Launcher: Mastering Your Own Domain

Source Code Accompanies This Article. Download It Now.

- launcher.txt
- launcher.zip

Launcher is a .NET GUI app that loads and executes other programs in multiple app domains.

October 01, 2004
URL:http://www.drdobbs.com/windows/launcher-mastering-your-own-domain/184405853

Loading and unloading managed programs into multiple domains

Gigi is a software developer specializing in object- and component-oriented programming using C++. He can be contacted at gigi_sayfanplaystation.sony.com.

Programmer productivity hinges on many factors, including talent, problem-domain experience, programming language, and tools, quality of tools, and—most importantly—the level of focus and degree of concentration, which is in constant flux. The ideal state of mind to program in is in the "flow," where you are fully focused on the problem, time flies, and you perform amazing programming feats. Some people call this "being in the zone" or "deep hack mode." Unfortunately, the smallest interruption can break the flow. If you work with the wrong tools or programming language, you might never achieve flow—you will instead fight your development environment, try to figure out obscure compiler messages, browse help files, and spend mental effort on side issues.

For instance, a friend and I are developing an online game using C# for the .NET platform. There is a server and multiple clients that share interfaces and common data types through a shared assembly. My development cycle includes editing the code, building the solution, and launching the server/client programs for testing. However, it takes time to launch a process, load the CLR, load the necessary system assemblies, load the server/client assembly, and start running. This ceremony gets old really fast, interrupts my flow, and strikes me as unnecessary. I should only load/unload the modified code.

The CLR provides the means to do this via *AppDomain*s. A single OS process can host multiple *AppDomain*s. The novel thing is that not only can DLLs be loaded into an *AppDomain*, but executable programs, too. Consequently, a single process can host multiple managed programs. In this article, I present Launcher, an application that loads/unloads other managed programs into multiple *AppDomain*s in its own process, executes them, and unloads them on demand. This application solves my problem because I can load/unload only my client and server assemblies into existing processes. Moreover, common system assemblies may already be loaded into the process, so the server and client programs don't have to load them again when they start executing. In addition to faster load/unload time, the memory footprint is reduced because the CLR and many assemblies that had to be loaded for each program in its own process can now be shared. While this discussion applies to CLR 1.1, I also mention some possible changes in Whidbey.

AppDomains, Processes, and Threads

The CLR is a managed runtime inside a managed runtime, which is the hosting OS. The OS has its own execution and isolation model for running code, which uses processes and OS threads. The CLR has a separate execution, security, and isolation model, which uses *AppDomain*s and CLR threads. *AppDomain*s are much lighter then OS processes. When a managed program starts executing, it is usually loaded into the default *AppDomain*. However, the code in the default *AppDomain* may create other *AppDomain*s in the same process, and load and execute other programs into them. The CLR verifies managed code upon just-in-time (JIT) compilation to be typesafe and guarantees that a fault in one *AppDomain* doesn't affect other ones in the same process.

A managed program runs inside a regular Win32 process that hosts the CLR as a COM object, loads the managed program, and executes it. This means that the CLR is subjected to the rules and regulations of the OS. Specifically, code always executes in the context of some OS thread. The CLR has its own concept of threads that are separate from OS threads. The mapping between OS threads and CLR threads is not documented officially, and it is considered an implementation detail that may change in a future version of the CLR. CLR threads may cross *AppDomain*s, but they have a separate thread local storage (TLS) in each *AppDomain* (accessible through the *Thread.GetData* and *Thread.SetData* APIs). The TLS of the OS thread associated with the CLR thread has its own *AppDomain*-agnostic TLS, which never changes when crossing *AppDomain*s. In the current implementation (Version 1.1), the same CLR thread is associated with the same OS thread whenever the OS thread enters the same *AppDomain*, but don't count on it.

When a program (executable assembly) is loaded and executed in an *AppDomain*, it usually loads many other assemblies. By default, each *AppDomain* maintains its own copy in memory of the loaded assemblies even if other *AppDomain*s in the same process have already loaded these assemblies. The only assembly that is always shared between all *AppDomain*s is *mscorlib*.

