USING THE ADOBE CREATIVE SUITE 6 SDK

TECHNICAL NOTE

Adobe

Using the Adobe Creative Suite 6 SDK

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

Contents

Getting Started with the Adobe Creative Suite SDK

The Adobe® Creative Suite® SDK is a set of ActionScript® libraries that make it possible to build Creative Suite Flash®-based extensions in CS5 and higher. Developers can include these libraries in their projects in order to create cross-application plug-ins that use the Adobe Flex® framework and AIR® 2.0 API, and access the document object model (scripting DOM) of Creative Suite applications through ActionScript objects.

About Creative Suite extensions

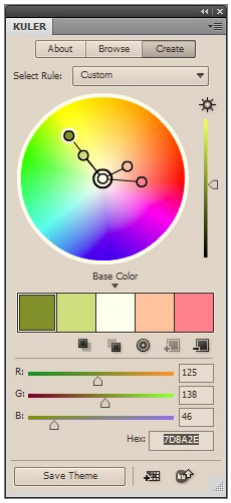
This section provides an overview of the Adobe Creative Suite extensibility technology, which provides a common infrastructure for development and deployment of extensions that work across a set of supported Adobe Creative Suite applications. An Adobe Creative Suite extension is a set of files that together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

The Adobe Creative Suite SDK provides developers with a consistent platform in which to develop and deploy extensions across the suite. Adobe Creative Suite extensions run in much the same way in all Adobe Creative Suite products (CS5 and higher), providing users with a rich and uniform experience.

Adobe Creative Suite extensions use ActionScript to create cross-platform user interfaces. Extensions also have access to the host application's scripting interface, and can use these scripting APIs to interact with the application.

Tight integration with the suite products 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 Creative Suite applications to their needs.

The Kuler panel, developed by Adobe and available in some products (CS5 and higher), is an example of a Adobe Creative Suite 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 Creative Suite extensibility architecture

The Adobe Creative Suite 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 Creative Suite 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 CS applications are made aware of the extensions (services or extended features) available to them by the CS 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.

The Service Manager communicates with Adobe Extension Manager to provide new content or updates to existing extensions. Once installed or updated, extension files are saved in a common location in the file system. CS applications can load extensions from this common location.

Anatomy of an extension

A deployed Adobe Creative Suite extension has these components:

File or Folder	Description
`MyExtension.swf`	The Flash file that provides the interface to the extension. The SWF file is a compiled AIR or Flex application. It can embed the SDK ActionScript libraries that allow the extension to communicate with the host application and the extensibility infrastructure. See the "Using Adobe Creative Suite SDK" on page 9 for basic information on creating an extension project.
`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 "Creating a manifest file" on page 12.
`icon_*.png`	Optional icons used to represent the extension when docked. You can provide icons for different states (normal, rollover, or disabled). For CS6 targets, you can provide icons for different color 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

The CS Service Manager is a program that runs in the background whenever extensions are invoked by Adobe Creative Suite products. This service 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; see Configuring an Adobe Creative Suite Extension. Every time an extension is installed, uninstalled, or updated, the CS Service Manager reloads the extension's

manifest to reflect those changes. The next time a CS application is re-started, the CS Service Manager notifies that application of the changes.

Users can install your packaged and signed Adobe Creative Suite extension through the Extension Manager; see "Packaging and signing your extension for deployment" on page 23. The Extension Manager installs all extensions in a common location, the `extensions/` folder, that all the Creative Suite applications can access.

- The name of the CS Service Manager root folder (`<ServiceMgr_root>`) depends on the Creative Suite version: `CSxServiceManager`, where is the version such as 5, 5.5, or 6.
- The exact location of the folder is platform-specific:
  - In Windows: `C:\Program Files\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 Creative Suite SDK

The Creative Suite SDK comprises two major components:

- A set of application-specific DOM libraries known as the Creative Suite ActionScript Wrapper (CSAW) libraries. The CSAW library names all start with `csaw`; for example, `csaw_indesign.swc` is the CSAW library for Adobe InDesign®.
- Two common infrastructure support libraries:
  - Adobe Player for Embedding (APE), implemented by the file `apedelta.swc`. This file is needed when using any of the CSAW libraries.
  - Creative Suite Extensible Services (CSXS). This library 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.
    - For CS5.x targets, use version 2, implemented by the file `CSXSLibrary-2.0-sdk-3.4-public.swc`
    - For CS6 targets, use version 3, implemented by the file `CSXSLibrary-3.0-sdk-4.5-public.swc`

Development environment requirements

The development environment for Creative Suite SDK is Flash Builder™, which is available from http://www.adobe.com/products/flashbuilder/.

To use the Creative Suite SDK, you must have:

- Flash Builder 4 or 4.5, or Eclipse with the Flash Builder plug-in, or the standalone Flex SDK provided with this SDK.
- Adobe Extension Manager CS5 or higher.

- Adobe Creative Suite 5 or higher, or at least one of the applications that supports extensions.

