CUTTING EDGE

Managing Dynamic Content Delivery In Silverlight, Part 2

Dino Esposito

Contents

Last month I discussed how to serve a Silverlight application some dynamically generated content at startup and even in an on-demand scenario. Many examples exist out there to show you how to use the WebClient class and its asynchronous call model to download a URL-based resource. In particular, I focused on what it takes to download a XAP package that includes XAML and managed code (see the January 2009 installment of Cutting Edge.

The basic idea is that you download a zipped stream and then extract any assembly you need. Next, you instantiate any class you need that is contained in the assembly. The class is a XAML user control—an entire piece of a XAML visual tree—that you can then append to any placeholders in the current XAML document object model (DOM).

To the browser, any downloaded XAP resources are totally indistinguishable from any other type of resource. Hence, the browser caches a XAP package just as any other downloaded resource. This built-in mechanism offers a first level of optimization that saves you repeated round-trips to get the same package over and over again. The WebClient class—the core of the downloader component discussed in last month's column—is based on the browser's connectivity engine and doesn't download resources available locally and not expired yet.

At the end of the day, you have a downloader component to defer the loading of any external packages you may need. In addition, you also have free caching capabilities for any dynamically downloaded resources. Among other things, you might like to add a permanent local cache of packages that is also able to deal with dynamic changes to the visual tree. Let's see how to achieve this.

Motivation for a Permanent Cache

When it comes to dynamic Silverlight content, there are two main aspects over which you might want to gain some more control.

The first is the expiration policy of the downloaded content. You might want to control exactly when a given package expires and needs to be downloaded again. Furthermore, you might want to tie the expiration to some external event such as a particular user action or changes to other cached resources. If you know how the ASP.NET Cache works, you know what I mean.

The ASP.NET Cache, in fact, allows you to cache data and give each cached item its own expiration policy—based on file changes, date/time, or even changes to other cached items. A similar engine doesn't exist in Silverlight 2, but large, dynamic, and highly customizable applications would gain significant benefits from it.

The second aspect of standard Silverlight resource caching that you might want to change regards the exposure of your packages to the user's activity. In other words, any XAP package saved in the browser's cache is at the mercy of the user. If the user clears the cache acting on the browser's interface, all of your XAP packages will be inevitably lost.

An application-managed permanent cache addresses both issues. XAP packages stored in this permanent cache would not be affected by the user clearing the browser's cache. To permanently store Silverlight XAP packages, you need to gain access to the local file system. For security reasons, Silverlight doesn't let applications access the entire local file system. However, the Isolated Storage API is here to help. For more on Silverlight security, see "CLR Inside Out: Security in Silverlight 2."

Foundations of Isolated Storage

Isolated storage was not specifically created for Silverlight. Isolated storage has been part of the Microsoft .NET Framework since version 1.0. Aimed at partially trusted applications, isolated storage enables such apps to store data on the local computer in full respect of any ongoing security policy. A classic fully trusted .NET application probably has no need to ever go through the isolated storage layer to save its own data, but for a partially trusted application, isolated storage is the only option for saving data on the client.

From a Silverlight perspective, isolated storage is a powerful tool and the only possible way to persist relatively large chunks of data in a cross-browser manner and without any of the restrictions that affect, for example, HTTP cookies. It is important to understand this point: in Silverlight, isolated storage is the only possibility you have to cache data on the local machine. If a Silverlight application needs to save some data—any kind of data—locally, then it can only do so through isolated storage. In addition, with isolated storage each application can keep its own data isolated from any other applications or from any other applications outside the site.

If you want a general, .NET-oriented introduction to isolated storage and its most common usage scenarios, you should read the .NET Framework Developer's Guide to Isolated Storage. The article mentions a couple of scenarios where using isolated storage is not appropriate. In particular, the guidelines say that you should not be using isolated storage to store sensitive data, code, or configuration settings (other than user preferences). Such guidelines stem from a general security awareness and do not necessarily imply any hazards inherent in the use of isolated storage.