The loader optimization attribute loads assemblies to be shared between *AppDomain*s. The benefit is a smaller working set and quicker load time if many assemblies are shared. The downside is that access to static fields is slightly slower for shared assemblies because an indirection is needed to find the static field address of the active *AppDomain*. Regardless of the sharing mode, every *AppDomain* always maintains a distinct copy of static fields to facilitate the isolation of *AppDomain*s. If static fields were shared, then code in one *AppDomain* that modified a shared static field might wreak havoc in another *AppDomain,* which might not suspect it. Another disadvantage of shared assemblies is that they cannot be unloaded.

The loader optimization attribute tells the JIT compiler how assemblies should be loaded. There are three values:

- *SingleDomain* (the default) causes the *AppDomain* to load a private copy of each necessary assembly's code.
- *MultiDomain* loads one copy of each assembly per process.
- *MultiDomainHost* loads all strongly named assemblies per-process, although in future versions, it may only load assemblies from the global assembly cache (GAC) per-process.

I recommend the *MultiDomainHost* option for most multidomain scenarios. Private assemblies that are used by a single program should not be shared (since the performance of static fields will suffer) at the process level. *MultiDomainHost* is reasonable since GAC assemblies are usually used by many programs. Because most applications use only the default *AppDomain*, it makes sense that the default is *SingleDomain* to get fast access to static fields.

Working with AppDomains

The *AppDomain* class exposes several methods, properties, and events for creating *AppDomain*s, getting information on their state, unloading *AppDomain*s, and communicating between *AppDomain*s. Here, I show how to create a new *AppDomain*, load a program into it, and execute it. Then I handle events, exchange data between *AppDomain*s, execute code, and create objects in another *AppDomain*.

The first step is to create a strongly named assembly that can be shared in a process by multiple *AppDomain*s. I created a dummy key using the strong-name tool (sn.exe), added a postbuild event to the project of the shared assembly to install it to the GAC for easy resolution, and changed a couple of attributes in AssemblyInfo.cs; see Listing One and the comments in AssemblyInfo.cs for details.

Because I want my multidomain program to share assemblies, I set the loader optimization attribute to *MultiDomainHost* (Listing Two). Shared assemblies must be strongly named. A strongly named assembly has a cryptographic signature that is useful in security-oriented code scenarios.

Next, I create a new *AppDomain* by calling the *AppDomain.CreateDomain* static method. This method has several overloaded versions, the simplest one just accepts a domain name. There are many attributes that control the behavior of *AppDomain,* which can be specified through the *AppDomainSetup* class. Once an *AppDomain* has been created, it can also be destroyed or unloaded. The terminology is a little inconsistent, but it hints to asymmetry in the way code and resources are managed in the CLR. You can load specific assemblies into an *AppDomain*, but you cannot unload an assembly from an *AppDomain*. The only way to unload an assembly is to unload the entire *AppDomain* (with all the other assemblies that are loaded into it). Listing Three shows how to create an *AppDomain* and immediately unload it.

But just creating new *AppDomain*s and unloading them isn't very interesting—I want to run some code in the new *AppDomain*. There are many ways to load code into the *AppDomain* and execute it. The simplest method is to call the *ExecuteAssembly* method of the new *AppDomain* (Listing Four). This method loads the specified assembly and executes it immediately in the new *AppDomain*. The calling thread is blocked until the program in the new *AppDomain* exits. It is possible to use asynchronous method invocation to allow the calling thread to continue working. The DomainMaster library (available electronically; see "Resource Center," page 5) implements this approach.

Okay, you can launch programs in new *AppDomain*s, but what if you want information on their execution? The *AppDomain* class exposes several events (see Table 1). You can register for these events just like any other CLR event. The important thing is, of course, to register the event handlers before loading the program because most events occur before the program actually starts executing (Listing Five). The event handlers execute in the *AppDomain* they were registered in. This means that the code of the event handlers must be loaded into that *AppDomain*. The assembly loader must be able to locate the assembly that contains the event handlers. This is an involved process, but the bottom line is that the assembly must be either on the probe path (which is always under the application base directory), under the dynamic base directory (if specified), or in the GAC.

