CEP for the InDesign Developer

version 1.0.0

What is CEP?

Adobe's Common Extensibility Platform, or CEP, is a suite of technologies embedded in Adobe's Creative Cloud (CC) desktop applications. CEP was originally known as CSXS—it first appeared as part of Adobe Creative Suite 4 in 2008. CEP delivers a runtime environment that is integrated with the ExtendScript scripting support of the host application. While this document is primarily about CEP and InDesign, CEP is also implemented in other Creative Cloud applications, including Photoshop, Illustrator, Premiere Pro, and Dreamweaver.

You can think of CEP as a web browser that's built into InDesign and other CC applications—but it's a web browser with a connection to InDesign's scripting document object model (DOM). In specific, CEP is built around a custom implementation of Google's Chromium Embedded Framework (or CEF). CEF gives developers a way to include a version of Google's Chrome browser in an application, including support for Google's V8 JavaScript engine. In CEP, Adobe has created a custom version of CEF that runs inside Adobe's desktop applications, such as InDesign.

CEP connects to CC applications via interprocess communication with PlugPlug, an Adobe shared technology component. Through PlugPlug, CEP enables communication between the V8 JavaScript environment and the ExtendScript DOM in the host application.

CC extensions built on the CEP foundation typically include two main parts: a user interface built in HTML and JavaScript, and one or more ExtendScript files that manage the work of manipulating the InDesign DOM. The HTML/JavaScript side of an extension can connect to the web, just as if it were a normal, non-embedded browser, but it's not required that your CC extension do so. In fact, most of the work done in a CC extension for InDesign will probably take place on the ExtendScript side—creating documents, adding text, placing graphics files, and so on.

CEP also extends CEF by adding node.js, a popular open-source framework based on V8 JavaScript. Node.js provides APIs for file system access, web interaction (such as file upload/download), cryptography, events, and other functions. In addition, node.js is extensible, and thousands of Node Packaged Modules (NPM) provide features such as authentication, database connectivity, and zip packaging. For more on node.js, see <link to http://nodejs.org>; for more on NPM, see <link to https://www.npmjs.org/>.

In InDesign, support for scripting is built into plug-ins developed in C++ using APIs provided by the InDesign SDK. All plug-ins, whether developed by Adobe or by third parties, can add objects, methods, and properties to the InDesign DOM. The InDesign DOM is dynamic: if you add a plug-in, the scripting model implemented in that plug-in appears in the DOM; remove the plug-in, and the supported scripting features are no longer available.

Finally, CEP integrates the Adobe IPC Toolkit, an interprocess communications system for Adobe CC applications. Like CEP, the Adobe IPC Toolkit is enabled by PlugPlug. With the IPC Toolkit, you can, for example, control Photoshop or Illustrator from an InDesign extension.

This document provides a conceptual overview of how these parts work together to form the CEP platform for creating CC extensions, with a special emphasis on InDesign solutions.

About Flash-based Extensions

CEP supersedes Adobe's previous, Flash-based extensibility platform, which were known as Creative Suite Extensions, or CSXS. Support for Flash-based CS Extensions will be removed from CC applications in the near future, if it has not already been removed. At the time of this writing, December 2014, support for Flash-based extensions is still available in the most recent version of InDesign (InDesign CC 2014.1), but you should not rely on this support continuing.

If you need to support Adobe Creative Suite applications (for example, InDesign CS6) in addition to CC applications, you'll need to maintain two versions of your extension—one for CS applications; another for CC applications.

What Does CEP Offer the InDesign Developer?

CEP makes it easy to develop solutions for InDesign using the application's support for scripting. If your project does not require that you add entirely new features to InDesign (a new text composer, for example), but can be implemented purely using InDesign scripting to manipulate the DOM, it makes more sense to develop as a CC extension than it does to build a plug-in using the C++ SDK.