So, can you safely store in XAP packages you download into Silverlight isolated storage? In Silverlight, unlike in the desktop CLR, any piece of executable code is untrusted by default and is not allowed to invoke critical methods or elevate privileges of the calling stack. In Silverlight, whatever code you store for later execution will be unable to do anything dangerous. This is no more risky than executing any other piece of Silverlight code. By building a permanent cache of Silverlight packages, you end up storing locally a segment of the Silverlight application you are consciously executing.

In Silverlight, the role of isolated storage is analogous—as far as persistence is concerned—to the role of HTTP cookies in classic Web applications. In Silverlight, you should look at isolated storage as a set of larger cookies that can contain any sort of data, including executable code. In this case, though, the Silverlight core CLR provides protection. According to the Silverlight security model, in fact, the core CLR will throw an exception any time the application code wants to execute critical methods. Unlike HTTP cookies, isolated storage in Silverlight is not linked to network I/O and no content is transmitted with requests.

Data in isolated storage is isolated by application and no other Silverlight application can access the store. The data is stored on the local file system, however, so an administrator of the machine is certainly able to access it.

Again, the overall model is not really different from what happens with HTTP cookies. An administrator can always locate and even alter the content of cookies. If it's worthwhile in your context, you can use encryption to add another level of data protection.

If you're still worried about having some downloaded executable code lying around your machine, you should refresh your understanding of the Silverlight security model. In brief, the Silverlight core CLR throws an exception any time application code attempts to execute a critical method. In the Silverlight Base Class Library (BCL), methods and classes that perform operations requiring high privileges are marked with a special SecurityCritical attribute. Note that this is the case with most of the content of the System.IO namespace.

The Silverlight security acknowledges that some platform classes may need to place safe calls to critical methods. Such classes and methods are then marked with the SecuritySafeCritical attribute. This is the case with classes in the System.IO.IsolatedStorage API (see **Figure 1**). The key point about Silverlight security is that no piece of application code can ever be marked with the SecurityCritical or SecuritySafeCritical attribute. This attribute is reserved for classes in assemblies digitally signed by Microsoft and loaded into memory from the Silverlight installation directory.

