# Wer startet Ihre .NET-Applikation?

## Tiefe Einblicke in die Klasse AppDomain

**von Frank Listing, MicroConsult GmbH**

Die Klasse AppDomain fristet ihr Dasein weitgehend unbeachtet von den meisten Entwicklern. Und doch arbeitet sie klaglos und unermüdlich, denn sie ist – oft unbemerkt – an der Ausführung aller .NET-Applikationen beteiligt. Dieser Artikel beschäftigt sich mit ein paar interessanten Eigenschaften und Anwendungsmöglichkeiten der Klasse AppDomain.

Die AppDomain wurde mit .NET neu eingeführt und repräsentiert das neue Prozessmodell dieser Software-Technologie. Sie hat ihren eigenen virtuellen Speicher und einen eigenen Sicherheitskontext. Der Start einer .NET-Applikation wird über eine Instanz der Klasse *AppDomain* gesteuert. Sie ist verantwortlich für das Laden der Assemblies und Sicherheitsprüfungen sowie das Finden der Ressourcen. Vielleicht wird sie eines Tages den Prozess ersetzen, wie wir ihn heute vom Windows-Betriebssystem kennen.

Heutzutage brauchen wir den Prozess aber noch, denn er ist fest im Betriebssystem verankert. Wo ist dann die AppDomain? Im Falle einer normalen .NET-Anwendung kann man sie sich als eine Art „Tapete“ vorstellen, mit der wir den Prozess auskleiden. Er startet die .NET-Laufzeitumgebung, die dann ein AppDomain-Objekt erzeugt, welches wiederum die Kontrolle übernimmt und die Assemblies ausführt.

### Mehrere AppDomains in einem Prozess

Im Zusammenspiel von Prozess und AppDomain gibt es eine Besonderheit: Ein Windows-Prozess kann mehr als eine AppDomain beherbergen. Das heißt, dass innerhalb eines Prozesses mehrere .NET-Applikationen gleichzeitig laufen können. Die Klasse AppDomain sorgt dafür, dass jede der Anwendungen ihren eigenen Ausführungskontext besitzt und diese damit sauber voneinander getrennt sind. Wollen sie miteinander kommunizieren, wird ein RPC-Aufruf wie beispielsweise Remoting nötig, obwohl die Applikationen innerhalb desselben Prozesses ausgeführt werden.

Von der Möglichkeit, mehrere Instanzen der Klasse *AppDomain* in einem Prozess anzulegen, wird relativ selten Gebrauch gemacht. Eine prominente Ausnahme bildet der Internet Information Server. ASP.NET-Applikationen werden dort in separaten AppDomains gestartet, was unter anderem folgende Vorteile mit sich bringt:

- Jede Applikation hat einen eigenen Sicherheitskontext.
- Die Speicherbereiche sind gegeneinander abgeschottet.
- Falls eine Anwendung Probleme bereitet, wird die AppDomain einfach entfernt. Der IIS läuft stabil weiter.

Das folgende Beispiel zeigt, wie ein Entwickler mehrere .NET-Applikationen in einem Windows-Prozess starten kann (Listing 1):

**Listing 1**

```
private void _btnStart_Click(object sender, System.EventArgs e)
{
    Thread t= new Thread(new ThreadStart(StartApplikation));
    t.Start();
}

private void StartApplikation()
{
    int count= Interlocked.Increment(ref _count);
    AppDomain ad= AppDomain.CreateDomain("AppDomain"+ count.ToString());
    _lastStarted= ad;

    try
    {
        ad.ExecuteAssembly(_tbApplication.Text);
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.Message, "Start");
    }

    // die Reste der AppDomain entsorgen
    // (Haupt-AppDomain muß zuletzt geschlossen werden)
    AppDomain.Unload(ad);
    Interlocked.Decrement(ref _count);
}

private void StarterWnd_Closing(object sender, System.ComponentModel.CancelEventArgs
e)
{
    // Sicherstellen, daß die Haupt-AppDomain
    // zuletzt geschlossen wird
    if(_count > 0)
    {
        e.Cancel= true;
        MessageBox.Show("Bitte schließen Sie erst alle Kindfenster!");
    }
}
```