CC extensions have the following advantages over C++ plug-ins:

- With each new version of InDesign, plug-in developers must recompile their plug-ins with a new version of the SDK. CC extensions, by contrast, will continue working, or will require only very minimal changes to work with a new version.

- You do not need to restart InDesign each time you need to make a change to your CC extension, as you do with C++ plug-ins. This speeds up the develop/build/test process considerably.

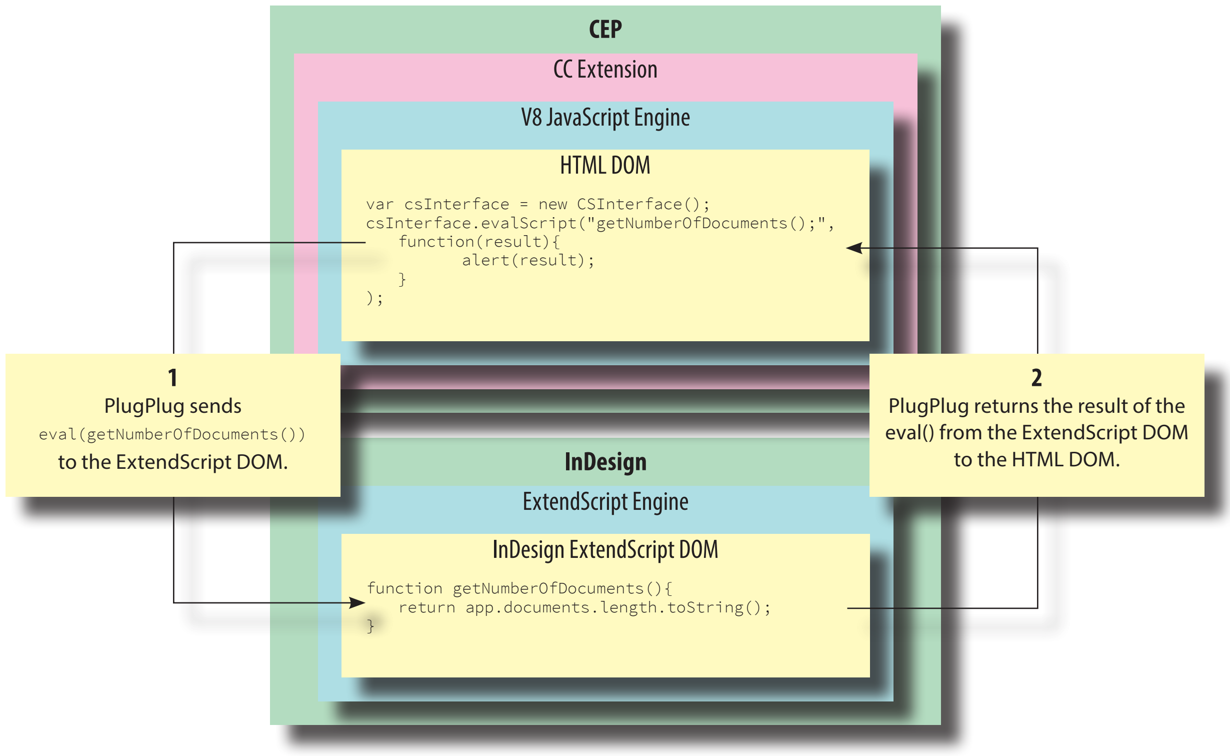
- CC extensions use standard web development technologies. This means that an enormous pool of resources is available—frameworks, add-ons, tutorials, and answers to questions. This also means that it's much easier to find help—there are simply a lot more web/HTML/JavaScript developers than there are InDesign C++ developers. In addition, the incorporation of node.js into CEP opens the door to literally thousands of NPM libraries.

- You can use lightweight, cross-platform web development tools, such as Brackets, to edit your code, rather than using separate, platform-specific IDEs (XCode on the Mac OS; Visual Studio in Windows).

