Application Domains and Contexts and Threads, Oh My!

Michael Stiefel
co-author “Application Development Using C# and .NET”

Why Understand App Domains?

- By understanding Application Domains you will have an opportunity to unify and tie together various concepts and mechanisms with .NET.
- You will meet them sooner or later.
- But when will I meet them?

When you least expect to...

Dorothy: “Do you suppose we'll meet any strange concepts when programming with .NET?"
Tinman: "We might."
Scarecrow: “Concepts that...that eat straw?"
Tinman: "Some. But mostly application domains, and contexts, and threads."
All: “Application Domains and Contexts and Threads, oh my!
Application Domains and Contexts and Threads, oh my!"

Scalability

- The more applications that can run simultaneously, the more *scalable* the solution.
- For applications to run together, they must be isolated from each other.
  - If one application crashes, the others must continue to run.
  - One application cannot *directly* access the data, or *directly* call methods, in another application.

Win 32 Application Isolation

- Applications are isolated in separate processes.
  - Each process has its own address space.
  - A *process* context switch is usually implemented in the processor.
- Processes are inefficient for really scalable solutions (tens to hundreds of thousands).
  - Process switches are slow.
  - Interprocess communication is much slower than intraprocess communication.

.NET Application Isolation

- The Common Language Runtime uses Application Domains to provide isolation in software.
- A type-safe assembly's types and methods can only be accessed in well defined ways.
- Hence, the CLR can prevent direct access to types from one application domain to another.
- Application Domains can contain multiple assemblies.

Application Domains

- Security Evidence
- Configuration Information
- Loaded Assemblies
- Code Isolation
- Think of ASP.NET as an example.

Code Access Security Policy

- Assemblies are assigned a certain level of trust.
  - For a given level of trust certain permissions are allowed, others are denied.
  - Examples of permissions are the right to access a file, use the clipboard, or make or accept Web connections.
- A particular level of trust is defined by a Code Group.
  - A code group is defined by various characteristics such as whether the assembly is running on the local computer or on a particular web site.

Security Evidence

- Evidence is the information associated with an assembly that allows the CLR to determine to which Code Group the assembly belongs.
- Examples of evidence:

  System.Security.Policy.Zone : MyComputer

  System.Security.Policy.Url : file://E:/AppDomainSecurity.exe
- The **Evidence** class represents the collection of evidence. Classes such as **Zone** and **Url** model the individual pieces of evidence.

Configuration Files

- If an assembly has a strong name, you can specify the version policy.
- You can also specify where to locate assemblies that have not yet been loaded.
  - The directory in which the application runs is known as the application base.
- The **AppDomainSetup** class models this information.

AppDomainSetup Class Properties

- **ApplicationBase** root application directory
- **ConfigurationFile** Gets or sets the name of the configuration file for an application domain.
- **LoaderOptimization** Specifies the optimization policy used to load an executable.
- **PrivateBinPath** Gets or sets the list of directories that is combined with the ApplicationBase directory to probe for private assemblies.
- **PrivateBinPathProbe** Gets or sets the private binary directory path used to locate an application.
- Private assemblies allow for building isolated apps.

AppDomain Class

- Represents an application domain.
- To create an application domain instance:

```
public static AppDomain CreateDomain(
        string appDomainName,
        Evidence appDomainEvidence,
        AppDomainSetup appDomainSetupInformation)
```

- Application domain creator can specify security evidence, and configuration.
  - Allows for building applications that cannot interfere with one another.

Executing Code

- Once an assembly has been created you can invoke methods to start executing code.
  - ExecuteAssembly
    - Execute code in assembly starting with entry method.
  - Load
    - Loads an assembly.
  - CreateInstance
    - Create a type, invoke methods through reflection.
- Can modify the evidence and culture.

AppDomain Example

```
Evidence ev = AppDomain.CurrentDomain.Evidence;
Evidence evidence = new Evidence(ev);
evidence.AddHost(new Url(@”file://f:/”);
evidence.AddHost(new Test());

AppDomainSetup setupInfo = new AppDomainSetup();
setupInfo.ConfigurationFile = "foo.config";

AppDomain appDomain = AppDomain.CreateDomain("NewAppDomain", evidence,
setupInfo);
appDomain.ExecuteAssembly("TargetAssembly.exe");
```

Unloading Application Domains

- When an application finishes executing the **AppDomain** can be unloaded.
  - Individual assemblies in an application domain cannot be unloaded.
  - The default (initial) application domain of a process cannot be unloaded.

TypeResolve Event

- If an assembly load fails, the type resolve event is raised.
- If handled, the application domain can provide the necessary assembly using any rules it wants, including building it on the fly.

Threads

- A process can have one or more threads of execution.
  - Threads are scheduled by the system.
  - A thread's context includes the machine registers and the stack.

Thread Class

- The currently executing thread is found from the static property **Thread.CurrentThread**.
- .NET threads run as delegates define by the **ThreadStart** class:

```
public delegate void ThreadStart();
```

- A thread instance is represented by the Thread class, the **Start** method causes the thread to run.
- The **Join** method causes a thread to wait on another thread.

Thread Synchronization

- Using threads can cause race conditions.
  - Does i = i + 1 represent a single statement?
  - What could go wrong?