**/Listing 1**

Der Einstieg ist die Methode *_btnStart_Click*, in der wir zuerst einen neuen Thread erzeugen. In diesem werden wir unser Programm starten. Dieser zusätzliche Thread ist notwendig, weil je nach Art des Programms sonst unterschiedliche Effekte auftreten können: Eine Konsolen-Applikation bleibt beispielsweise einfach stehen und wartet, bis das gestartete Programm beendet ist, da sie nur einen Thread besitzt und dieser mit der Ausführung des „Kindprogramms“ beschäftigt ist. Bei einer Windows-Applikation ist das Verhalten unbestimmter. Da hier von Haus aus mehrere Threads die Arbeit verrichten, können wir hier zum Beispiel über einen Button-Klick mehrere Kindanwendungen starten. Dabei kann es passieren, dass beim Beenden einer einer dieser Applikationen eine weitere mit „abgeschossen“ wird.
Deshalb erzeugen wir erst mal einen Thread. Hier lassen wir von der statischen Methode *CreateDomain* der Klasse *AppDomain* ein neues AppDomain-Objekt erstellen. Auf diesem führen wir die Methode *ExecuteAssembly* aus und geben ihr natürlich die Information über das gewünschte Assembly mit. Nun wird dieses ausgeführt und der Thread bleibt an dieser Stelle stehen und wartet, bis das Assembly beendet wurde. Danach muss die AppDomain entladen werden, damit sich die Starter-Applikation nicht undefiniert verhält bzw. definiert am Ende abstürzt.

Es ist unbedingt zu beachten, dass die Haupt-AppDomain als letzte beendet wird, sonst versucht die Laufzeitumgebung unter Umständen auf Ressourcen zuzugreifen, die bereits entsorgt wurden. Daraus resultiert der Absturz. Um die Einhaltung dieser Regel sicherzustellen, wurde der Zähler (Variable *_count*) eingebaut, der in der Methode *StarterWnd_Closing* dafür sorgt, dass die Testapplikation erst geschlossen wird, wenn alle zusätzlichen AppDomain-Objekte entladen sind. Weiterhin wird die Ausführung des Assemblies von einem try-catch-Block eingeschlossen. Dies

fängt Abstürze der Kindprogramme sicher ab und die AppDomain lässt sich hinterher sauber entsorgen.

Im vorliegenden Beispielprogramm kann über einen Button ein .NET-Assembly innerhalb desselben Prozesses wie die Starter-Applikation ausgeführt werden. Beim Start mehrerer Kind-Applikationen zeigt der Windows-Task-Manager trotzdem nur einen Prozess an.

Listing_AppDomain_1.tif

Abb. 1: Rechner.exe wird vom Task-Manager nicht angezeigt.

Weil wir einmal dabei sind, können wir auch gleich noch eine andere interessante Methode der Klasse *AppDomain* ausprobieren: *Unload*. Wenn wir im Beispielprogramm den Button *Unload* drücken, wird die Methode _btnUnload_Click aufgerufen und diese versucht, die zuletzt gestartete AppDomain zu beenden (Listing 2).

**Listing 2**

```
private void _btnUnload_Click(object sender, System.EventArgs e)
{
    try
    {
        AppDomain.Unload(_lastStarted);
        Interlocked.Decrement(ref _count);
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.Message, "Unload");
    }
}
```

**/Listing 2**

In der hier gezeigten Konstellation Starter-Applikation und Testprogramm führt das aber meistens zu einer *CannotUnloadAppDomainException*, weil ein Thread der Anwendung nicht rechtzeitig beendet werden kann und ein Timeout zuschlägt. Hinterher stürzt dann allerdings die Applikation ab, weil der ein oder andere Thread doch schon beendet wurde – Operation gelungen, Patient tot.

