

The last step is to modify `main.js` to call this function using `CSInterface` when the extension's button is clicked:

```
window.adobe_cep.showDevTools();

// Get a reference to a CSInterface object
var csInterface = new CSInterface();
var button = window.document.getElementById("btn");
button.onclick = function() {
    // Call function defined in host/ps.jsx
    csInterface.evalScript("addDocument()");
    };
```

Now, when we reload the extension and click the button, Photoshop creates a new document.

Packaging your extension

When you have finished developing your extension, you can use the command-line tool `ZXPSignCmd` to sign it and package it for deployment in the ZXP format. For complete details, see "Using the CC Extension Signing Toolkit" on page 35.

3 Using the New Project Wizard

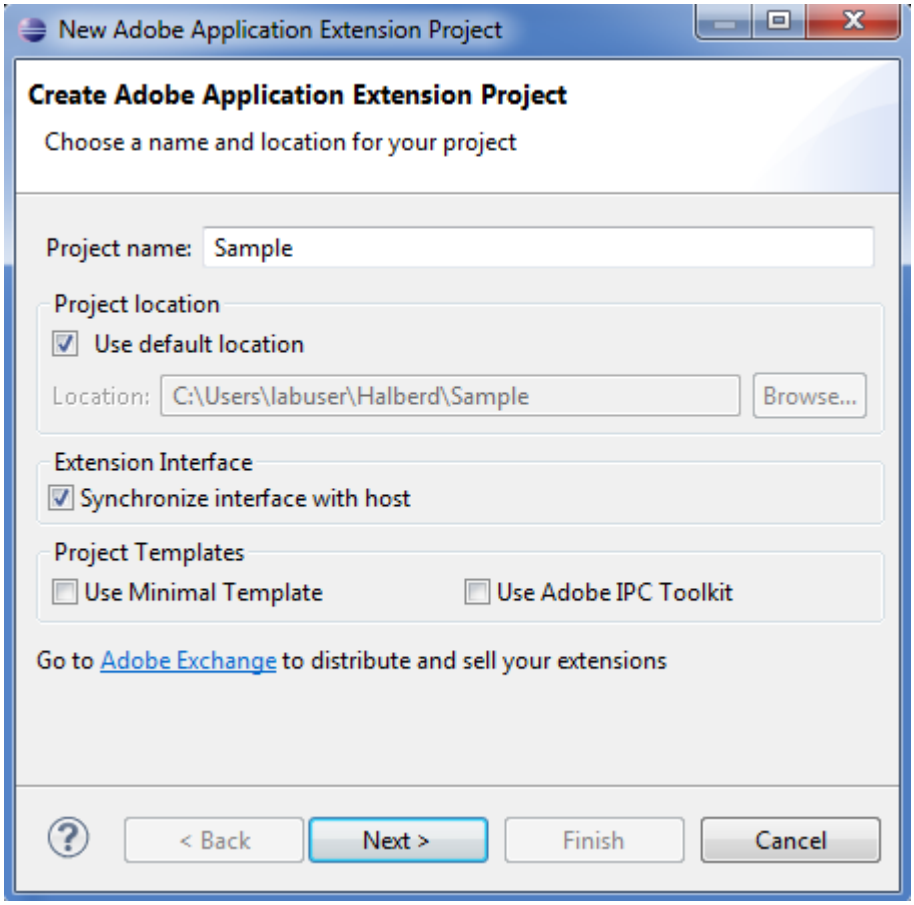
The New Project Wizard in Adobe Extension Builder 3 Preview helps you create a basic project, customizing settings as you go.

The wizard provides an easy way to customize features of the appearance (such as the panel size, or making the the UI theme of the extension match the UI theme chosen by the application user) and behavior (such as creating boiler-plate code for communicating with other Creative Cloud applications, or specifying JavaScript libraries to be included). You can also adjust these settings after the wizard creates your initial project.

This tutorial guides you through each wizard page.

Starting the wizard

To open the New Project Wizard in Adobe Extension Builder 3 Preview, choose **File > New > Project > Adobe Extension Builder 3 > Application Extension Project**. (If the current perspective is already "Adobe Extension Builder 3", you can just choose **File > New Application Extension Project**.)



On the first page, give your new project a name, and decide where the root folder should be. The default location is the current Eclipse workspace.

The link to Adobe Exchange opens the web marketplace where you can share your extension or offer it for sale.