Supported applications

The Creative Suite SDK works with most of the Creative Suite products. The following Creative Suite applications support Creative Suite SDK extensions. Most of these applications also support an application-specific scripting object model that allows direct manipulation of application objects. For those that do not yet have ActionScript wrapper libraries in the Creative Suite SDK, it is possible to access the ExtendScript or JavaScript scripting DOM directly through CSXS library.

It is possible to build an extension that works in all of the Creative Suite applications; for instance, one that connects to an Adobe LiveCycle server for workflow information. (Note however, that Adobe Bridge CS6 does not support extensions.)

Application	Host name	CS5 Version	CS5.5 Version	CS6 Version	ActionScript wrappers
Adobe Bridge	`BRDG`	4	4.1	—	Yes (CS5.x only)
Dreamweaver®	`DRWV`	11	11.5	12	No
Fireworks®	`FWKS`	11	11.1	12	No
Flash Pro	`FLPR`	11	11.5	12	Yes (CS6 only)
InDesign®	`IDSN`	7	7.5	8	Yes
InCopy®	`AICY`	7	7.5	8	Yes
Illustrator®	`ILST`	15	15.1	16	Yes
Photoshop® /Photoshop Extended	`PHSP`	12	12.1	13	Yes
Adobe Premiere® Pro	`PPRO`	5	5.5	6	Yes (CS6 only)

Setting up the environment

To begin using Adobe Flash Builder to create a Creative Suite SDK project, you must install it and add the CS Flex SDK 3.4 to it to support targets in CS5 or CS5.5., or the CS Flex SDK 4.5 to leverage extensibility technology only available in CS6.

To add the CS Flex SDK 3.4 :

1. Locate the CS Flex SDK provided with the Creative Suite SDK:

   ```
   <CS_SDK_root>/CS Flex SDK 3.4.0/
   <CS_SDK_root>/CS Flex SDK 4.5.0/
   ```

   (If you use Flash Builder 4.6, you must replace the included SDK 4.6 with SDK 4.5.)

2. Download Adobe Flash Builder 4.5 or higher from http://www.adobe.com/products/flashbuilder/ and run the installer.

3. If you want to use the AIR 2.0 libraries, download http://labs.adobe.com/wiki/index.php/AIR_2:Release_Notes, and follow the instructions in "How to overlay the Adobe AIR SDK for use with the Flex SDK."

4. Launch Flash Builder.
5. In Flash Builder, choose **Window > Preferences**; in the Preferences dialog, select **Flash Builder > Installed Flex SDKs**.
6. Click **Add**, and navigate to the CS Flex SDK folder that is part of the Creative Suite SDK. Select the checkbox by the folder to make it the default.

You are now ready to make your first Creative Suite SDK project.

Using Adobe Creative Suite SDK

This section describes the general procedure for using Adobe Creative Suite SDK, and provides detailed walkthroughs and examples that illustrate the procedures in the context of sample code that is included with the SDK.

Add libraries to your Flex project

1. In Flash Builder, create a new Flex project that targets the Desktop by choosing **File > New > Flex Project**, and using the New Project wizard. In the wizard, choose "Desktop (runs in Adobe AIR)" as the application type.

   Once you have created a Flex project, you must add some or all of the SDK libraries to your project in order to use their APIs in your extension.

2. Decide which version or versions of the Creative Suite your project targets. If you plan to target multiple versions of the Creative Suite with a single project (for example, both CS5.x and CS6), it is recommended that you choose the CSAW libraries for the minimum target version. This helps ensure that you do not use any API features that are unavailable in earlier versions. For details of specific APIs, see the API reference documentation.

3. Decide which Flex version of the CSAW libraries to use.

   For targets in CS5.x or CS6, you can use versions compiled with Flex SDK 3.3, 3.4, or 3.5. The Flex 3.4 version is recommended, unless you have a compelling reason not to use it; for example, you have external libraries that were compiled with Flex 3.5 SDK.

   If your minimum version is CS6, compile against the default Flex 4.5.

4. Find the CSAW libraries associated with the chosen Creative Suite and Flex versions. The libraries are stored in this folder structure:

   `<CS_SDK_root>/libs/cslibs/<CSSDK version>/<Flex SDK version>/release/...`

5. Add the wrapper libraries you need. Add `csawlib.swc` if you want to use all the application wrappers, or the product-specific wrappers if you are targeting only one or two applications (for example `csaw_photoshop.swc`). To do this:

