Demystify .NET App Domains and Contexts

Do you want fault isolation and other plumbing perks without the usual overhead hit? Learn to wield these sophisticated .NET processes.

by Juval Lowy

Technology Toolbox

- ☑ VB.NET
- ☑ C#
- ☐ SQL Server 2000
- ☐ ASP.NET
- ☐ XML
- ☐ VB6

All .NET components and applications must run in the Common Language Runtime (CLR) managed environment. But the underlying operating system knows nothing about managed code. It provides processes and raw memory only. So managed code can't execute in the native operating system process. Instead, it uses a logical .NET process—application domain—that hosts assemblies and provides them with process-wide services. Each app domain interacts with the runtime DLL in the physical process space, providing its hosted assemblies with a managed heap, garbage collection, Just-in-Time (JIT) compiler, and assembly resolver and loader.

Moreover, because an app domain is only a logical abstraction on top of a physical process, a process actually can host multiple app domains (see Figure 1). This provides fault isolation. By distributing unmanaged applications (typically clients and servers) to multiple processes, a single process can fail without bringing down the whole application. You can handle the error or perform a graceful exit. Distribution across processes also enhances security because server objects often require incoming client call authorization and authentication checks.

In the past, you paid an overhead penalty for these distribution benefits because of cross-process calls and the overhead involved in managing multiple processes. .NET app domains are completely independent of one another—even if they share the same physical process. App domains can start and shut down separately from their hosting process, and you can debug them separately. App domains have security boundaries and their own loaded set of assemblies.

On the other hand, the overhead involved in making calls across app domains in the same process is more like that of same-process calls than cross-process calls—yet you pay only for creating and managing a single process.

The .NET application frameworks take full advantage of these important benefits. For example, ASP.NET puts every Web app in its own app domain, but they all share the same physical process. You can easily create new app domains yourself with the AppDomain class' static method,

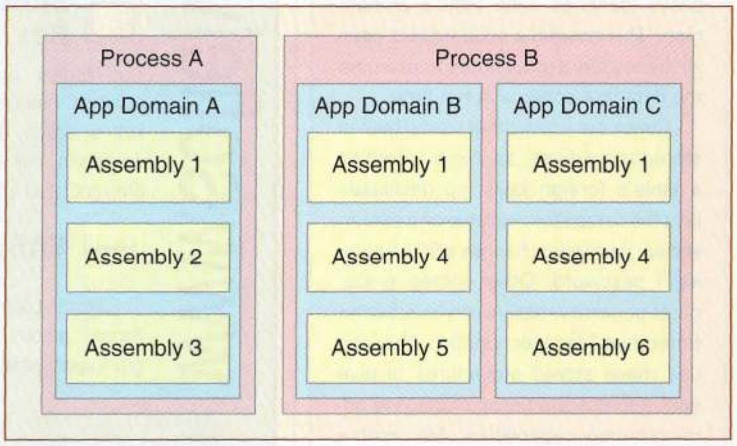


Figure 1 Discover Process and App Domains. An app domain is a logical managed process that hosts assemblies. A physical process can host multiple app domains, and app domains can start and shut down independently.

Resources

"Remoting Object Model Concepts" and "Context Bound Object Category," both in the MSDN library: http://msdn.microsoft.com/library

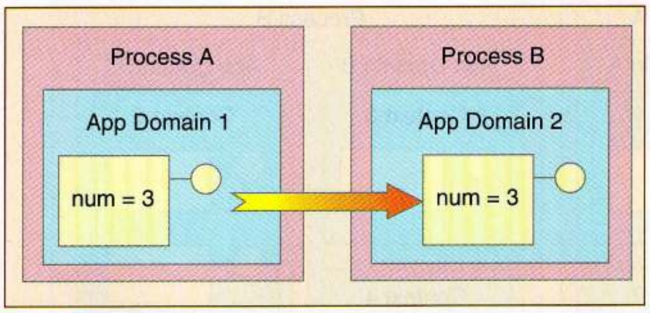


Figure 2 Marshal by Value. You can marshal any serializable object marshaled by value across app domain boundaries. Once marshaled by value, the new object is a cloned copy of the original and will change state independently.