That said, there's no reason not to "mix and match"—you can create a scriptable C++ plug-in and write the user interface for the InDesign ExtendScript DOM, which can then interact with a user interface written in HTML. For more on this approach, see "Sending Messages to InDesign Plug-ins," later in this document.

CEP Component Architecture

It’s important to understand the “tag team” approach used by CC extensions—that there’s an HTML/V8 JavaScript “side” and a host application/ExtendScript “side,” and that CS events (previously known as “CSXS events”), enabled by PlugPlug, are the glue that holds the two sides together. There are also two JavaScript engines involved, one for V8, and one for ExtendScript, and the two engines do not co-exist in a single, global, context, as shown in the following diagram.



```
var csInterface = new CSInterface();
csInterface.evalScript("getNumberOfDocuments();",
    function(result){
          alert(result);
    }
);
```

1
PlugPlug sends
`eval(getNumberOfDocuments())`
to the ExtendScript DOM.

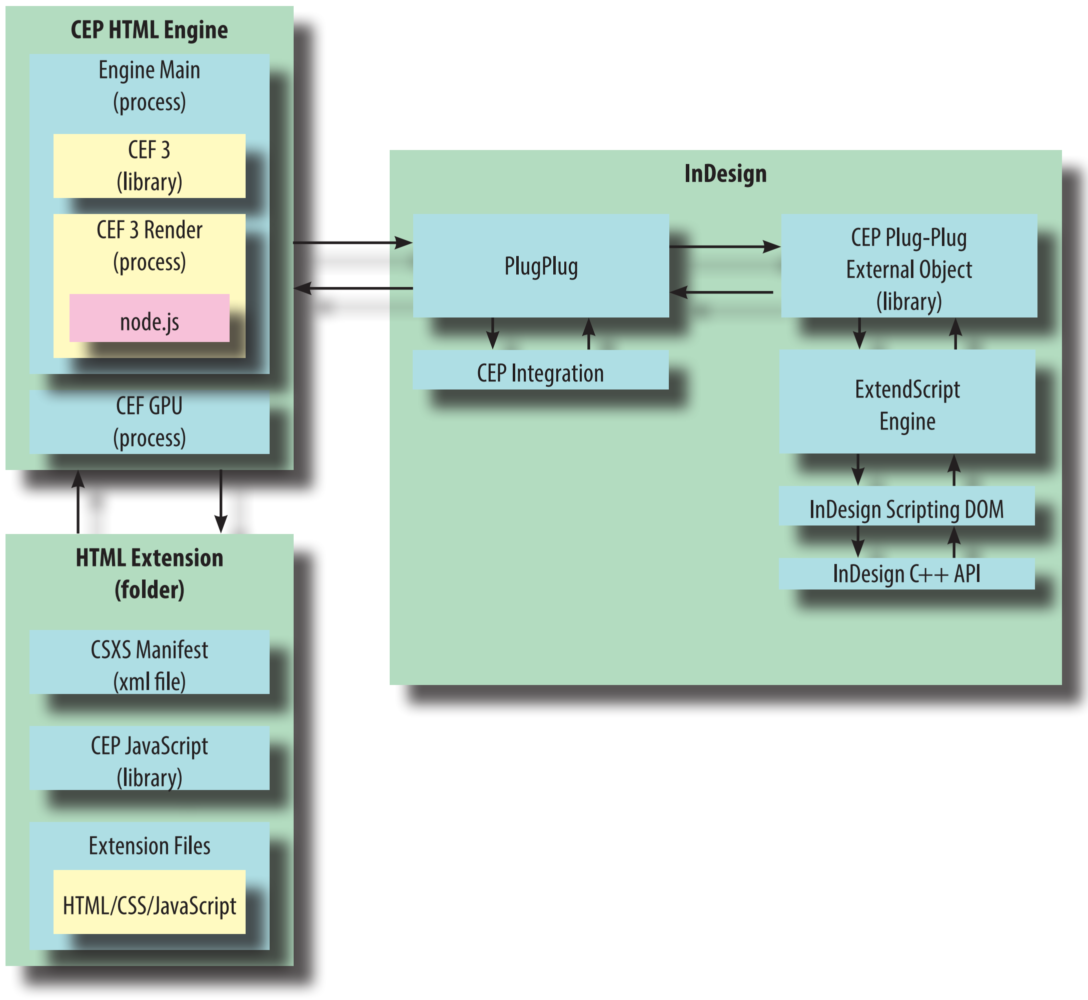
2
PlugPlug returns the result of the eval() from the ExtendScript DOM to the HTML DOM.

```
function getNumberOfDocuments(){
    return app.documents.length.toString();
}
```