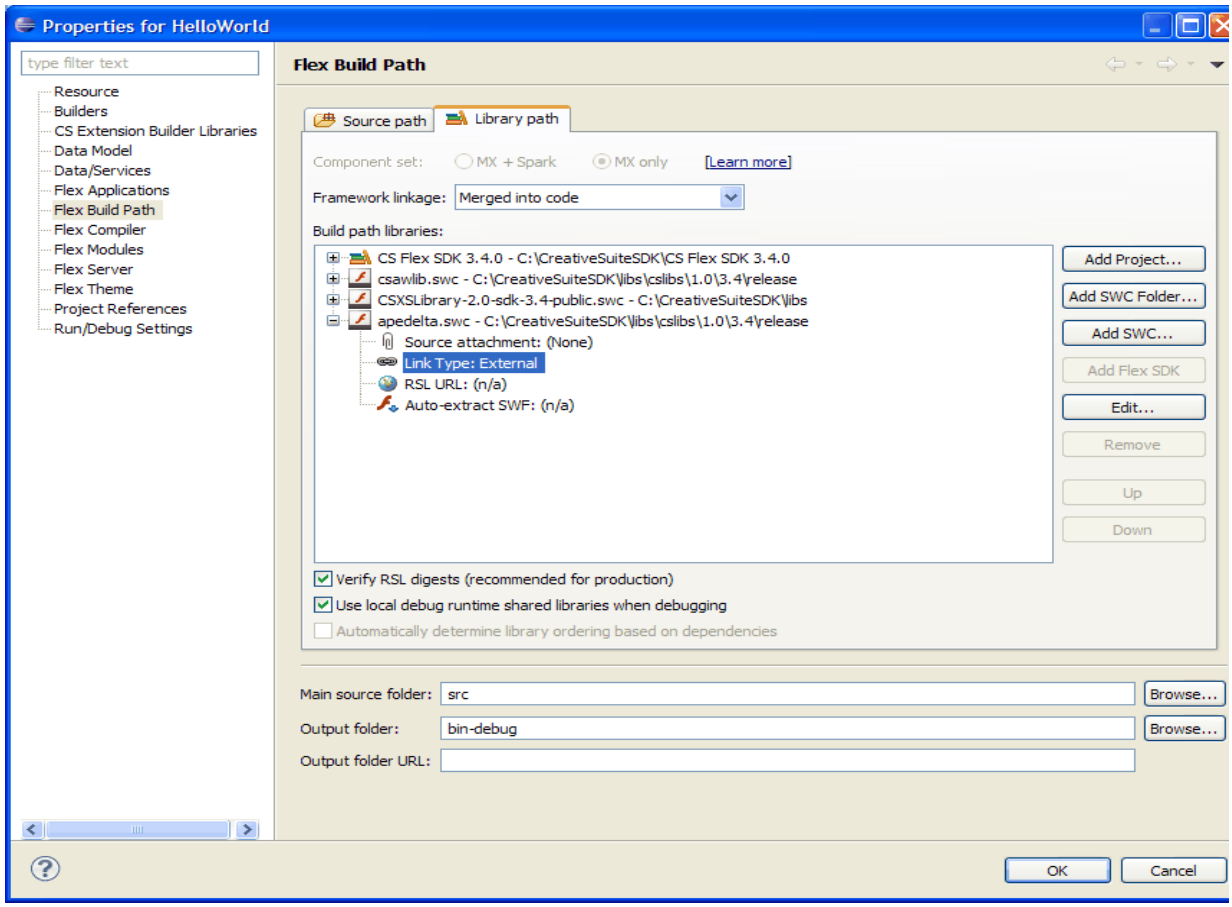
   ▷ Select your project in the Package Explorer and choose Properties from the context menu.

   ▷ In the Properties dialog, select Flex Build Path. Click **Add SWC**, specify the desired library or libraries, and click **OK**.

6. If you use any wrapper libraries, you must also include `apedelta.swc` in your project. To do this:

   ▷ Select your project in the Package Explorer and choose Properties from the context menu.

- In the Properties dialog, select Flex Build Path. Click **Add SWC**, specify `apedelta.swc`, and click **OK**.
- Open `apedelta.swc` in the Library path pane and ensure that the Link Type is External. (This library defines the Flash Player API; if it is compiled into your extension, you will get run-time errors.)



7. Include the CSXSLibrary if you plan to send events to other extensions, execute ExtendScript code, use `CSXSWindowedApplication`, `CSExtension`, or any other part of the CSXS API. For details, see the reference documentation for CSXSLibrary.

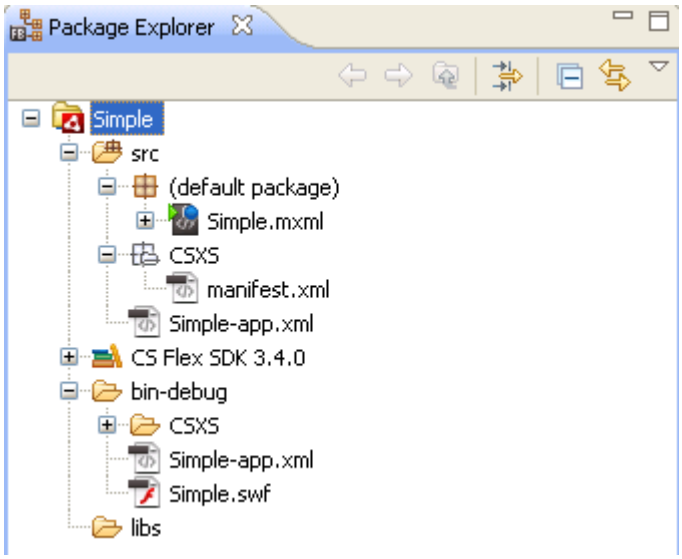
Create and add manifest file

A Creative Suite extension must have a *manifest file*, which tells the Extension Manager how to install that extension and specifies how it should be loaded and executed in the application. The extension manifest is an XML file.