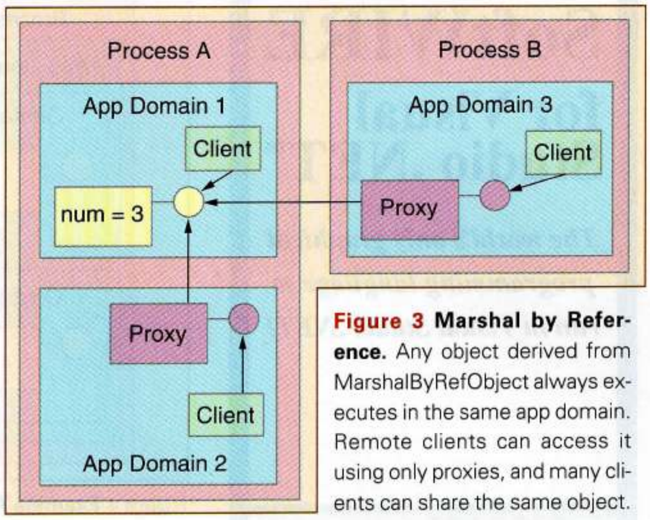


Figure 3 Marshal by Reference. Any object derived from MarshalByRefObject always executes in the same app domain. Remote clients can access it using only proxies, and many clients can share the same object.

CreateDomain(). This method returns a new AppDomain object, used to create new objects in the new app domain. You call the CreateInstance() method of the AppDomain object, providing the required type and its containing assembly.

App domain indirection also enhances portability. Different operating systems employ slightly different processes and process-management APIs, but it's all encapsulated from developers. You need only be aware of app domains. You can rely on the CLR's native provider to implement app domains properly.

Call Across App Domains

You can access objects in the same app domain without much fuss. But when you try to call methods on a remote object in another app domain, you'll find that by default, you can't access objects beyond their hosting app domain. You must allow for such access explicitly in your design and class definition.

You can cross the app domain boundary by value or by reference. By value means when a client in App Domain 2 calls a method of an object in App Domain 1, the object is copied to App Domain 2, giving the client its own cloned copy of the object. Changes made to the object state in App Domain 2 affect only the local copy. This resembles COM marshal by value and is often referred to as marshaling by value. To enable this, your managed class must be serializable—it must provide a generic, automated way to serialize its state to a stream so that .NET in the receiving App Domain can construct a new object, populating its member variables (its state) with the information in the stream.

.NET serialized classes can either use the Serializable attributes or implement the ISerializable interface for custom serialization. Recall that serialization affects all object fields, public and private alike.

By itself, .NET has no way of knowing which part of the object state is serializable as is. The object might hold resources or objects not marshalable by value, or parts of states that should be reacquired in the new app domain. If you use the Serializable attributes, you give your consent to marshal the object state automatically. For example, consider this class definition:

```
//C#
[Serializable]
public class MyClass
{
   public int num;
}
'VB.NET
<Serializable()> Public Class VBClass
   Public num As Integer
End Class
```

If the object's num member value is 3 and a remote client accesses it across an app domain boundary, .NET will marshal the object by value. When the remote client assigns 4 to num, this assignment affects its own new distinct copy (see Figure 2).

You have a second remoting option: marshalling by reference. When a client tries to access a remote object, .NET provides the client with a proxy object that implements the same set of public methods, properties, and fields as the object itself. The proxy forwards proxy calls on to the actual object. Clients in the same app domain as the object get a direct reference to the object; no proxies are involved.

To designate an object for marshaling by reference, the object must derive directly (or have one of its base classes derive) from MarshalByRefObject (see Figure 3):

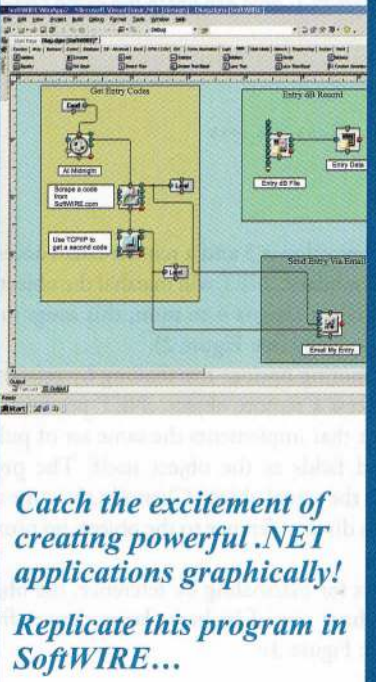
```
//C#
public class MyClass : MarshalByRefObject
{
   public int num;
}
'VB.NET
Public Class VBClass
   Inherits MarshalByRefObject
   Public num As Integer
End Class
```