Synchronize the panel UI with the host UI

"Synchronize interface with host" is selected by default. This means that the New Project Wizard generates boilerplate code in the new project to set the background color of panel, text size and font family, and color of button to be same with the host application. For a Photoshop extension, the color theme of the extension panel also synchronizes with the color theme the host application.

- If you want to customize other styles, you must write the code yourself.
- If you deselect synchronization, the extension panel has the default UI values, which include a white background.

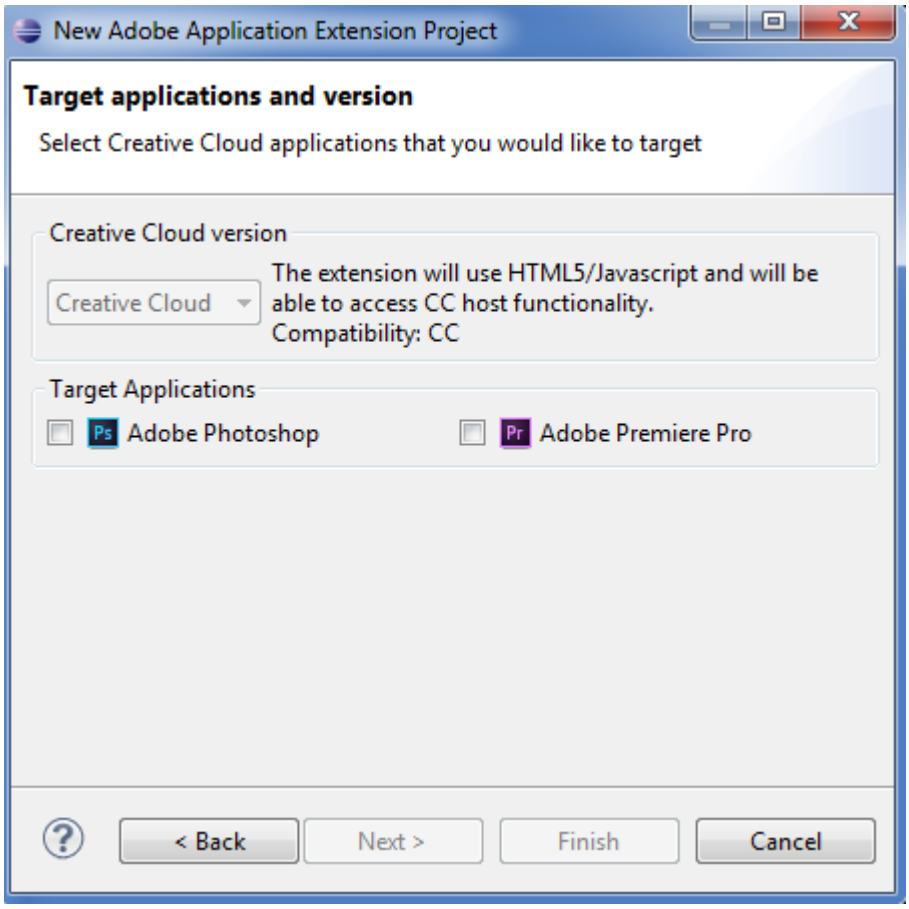
Templates

Templates generate extra code in your extension for common requirements. By default, no templates are selected, and the wizard generates buttons in the extension panel for each target host application, along with basic HTML. JavaScript, and CSS code, and a JSX (ExtensdScript) file in the `jsx` folder for each target host application.

- If you select **Use Minimal Template**, the wizard also generates code for a text input box and a button in the panel. The template JavaScript code causes the button to get the text that the user has entered and display it in a dialog box.
- If you select **Use Adobe IPC Toolkit**, the wizard generates JavaScript code that you can customize for inter-application communication, and includes a JavaScript file named `Vulcan-4.0.0.js` in the JavaScript file.

Click **Next** to select the target applications for your extension.