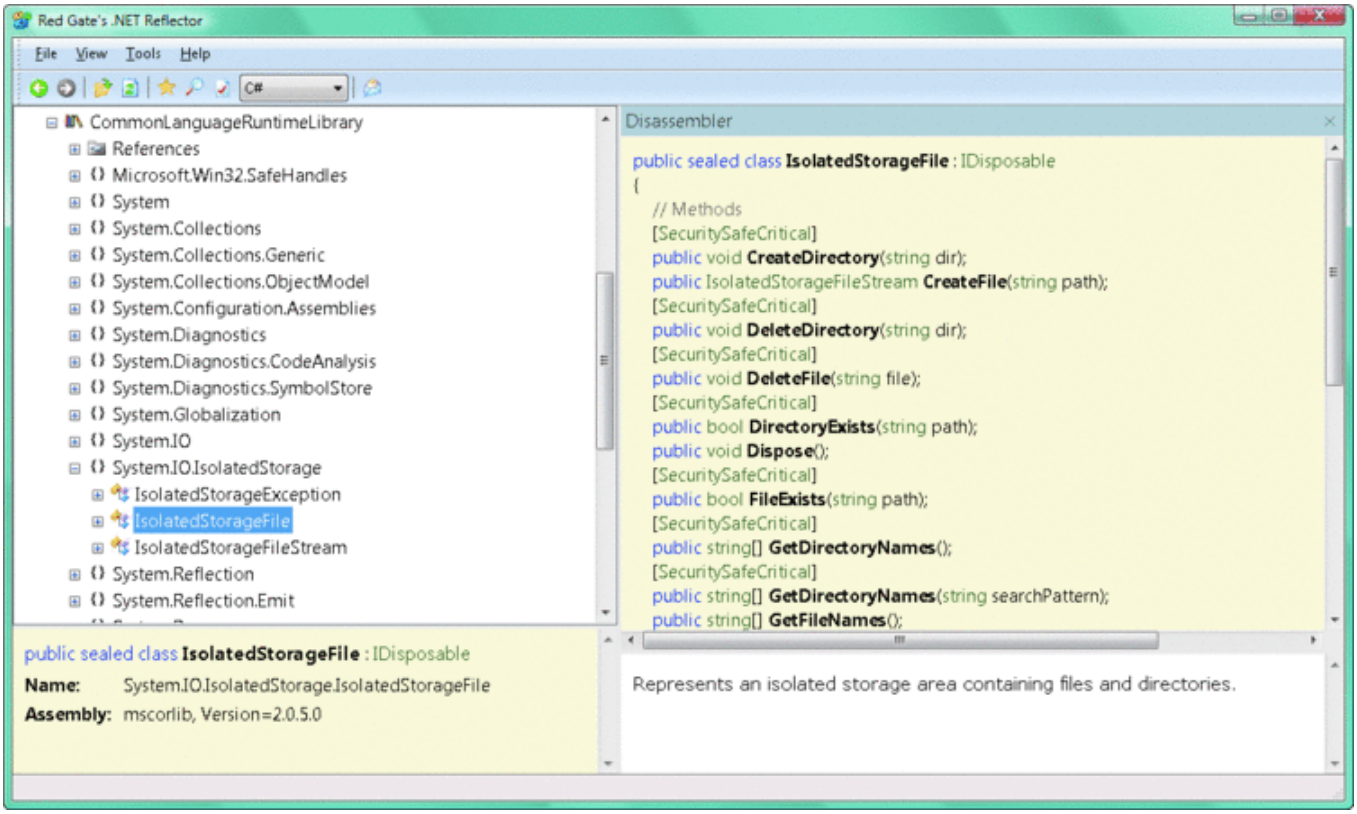

Figure 1 **Browsing inside the Isolated Storage API**

As you can see, even in the very unfortunate (and unlikely) circumstance that some malicious guy penetrates your computer and replaces downloaded Silverlight content, the damage is limited to regular operations executable in transparent mode.

Isolated Storage API

The Silverlight BCL comes with its own implementation of the isolated storage that is tailor-made for the Web scenario. Isolated storage provides access to a subtree of the whole local file system, and no method or property ever allows running code to figure out where the file store is physically located on the user machine. A Silverlight application is not allowed to use absolute file system paths through isolated storage. Likewise, drive information is not available and is unsupported and the same applies to relative paths that contain ellipsis like this one:

Copy Code

```
\..\..\myfile.txt
```

The isolated storage subtree is rooted in a folder located under the current user path. For example, on Windows Vista, the root of the isolated storage folder is located under the Users directory.

A Silverlight application gains access to the application-specific isolated storage entry point through a method call:

Copy Code

```
using (IsolatedStorageFile iso =
       IsolatedStorageFile.GetUserStoreForApplication())
{
  ...
}
```

The static method GetUserStoreForApplication returns the token you use for any further access to the isolated storage. When the first call to GetUserStoreForApplication is made, the application-specific subtree is created, if it doesn't exist already.

The Silverlight isolated storage API supplies classes to work with files and directories within the protected file system subtree. Happily, the list of classes you need to know is fairly short; you'll find them listed in **Figure 2**.

Figure 2 Classes in the Isolated Storage API

Class	Description
IsolatedStorageException	Represents the exception thrown when an operation in isolated storage fails.
IsolatedStorageFile	Represents an isolated storage area containing files and directories.
IsolatedStorageFileStream	Exposes a file within isolated storage.

In the IsolatedStorageFile class, there are a number of methods to create and delete files and directories, to check for the presence of files and directories, and to read and write new files. You employ streams to work with files. Optionally, you can wrap up streams with stream readers, which are much more comfortable objects to work with. **Figure 3** shows a brief example of how to create an isolated storage file using a stream writer.