Die Hilfe zur Klasse AppDomain enthält Informationen zu weiteren durchaus beachtenswerten Properties und Methoden, die hier nicht beleuchtet werden können. Stattdessen setzen wir uns mit ein paar Ereignissen auseinander, welche für Entwickler besonders hilfreich sind. [1]

### Das UnhandledException-Event

Das UnhandledException-Ereignis tritt auf, wenn eine Ausnahme im Programm aufgetreten ist, die wir nicht behandelt haben. Es bildet quasi eine Klammer um die Applikation und ermöglicht einen halbwegs geregelten Ausstieg. Wenn in der Handler-Methode das Property *IsTerminating* der mitgeschickten *UnhandledExceptionEventArgs* mitteilt, dass die Applikation beendet wird, ist eine Rettung nicht möglich. Wenigstens erhält der Benutzer aber die Information, dass es uns leid tut und wir diesen Absturz nicht vorhergesehen haben (Frei nach Microsoft: „Unerwarteter Fehler in…“). Natürlich besteht auch die Möglichkeit, Indizien über den Absturz zu sichern und in ein Log-File, das Event-Log oder andere geeignete Kanäle zu schreiben. Damit erhält der zuständige Entwickler Hinweise zur Fehlerbehebung. Diese liefert das Exception-Objekt, welches uns über das Property *ExceptionObject* von der Laufzeitumgebung zur Verfügung gestellt wird. Hier können wir die üblichen Informationen wie die Fehlernachricht und den Stack-Trace auslesen, um den Fehler zu lokalisieren.

### AssemblyResolve-Event

Dieses Ereignis wird ausgelöst, wenn die CLR ein Assembly auf dem üblichen Weg über das Applikationsverzeichnis oder GAC nicht finden kann. Dies kann der Fall sein, wenn die Installation

mangelhaft ist oder der Benutzer seine Festplatte zu intensiv aufgeräumt hat. Warum aber nicht sogar absichtlich dafür sorgen, dass die CLR ein Assembly nicht auflösen kann?
Die Möglichkeit, in den Lademechanismus einzugreifen, eröffnet uns interessante Perspektiven. Applikationen, die unter Aufsicht der CLR laufen, haben eine unangenehme Eigenschaft: Sie sind leicht dekompilierbar. Eine bunte Oberfläche kann jeder bauen, da müssen wir uns nicht extra schützen.. Aber in den Tiefen des Programms gibt es hin und wieder einen geschickt ausgedachten Algorithmus, eine Formel, die nicht von jedem Anfänger erkannt werden soll.
Zwar leistet der Dotfuscator dabei gute Dienste, aber warum lassen wir das Assembly nicht ganz von der Bildfläche verschwinden und legen es auf einen zentralen Server, auf den kein Benutzer direkten Zugriff hat? Wie dies möglich ist, zeigt das folgende Beispiel, welches das AssemblyResolve-Ereignis verwendet.

Dafür müssen wir in unsere Applikation einen Mechanismus einschleifen, der sich um die nicht gefundenen Assemblies kümmert. Dieser wurde in einer Hilfsklasse (*ResolverUtil*) verpackt und für seine flexible Wiederverwendung in einer DLL implementiert (Listing 3).

**Listing 3**

```
public class ResolverUtil
{
    private static ResolverServer.IResolver _resolver;

    public static void Init(string server)
    {
        try
        {
            string url= "tcp://"+ server+ ":8111/Resolver";
            _resolver= (ResolverServer.IResolver)Activator.GetObject(
                typeof(ResolverServer.IResolver),
                url);

            AppDomain cD= AppDomain.CurrentDomain;

            // Anmelden des eigenen Resolvers bei der zuständigen AppDomain
            cD.AssemblyResolve+= new ResolveEventHandler(MyResolver);
        }
        catch
        {}
    }

    private static Assembly MyResolver(object sender, ResolveEventArgs args)
    {
        try
        {
            byte[] byAsm= _resolver.GetAssembly(args.Name);

            if(byAsm != null)
            {
                Assembly assembly= Assembly.Load(byAsm);
                return assembly;
            }
        }
        catch
        {}

        return null;
    }
}
```

**/Listing 3**

Die Klasse unterteilt sich in zwei Bereiche: Einerseits in die Initialisierung und andererseits in den eigentlichen Resolver. In der Initialisierung wird eine Verbindung zum Server bereitgestellt, welcher uns die fehlenden Assemblies liefern soll. Hierzu benutzen wir Remoting. Den Namen des Servers lassen wir uns vom Aufrufer bereitstellen. In der Methode *MyResolver* wird über diese Verbindung beim Server angefragt, ob das fehlende Assembly vorrätig ist und im Erfolgsfall geladen sowie an die Laufzeitumgebung durchgereicht.