The “tag team”: A CC Extension is made up of two JavaScript engines

The V8 JavaScript engine can communicate with the ExtendScript engine in two ways: using the evalScript() method of the CSInterface, and by sending CS events (again, via the CSInterface). The ExtendScript engine can communicate with the V8 engine by sending CS events via the PlugPlug External Object library.

Though CEP loads the HTML content of your CC extension inside a window in the host application, CEP is a separate process from the host application, and uses PlugPlug to communicate with the host application. The following diagram shows another view of the relationship between the parts that make up CEP.



Component Architecture Diagram

The following sections go into a bit more detail on the elements shown in the diagram.

PlugPlug

PlugPlug is a shared library component and is responsible for communication between the host application and other CEP components. PlugPlug controls the life cycle of the CEP HTML engine. PlugPlug is responsible for sending/receiving CSXS events between native APIs, HTML extensions, and Flash extensions within host application.

PlugPlug External Object Library

PlugPlug External Object is an ExtendScript library which provides the APIs that enable communication between the V8 JavaScript engine and the ExtendScript DOM of the host application.

Note that the values that you send from one JavaScript engine to another are strings. This means two things to CC extension developers:

- If you need to send an object from one JavaScript engine to another (V8 to ExtendScript, for example), you'll need to have some way to convert it to a string, and a way to convert the string back to an object. JSON is the best way to accomplish this. The V8 engine on the HTML side of your extension supports JSON; on the ExtendScript side, you'll need to include a JSON external library in your script. You can find the standard library, json2.js, at <link to https://github.com/douglascrockford/JSON-js>.

- You'll need to use escape() (as you send a string) and unescape() (on the receiving side) as you send a string from one JavaScript context to another.

Note: The InDesign CC 2014.1 release includes the PlugPlug External Object library as part of the application's ExtendScript DOM. To use the library with previous versions of InDesign CC, you'll need to include a copy of the library in your scripts. This library is available from <link to https://github.com/Adobe-CEP/CEP-Resources/releases/tag/1>.

CEP HTML Engine

The CEP HTML Engine is a standalone executable responsible for rendering HTML extensions and running V8 JavaScript code. It uses open source project CEF version 3 (<link to http://code.google.com/p/chromiumembedded/>). PlugPlug launches an HTML engine process when an extension is loaded, and closes it when the extension is unloaded. PlugPlug uses interprocess communication to communicate with the HTML engine. The HTML engine's main process is the CEF's browser process, and it may have one or more Render and GPU process.

For more information on the Render and GPU processes mentioned in the diagram, please refer to the links below:

http://www.chromium.org/developers/design-documents/multi-process-architecture

http://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome

CEP HTML Extensions

Each HTML extension is a folder, containing the manifest.xml file and one or more HTML, JavaScript, CSS files. Extension developers can call CEP JavaScript libraries (provided as .js files) from V8 JavaScript in the HTML engine, and the functions in these libraries provide a way to communicate with the ExtendScript DOM inside InDesign. You can think of an HTML extension as a web page, or a web site—albeit one that can connect to InDesign documents, pages, and page items.