Put Things in Context

.NET component services manage component connectivity and services plumbing so you can focus on business logic. For example, you get automatic thread concurrency synchronization, and .NET Enterprise Services (née COM+). The latter provides essential services such as object pooling, object activation, transactions, *Introducing*

SoftWIRE for Visual Studio .NET™

The world's only graphical programming language to run in Visual Studio .NET!



Catch the excitement of creating powerful .NET applications graphically!

Replicate this program in SoftWIRE…

Continued on next page.

SoftWIRE

www.softwire.com

SoftWIRE, Inc. • 16 Commerce Blvd.
Middleboro, MA 02346 • (508) 946-8900

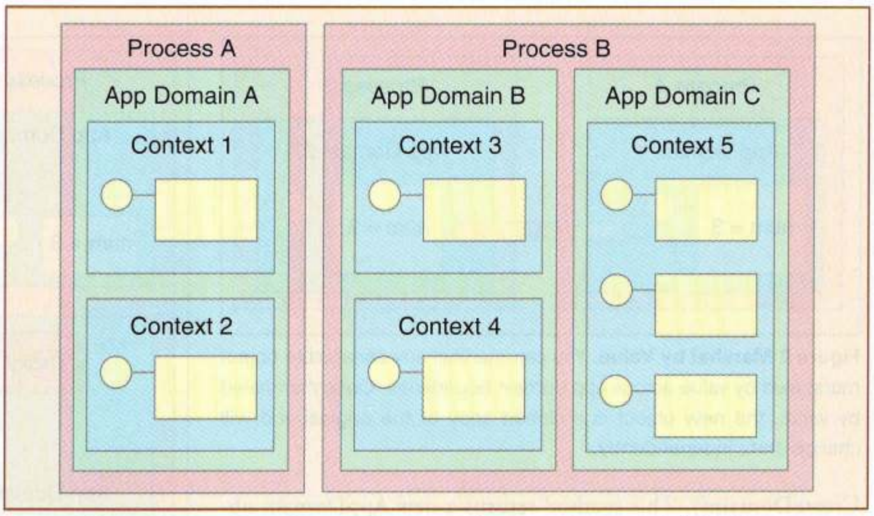


Figure 4 Examine App Domains and Contexts. App domains are subdivided into contexts providing specific runtime environments to objects. App domains can have many contexts, and a context can host many objects.

access security, disconnected work, publishing and subscribing to events, and more.

To supply these services, .NET must intercept the calls a client makes on the object and do some pre- and post-call processing. .NET does this by introducing a proxy between client and object.

For example, the proxy is associated with a lock to provide automatic thread safety and concurrency synchronization. If the object is available (not servicing another thread), the proxy locks the lock and forwards the call to the object. While the call is in progress, .NET blocks incoming calls from other clients on other proxies when they try to access the lock. When the call returns from object to proxy, it unlocks the lock, allowing other clients to use the object. By intercepting the call and performing pre- and post-call processing, .NET provides a valuable component service. For another example—access security—the proxy can verify client authorization before letting it call the object.

The key to such services is ensuring that you always have a proxy between client and object. But app domain affords too coarse an execution scope for this. Even though cross-app domain calls always go through a proxy, same-app domain calls use direct reference. To address this, .NET subdivides app domains into contexts within which objects execute (see Figure 4).

A context is a logical grouping of objects that rely on the same set of services. Calls into a context go through one or more proxies to ensure that an object always gets the runtime environment it needs. But if an object doesn't need these services, it shouldn't carry the overhead needed for cross-context access. So all .NET objects are classified by whether they need component services.

Objects are context-agnostic by default, lacking context affinity and always executing in the contexts of their calling clients. Because such objects "jump" from one context to another, they're also referred to as context-agile objects. Objects marshaled by value are context-agile by definition. You can still marshal a context-agile object by reference across app domains. Its agility takes place inside an app domain only.