Events are cool but they are handled in the context of the *AppDomain* that they occur in. You may want to communicate information between *AppDomain*s and there are various ways to do so. One category is through OS-level IPC (simple files, P/Invoke Win32 IPC); the other category is *AppDomain*-specific communication methods, which is what I focus on here. The simplest method is using the *GetData*/*SetData* methods. You can *Get*/*Set* a value or a full-fledged object from/to a different *AppDomain*. Objects are passed by value (a copy of the original object) or by reference (a proxy to the original object). Listing Six demonstrates passing the creating *AppDomain* object to the created *AppDomain*. The *GetData*/*SetData* methods are intuitive to use and provide a dictionary API where each value/object is identified by a string.

The CLR allows executing arbitrary code in a different *AppDomain* via the *DoCallback()* method. The code requires defining a delegate of type *CrossAppDomainDelegate*. Listing Seven demonstrates an interaction between two *AppDomain*s. The default *AppDomain* creates another *AppDomain* and calls *SetData()* to pass itself to the other *AppDomain*. Then it executes some code in the other *AppDomain* that uses the data that was set earlier. The assembly that contains the *CrossAppDomainDelegate* code, as well as any other assemblies it references, must be loaded into the other *AppDomain* prior to calling the *DoCallback()* method.

The most advanced way of communication is creating instances of objects in a different domain. When creating an instance using the *CreateInstance()* method, the return value is an *ObjectHandle*. which can't be used directly in the current *AppDomain* without unwrapping it first. The benefit of an *ObjectHandle* is that it does not require loading all the metadata of the wrapped object. It decouples the creation of the object from its use. The creating *AppDomain A* may blindly create an object instance *O* in *AppDomain B*, then pass *O* to *AppDomain C* where it is unwrapped and used. The metadata of object *O* does not have to be loaded into *AppDomain A*. A simpler scenario is when *AppDomain* A creates an object instance *O* in *AppDomain B* and would like to use it. Listing Eight demonstrates that by using the *CreateInstanceAndUnwrap()* method to create an *AppDomainExplorer* instance in the other *AppDomain*. Since *AppDomainExplorer* is derived from *MarshalByRefObject*, the returned object is a proxy to an instance in the other *AppDomain* and not a local copy. Creating *AppDomain* loads the assembly, which contains the type and its metadata, into the other *AppDomain*.

What actually happens when an *AppDomain* is unloaded? The *DomainUnload* event is called and all threads are aborted. However, due to the subtle interactions between OS threads and CLR threads, some threads cannot be aborted. In this case, an exception is thrown that claims that a thread cannot be unwound, but there is no good way to handle it. The *AppDomain* won't be unloaded along with all its assemblies and that's it. I encountered this situation when I called *Console.ReadLine()*. When a nondefault *AppDomain* is unloaded, an *AppDomainUnloadException* is thrown in the default *AppDomain* as an unhandled exception. I tried to catch it explicitly by wrapping the call to *AppDomain.Unload* with a *try-catch* block, but it didn't help. I guess the exception is thrown in the unloaded *AppDomain*, which is already in a state that doesn't let it handle it. The text of the unhandled exception is: "An unhandled exception of type 'System.AppDomainUnloadedException' occurred in Unknown Module." GUI applications cannot be unloaded cleanly without quitting first. If you try to unload an active GUI application, it crashes unpleasantly.

In general, the Console doesn't play nicely with multidomain programs. The main reason is that there is one console per process and it is not trivial to share it properly between multiple console apps in various *AppDomain*s. This is a basic feature of the OS—one console at most per one process. Even using Win32 APIs, you can just replace a console with another console or create a console for a GUI app (*AllocConsole()*), but you can't have more than one console per process. So, all the console apps in a multidomain program share the same console. If the default *AppDomain* contains a GUI app, then a console is created for the first console app that is loaded into another *AppDomain*. The title of the console always reflects the latest console application that was loaded. There is no way to switch "focus" between apps in different *AppDomains*. This leads to a serious problem with *Console.ReadLine()*. When users press the Enter key, which app in which *AppDomain()* should handle it? It turns out to be a moot point because, even if there is a single app that called *Console.ReadLine()*, pressing Enter has no effect. Threads that wait on *ReadLine()* cannot be aborted and their *AppDomain*s won't unload. Finally, if the app in your default *AppDomain* is a GUI app and you load a console app into another *AppDomain*, you will not be able to get rid of the console. Even if you never call *Console.ReadLine()* and unload all the *AppDomain*s that contain console apps, the console window will remain there. It sort of becomes your main window. A console window created by closing the main form doesn't terminate the program, while closing the console window does, regardless of whether there are still active *AppDomain*s that contain console apps. I reported all these issues to Chris Brumme (a senior architect on the CLR team), who provided me with invaluable inside information and said that he is aware that *Console.ReadLine()* is blocked in an OS API that cannot be safely interrupted (http://blogs.msdn.com/cbrumme/). These issues are not likely to be solved in Whidbey despite the serious overhaul the Console is getting.