Note: Unless you've enabled debugging on your system, all HTML extensions must be signed to run (InDesign will not load unsigned extensions). Refer to "Running and Debugging your Extension" for more on enabling debugging, and "Packaging and Signing your Extension for Deployment," both from *Using the Extension SDK* (also known as the file "Adobe Application SDK.pdf") for more on signing extensions.

CSInterface: The Key to CEP

The functions in the JavaScript library file “CSInterface.js” (sometimes seen with version-specific file names such as “CSInterface-5.2.js,” “CSInterface-5.0.js,” and others) provide an API for the V8 JavaScript engine of the extension to use to communicate with PlugPlug, and, via PlugPlug, with the ExtendScript engine.

The most important of these functions is CSInterface.evalScript(), which sends a string to the ExtendScript DOM and performs an eval() on the string in the ExtendScript engine corresponding to the extension. The ExtendScript engine for a given extension will be named using the following pattern:

extension_id + “_Engine_Id”

If, for example, your extension id (as specified in <Extension> element of the extension’s manifest.xml file) is “com.adobe.test”, then the ExtendScript engine name will be “com.adobe.test_Engine_Id”.

External ExtendScripts can use the #targetengine preprocessor directive to interact with the extension’s ExtendScript engine.

For more on CSInterface.evalScript() and the following functions, refer to *Using the Extension SDK* (also known as the file “Adobe Application SDK.pdf”).

CSInterface Environment Functions

Through CSInterface, CEP offers several methods for getting information about the host application and the operating system. Here’s a short summary:

getHostEnvironment(): This function returns information about the current application skin.

getOSInformation(): This function returns the current operating system.

getHostCapabilities(): This function returns an an object that specifies the support for various CEP functions provided by the host application. For InDesign CC 2014.1, for example, this function returns EXTENDED_PANEL_MENU = false, which means that true panel menus are not supported by the application (which, in turn, means that you’ll have to create your own using JavaScript and CSS).

getSystemPath(): This function returns various platform-standard file paths, such as the user’s documents folder.

openURLInDefaultBrowser(): Opens the specified URL in the system default web browser. You can use this function to direct users to a help page, or to other web resources.

CSInterface Extension Functions

CS Interface offers two methods for interacting with CC extensions:

requestOpenExtension(): Opens a CC extension. If your panel extension has an “Options” or “Preferences” dialog box, it’s quite useful to be able to open that extension from the panel.

closeExtension(): Closes a CC extension.

CSInterface and Events

CSInterface provides methods for sending and receiving CS Events:

dispatchEvent(): This function sends a CS event, which can be monitored by other CC extensions.

addEventListener(): Adds an event listener for a specified CS event.

removeEventListener(): Removes a CS event listener.

InDesign supports the following CS events:

documentAfterActivate: Fires when a document activates (after creating a new document, opening a document, or when a document regains focus). The event data contains the URI to the document (or the document name, if the document has not been saved).

documentAfterDeactivate: Fires when a document loses focus.

applicationBeforeQuit: Fires when the user quits the application, but before the application has shut down.

applicationActivate: Fires when the application gains focus.

documentAfterSave: Fires when the application has been saved. The event data contains the URI of the saved document.

Note that InDesign's ExtendScript DOM supports a much wider range of application events. You can use the PlugPlug External Library to send a CS event when these events fire. To do this, you create an event listener in the ExtendScript engine (if the first function is created via CSInterface.evalScript(), it will be created in the ExtendScript engine corresponding to the CC extension):

```
//Event listener:
function addSelectionChangedListener(){
  var eventListener = app.addEventListener(Application.AFTER_SELECTION_CHANGED,
    handleSelectionChange);
}

//Event handler:
function handleSelectionChange(event){
  //Construct a CSXS event and send it with data.
  var csxsEvent = new CSXSEvent();
  csxsEvent.type = idSelectionChanged";
  csxsEvent.data = "Selection changed!";
  csxsEvent.dispatch();
}
```

The above example assumes that you have loaded the PlugPlug External Object library. In InDesign CC 2014.1/CEP 5.2, this library can be loaded using the following:

```
var plugplugLibrary = new ExternalObject("lib:\PlugPlugExternalObject");
```

In earlier versions of InDesign CC, you will have to load the PlugPlug External Object library as a file:

```
//Where "file" is the File object containing the library:
var plugplugLibrary = new ExternalObject("lib:" + file.fullName);
```

The PlugPlug External Object library file is available from <link to https://github.com/Adobe-CEP/CEP-Resources/releases/tag/1>.