Figure 3 Creating an Isolated Storage File

Copy Code

```
using (IsolatedStorageFile iso =
      IsolatedStorageFile.GetUserStoreForApplication())
{
    // Open or create the low level stream
    IsolatedStorageFileStream fileStream;
    fileStream = new IsolatedStorageFileStream(fileName,
        FileMode.OpenOrCreate, iso);

    // Encapsulate the raw stream in a more convenient writer
    StreamWriter writer = new StreamWriter(stream);

    // Write some data
    writer.Write(DateTime.Now.ToString());

    // Clean up
    writer.Close();
    stream.Close();
}
```

Once you wrap a low-level stream in a more convenient stream writer or reader, the code you use to write (or read) some data is nearly identical to the code you would use in a classic .NET application. Let's see how to take advantage of the isolated storage API to save any downloaded XAP package locally and reload it later.

Building a Permanent Cache of Packages

In last month's column, I used a downloader wrapper class to hide some of the boilerplate code you need to download a XAP package and extract assemblies and other resources. The Downloader class, however, is not just a helper class. Conceptually, it represents a nontrivial piece of logic that you might want to isolate from the rest of the application code for a number of reasons.

The first reason that springs to mind is testability. By exposing the functionality of a downloader component through an interface, you gain the possibility of quickly and effectively mocking the downloader for testing purposes. In addition, an interface represents the tool you leverage to replace a simpler downloader with a

more sophisticated one that, perhaps, just happens to support package caching. **Figure 4** shows the architecture of the design you should aim for.