Launcher

Launcher (available electronically) is a .NET GUI app that loads and executes other programs in multiple app domains and allows unloading and reloading them. It is an extraordinary deployment and debugging tool. Launcher's GUI lets users add new applications to the managed list. Applications on the list can be executed (loaded) or stopped (unloaded). The application list is stored in a file so Launcher remembers the applications across invocations.

Launcher is really just a GUI front end to the DomainMaster engine, which lives in a separate assembly (DomainMasterLib). The DomainMaster engine is a Singleton. This modularization cleanly separates the core AppDomain management from the GUI front end. It also allows experimenting with different front ends or easily creating a dedicated custom Launcher. For example, always launch the BoogaBooga server, wait five seconds, launch three BoogaBooga clients, and finally launch the standalone Logger application.

The public interface of the DomainMasterLib assembly is defined in the Interfaces.cs file (available electronically). You acquire the main interface (*IDomainMaster*) from the static property *DomainMaster.Instance*. This is the only way since the constructors are private. DomainMaster stores a little information about every application in the *AppInfo* class: the executable path, the *AppDomain* object it belongs to (null if the application is not running), and a state enumeration (IDLE, RUNNING, SHUTTING_DOWN, or NONE). It exposes an outgoing interface (*IDomainMasterEvents*) for notifying the front end on interesting events such as assembly load and *AppDomain* unload. The *IDomainMaster* interface is the active interface, which the front end uses to communicate with DomainMaster. The front end may add new applications to the managed list of DomainMaster, it may launch idle applications, terminate running applications, and get a collection of all the managed apps through the *Apps* property. To receive the events, the front end needs to implement the *IDomainMasterEvents* interface and call the *AttachSink()* method.

The API is not perfect. You can't remove an application once it was added (except trough editing the persistent apps.txt file) and the fact that the DomainMaster is a Singleton allows only a single applications collection and a single sink (per process). However, it is good enough for my needs and it can be easily modified if necessary.

The *DomainMaster* class operates much like the sample code previously discussed. The major difference is that the DomainMaster is using asynchronous method calls to avoid blocking its users. The *ExecuteAssemblyDelegate* is declared at the top, and in the *Launch()* method I wrap it around the new *AppDomain ExecuteAssembly* method and invoke it asynchronously using *BeginInvoke()*. I don't wait or check the result of the invocation. If something went wrong, I get an exception. The rest of the code is just bookkeeping of the apps collection. The event handlers notify the user through the *IDomainMasterEvents* interface about interesting events and the *Terminate()* method is used to unload running applications.

I include two test applications called *Client* and *Server* (available electronically). These applications talk through Remoting and can be launched through Launcher. Remoting is the official way to communicate between applications in different *AppDomain*s, which may be in the same process, a different process on the same machine, or on a different machine altogether. The test applications are both console programs and demonstrate all the console-related issues I keep whining about.

Conclusion

In the future, Windows will likely move to direct mapping of OS concepts to CLR concepts. The current discrepancies are difficult to overcome—this is clear when trying to fit AppDomains into the OS world of processes, hard threads, and consoles. This probably improves the performance because managed-unmanaged costs a lot today (about 50 CPU instructions). As more and more APIs are migrated to managed code, this becomes less of a problem because you will not have to leave the managed world most of the time. Still, for interoperability with legacy Win32 and COM code, it is important to improve this aspect of the CLR.