Auf diese Weise müssen wir im Hauptprogramm nur noch in der Main-Methode diesen Mechanismus initialisieren (Listing 4). Hier wird zuerst der Rechnername des Servers aus der

Applikations-Konfigurationsdatei gelesen und danach die Methode *Init* der Resolver-Hilfsklasse aufgerufen. Dieser geben wir den Server-Namen mit.

**Listing 4**

```
static void Main()
{
    string server=
        System.Configuration.ConfigurationSettings.AppSettings["Server"];
    Resolver.ResolverUtil.Init(server);

    Application.Run(new RechnerWnd());
}
```

**/Listing 4**

Damit ist der Client-seitige Mechanismus soweit komplett, um einzelne fehlende Assemblies von einem entfernten Server laden zu können. Wenn die CLR ein benötigtes Assembly nicht auflösen kann, wird die Anfrage an die Methode *MyResolver* weitergeleitet. Diese delegiert das Problem zum Server und fordert von ihm das Assembly an. Falls es dieser vorhält, schickt er es als Byte-Stream über die Leitung zur Methode *MyResolver*, die den Stream in ein Assembly lädt und an die CLR weiterreicht. Dabei hilft die statische Methode *Assembly.Load*, die in einer Variante als Parameter einen Byte-Stream erwartet.

Falls sich das Assembly nicht auflösen lässt, teilen wir das der Laufzeitumgebung mit, indem wir die Methode mit der Null-Referenz zurückkehren lassen. In diesem Fall versucht die CLR noch einmal, das Assembly unter Einbeziehung aller Rückfallkonzepte zu finden. Dieses Verhalten tritt auf, wenn beispielsweise eine länderspezifische Variante einer System-DLL nicht gefunden wird. Die Laufzeitumgebung fragt erst über unseren Mechanismus nach der DLL. Da wir diese auch nicht liefern können, lädt die Laufzeitumgebung dann die sprachneutrale Variante dieser DLL.

### Der Server

Der Server hat die Aufgabe, alle benötigten Assemblies zur Verfügung zu stellen, die ihm der Administrator bekannt gibt. Falls mehrere in einem Verzeichnis stehen, werden Abhängigkeiten automatisch erkannt und diese Assemblies dann gleich mit geladen.
Technisch unterteilt sich das Ganze in den eigentlichen Server, der die Assemblies vorhält und auf die Client-Anfragen wartet, sowie in eine Host-Applikation, mit deren Hilfe die nötigen Assemblies in den Server geladen werden. Der Server-Teil ist einfach gehalten (Listing 5).

**Listing 5**

```
public interface IResolver
{
    byte[] GetAssembly(string name);
}

public class Resolver : MarshalByRefObject, IResolver
{
    private static Assemblies _assemblies= null;

    public static Assemblies Assemblies
    {
        get
        {
            return _assemblies;
        }
        set
        {
            _assemblies= value;
        }
    }

    public byte[] GetAssembly(string name)
    {
        if(_assemblies != null && _assemblies.References.Contains(name))
        {
            return (byte[])_assemblies.References[name];
        }

        return null;
    }
}

public class Assemblies
{
    public Hashtable References= new Hashtable();
    public Hashtable Warnings= new Hashtable();
}
```

**/Listing 5**

Die öffentliche Schnittstelle enthält lediglich eine Methode zum Holen eines Assemblies. Sie wurde in ein Interface ausgelagert, das in einer eigenen DLL untergebracht ist. Diese Aufteilung bewahrt uns davor, den gesamten Server auf dem Client installieren zu müssen. [2]Den Löwenanteil der Arbeit erledigt die Server-Klasse, die für die Speicherung und Weitergabe der Assemblies zuständig ist. Sie beinhaltet die Implementierung des Interface zum Client und zusätzlich ein Property, das es der Host-Applikation gestattet, die Assemblies im Server abzulegen.

Daneben gibt es noch eine einfache Hilfsklasse, welche die Hash-Tabellen für die Assemblies und zugehörige Warnungen zusammenfasst. Die Warnungen werden nur gespeichert, um den Benutzer nicht mit doppelten Meldungen zu langweilen.

Abschließend ist noch die Host-Applikation erforderlich, welche die Assemblies in den Server lädt. Sie ist  auf beiliegender Heft-CD zu finden. Hier beschränken wir uns auf den Teil, der die Abhängigkeiten automatisch auflöst (Listing 6).