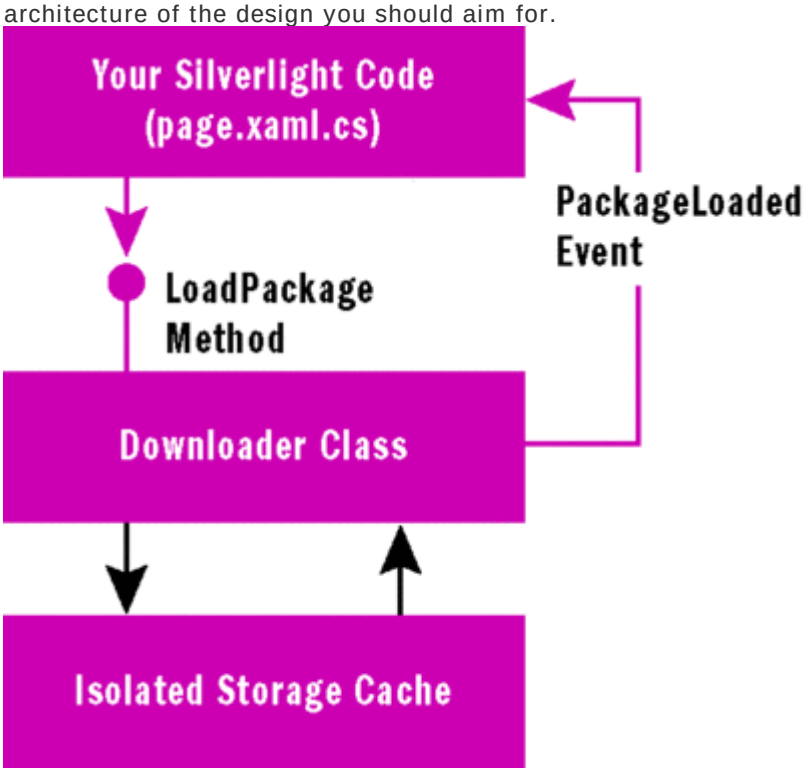

Figure 4 **The Downloader Component and the Rest of the Application**

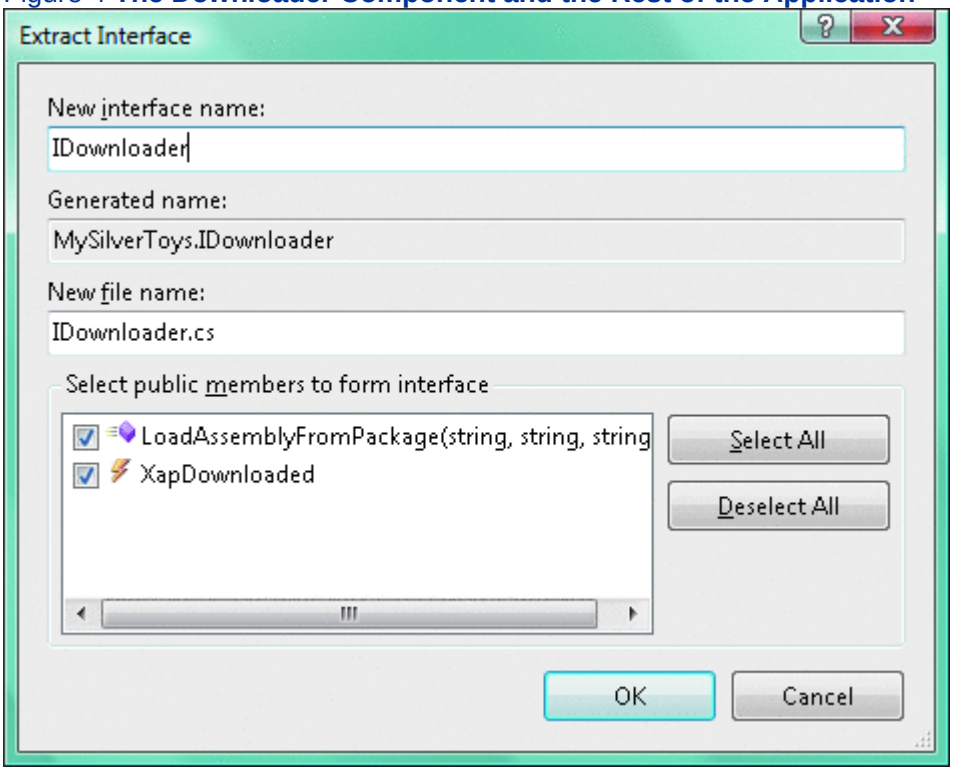

Figure 5 **Extracting an Interface**

In last month's source code, the Downloader class was a monolithic piece of code. For a more flexible design, let's extract an interface out of it. As **Figure 5** shows, Visual Studio has a context menu that,

although not as rich as that of commercial refactoring tools, does provide some help with the extraction of an interface out of a class.

Now that the core of your Silverlight application talks to the IDownloader interface, all the logic for package caching must go only inside the actual downloader class:

Copy Code

```
interface IDownloader
{
    void LoadPackage(string xapUrl, string asm, string cls);
    event EventHandler<Samples.XapEventArgs> XapDownloaded;
}
```

In particular, the LoadPackage method will be rewritten to incorporate the logic that would check for the existence of the specified XAP package within the isolated storage and download it from the Internet otherwise. **Figure 6** shows a large section of the code for the Downloader class. The method first attempts to get the stream for the XAP package from the internal cache. If this attempt fails, the method proceeds and downloads the package from the host server. (This is just what I discussed in detail last month.)