Selecting target applications



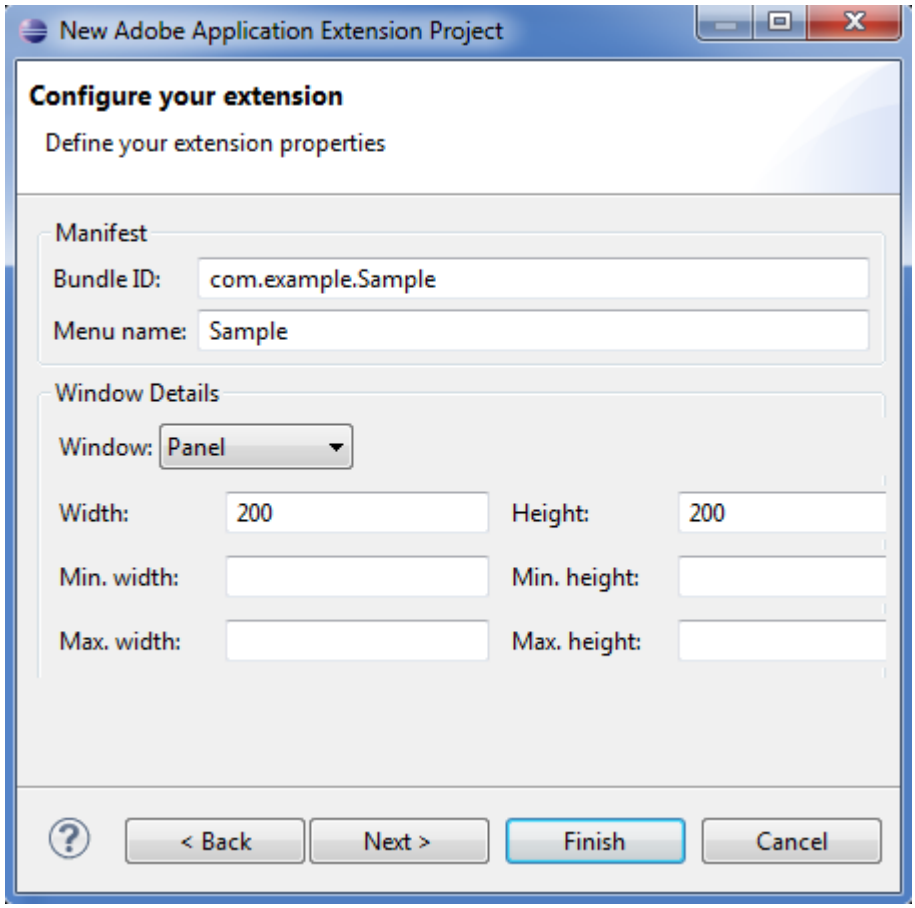
Your extension can only be installed in the target applications you select.

You must select at least one target application.

The link to Adobe Exchange opens the web marketplace where you can share your extension or offer it for sale.

Configuring extension properties

After you have selected a target applications, you have the option of clicking **Finish** immediately to create the new project, allowing the next two pages of setting options to default. If you click **Next**, you can configure additional extension properties:



The bundle ID must be unique to avoid conflict with other extensions. You can customized it with your company name.

You can customize the size and resizeability of the panel.

If minimum and maximum values are equal, the window cannot be resized in that dimension.

Menu name

You can enter a **Menu name** value to customize the label string for the menu item in the **Windows > Extensions** menu that invokes your extension. The default is the project name. If you leave **Menu name** blank, there is no menu item, but you can open your extension programmatically; for example:

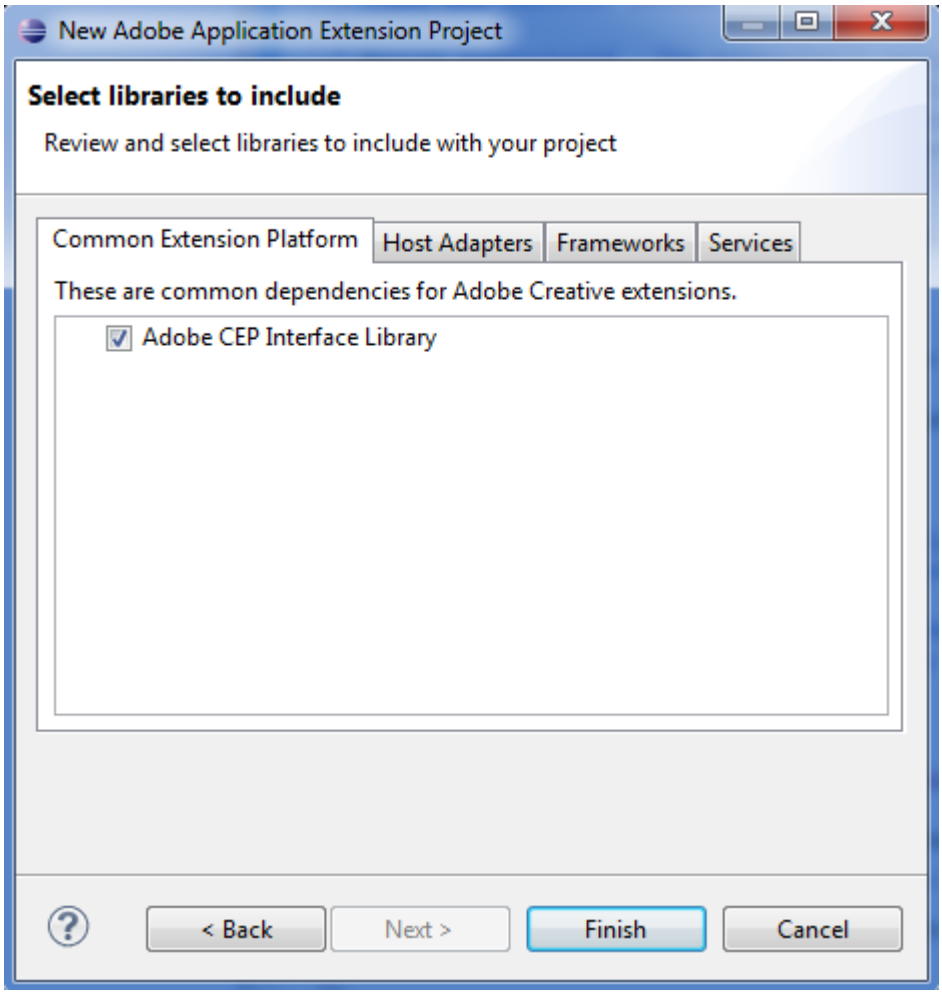
```
new CSInterface().requestOpenExtension(extensionId, "");
```

Window types

An extension runs in a separate window or dialog according to the Window type you choose:

- The Panel type (the default) is like a native panel or palette within the host application. It can be docked, participates in workspaces, has fly-out menus, and is re-opened at start-up if open at shutdown. Kuler is an example of the Panel type of extension.
- The ModalDialog type opens a new window and forces the user to interact with the window before returning control to the host application. Use for dialogs and message boxes.
- The Modeless type opens a new window, but allows the user to continue interacting with the host application.

Again, when you have finished this page you can click **Finish**, allowing values from the final page to default. Click **Next** to review and select the JavaScript libraries to include in your project.



These tabs list all of the JavaScript libraries that are provided with Extension Builder 3.

Select libraries to include them in your project. Included libraries are automatically placed in your project's `lib` folder.

Additional configuration options

You can configure libraries after the project is created; right-click the project and choose **Properties > Adobe Extension Builder > Libraries** to open a dialog that looks just like this page. Libraries are available in these categories:

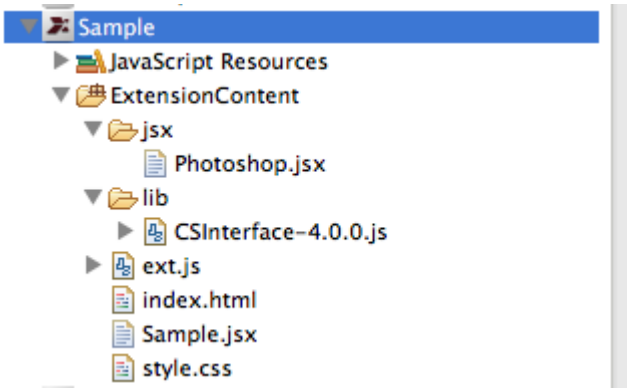
- Common Extensibility Platform (CEP)
  - **Adobe CEP Interface Library** is selected by default. You should always include it. This library enables an extension to communicate with the host application to get the information, evaluate the JavaScript code in application DOM and so on. '
- Host Adapters
  - **Creative Cloud Host Adapter Libraries** enable deeper integration with Creative Cloud applications is straight JavaScript. Use these to augment your C++ or ExtendScript code.
  - **Adobe CSHA** and **Adobe CSHA Photoshop** are the JavaScript host adapters for use with hybrid extensions. Adobe CSHA is for all host applications that have JavaScript APIs, and thus includes

Adobe CSHA Photoshop. (Photoshop is currently the only supported host application with a JavaScript API.)

