

 Filter by title

- The Windows Shell
- ▾ Shell Developer's Guide
- Shell Developer's Guide
- Security Considerations: Microsoft Windows Shell
- Guidance for Implementing In-Process Extensions**
- Developing with Windows Explorer
- Shell and Shlwapi DLL Versions
- > Implementing a Custom File Format
- > Shell Extensibility (Creating a Data Source)
- > Implementing Control Panel Items
- > Supporting Shell Applications
- > Legacy Shell Topics
- > Shell Reference
- > Deprecated API
- > Shell Samples
- Shell Glossary

⋮ / Desktop Environment / Windows Shell /

⊕ ✎ ⋮

Guidance for Implementing In-Process Extensions

Article • 01/07/2021 • 5 contributors [Feedback](#)

In this article

- [Version Conflicts](#)
- [Performance Issues](#)
- [Issues Specific to the .NET Framework](#)
- [Acceptable Uses of Managed Code and Other Runtimes](#)

In-process extensions are loaded into any processes that trigger them. For example, a Shell namespace extension can be loaded into any process which accesses the Shell namespace either directly or indirectly. The Shell namespace is used by many Shell operations, such as the display of a common file dialog, the launch of a document through its associated application, or the obtaining of the icon used to represent a file. Because in-process extensions can be loaded into arbitrary processes, care must be taken that they do not negatively impact the host application or other in-process extensions.

One runtime of particular note is the *common language runtime (CLR)*, also known as *managed code* or the *.NET Framework*. **Microsoft recommends against writing managed in-process extensions to Windows Explorer or Windows Internet Explorer and does not consider them a supported scenario.**

This topic discusses factors to consider when you determine whether any runtime other than the CLR is suitable for use by in-process extensions. Examples of other runtimes include Java, Visual Basic, JavaScript/ECMAScript, Delphi, and the C/C++ runtime library. This topic also provides some reasons that managed code is unsupported in in-process extensions.

Version Conflicts

A version conflict can arise through a runtime that does not support the loading of multiple runtime versions within a single process. Versions of the CLR prior to version 4.0 fall into this category. If the loading of one version of a runtime precludes the loading of other versions of that same runtime, this can create a conflict if the host application or another in-process extension uses a conflicting version. In the case of a version conflict with another in-process extension, the conflict can be difficult to reproduce because the failure requires the right conflicting extensions and the failure mode depends on the order in which the conflicting extensions are loaded.

Consider an in-process extension written using a version of the CLR prior to version 4.0. Every application on the computer that uses a file **Open** dialog box could potentially have the dialog's managed code and its attendant CLR dependency loaded into the application's process. The application or extension that is first to load a pre-4.0 version of the CLR into the application's process restricts which versions of the CLR can be used subsequently by that process. If a managed application with an **Open** dialog box is built on a conflicting version of the CLR, then the extension could fail to run correctly and could cause failures in the application. Conversely, if the extension is the first to load in a process and a conflicting version of managed code tries to launch after that (perhaps a managed application or a running

 **Download PDF**

application loads the CLR on demand), the operation fails. To the user, it appears that some features of the application randomly stop working, or the application mysteriously crashes.

Note that versions of the CLR equal to or later than version 4.0 are not generally susceptible to the versioning problem because they are designed to coexist with each other and with most pre-4.0 versions of the CLR (with the exception of version 1.0, which cannot coexist with other versions). However, issues other than version conflicts can arise as discussed in the remainder of this topic.

Performance Issues

Performance issues can arise with runtimes that impose a significant performance penalty when they are loaded into a process. The performance penalty can be in the form of memory usage, CPU usage, elapsed time, or even address space consumption. The CLR, JavaScript/ECMAScript, and Java are known to be high-impact runtimes. Since in-process extensions can be loaded into many processes, and are often done so at performance-sensitive moments (such as when preparing a menu to be displayed the user), high-impact runtimes can negatively impact overall responsiveness.