Monitor Class

- The **Monitor** class allows you to set up a critical section where only one thread can execute at a time.
  - Monitor.Enter(object o), Monitor.Exit(object o)
- You can wait and signal objects.
  - Monitor.Wait, Monitor.Pulse, Monitor.PulseAll
- You do not have to block on Monitor.Enter.
  - Monitor.TryEnter

Interlocked Class

- The **Interlocked** class has methods for insuring the increments, decrements, and exchanges for single values update correctly.

```
Interlocked.Increment(ref OutputCount);
```

Synchronization Attributes

- Deriving your class from **ContextBoundObject**, you can use the **SynchronizationAttribute** class to let the system handle the synchronization for you for all instance methods on the object
- The SynchronizationAttribute class has 4 values:
  - REQUIRED, REQUIRES_NEW, SUPPORTED, NOT_SUPPORTED

```
[Synchronization(SynchronizationAttribute.REQUIRED)]
class Counter : ContextBoundObject
{
...
```

Threads and Application Domains

- A process can have multiple application domains and multiple threads: what is the relation between a thread and an application domain?
- Application domains run on the thread that created them.

Code Sample

```
ThreadStart x = new ThreadStart(RunCode);
Thread t = new Thread(x);
t.Start();
...
static void RunCode()
    {
        ...
        AppDomain appDomain =
        AppDomain.CreateDomain("NewAppDomain",
                               evidence, setupInfo);
        appDomain.ExecuteAssembly("TargetAssembly.exe");
    }
```

CLR Hosting

- If you need to use .NET from within a legacy application, you can write a CLR Host.
- An CLR Host must load the CLR, setup the application domains, and then execute their code.
- ASP.NET uses an ISAPI filter to start the CLR and load the Web services apps.
- Yukon will use CLR Hosting so that you can write stored procedures in .NET languages.

Starting up the CLR

- Since side-by-side versions of the CLR can co-exist, you can specify which version to startup.
  - If unspecified, the latest version is used.
  - In Version 1, only one CLR version runs at a time in a process.
- Specify which CLR execution engine you want: workstation or server.
  - Uniprocessor machines use workstation version.
- Specify which garbage collector you want.
- Request ICorRuntimeHost interface to start creating application domains.

CorBindToRuntimeEx

```
CComBSTR bstrVer(L"v0.0.0000");
CComBSTR bstrCLRType(L"wks");
CComPtr<ICorRuntimeHost> pHost;

HRESULT hr = CorBindToRuntimeEx(NULL, bstrCLRType,
                  STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN |
                  STARTUP_CONCURRENT_GC,  CLSID_CorRuntimeHost,
                  IID_ICorRuntimeHost, (void **)&pHost);
if  (!SUCCEEDED(hr))
{
     ...
}
```

Server and Workstation Engines

- Server version takes advantage of multiple processors.
  - Garbage collection can take place on each processor in parallel.
- Single processor machines always get workstation version.

Garbage Collection Settings

- Concurrent Garbage Collection occurs when garbage collection is done on background threads.
  - More responsive UI, but slower overall performance.
- Nonconcurrent Garbage Collection (default) is done on the user code thread.
  - Always better for server applications, better performance

ICorRuntimeHost

- Can stop and start CLR.
- Create the default domain.

```
pHost->Start();
CComPtr<IUnknown> punkDefaultDomain;
hr = pHost->GetDefaultDomain(&punkDefaultDomain);
CComPtr<_AppDomain> pDefaultDomain;
hr = punkDefaultDomain->QueryInterface(__uuidof(_AppDomain),
    (void**) &pDefaultDomain);

...
pHost->Stop();
```

Managed and UnManaged Hosts

- The hosting code is divided into a unmanaged and managed parts.
  - It is far easier to manage application domains from managed code.
  - Avoids transitions from managed to unmanaged code.
- The next step is to start up the managed host and invoke the startup method in the process' default domain.

Code Sample

```
CComBSTR bstrAssembly(L"AppHost");
CComBSTR bstrType(L"ManagedHost");
hr = pDefaultDomain->CreateInstance(bstrAssembly, bstrType, &pObjectHandle);
...
hr = v.pdispVal->QueryInterface(__uuidof(_Object), (void**) &pobj);
...
_Type* pType;
hr = pobj->GetType(&pType);
...
CComBSTR bstrArgument(L"StartManagedHost");
_MethodInfo* mi;
hr = pType->GetMethod_6(bstrArgument, &mi);
...
CComVariant vReturnValue;
hr = mi->Invoke_3(vHandle, NULL, &vReturnValue);
```

Managed Host

- The managed host then manages a thread pool to run the application domains it manages.
- The appropriate security, configuration files and location are used as appropriate.

Code Isolation

- How can you use data or call methods that live in another application domain?
  - Marshal by Value (the data gets copied)
  - Marshal by Reference (you get a reference to the data)
- This applies to function arguments or data access.

Marshal By Value

- Serialization makes a copy of the object.
  - Use Serialization attribute or implement ISerializable
  - Unfeasible for large objects (performance)
  - Duplicates object, no network hit to access object

```
[Serializable]
 class Counter
 {
 ...
```

Marshal By Reference

- Derive object from **MarshalByRefObject** and you will get a reference to the data in the other application domain.
- Uses proxies to access object
- Use if state must stay in one app domain or object is too large to copy

```
class Counter : MarshalByRefObject
{
...
```

Context

- Contexts are used inside of application domains to control access to objects that require a special execution environment.
- Derive the object from **ContextBoundObject** which derives from **MarshalByRefObject**.
  - Proxies are used even with an application domain so the infrastructure can provide the necessary services.
  - I.E.: different threading or transaction requirements.
  - Objects in the same context do not use proxies.

Summary

- Application Domains provide application isolation in software.
- Threads can run across application domains providing independent execution paths in an application.
- Contexts provide a means for the infrastructure to provide services to make different application objects work together.