- Frameworks
  - **JQuery 1.9.1** is a third--party JavaScript library. This is provided here as a convenience, but you are not restricted to libraries that are listed here. You can include any JavaScript libraries that are compatible with your platform.
- Services
  - **Adobe IPC Toolkit** is the library that is used to send and receive message to other Creative Cloud applications.

New project structure

The structure of the project created with a default configuration and with Photoshop selected as the target application in the New Project Wizard looks like this:




```
Sample/                          // project root
   ExtensionContent/             // project source code
      index.html                    // main page source
      Sample.jsx                    // main ExtendScript source
      style.css                     // CSS stylesheet
      jsx/                       // included ExtendScript libraries
         Photoshop.jsx              // host adapter for Photoshop
      lib/                       // included JavaScript libraries
         CSInterface-4.0.0.js       // CEP library
```

The root folder is named with the project name, and the folder `ExtensionContent` contains the source code for the project. When extension is opened in a host application, the content of the main page, `index.html`, is shown in an embedded browser.

`<ProjectName>``.jsx` is the main script file. The ExtendScript code in this file can access the ExtendScript DOM of the host application. All global functions and variables defined in the main script file " are availalbe to the extension at run time. Invoke a function that is defined in this file by calling the CEP `evalScript()` method:

```
CSInterface.evalScript("MyFn arg1 arg2");
```

Functions, methods, and variables that access application-specific functionality should be defined in the host-specific ExtendScript files in the `jsx/` folder. You can add your own JSX files here to organize your extension's logic.

To make functions and variables defined in a JSX file in the `jsx/` folder available to the DOM of the host application, you must load it in the host application, like this:

```
function loadJSX() {
    var csInterface = new CSInterface();
    var extensionRoot = csInterface.getSystemPath(SystemPath.EXTENSION) + "/jsx/";
    csInterface.evalScript('$._ext.evalFiles("' + extensionRoot + '")');
}
```

Define the functions and variables in the separate JSX file as properties of a special object "`$`", preferably in a host-specific namespace . For example, the generated file `jsx/Photoshop.jsx` defines functions like this:

```
$._ext_Photoshop = {}
```

4 Running and Debugging your Extension

Once you have created the project you can run the extension within your chosen host application. Before you run for the first time, however, you must let the operating system know that you are still in development, so that it won't expect your extension to be signed. You do this by setting a platform-specific debugging flag.

After you have set the debugging flag, you must copy your Output folder to the extension deployment folder. The host application automatically loads supported extensions on launch.

Setting the OS debug mode

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.

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.4\...`
3. Change value for key `PlayerDebugMode` to 1 to enable or to 0 to disable the debug mode.
4. Close the registry editor.

In Mac OS:

1. Navigate to the folder `<user>/Library/Preferences/...`
2. Find the PLIST file:
   `com.adobe.CSXS.4.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 or to 0 to disable the debug mode.
5. Save the file.

NOTE: 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**.

Loading the extension

To run and debug your extension in its target application, you must load it into the target's `extensions` folder (in the application install folder), or into one of the shared extension deployment folders.

Go to the Extension Builder workspace folder that contains your project, find your project's Output folder (the default name is `bin-debug`). Copy your Output folder to the deployment folder; the name and

location of this folder depends on the version you are targeting and your platform. (The name of the Adobe Service Manager root folder (`<ServiceMgr_root>`) is `CEPServiceManager4` for Creative Cloud.)

- These are the system-wide deployment folders for all users:
  - In Windows:
    `C:\Program Files\Common Files\Adobe\<ServiceMgr_root>\extensions\`
  - In Mac OS:
    `/Library/Application Support/Adobe/<ServiceMgr_root>/extensions/`
- For a specific user, these are the default locations of the deployment folder.
  - In Windows:
    `C:\<username>\AppData\Roaming\Adobe\<ServiceMgr_root>\extensions\`
  - In Mac OS:
    `~/Library/Application Support/Adobe/<ServiceMgr_root>/extensions/`

On launch, an application searches for extensions in this order: its own `extensions` folder first, then 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.

When you start the host application, your extension's menu (as defined in the manifest file) appears in the **Window > Extensions** menu.

5 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 8, ""), 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 10.
`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 8, "".