A high-impact runtime that consumes significant resources can cause a failure in the host process or another in-process extension. For example, a high-impact runtime that consumes hundreds of megabytes of address space for its heap can result in the host application being unable to load a large dataset. Furthermore, because in-process extensions can be loaded into multiple processes, high resource consumption in a single extension can quickly multiply into high resource consumption across the entire system.

If a runtime remains loaded or otherwise continues to consume resources even when the extension that uses that runtime has unloaded, then that runtime is not suitable for use in an extension.

Issues Specific to the .NET Framework

The following sections discuss examples of issues found with using managed code for extensions. They are not a complete list of all possible issues that you might encounter. The issues discussed here are both reasons that managed code is not supported in extensions and points to consider when you evaluate the use of other runtimes.

Re-entrancy

When the CLR blocks a single-threaded apartment (STA) thread, for example, due to a `Monitor.Enter`, `WaitHandle.WaitOne`, or contended [lock](#) statement, the CLR, in its standard configuration, enters a nested message loop while it waits. Many extension methods are prohibited from processing messages, and this unpredictable and unexpected reentrancy can result in anomalous behavior which is difficult to reproduce and diagnose.

The Multithreaded Apartment

The CLR creates *Runtime Callable Wrappers* for Component Object Model (COM) objects. These same Runtime Callable Wrappers are destroyed later by the CLR's finalizer, which is part of the multithreaded apartment (MTA). Moving the proxy from the STA to the MTA requires marshaling, but not all interfaces used by extensions can be marshalled.

Non-Deterministic Object Lifetimes

The CLR has weaker object lifetime guarantees than native code. Many extensions have reference count requirements on objects and interfaces, and the garbage-collection model employed by the CLR cannot fulfill these requirements.

- If a CLR object obtains a reference to a COM object, the COM object reference held by the Runtime Callable Wrapper is not released until the Runtime Callable Wrapper is garbage-collected. Nondeterministic release behavior can conflict with some interface contracts. For example, the **IPersistPropertyBag::Load** method requires that no reference to the property bag be retained by the object when the **Load** method returns.
- If a CLR object reference is returned to native code, the Runtime Callable Wrapper relinquishes its reference to the CLR object when the Runtime Callable Wrapper's final call to **Release** is made, but the underlying CLR object is not finalized until it is garbage-collected. Nondeterministic finalization can conflict with some interface contracts. For example, thumbnail handlers are required to release all resources immediately when their reference count drops to zero.

Acceptable Uses of Managed Code and Other Runtimes

It is acceptable to use managed code and other runtimes to implement out-of-process extensions. Examples of out-of-process Shell extensions include the following:

- Preview handlers
- Command-line-based actions such as those registered under **shell\verb\command** subkeys.
- COM objects implemented in a local server, for Shell extension points that allow out-of-process activation.

Some extensions can be implemented either as in-process or out-of-process extensions. You can implement these extensions as out-of-process extensions if they do not meet these requirements for in-process extensions. The following list shows examples of extensions that can be implemented as either in-process or out-of-process extensions:

- **IExecuteCommand** associated with a **DelegateExecute** entry registered under a **shell\verb\command** subkey.
- **IDropTarget** associated with the CLSID registered under a **shell\verb\DropTarget** subkey.
- **IExplorerCommandState** associated with a **CommandStateHandler** entry registered under a **shell\verb** subkey.

Feedback

Was this page helpful?

Yes

No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Additional resources

Training

Module
[Explore extensions and the extension framework in finance and operations apps - Training](#)




Finance and operations apps are customized by using extensions, which let you add functionality to model elements and source code in the Application Object Tree (AOT) by using Visual Studio.

Events

Nov 20, 12 AM - Nov 22, 12 AM

Gain the competitive edge you need with powerful AI and Cloud solutions by attending Microsoft Ignite online.
[Register now](#)

 English (United States)  Your Privacy Choices  Theme 

[Manage cookies](#) [Previous Versions](#) [Blog](#)  [Contribute](#) [Privacy](#)  [Terms of Use](#) [Trademarks](#)  © Microsoft 2024