How about Launcher? I hope that Whidbey solves the console issues I mentioned. But, in the meantime, a couple of possible directions I intend to follow include intercepting console events and output and writing them to a GUI console simulator, and hosting the CLR as a COM object and using the low-level interfaces it provides to improve the behavior of Launcher.

DDJ

Listing One

```
// The post build event
call "$(DevEnvDir)..\Tools\vsvars32.bat"
sn -Vr $(TargetDir)$(TargetFileName)
gacutil -u $(TargetName)
gacutil -i $(TargetDir)$(TargetFileName)

// From AssemblyInfo.cs
[assembly: AssemblyKeyFileAttribute("..\\..\\public.snk")]
[assembly: AssemblyDelaySignAttribute(true)]
```

Back to article

Listing Two

```
namespace DomainMasterSample
{
   class MultiDomainApp
   {
      [LoaderOptimization(LoaderOptimization.MultiDomainHost)]
      static void Main()
      {
      }
}
```

Back to article

Listing Three

```
namespace DomainMasterSample
{
   class MultiDomainApp
   {
      [LoaderOptimization(LoaderOptimization.MultiDomainHost)]
      static void Main()
      {
         // Create a new AppDoamin
         AppDomain ad = AppDomain.CreateDomain("Playground Domain");
         // Unload the domain
         AppDomain.Unload(ad);
      }

   }
}
```

Back to article

Listing Four

```
namespace DomainMasterSample
{
   class MultiDomainApp
   {
      [LoaderOptimization(LoaderOptimization.MultiDomainHost)]
      static void Main()
      {
         // Create a new AppDomain
         Console.WriteLine(AppDomain.CurrentDomain.FriendlyName);
         AppDomain ad = AppDomain.CreateDomain("Playground Domain");
         // Register domain event handlers
         ad.AssemblyLoad += new AssemblyLoadEventHandler(AssemblyLoadHandler);
         ad.DomainUnload += new EventHandler(DomainUnloadHandler);
         // Load and execute assembly in domain
         ad.ExecuteAssembly("Playground.exe");
         // Unload the domain
         AppDomain.Unload(ad);
      }
      static public void AssemblyLoadHandler(object sender,
                                           AssemblyLoadEventArgs args)
      {
         AppDomain ad = sender as AppDomain;
         Debug.Assert(ad == AppDomain.CurrentDomain);
         string a = args.LoadedAssembly.GetName().Name;
         AppDomain c = ad.GetData("Creator") as AppDomain;
         Console.WriteLine("Assembly Loaded: '{0}' in '{1}'
                     created by '{2}'", a, ad.FriendlyName, c.FriendlyName);
      }
      static public void DomainUnloadHandler(object sender, EventArgs args)
      {
         AppDomain ad = sender as AppDomain;
         Debug.Assert(ad == AppDomain.CurrentDomain);
      }
   }
}
```

Back to article

Listing Five

```
namespace DomainMasterSample
{
   class MultiDomainApp
   {
      [LoaderOptimization(LoaderOptimization.MultiDomainHost)]
      static void Main()
      {
         // Create a new AppDomain
         Console.WriteLine(AppDomain.CurrentDomain.FriendlyName);
         AppDomain ad = AppDomain.CreateDomain("Playground Domain");
         // Register domain event handlers
         ad.AssemblyLoad += new AssemblyLoadEventHandler(AssemblyLoadHandler);
         ad.DomainUnload += new EventHandler(DomainUnloadHandler);
         // Load and execute assembly in domain
         ad.ExecuteAssembly("Playground.exe");
         // Unload the domain
         AppDomain.Unload(ad);
      }
      static public void AssemblyLoadHandler(object sender,
                                            AssemblyLoadEventArgs args)
      {
         AppDomain ad = sender as AppDomain;
         Debug.Assert(ad == AppDomain.CurrentDomain);
         string a = args.LoadedAssembly.GetName().Name;
         AppDomain c = ad.GetData("Creator") as AppDomain;
         Console.WriteLine("Assembly Loaded: '{0}' in '{1}'
                      created by '{2}'", a, ad.FriendlyName, c.FriendlyName);
      }
      static public void DomainUnloadHandler(object sender, EventArgs args)
      {
         AppDomain ad = sender as AppDomain;
         Debug.Assert(ad == AppDomain.CurrentDomain);
      }
   }
}
```