The Creative Suite SDK includes the complete XSD file for the Extension Manifest schema and a sample manifest file. The section "Creating a manifest file" on page 12 describes some of the options and capabilities.

1. Create a manifest file based on the sample. The file must be named `manifest.xml`.
2. Place the manifest file in your project source folder, in a package named `CSXS`. The build process will automatically copy the manifest to the Output folder.

After you have created your manifest file, your project should look something like this in the Package Explorer:



Import a package

In the Flex project code, you must import the package from the Adobe Creative Suite SDK that contains the ActionScript wrappers for the scripting functionality you intend to use. Do this by adding an import statement to the top of the `<mx:Script>` tag in an MXML file (or `<fx:Script>` if you are using Flex 4.5). For example, using Flex 4.5:

```
<fx:Script>
<![CDATA[
import com.adobe.csawlib.indesign.InDesign;
    import com.adobe.indesign.Application;
]]>
</fx:Script>
```

Import the package that corresponds to the target application (InDesign in this example), in order to provide access to the scripting DOM of that application. For a complete listing and description of the available packages and their contents, see the API Reference documentation.

Once you have created the extension project, added the libraries, created and added the manifest, and imported the libraries you need to use, you are ready to write your extension, adding the functionality to the MXML file. For example, using Flex 4.5:

```
<?xml version="1.0" encoding="utf-8"?>
<csxs:CSExtension xmlns:fx="http://www.adobe.com/mxml/2009"
                  xmlns:s="library://ns.adobe.com/flex/spark"
                  xmlns:csxs="com.adobe.csxs.core.*">

    <fx:Script>
        <![CDATA[
            import com.adobe.csawlib.indesign.InDesign;
            import com.adobe.indesign.Application;
```

```
            public static function run():void
            {
                var app:com.adobe.indesign.Application = InDesign.app;
                app.documents.add();
            }

        ]]>

    </fx:Script>

    <s:VGroup height="100%" width="100%">
        <s:Button label="Run ID code" click="run()"/>
    </s:VGroup>
</csxs:CSExtension>
```

Creating a manifest file

Extensions created with the Adobe Creative Suite SDK require a manifest file. The manifest is an XML file that describes the extension, tells the CS Extension Manager how to install it, and gives the author control over extension-specific options such as the extension life cycle, UI, and menus. The complete schema for the XML, which you can use to validate the syntax, is included in the Creative Suite SDK installation:

```
<CS_SDK_root>/docs/ExtensionManifest-3.0.xsd
```

The typical simple extension is a single SWF; however, the Creative Suite SDK also supports bundling multiple SWFs 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 Creative Suite SDK.

ExtensionManifest

The root element for an extension manifest XML file:

```
<ExtensionManifest
    Version="3.0"
    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 "Localizing an extension" on page 18), 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 an SWF 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 CSXS 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 Creative Suite applications will run the extension under what conditions.

This element must list each of the Creative Suite host applications targeted by your extension, the supported locales, and the runtime requirements. In this example, an extension that targets InDesign CS6 requires CSXS 3:

```
<ExecutionEnvironment>

    <HostList>
        <Host Name="IDSN" Version="10" />
    </HostList>

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

    <RequiredRuntimeList>
        <RequiredRuntime Name="CSXS" Version="3.0" />
    </RequiredRuntimeList>

</ExecutionEnvironment>
```

HostList/Host

The `HostList` element contains a list of `Host` elements for all supported hosts. Each `Host` tag specifies a supported Creative Suite product.

Attributes The `Host` tag contains the following attributes:

`Name`	Required, the host name of the host application. See "Supported applications" on page 8.
`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

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 "Localizing an extension" on page 18.

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 both CS5.x and CS6 applications, use CSXS 2.0:

```
<RequiredRuntimeList>
   <RequiredRuntime Name="CSXS" Version="2.0" />
</RequiredRuntimeList>
```

For extensions that target only CS6 applications, use CSXS 3.0:

```
<RequiredRuntimeList>
   <RequiredRuntime Name="CSXS" Version="3.0" />
</RequiredRuntimeList>
```

DispatchInfoList/Extension/DispatchInfo

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

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

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

```
      <DispatchInfo >
         <Resources>
            <SwfPath>./Simple.swf</SwfPath>
         </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 8.

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:

`SwfPath`	Contains the path to the extension's SWF 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 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`: Fired when a document has been activated.
	▶ `documentAfterDeactivate`: Fired when the active document has been deactivated.
	▶ `applicationActivate`: Fired when the application gets an "activation" event from the OS.
	▶ `applicationBeforeQuit`: Fired when the application is about to shut down.
	▶ `documentAfterSave`: Fired 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`
	`ToolTip`

`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 "Localizing the extension's manifest file" on page 19. 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 19. 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 19.

Localizing an extension

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

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.