**Listing 6**

```
private void GetAssembly(string strDir, string strFile)
{
    string strPath= strDir+ "\\"+ strFile;
    byte[] rawAssembly;
    try
    {
        rawAssembly= loadFile(strPath);
    }
    catch
    {
        if(!_assemblies.Warnings.Contains(strFile))
        {
            _assemblies.Warnings.Add(strFile, strFile);
            _tbStatus.Text+= "WARNUNG: Assembly "+ strFile+
                " konnte nicht geladen werden.\r\n";
        }
        return;
    }
    Assembly asm= Assembly.Load(rawAssembly);

    _assemblies.References.Add(asm.GetName().FullName, rawAssembly);
    _tbStatus.Text+= "Assembly: "+ strFile+ " hinzugefügt.\r\n";

    // referenzierte Assemblies lesen
    AssemblyName[] ans= asm.GetReferencedAssemblies();

    foreach(AssemblyName an in ans)
    {
        try
        {
            // Haben wir das Modul schon geladen?
            string strNew= an.Name+ ".dll";
            if(!_assemblies.References.Contains(an.FullName))
            {
                GetAssembly(strDir, strNew);
            }
        }
        catch
        {
        }
    }
}
```

**/Listing 6**

Nachdem der Benutzer über einen Dialog ein Assembly ausgewählt hat, wird dieses und gegebenenfalls auch weitere Assemblies in den Speicher geladen und dem Server hinzugefügt. Diese Arbeit erledigt die Methode *GetAssembly*.

Zuerst wird das ausgewählte Assembly geladen und gespeichert. Die Methode *GetReferencedAssemblies* der Klasse *Assembly* ermittelt dann alle referenzierten Assemblies. Der Server versucht nun, diese zu laden – natürlich auch unter Berücksichtigung weiterer Referenzen. Dazu wird die Methode *GetAssembly* mehrfach rekursiv aufgerufen. Bei den nicht im angegebenen Verzeichnis vorhandenen Assemblies – dazu gehören beispielsweise die der .NET-Laufzeitumgebung – wird der Ladevorgang fehlschlagen. Dies wird dem Benutzer als Warnung mitgeteilt, der dann entscheidet, ob daraus Probleme resultieren. Nun muss der Server nur noch warten, bis ein Client ein Assembly anfordert.

### Ressourcen

Einen analogen Ansatz gibt es für die Suche nach Ressourcen, wofür das *ResourceResolve*-Event verwendet wird. Auch bei diesem Event wird als Ergebnis von der Laufzeitumgebung ein Assembly (mit der passenden Ressource) erwartet, und das prinzipielle Vorgehen entspricht dem des AssemblyResolve-Event.

## Fazit

Die AppDomain ist also nicht nur eine unbedeutende Klasse, welche die .NET-Laufzeitumgebung braucht, um ihre Applikationen zu starten. Sie bietet vielmehr interessante Möglichkeiten, um Programme besser an die Erfordernisse von Projekten anzupassen.

Der hier vorgestellte Lösungsansatz zum Schutz des geistigen Eigentums lässt sich je nach Bedarf beliebig variieren: Die Assemblies können beispielsweise von einem Web-Service kommen oder aus einer Ressource geladen werden. Eventuell werden sie verschlüsselt und können nur auf bestimmten Rechnern (zum Beispiel abhängig von einer Host-ID) entschlüsselt werden. Mit der AppDomain kann der Entwickler also seiner Fantasie freien Lauf lassen.

*Dipl-Ing. Frank Listing ist Trainer und Projektcoach bei der Münchener MicroConsult GmbH (www.microconsult.com), die seit 30 Jahren Hard- und Software-Projekte in der Industrie mit Training, Coaching und Engineering begleitet. Seine Schwerpunkte liegen auf COM, .NET, Software-Engineering und Test. Der Autor hat jahrelange Projekterfahrung in der Windows-Entwicklung (COM, MFC, C++ und .NET) und ist erreichbar unter f.listing@microconsult.com.*

### Links & Literatur

[1] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemappdomainmemberstopic.asp

[2] http://www.thinktecture.com/Resources/default.html