The other type, context-bound objects, always execute in the same context. Their context affinity is fixed for life. All calls to context-bound objects go through proxies. To qualify as a context-bound object, the object must derive directly (or have one of its base classes derive) from the class ContextBoundObject:

```
//C#
public class MyClass :
ContextBoundObject
{...}

'VB.NET
Public Class VBClass
   Inherits ContextBoundObject
   ...
End Class
```

To see how context-bound objects can exploit .NET component services, consider the class Synchronization attribute in the System.Runtime.Remoting.Contexts namespace, used for the automatic synchronization I mentioned earlier:

```
//C#
[Synchronization]
public class MyClass : ContextBoundObject
{
public void DoSomething(){}
//other methods and data members
}
'VB.NET
<Synchronization()> Public Class
   VBClass
   Inherits ContextBoundObject
   Public Sub DoSomething()
   End Sub
   'other methods and data members
End Class
```

By adding the Synchronization attribute, .NET automatically ensures that only one thread may access the object at a time—no need to implement this functionality yourself. In fact, contexts are extensible; you can define your own context attributes for custom services and extensions such as call logging and tracing.

A context-bound object is, by definition, marshaled by reference across app domains because an object must never leave its context. To enforce this, the class ContextBoundObject derives directly from MarshalByRefObject. .NET decides on the object activation context based on the services the object requires and the context of its creating client when the client creates the object.

A creating client's context is "good enough" for an object's needs if the context has adequate runtime properties. In this case, the object is placed in its creating client context. But if the object requires services not supported by the creating client context, .NET creates a new context and places the new object in it.

Note that .NET doesn't try to find out if there is already another appropriate context for the object in that app domain. The algorithm is simple: The object will either share its creator's context or get a new one. This algorithm trades memory and context-management overhead for speed in allocating the new object to a context.

Also note that there's no limit to the number of contexts an app domain can host, nor to the number of objects a particular context can host. The only requirements are that a context must belong to exactly one app domain; that a context-bound object must belong to exactly one context; and that every app domain starts up with only one context, called the default context, which has no special properties or attributes. This makes the allocation algorithm simple to manage.

If you're a COM+ developer, these rules and behavior should look familiar because .NET context concepts adopt and extend those of COM.

Finally, I wanted to say a few words about passing object references around. Because you must always access context-bound objects using proxies, what happens if a client in the same context as the object passes an object reference to a client in a different context? If the same-context client had a direct reference to the object, how would .NET detect it and introduce a proxy between the new proxy and the new client?

In the COM+ world, you had to marshal object references between contexts manually using the Global Interface Table. Not in .NET. Instead, .NET always requires you to access such objects using a proxy, even by clients in the same context. It uses a *transparent proxy* that takes up minimal overhead, and it does nothing with the services the object uses.

However, the transparent proxy detects when the same-context client tries to pass its transparent proxy to another client in a different context. It then creates a new transparent proxy in the second client context and hooks it up with the actual proxy from the new context to the object. VSM

Juval Lowy is a software architect and principal of IDesign, a consulting and training company focused on .NET design and migration. Juval authored *COM and .NET Component Services* (O'Reilly). Juval is a frequent speaker at software development conferences and chairs the program committee of the .NET California Bay Area User Group. Contact him at www.componentware.net.

Go Online

Use these Locator+ codes at www.visualstudiomagazine.com to go directly to these related resources.

VS0202 Download all the code for this issue of *VSM*.

VS0202BB_T Read this article online.

Continued from previous page.

...and be eligible to win this Toshiba 50" TheaterWide-HD® Plasma TV Monitor and Sony Progressive-Scan DVD Player!

Toshiba 50HP81 $15,999

Sony DVP-S9000ES $1,400

No purchase required. Simply download SoftWIRE from the SoftWIRE website. Install it in your Visual Studio for VB6 or .NET. Build the application following the instructions on our site, and run it.

The running application creates your entry.

Drawing to be held March 15th, 2002. See our website for complete rules and details.

SoftWIRE™

www.softwire.com

SoftWIRE, Inc. • 16 Commerce Blvd. Middleboro, MA 02346 • (508) 946-8900