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>
```

Localizing the extension's Flex UI

You must make the localization resources available as part of initializing your extension's Flash component. To do this, call `initResourceBundle()` during the initialization:

```
CSXSInterface.getInstance().initResourceBundle();
```

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

In your Flex UI, use the `ResourceManager` object in your ActionScript code to directly access the resources in your messages.properties file. Use `resourceManager.getString()`, passing the name of the resource bundle, "messages", and the key of the property to retrieve:

```
resourceManager.getString('messages', 'myKey');
```

For more information on retrieving information from your resource file, see http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/mx/resources/IResourceManager.html.

Running and debugging your extension

Once you have created the project you can run the extension within your chosen host application (such as InDesign, Illustrator, or Photoshop). 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 CS Service Manager's deployment folder. The Service Manager automatically loads the extension into any application that supports it, when that application is launched.

To start debugging, you must first define a debug run configuration in Flash Builder. You can then debug the extension with Flash Builder while it is running in the application.

Setting the OS debug mode

To enable debugging mode in the application-embedded Flash Player, you must set an OS-specific flag. The location of this flag will change, depending on which version of the Creative Suite you are targeting for your extension. You need to do this in order to run your extension in the host application, even if you are not yet debugging it; the debug mode allows you to run your extension before it is packaged and signed for deployment.

In Windows:

1. Choose **Run** from the Windows Start menu, and enter `regedit` to open the registry editor.
2. Navigate to the key
   CS5: `HKEY_CURRENT_USER\Software\Adobe\CSXS2Preferences\...`
   CS5.5: `HKEY_CURRENT_USER\Software\Adobe\CSXS.2.5Preferences\...`
   CS6: `HKEY_CURRENT_USER\Software\Adobe\CSXS.3Preferences\...`
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:
   CS5: `com.adobe.CSXS2Preferences.plist`
   CS5.5: `com.adobe.CSXS.2.5.plist`
   CS6: `com.adobe.CSXS.3.plist`
3. Open this file with the XCode Property List editor, or with your preferred text editor.
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**.

Debug file for Photoshop and Dreamweaver

These applications require a `debug` file (in addition to `PlayerDebugMode`) to enable the debugger:

- ▷ Photoshop CS5, CS5.5, CS6
- ▷ Dreamweaver CS5.5, CS6

Create an empty file named `debug` (with no extension or contents) in the version-specific and platform-specific location:

- In Windows: Create the `debug` file in the same directory as your Photoshop or Dreamweaver executable. For example:

  ```
  C:\Program Files\Adobe Photoshop CS6\debug
  C:\Program Files\Adobe Dreamweaver CS6\debug
  ```

- In Mac OS: Create the debug file inside the package Contents folder for the application. For example, for Photoshop CS6 the path is:

  ```
  /Applications/Adobe Photoshop CS6/Adobe Photoshop CS6/Contents/debug
  ```

Loading the extension

To run and debug your extension in its target application, you must load it into the CS Service Manager.

Go to the Flash 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 is dependent on the version of the Creative Suite you are targeting and your platform:

- The name of the CS Service Manager root folder (`<ServiceMgr_root>`) depends on the Creative Suite version. It is `CSx`*`x`*`ServiceManager`, where *`x`* is the CS version (5, 5.5, or 6).
- For a specific user, these are the default locations of the deployment folder:
  - In Windows XP:
    `C:\Documents and Settings\<user>\Application Data\Adobe\<`*`ServiceMgr_root`*`>\extensions\`
  - In Windows 7/Vista:
    `C:\Users\<user>\AppData\Roaming\Adobe\<`*`ServiceMgr_root`*`>\extensions\`
  - In Mac OS:
    `/Users/<user>/Library/Application Support/Adobe/<`*`ServiceMgr_root`*`>/extensions/`
- 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/`

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

Creating a debug run configuration in Flash Builder

For each new project you want to debug with Flash Builder, you must define a debug run configuration. To do this:

1. Open your project in Flash Builder and select it in the Package Explorer.
2. In Flash Builder 4, choose **Run > Debug > Other**.
   In Flash Builder 4.5, choose **Run > Debug Configurations**.
3. Select Web Application and click **New** to create a configuration for Web Application.
4. Enter "Debug <*extension_name*>" in the Name box, and the name of your project in the Project box.

5. Deselect the "Use defaults" option.
6. Replace the value in the "Url or path to launch" box with `about:blank`.
7. Click **Apply** and close the dialog.

Debugging your extension in a host application

Once you have completed all of these prerequisites (setting the OS debug flag, loading the extension into the CS Service Manager, and setting up debug run configuration in Flash Builder), you are ready to start debugging.

To debug your extension with Flash Builder while it is running in the target application:

1. Open and select your project in Flash Builder.
2. Set a breakpoint in your ActionScript code.
3. To start a new debug session, choose **Run > Debug > *your_debug_config_name***.
   If you see a warning dialog, dismiss it and continue.
4. Launch the host application and run your extension by choosing its menu item from the **Window > Extensions** menu.

For information about how to use the debugger in Flash Builder, see the Flex SDK documentation: http://opensource.adobe.com/wiki/display/flexsdk/Developer+Documentation.

Creating a hybrid extension

A hybrid extension is a package that combines a Creative Suite 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 Flash-based 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 Creative Suite products, in addition to your Creative Suite Flash-based 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 Creative Suite 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 ActionScript.

Writing hybrid extensions

If you are already familiar with writing Creative Suite extensions and native application extensions (for example, a Photoshop or InDesign C++ extension, a Flash Pro 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 Creative Suite extension using Adobe Creative Suite 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 Flash-based and native API components or your hybrid extension. For example, you can create your own socket implementation to pass messages between your native plug-in and the ActionScript code of your Creative Suite extension.

Adobe offers the Native Application Toolkit that shows you how to use Adobe's PlugPlug library to communicate between C/C++ and ActionScript. 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, InDesign, and Flash Pro 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 Creative Suite SDK component as described in "Running and debugging your extension" on page 19.
- 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 26

Packaging and signing your extension for deployment

The Extension Manager package file 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.

An Extension Manager package 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 compiled Flex project in SWF format.
- 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 toolkit that you can use to package and sign extensions so they can be installed in Creative Suite applications using Extension Manager. The toolkit includes:

- The UCF tool, a command-line tool used to create Universal Container Format (UCF) packages
- A document that provides the information you need to create packages for deployment: *Technical Note: Packaging, Signing, and Deploying Extensions with Extension Manager*.

See the Adobe Creative Suite SDK page, http://www.adobe.com/devnet/creativesuite/, to download the signing and packaging toolkit.

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 SWF file is located at `<staging folder>/Simple.swf`, the path in the manifest should be specified as `./Simple.swf`.

For a hybrid extension, you must package and sign the Creative Suite 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 26.

Using UCF

To package the extension, use the Universal Container Format (UCF) command-line tool. The Adobe UCF tool is implemented in Java; the JAR file is packaged with this document. Running the tool requires that the `java` command is available in your shell's path. UCF requires JRE 1.5 or newer to run, but JRE 6 is recommended. This is the default in Mac OS X; in Windows, you must install JRE 1.5 or better.

Invoke UCF directly using the JAR file:

```
java –jar ucf.jar ...
```

All of the options and arguments to this command are described in the Tech Note. You must specify the signing option to produce a signed package that can run in a standard Creative Suite environment.

For example, suppose you want to package and sign an extension of any type that has been staged for packaging in a folder called `myExtension` in the current working directory. Use this shell command:

```
java -jar ucf.jar -package -storetype PKCS12 -keystore myCert.pfx -storepass mypasswd
   myExtension.zxp -C "./myExtension" .
```

This creates a package named `myExtension.zxp`, signed with the `myCert.pfx` certificate.

After packaging and signing the extension these two files are added to the final ZXP archive:

- `mimetype`

  A file with the ASCII name of `mimetype`, which 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. These are the possible validation results:

Signature	Signing certificate	Extension Manager action
No signature	N/A	Shows error dialog and aborts installation
Signature invalid	Any certificate	Shows error dialog and aborts installation
Signature valid	Adobe certificate	Silently installs extension
	OS-trusted certificate	Silently installs extension
	other certificate	Prompts user for permission to continue the installation

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 Creative Suite SDK portion separately, as described in "Creating the deployment package" on page 24.
- 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 Creative Suite 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 26.

   For example, for a hybrid extension that includes a Creative Suite 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 Creative Suite SDK extension component. See "Packaging a hybrid extension" on page 26. 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 theCreative Suite 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 Creative Suite 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>
```

PHOTOSHOP NOTE: If your hybrid extension project supports Photoshop as a host application, and will be installed using Extension Manager CS5, you must generate two different ZXP files, one for 32-bit support and one for 64-bit support. The addition of the Products attribute in the MXI file allows you to target a specific version of Photoshop with Extension Manager CS5.5, but with Extension Manager CS5 cannot interpret this attribute, and attempts to install both files in both versions. Make sure you add the bit attribute to the product element, and that the correct version of the plug-in is specified in the Bundle Manifest Editor before exporting the extension.

For example, for a Win32 platform, use:

```
<product maxversion="12.1" name="Photoshop" primary="true" version="12.0" bit=”32”/>
...
<file destination="$automate" platform="win"
          source="cs_resources/win32/MyPlugin.8li"/>
```

For a Win64 platform, use:

```
<product maxversion="12.1" name="Photoshop" primary="true" version="12.0" bit=”64”/>
...
<file destination="$automate" platform="win"
          source="cs_resources/win64/MyPlugin.8li"/>
```

Installing a packaged and signed extension

Adobe Extension Manager, a tool that is included with all Creative Suite 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

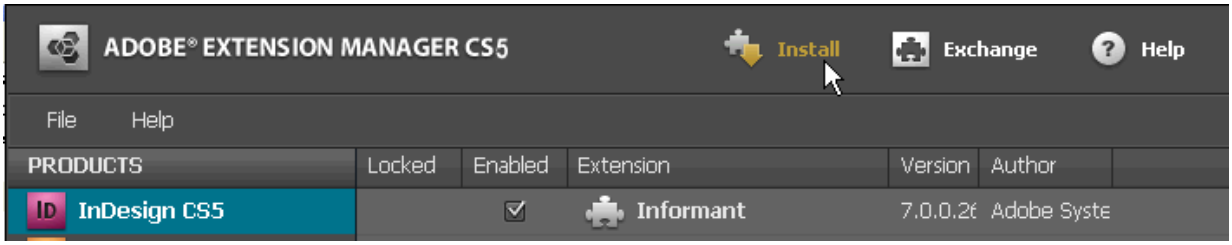
To install the signed ZXP file follow these steps:

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 25.
4. Once the installation has completed, check that your extension appears in all of the products that it supports.