Back to article

Listing Six

```
namespace DomainMasterSample
{
   class MultiDomainApp
   {
      [LoaderOptimization(LoaderOptimization.MultiDomainHost)]
      static void Main()
      {
         // Create a new AppDomain
         Console.WriteLine(AppDomain.CurrentDomain.FriendlyName);
         AppDomain ad = AppDomain.CreateDomain("Playground Domain");
         // Exchange data between AppDomains
         ad.SetData("Creator", AppDomain.CurrentDomain);
         AppDomain creator = ad.GetData("Creator") as AppDomain;
         Debug.Assert(creator == AppDomain.CurrentDomain);
         // Unload the domain
         AppDomain.Unload(ad);
      }
   }
}
```

Back to article

Listing Seven

```
namespace DomainMasterSample
{
   class MultiDomainApp
   {
      [LoaderOptimization(LoaderOptimization.MultiDomainHost)]
      static void Main()
      {
         // Create a new AppDomain
         Console.WriteLine(AppDomain.CurrentDomain.FriendlyName);
         AppDomain ad = AppDomain.CreateDomain("Playground Domain");
         // Exchange data between AppDomains
         ad.SetData("Creator", AppDomain.CurrentDomain);
         // Execute some code in a different AppDomain
         CrossAppDomainDelegate cadd =
                     new CrossAppDomainDelegate(CrossAppDomainCallback);
         ad.DoCallBack(cadd);
         // Unload the domain
         AppDomain.Unload(ad);
      }
      static public void CrossAppDomainCallback()
      {
         AppDomain ad = AppDomain.CurrentDomain;
         AppDomain c = ad.GetData("Creator") as AppDomain;
         Console.WriteLine("CrossAppDomainCallback() running in '{0}'
                       created by '{1}'", ad.FriendlyName, c.FriendlyName);
      }
   }
}
```

Back to article

Listing Eight

```
namespace DomainMasterSample
{
   public class AppDomainExplorer : MarshalByRefObject
   {
      public void Explore()
      {
         AppDomain ad = AppDomain.CurrentDomain;
         Console.WriteLine("----- AppDomainExplorer.Explore() -----");
         Console.WriteLine("Name: {0}", ad.FriendlyName);
         Console.WriteLine("Setup Info: {0}", ad.SetupInformation.ToString());
         Console.WriteLine("Assemblies:");
         foreach (Assembly a in ad.GetAssemblies())
         {
            Console.WriteLine(a.FullName);
         }
      }
   }
   class MultiDomainApp
   {
      [LoaderOptimization(LoaderOptimization.MultiDomainHost)]
      static void Main()
      {
         // Create a new AppDomain
         Console.WriteLine(AppDomain.CurrentDomain.FriendlyName);
         AppDomain ad = AppDomain.CreateDomain("Playground Domain");
         // Create object in another AppDomain
         string path = Path.Combine(Directory.GetCurrentDirectory(),
                                                "DomainMasterSample.exe");
         Assembly a = Assembly.LoadFile(path);
         ad.Load(a.GetName());
         AppDomainExplorer ade =
                ad.CreateInstanceAndUnwrap(a.GetName().FullName,
                "DomainMasterSample.AppDomainExplorer") as AppDomainExplorer;
         ade.Explore();
         // Unload the domain
         AppDomain.Unload(ad);
         Console.ReadLine();
      }
   }
}
```

Back to article

AppDomain	Events
AssemblyLoad	Occurs when an assembly is loaded.
AssemblyResolve	Occurs when an assembly resolution fails.
DomainUnload	Occurs when an AppDomain is about to be unloaded.
ProcessExit	Occurs on the default AppDomain when the parent process exits.
ResourceResolve	Occurs when a resource resolution fails.
TypeResolve	Occurs when a type resolution fails.
UnhandledException	Occurs when an exception is not caught by an event handler.

Table 1: ***Events exposed by the* AppDomain *class.***