Figure 6 Cache Support for the Downloader Component

Copy Code

```
public void LoadPackage(string xap, string asm, string cls)
{
    // Cache data within the class
    Initialize(xap, asm, cls, PackageContent.ClassFromAssembly);

    // Have a look in the cache
    Stream xapStream = LookupCacheForPackage();
    if (xapStream == null)
        StartDownload();
    else
    {
        // Process and extract resources
        FindClassFromAssembly(xapStream);
    }
}

protected Stream LookupCacheForPackage()
{
    // Look up the XAP package for the assembly.
    // Assuming the XAP URL is a file name with no HTTP information
    string xapFile = m_data.XapName;

    return DownloadCache.Load(xapFile);
}

protected void StartDownload()
{
    Uri address = new Uri(m_data.XapName, UriKind.RelativeOrAbsolute);
    WebClient client = new WebClient();

    switch (m_data.ActionRequired)
    {
        case PackageContent.ClassFromAssembly:
            client.OpenReadCompleted +=
                new OpenReadCompletedEventHandler(OnCompleted);
            break;
        default:
            return;
    }
```

```
    client.OpenReadAsync(address);
}

private void OnCompleted(object sender, OpenReadCompletedEventArgs e)
{
    // Handler registered at the application level?
    if (XapDownloaded == null)
        return;

    if (e.Error != null)
        return;

    // Save to the cache
    DownloadCache.Add(m_data.XapName, e.Result);

    // Process and extract resources
    FindClassFromAssembly(e.Result);
}

private void FindClassFromAssembly(Stream content)
{
    // Load a particular assembly from XAP
    Assembly a = GetAssemblyFromPackage(m_data.AssemblyName, content);

    // Get an instance of the specified user control class
    object page = a.CreateInstance(m_data.ClassName);

    // Fire the event
    XapEventArgs args = new XapEventArgs();
    args.DownloadedContent = page as UserControl;
    XapDownloaded(this, args);
}
```

In Silverlight, downloading is an asynchronous process, so the internal method StartDownload raises a "completed" event when the package is entirely available on the client. The event handler first saves the content of the XAP package stream to a local file and then extracts resources from it. Note that in the sample code I'm only extracting assemblies; in a more general component you might want to extend caching capabilities to any other type of resource such as XAML for animations, images, or other auxiliary files.

The LoadPackage method in the Downloader class is used to download Silverlightuser controls to insert in the current XAML tree. Because the XAP package is a multi-file container, you have to specify which assembly contains the user control and the class name. The code in **Figure 6** simply extracts the specified assembly from the package, loads it into the current AppDomain, and then creates an instance of the specified contained class.

What if the assembly has some dependencies? The code in **Figure 6** just doesn't cover this scenario. As a result, if the assembly passed as an argument to LoadPackage has a dependency on another assembly (even in the same XAP package), you get an exception as soon as the execution flow reaches a class in the dependency assembly. The point is that all assemblies in the package should be loaded in memory. For this to happen, you need to access the manifest file, read about deployed assemblies, and process them. **Figure 7** shows how to load into memory all assemblies referenced in the manifest file.

Figure 7 Loading All Assemblies in the Manifest

Copy Code

```
private Assembly GetAssemblyFromPackage(
     string assemblyName, Stream xapStream)
{
    // Local variables
    StreamResourceInfo resPackage = null;
    StreamResourceInfo resAssembly = null;
```

```
    // Initialize
    Uri assemblyUri = new Uri(assemblyName, UriKind.Relative);
    resPackage = new StreamResourceInfo(xapStream, null);
    resAssembly = Application.GetResourceStream(
                              resPackage, assemblyUri);

    // Extract the primary assembly and load into the AppDomain
    AssemblyPart part = new AssemblyPart();
    Assembly a = part.Load(resAssembly.Stream);

    // Load other assemblies (dependencies) from manifest
    Uri manifestUri = new Uri("AppManifest.xaml", UriKind.Relative);
    Stream manifestStream = Application.GetResourceStream(
        resPackage, manifestUri).Stream;
    string manifest = new StreamReader(manifestStream).ReadToEnd();

    // Parse the manifest to get the list of referenced assemblies
    List<AssemblyPart> parts = ManifestHelper.GetDeploymentParts(manifest);

    foreach (AssemblyPart ap in parts)
    {
        // Skip over primary assembly (already processed)
        if (!ap.Source.ToLower().Equals(assemblyName))
        {
            StreamResourceInfo sri = null;
            sri = Application.GetResourceStream(
                resPackage, new Uri(ap.Source, UriKind.Relative));
            ap.Load(sri.Stream);
        }
    }

    // Close stream and returns
    xapStream.Close();
    return a;
}
```

The manifest file is an XML file, as shown here:

Copy Code

```
<Deployment EntryPointAssembly="More" EntryPointType="More.App"
            RuntimeVersion="2.0.31005.0">
  <Deployment.Parts>
    <AssemblyPart x:Name="More" Source="More.dll" />
    <AssemblyPart x:Name="TestLib" Source="TestLib.dll" />
  </Deployment.Parts>
</Deployment>
```

To parse this file, you can use LINQ-to-XML. The source code contains a sample ManifestHelper class with a method that returns a list of AssemblyPart objects (see **Figure 8**). It is worth noting that in Beta versions of Silverlight 2, you could use the XamlReader class to parse the manifest file to a Deployment object:

Copy Code

```
// This code throws in Silverlight 2 RTM
Deployment deploy = XamlReader.Load(manifest) as Deployment;
```

Figure 8 Parsing the Manifest Using LINQ-to-XML

Copy Code

```
public class ManifestHelper
{
   public static List<AssemblyPart> GetDeploymentParts(string manifest)
   {
```

```
      XElement deploymentRoot = XDocument.Parse(manifest).Root;
      List<AssemblyPart> parts =
          (from n in deploymentRoot.Descendants().Elements()
           select new AssemblyPart() {
                Source = n.Attribute("Source").Value }
          ).ToList();

          return parts;
   }
}
```

In the release version, the Deployment object has been transformed into a singleton and subsequently it can't be used to load assemblies dynamically. The XML in the manifest therefore must be parsed manually.

Expiration Policies

As implemented thus far, the assembly cache is permanent and there's no way for the user to get updated packages. The only possible workaround is to have a machine administrator locate isolated storage files for the application and delete them manually through Windows Explorer.

Let's see what it would take to add a simple expiration policy that makes any cached XAP file obsolete after a given amount of time after download. The proper place to set an expiration policy is the DownloadCache class, as in **Figure 6**. When you add a XAP file to the cache, you need to store some information about the download time. When you access the cache to pick up a package, you should verify whether the package is expired (see **Figure 9**).

Figure 9 Test for Expiration

Copy Code

```
public bool IsExpired(string xapFile)
{
    bool expired = true;
    if (m_ItemsIndex.ContainsKey(xapFile))
    {
        DateTime dt = (DateTime)m_ItemsIndex[xapFile];

        // Expires after 1 hour
        expired = dt.AddSeconds(3600) < DateTime.Now;
        if (expired)
            Remove(xapFile);
    }

    return expired;
}
```

The implementation of such an apparently simple algorithm is hindered by one notable fact. In Silverlight, you have no way to access file attributes like last update or creation time. This means that you are responsible for managing time information. In other words, when you add a XAP to the cache you also create an entry in some custom and persistent dictionary that tracks when the package was downloaded. Needless to say, this information has to be persisted in some way to the isolated storage. **Figure 10** illustrates this concept.

Figure 10 Persisting Download Details to Isolated Storage

Copy Code

```
public static Stream Load(string file)
{
    IsolatedStorageFile iso;
    iso = IsolatedStorageFile.GetUserStoreForApplication();

    if (!iso.FileExists(file))
    {
        iso.Dispose();
        return null;
```

```
    }

    // Check some expiration policy
    CacheIndex m_Index = new CacheIndex();
    if (!m_Index.IsExpired(file))
        return iso.OpenFile(file, FileMode.Open);

    // Force reload
    iso.Dispose();
    return null;
}
```

CacheIndex is a helper class that uses the Silverlight native application settings API to persist a dictionary of XAP names and download times to the isolated storage. The m_ItemIndex member is a plain Dictionary object instantiated in CacheIndex constructor, as shown in **Figure 11**.