Testing extension installation

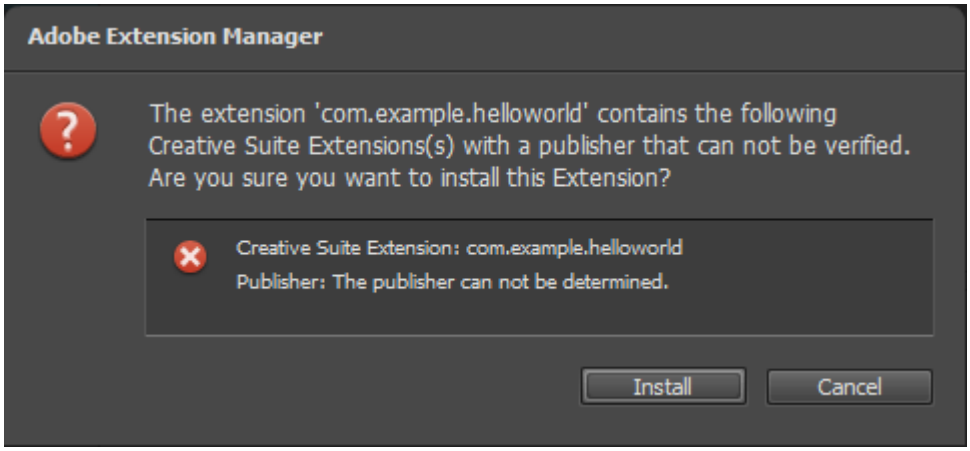
To test whether your package works properly, use Extension Manager to install your Creative Suite extension on your local versions of the Creative Suite applications.

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.

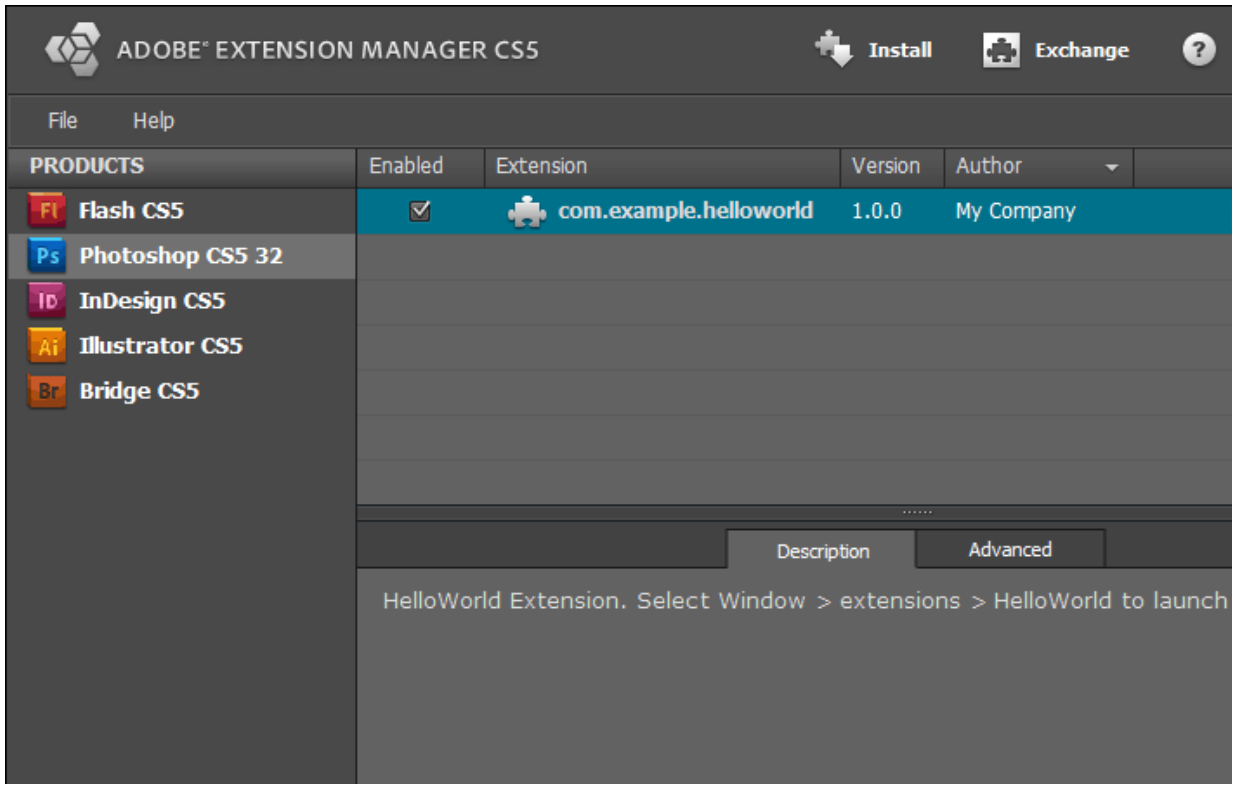


4. Once the installation has completed, check that your extension appears in all of the products that it supports.

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/SWFPath element. The path must be relative to the extension's root folder.
- Verify that the extension has been successfully copied to the CS Service Manager's extensions folder. For more details, refer to "Loading the extension" on page 21.