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 4.2

```
<RequiredRuntimeList>
   <RequiredRuntime Name="CSXS" Version="4.2" />
</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 10.

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 8, "." 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 48. 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 48.

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.

Communicating between components

You must choose the mechanism you want to use to communicate between the HTML and native API components of your hybrid extension. For example, you can create your own socket implementation to pass messages between your native plug-in and the JavaScript code of your extension.

Adobe offers the Native Application Toolkit that shows you how to use Adobe's PlugPlug library to communicate between C/C++ and JavaScript. The toolkit provides the libraries, documentation, and samples you need to build a hybrid extension where the components communicate with each other.

- You can include these libraries directly in Photoshop and InDesign native plug-ins.
- For information on using the PlugPlug libraries in Illustrator, see the FreeGrid sample in the Illustrator SDK.

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 4, "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 39.

7 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 37.

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 39.

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 37.	
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 35.
- 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 40.

   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 UCF 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 39. 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

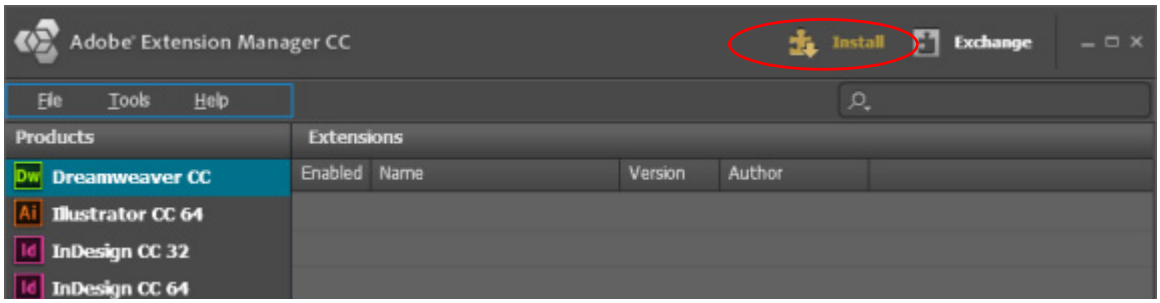
To install the signed ZXP file follow these steps:

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 37.
4. Once the installation has completed, check that your extension appears in all of the products that it supports.

Testing extension installation

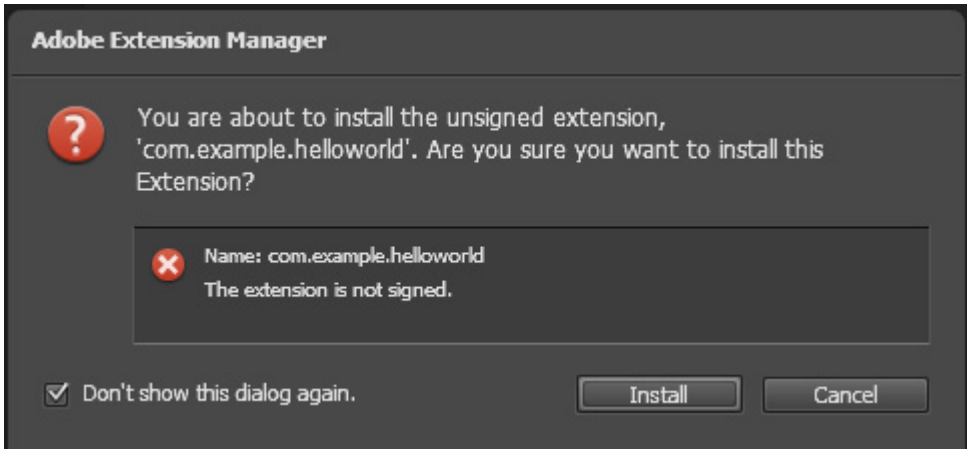