In the JavaScript for your extension, you can create an event listener and event handler to detect and respond to this event:

```
function addSelectionChangedEventListener(){
  csInterface.addEventListener("idSelectionChanged", function(event){
    //Do something here.
  });
  //Set up the event listener on the ExtendScript side:
  csInterface.evalScript("addSelectionChangedListener()");
}
```

Sending CS events from ExtendScript side of your extension isn't only for event monitoring. An ExtendScript function can send a CS event at any point, for any reason. If you're iterating through a loop in ExtendScript, for example, you could send CS events to update a progress bar on the JavaScript side of your extension. CS events are the main way that you communicate between the ExtendScript and JavaScript engines in your extension.

CEP and InDesign Plug-in Development

CEP uses the InDesign ExtendScript scripting DOM to automate document creation, layout, and typesetting in InDesign. The ExtendScript DOM, in turn, relies on scripting objects, properties, and methods exposed by the scripting interfaces provided by InDesign plug-ins. The capabilities of CEP, therefore, are determined by the scripting support that Adobe and third-party developers add to their plug-ins. CEP, scripting, and InDesign plug-ins are all part of the InDesign automation ecosystem.

Adding Scripting to an InDesign Plug-in

InDesign scripting interfaces are provided by each InDesign plug-in, and are merged into the InDesign ExtendScript DOM when the application starts. Third party developers can add scripting features to their plug-ins using the same techniques as the InDesign team uses in their plug-ins.

If you're creating an extension that includes a C++ plug-in (known as a "hybrid extension"), and if your extension needs to communicate with the plug-in, then you'll need to add scripting features to the plug-in to enable communication between the plug-in, the ExtendScript engine, and the V8 JavaScript engine.

Adding scripting features to an InDesign plug-in is covered in depth in the section "Scriptable Plug-in Fundamentals" of the "Plug-in Programming Guide" chapter of the *InDesign SDK Documentation*.

Sending Messages from InDesign Plug-ins

If you're creating an InDesign plug-in, you can use the PlugPlug libraries from the InDesign SDK to send CS event messages via PlugPlug. Event listeners in the V8 JavaScript engine of your extension can then react to these events. For more on adding PlugPlug CS event features to your plug-in, refer to <link to http://blogs.adobe.com/indesignsdk/html-extensions-in-indesign/>.