If the problem persists, check the application's log for possible errors; see "Checking log files for errors" on page 31.

Extension throws a security error upon loading

If your extension fails to load any of the ActionScript libraries provided with the SDK, it might throw a security error.

To prevent this, make sure your SWF file is compiled using the framework linkage "Merge into code" option, rather than the "Runtime shared library (RSL)" option.

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 Creative Suite extensions:

- "LogBook logs" on page 32
- "Flash Player's built-in logging " on page 32
- "Application logs" on page 33
- "CS Service Manager logs" on page 34

LogBook logs

You can use the LogBook application to track logging messages sent between the various platform components. To use this application for logging:

1. Include this code in your application main MXML file so LogBook can listen to and display the logging information traced by your extension:

```
var loggingTarget:LocalConnectionTarget = new LocalConnectionTarget("_test");
loggingTarget.filters=["*"];
loggingTarget.level = LogEventLevel.ALL;
loggingTarget.includeDate = true;
loggingTarget.includeTime = true; l
oggingTarget.includeCategory = true;
loggingTarget.includeLevel = true;

logger = Log.getLogger(this.className);

logger.info("my message");
```

2. Go to http://code.google.com/p/cimlogbook/downloads/list and download `LogBook-1.3.air`
3. Install the AIR application and start it.
4. Enter the local connection name, as passed into the constructor of `LocalConnectionTarget`. In the example, this is `'_test'`.
5. Start the host application and open your extension. You should see log messages in LogBook.
6. To listen to internal messages set the local connection name in logbook to `'_csxs2'`

Flash Player's built-in logging

The Flash Player runtime embedded in the Creative Suite application has some built-in logging functionality. It is very light-weight and easy to set up, but not as comfortable as LogBook. To use it, you must put trace() calls directly into your scripts, or create a TraceTarget instance. For example:

```
var traceTarget : TraceTarget = new TraceTarget();
traceTarget.filters = ["*"];
traceTarget.level = LogEventLevel.ALL;
traceTarget.includeDate = true;
traceTarget.includeTime = true;
traceTarget.includeCategory = true;
traceTarget.includeLevel = true;
```

To generate the `flashlog.txt` file:

1. Create a new file `mm.cfg` in the platform-specific folder:
   - ▷ In Windows XP: `C:\Documents and Settings\user\`
   - ▷ In Windows Vista and Windows 7: `C:\Users\`*`user`*`\`
   - ▷ In Mac OS: `/Library/Application Support/Macromedia/`
2. Edit the file and add these lines:

```
ErrorReportingEnable=1
TraceOutputFileEnable=1
```

The `flashlog.txt` file is then generated in the platform-specific folder:

- In Windows XP: `C:\Documents and Settings\user\Application Data\Macromedia\Flash Player\Logs`
- In Windows Vista and Windows 7: `C:\Users\user\AppData\Macromedia\Flash Player\Logs`
- In Mac OS: `/Users/user/Library/Preferences/Macromedia/Flash Player/Logs/`

Application logs

The Adobe Creative Suite 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 CSXS version and host application, `csxs2-HostID.log` or `csxs2.5-HostID.log`; for example, `csxs2-ILST.log` for an extension running in Illustrator CS5.

These logs are located at these platform-specific locations:

- In Windows XP: `C:\Documents and Settings\<user>\Local Settings\Temp`
- In Windows Vista: `C:\Users\<user>\Locale\Temp`
- In Windows 7: `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:
  - CS5: `HKEY_CURRENT_USER/Software/Adobe/CSXS2Preferences`
  - CS5.5: `HKEY_CURRENT_USER/Software/Adobe/CSXS.2.5Preferences`
  - CS6: `HKEY_CURRENT_USER/Software/Adobe/CSXS.3Preferences`
- In Mac OS X: PLIST file in `/Users/<user>/Library/Preferences/`
  - CS5: `com.adobe.CSXS2Preferences.plist`
  - CS5.5: `com.adobe.CSXS.2.5.plist`
  - CS6 `com.adobe.CSXS.3.plist`

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

CS Service Manager logs

The name of the CS Service Manager root folder (`<ServiceMgr_root>`) depends on the Creative Suite version. In CS6, the root folder is `CS6ServiceManager`.

The Service Manager keeps log files at these locations:

- In Windows XP: `C:\Documents and Settings\<user>\Application Data\Adobe\<ServiceMgr_root>\logs`
- In Windows Vista: `C:\Users\<user>\AppData\Roaming\Adobe\<ServiceMgr_root>\logs`
- In Mac OS X: `/Users/<user>/Library/Application Support/Adobe/<ServiceMgr_root>/logs`