To test whether your package works properly, use Extension Manager to install your extension on your local versions of the Adobe desktop applications.

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.

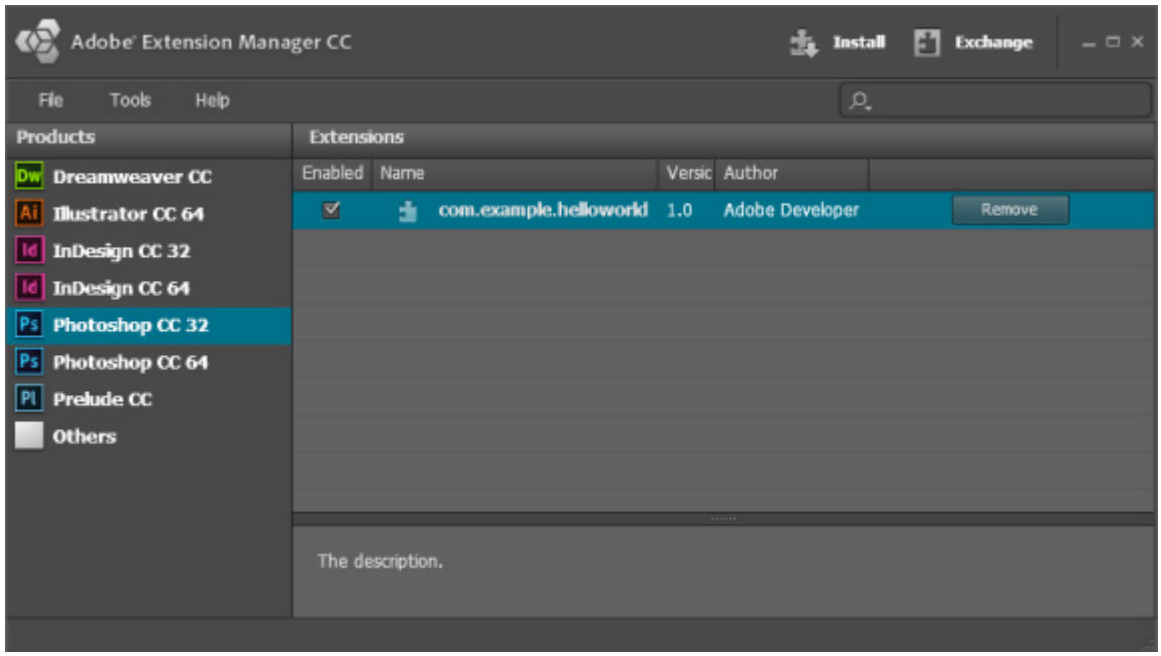


4. Once the installation has completed, check that your extension appears in all of the products that it supports.

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 "Loading the extension" on page 25.

If the problem persists, check the application's log for possible errors; see "Checking log files for errors" on page 44.

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.

Checking log files for errors

Several types of logs are available for help in debugging your Adobe extensions:

- "Application logs" on page 44
- "Adobe Service Manager logs" on page 45

Application logs

The Adobe extensibility infrastructure creates a log file for each of the applications running extensions. These files provide useful debug information for extension developers. The log files are generated in the platform's `temp` folder, and named according to the CEP/CSXS version and host application, `csxs4.2-HostID.log`; for example, `csxs4.2-ILST.log` for an extension running in Illustrator.

These logs are located at these platform-specific locations:

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

If you need more detailed information, you can increase the logging level. Possible log level values are:

"0": Off; no logs are generated
"1": Error; preferred and default level
"2": Warn
"3": Info
"4": Debug

"5": Trace
"6": All

Update the `LogLevel` key at these platform-specific locations:

- In Windows Registry Editor:
  `HKEY_CURRENT_USER/Software/Adobe/CSXS.4Preferences`
- In Mac OS X: PLIST file in `/Users/<user>/Library/Preferences/`
  `com.adobe.CSXS.4.plist`

You must restart your application for these changes to take effect.

Adobe Service Manager logs

To enable logging in Adobe Service Manager and the Communication Toolkit (Vulcan), edit the `StartupOptions.xml` file:

- Windows: `C:\Program Files\Common Files\Adobe\CEPServiceManager4\configuration`
- Mac OS: /`Library/Application Support/Adobe/CEPServiceManager4/configuration`

Add this element in the `<systemProperties>` element:

```
<vcentry key="loglevel">
    <vcint>6</vcint>
</vcentry>
```

The Service Manager keeps log files at these locations:

- In Windows: `C:\Users\<user>\AppData\Roaming\Adobe\<ServiceMgr_root>\logs`
- In Mac OS X: `/Users/<user>/Library/Application Support/Adobe/<ServiceMgr_root>/logs`

Remote debugging

CEP 4.2 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>
```

8 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 48.
- 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 48.

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>
```

9 CEP Events and Event Handling in JavaScript

CEP supports sending and receiving events within an extension, among extensions in the same host application, and among extensions in different applications. Both Flash and HTM extensions are based on a common communication layer with the same event data format. They can communicate with each other through the CEP event framework, and can also communicate with native code in the host application, as long as that application uses the *PlugPlugAddEventListener/PlugPlugDispatchEvent* architecture.

Function signatures

The CEP JavaScript event class, `CSEvent`, is the same as the ActionScript equivalent. The `CSInterface` class defines `dispatchEvent()` and `addEventListener()` methods that allow you to send events, and to set up and register event handler callbacks.

These are the function prototypes:

```
  /**
  * Class CSEvent.
  * Use to dispatch a standard CEP event
  *
  * @param type          Event type.
  * @param scope         The scope of event, "GLOBAL" or "APPLICATION"
  * @param appId         The unique ID of the application that generated the event
  * @param extensionId   The unique ID of the extension that generated the event
  *
  * @return CSEvent object
  */
function CSEvent(type, scope, appId, extensionId)

/**
  * Registers an interest in a CEP event of a particular type, and
  * assigns an event handler. The handler can be a named or anonymous
  * function, or a method defined in a passed object.
  *
  * The event infrastructure notifies your extension when events of this
  * type occur, passing the event object to the registered handler function.
  *
  * @param type      The name of the event type of interest.
  * @param listener  The JavaScript handler function or method.
  *                  Takes one argument, the Event object.
  * @param obj       Optional, the object containing the handler method,
  *                  if any. Default is null.
  */
CSInterface.prototype.addEventListener = function(type, listener, obj)

/**
  * Triggers a CEP event programmatically. Use to dispatch
  * an event of a predefined type, or of a type you have defined.
  *
  * @param event     A CSEvent object.
  */
CSInterface.prototype.dispatchEvent = function(event)
```

Using the event framework

If you are already familiar with CEP event handling in Flash extensions, it is practically the same in JavaScript. Here is a JavaScript code snippet that shows how you create an event type, define a handler for it, set up an event listener to invoke the handler, and dispatch the event.

The `CSInterface.addEventListener()` method supports both named and anonymous event-handler callback functions, as 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("com.adobe.cep.test", myEventHandler);

// Register an anonymous event handler
// (the second argument is the callback function definition)
csInterface.addEventListener("com.adobe.cep.test",
         function (event) {
             console.log("type=" + event.type + ", data=" + event.data);}
  )
```

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("com.adobe.cep.test", 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("com.adobe.cep.test", 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("com.adobe.cep.test", "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 = "com.adobe.cep.test";
  event.scope = "APPLICATION";
  event.data = "This is a test!";

  csInterface.dispatchEvent(event);

// Send an object as event data
  var event = new CSEvent("com.adobe.cep.test", "APPLICATION");
  var obj = new Object();
  obj.a = "a";
  obj.b = "b";
  event.data = obj;
  cSInterface.dispatchEvent(event);
```

Communication between Flash and HTML extensions

If the event data is a string, you can use exactly the same event handling model between Flash and HTML extensions as when both extensions are HTML-based.

However, if you dispatch an event from an HTML extension where the data is a JavaScript object, and want handle it in a Flash extension, you will have to convert the object. A JavaScript object is serialized as JSON string for transport. You must de-serialize it to create an XML-based object in your ActionScript event handler in the Flash extension.

Similarly, if you dispatch an event containing an ActionScript object from a Flash extension and want to handle it in JavaScript code, you must de-serialize it from XML to create a JSON-based object.

Communication between native host API and HTML extensions

For event passing from the native host API to the JavaScript code of an HTML extension, use the PlugPlug C++ event methods: `PlugPlugAddEventListener()` and `PlugPlugDispatchEvent()`.

Unlike Flash extensions, HTML extensions do not support window state-change events.

Event type support

These event types are defined and supported by Creative Cloud desktop applications. Currently, only the `Application` scope is supported for events.

Event type	Sent after	Supported in hosts
`documentAfterActivate`	Document has been activated (new document created, existing document opened, or open document got focus)	Photoshop 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	Mac OS only: Premiere Pro Prelude Windows and Mac OS: Photoshop InDesign/InCopy Illustrator

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 10.
`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.

10 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
```

This document provides detailed reference information for functions defined in this file. These functions are in three categories:

- "Extension control functions" on page 54
- "File I/O functions" on page 58

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.
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.
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

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
  ```

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`