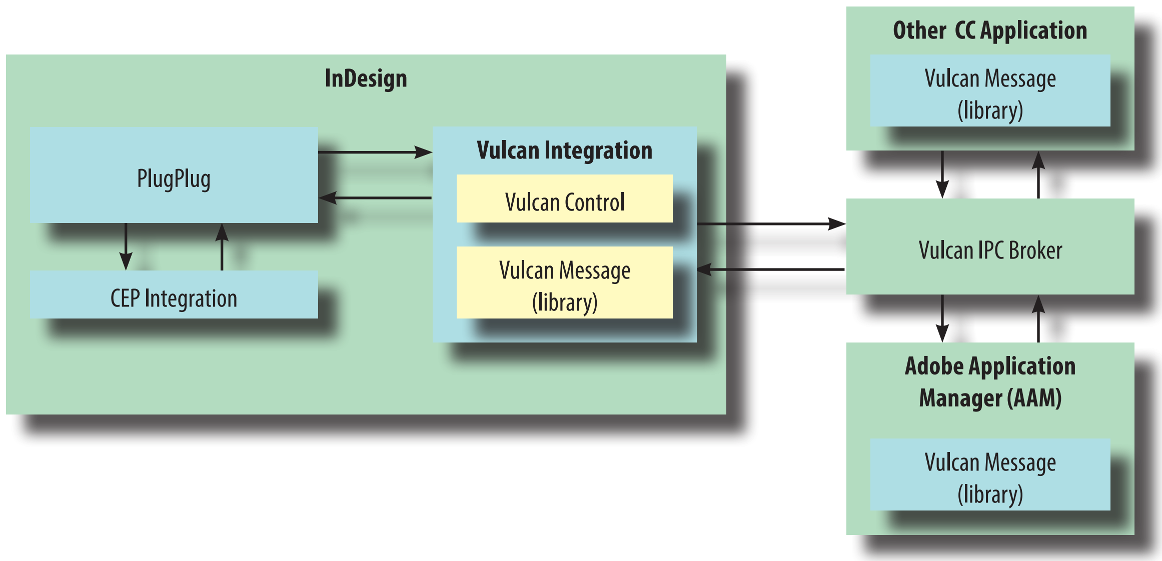
In addition, with the InDesign CC 2014.1 plug-in SDK, your extension can use PlugPlug to load and unload your CC extension. See <link to http://blogs.adobe.com/indesignsdk/new-plugplug-apis-for-hybrid-extension-developers/>

Adobe IPC Toolkit

The Adobe IPC Toolkit is a CC shared component that can be called from CEP to provide interapplication communication between CC desktop applications. It provides two libraries:

- Vulcan Message Library enables communication between CC desktop applications (or any application integrating the library and deploying the required infrastructure) and CC extensions.
- Vulcan Control Library can be used to control CC desktop applications (use this library to check for installed applications, check for running applications, launch the applications, etc.).

For more in-depth information on using the Adobe IPC Toolkit, refer to "IPC Message Handling," in *Using the Extension SDK* (also known as the file "Adobe Application SDK.pdf").



Adobe IPC Toolkit Component Diagram

Vulcan Control Library

Vulcan Control Library can be used to monitor and control Creative desktop applications (check to see whether the application is installed, whether an application is currently running, launch the application, etc.).

Vulcan Message Library

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

Vulcan IPC Broker

The Vulcan IPC Broker is the hub of IPC Toolkit messaging. It runs as a standalone process. It is launched by Vulcan Message library, and quits after last Vulcan Message library closes.

CEP JavaScript Extensions

In addition to CSInterface and the Adobe IPC Toolkit APIs, CEP offers a third API, CEPEngine_extensions. It is a JavaScript interface to a CEF extension. The CEPEngine_extensions supports the following areas:

File I/O: Manipulation of files and folders. A very useful API is OpenURLInDefaultBrowser(), which can be used to display help pages and other web-based information about your CC extension.

Process Control: Manipulation of external processes. A significant API is RegisterExtensionUnloadCallback(), which registers a callback function that will fire when your extension is unloaded. Use this function for anything you might want to do when the extension shuts down, such as writing persistent data to a file or to local storage.

Base64 encoding/decoding: Encode data and text to/from Base64. The functions provided by the APIs in this area are very useful for working with web assets, such as downloaded images.

There is no separate JavaScript library to include—these APIs are embedded in the CC extension framework.

For more on working with the APIs exposed by CEPEngine_extensions, refer to the Refer to the section "CEP Engine JavaScript Extension Reference" from *Using the Extension SDK* (also known as the file "Adobe Application SDK.pdf"), and the blog post "The Other API," at <link to https://medium.com/@HallgrimurTh/the-other-api-23357c99c774>.

Note: CEPEngine_extensions was introduced before support for node.js was implemented in CEP. In some cases, you might find it easier to use node.js and/or NPM to accomplish the same tasks (e.g., file system access). Note, in addition, that some of the tasks can be accomplished using InDesign's ExtendScript DOM (e.g., file/folder selection).