Figure 11 The CacheIndex Class

Copy Code

```
public class CacheIndex
{
    private const string XAPCACHENAME = "XapCache";
    private Dictionary<string, object> m_ItemsIndex;
    public CacheIndex()
    {
      IsolatedStorageSettings iss;
      iss = IsolatedStorageSettings.ApplicationSettings;
      if (iss.Contains(XAPCACHENAME))
         m_ItemsIndex = iss[XAPCACHENAME] as Dictionary<string, object>;
      else
      {
         m_ItemsIndex = new Dictionary<string, object>();
         iss[XAPCACHENAME] = m_ItemsIndex;
         iss.Save();
      }
   }
  ...
}
```

ApplicationSettings is a very nice feature of Silverlight 2. It basically consists of a string/object dictionary and is automatically read from storage upon application loading and saved back to the storage upon shutdown. Any (serializable) object you add to the dictionary is automatically persisted.

By creating a XAPCACHENAME entry in the dictionary, you arrange to persist the content of the XAP dictionary. The XAP dictionary contains one entry for each downloaded package paired with the time of the download, as you see in **Figure 12**. Note that the ApplicationSettings API also offers you a Save method to force persistence before application shutdown.

Figure 12 Adding Download Information

Copy Code

```
public void Add(string xapFile)
{
    m_ItemsIndex[xapFile] = DateTime.Now;
    IsolatedStorageSettings iss;
    iss = IsolatedStorageSettings.ApplicationSettings;
    iss.Save();
}

public void Remove(string xapFile)
{
    m_ItemsIndex.Remove(xapFile);
    IsolatedStorageSettings iss;
```

```
    iss = IsolatedStorageSettings.ApplicationSettings;
    iss.Save();
}
```

Putting It All Together

All the changes and details discussed so far occur behind the curtain of a public class—the Downloader class. By incorporating this class in your Silverlight application, you gain multiple levels of caching in a single shot. With proper coding you can cache downloaded user controls for the duration of the application session. If you download content that goes to, say, a tab item, you can then hide and show the tab item repeatedly without the need to download the package over and over again.

If you download through the WebClient class (as in last month's column), you pass through the browser engine and get browser-level caching. Any downloaded XAP package lives on the user's machine until the user clears the browser cache. Finally, the Downloader class you get with this column supports a permanent cache through isolated storage. This means that any time you download via WebClient, the XAP package is also saved to the local storage.

The Downloader class, however, also offers to pick up the XAP file from the storage if a valid package can be found. Note that this works across application sessions. You download the package once, you work and then you shut down the application. When you resume, the package is reloaded from the storage if it has not expired. What if the package expires? In this case, the downloader passes through WebClient. But at this point, WebClient might just return a previously browser-cached copy of the same package.

This is by design. To bypass the browser-level caching, you have to disable it in the original HTTP request, as discussed last month. You must cache properties at the page level or get the package through an HTTP handler where you can set expiration policies more precisely without affecting other resources that go with the page.

Some Final Notes

Caching the XAP package doesn't mean caching individual resources such as DLLs, XAML animations, or multimedia content. In the current implementation, resources are extracted from the XAP package every time they are used. However, this is an aspect you can further improve on with the Downloader class. Likewise, the package delivers a user control but doesn't track changes the user can force on its user interface. Tracking dynamic changes to the XAML tree is another topic and deserves an article of its own.

There are two ways to access a Silverlight private file system on the local user's machine. In all the code snippets for this column, I used the method GetUserStoreForApplication on the IsolatedStorageFile class. This method returns a token to access a section of the file system that is isolated per application, meaning that all and only the assemblies associated with an application will use the same store. You can also select a store and share it across all applications hosted on the same site. In this case, you get the token through the method GetUserStoreForSite.

Note too that local storage can be administered through the Silverlight configuration dialog box (right-click on a Silverlight app) and even disabled completely. In this case, when attempting to get the token, an exception will be raised. A disk quota also applies to local storage on a per-domain basis; its default value is 1MB. Remember this when planning for a permanent Silverlight cache.

Send your questions and comments for Dino to cutting@microsoft.com.

Dino Esposito is an architect at IDesign and the co-author of *Microsoft .NET: Architecting Applications for the Enterprise* (Microsoft Press, 2008). Based in Italy, Dino is a frequent speaker at industry events worldwide. You can join his blog at weblogs.asp.net/despos