4
The Caching Application Block

MANY ENTERPRISE APPLICATIONS focus on moving data and presenting data to users in unique and interesting ways. Yet, all of this data does not necessarily need to be retrieved from a backend database for every request. Data that is semi-static, nontransactional, and consumed frequently or is expensive to create, obtain, or transform is ideal for caching. Caching helps not only reduce the amount of data that is transferred between processes and computers, but it also helps reduce the amount of data processing that occurs in a system and the number of disk access operations that must occur. Unfortunately, leveraging caching techniques to improve the performance and scalability metrics in enterprise applications is an important area that is too often overlooked.

For application architects, it is not good enough to design applications that solve specific business problems. An application that "does the job" but does not perform or scale well will eventually see little use. Even if users are attracted to the application at first, as performance degrades and the application is deemed unreliable, users will turn away. Unless thorough stress testing is performed, these types of problems rarely show themselves early on. It is the architect's responsibility to ensure that applications and systems are designed to meet performance, scalability, and availability needs.

It would be ideal to have service-level agreements in place that detail the metrics that need to be met for a specific application or system; however,

reality does not always match ideology. It is often the case that systems and applications must be designed without specific information about how the application needs to perform or scale. This, however, is not an excuse to design a system that does not perform or scale well. Architects must strive for a design that overcomes these challenges. It is important to remember that caching isn't something that can typically be added to an application at any point in the development cycle; the application should be designed with caching in mind.

The Microsoft patterns & practices team has published a lot of excellent information on caching best practices, most notably the *Caching Architecture Guide for .NET Framework Applications.*[1] The section in the chapter that covers the design of the Caching Application Block details how the recommendations in this guide are core to the design of Enterprise Library's Caching Application Block. The chapter describes how the application block has been designed for extensibility and provides examples for how to extend it. It also shows how to configure and develop an application so that it can benefit from the features of the Caching Application Block.

Note that much of the information in this chapter is not new; rather, it is a combination of parts of the *Caching Architecture Guide for .NET Framework Applications* document and the Enterprise Library documentation for the Caching Application Block. Most of the new information in this chapter is where I show how to extend the Caching Application Block by way of a custom Cache Storage Provider, expiration policy, and callback. I don't repeat all the information found in these guides, but I focus on the parts that are specific to Enterprise Library's implementation for caching.

What Is the Caching Application Block?

The Enterprise Library's Caching Application Block is an implementation of the recommendations that are put forth in the *Caching Architecture Guide for .NET Framework Applications.* Its design contains all of the fundamental elements found in the Solution Blueprint suggested in this guide; namely,

1. Found at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/CachingArch.asp.

providing implementations for CacheManagers, a cache service, and cache storage. Furthermore, the design of the Caching Application Block provides extension points that allow an enterprise to add new implementations for certain critical areas.

For example, if the cache StorageProviders (aka BackingStores) that ship with the Caching Application Block are not sufficient for the needs of a particular enterprise, a new one can be developed and "plugged in" so that it is as easy to use as the ones that ship with the block. Additionally, a simple and consistent programming interface is exposed that allows the code for an application to be written so that it is agnostic as to the type of BackingStore that is used. This allows the code for an application to remain unchanged if a modification needs to be made to the BackingStore that is used for caching. Overall, the major objective for the Caching Application Block is to provide a set of classes and interfaces that make it easy for an application to cache data to help tune that application's performance, scalability, and availability.

Performance

By storing data as close as possible to the consumer of the data, repetitive data creation, processing of the data, and data retrieval can be avoided. Reference data like countries and states are excellent candidates for information that should be cached, because this type of information rarely changes. Therefore, it can be retrieved from the backend data source less frequently and cached on an application server or Web server. This reduces or eliminates the need to make multiple roundtrips to a database to retrieve this type of data as well the need to recreate the same data for each request. Eliminating these types of activities can dramatically improve an application's performance.

Scalability

Often the same data, business functionality, and user interface fragments are required by many users and processes in an application. For example, a combo box that lets users select a specific country in a form could be used by all the users of a Web application regardless of who that user is or even where they are in the Web application. If this information is processed for

each request, valuable resources are wasted recreating the same output. Instead, the page fragment can be stored in the ASP.NET output cache and reused for each request. This improves the scalability of the application because as the user base increases, the demand for server resources for these tasks remains constant and the resources that would be used to render these results can now be used for other purposes. Furthermore, this helps scale the resources of the backend database server. By storing frequently used data in a cache, fewer database requests are made, meaning that more users can be served.

Availability

Sometimes the services that provide information to an application may be unavailable. This is very common, for example, in occasionally connected smart client systems. By storing that data in another place, an application may be able to survive system failures such as Web service problems or hardware failures. Of course, this depends a lot on the type and amount of the actual data that is cached and if the application has been designed to cache information specifically to handle availability issues.

It is atypical and often inadvisable to cache *all* the information for an application, especially if that data is not relatively static or is transactional in nature. One exception to this rule, however, is if the application must be designed to be available even when a backend data store is not available. For example, it may be reasonable for the application to cache all of its information because the data store has scheduled periods where it may be offline or connectivity to the data source is unreliable. Each time a user requests information from the data store while it is online, the information can be returned and cached, updating the cache on each request. When the data store becomes unavailable, requests can still be serviced using the cached data until the data store comes back online.

Why Not Use the ASP.NET Cache?

We should. The .NET Framework includes support for caching in Web applications and Web services with the `System.Web.Caching` namespace. It should still be used to cache information in a Web application, especially when it comes to page and page fragment caching. However, there are

other scenarios in which a caching mechanism that is agnostic to the runtime environment would be a valuable addition to the application to increase performance and availability. The Caching Application Block is not a replacement for the ASP.NET cache; it should be used in situations where the ASP.NET cache is not an ideal fit. The Caching Application Block is a good choice for the following circumstances.

- For situations that require a consistent form of caching across different application environments. For example, it is a good idea to design the data layers or business layers of enterprise applications so that they can be used independently of the application environment in which they run; that is, the business or data layer can run just as well in a Windows application as it does in a Web service or Web application. Although it is possible to use the ASP.NET cache in non-Web scenarios,[2] it has not been tested and is not supported by Microsoft.
- For smart client applications, Windows Services, and console applications that use locally cached reference data to create requests and support offline operations or need a cache to improve performance.
- For situations that require a configurable and persistent BackingStore. The Caching Application Block supports both isolated storage and database BackingStores. This allows cached data to survive application restarts. Developers can create additional BackingStore providers and add them to the Caching Application Block using its configuration settings. The application block can also symmetrically encrypt a cache item's data before it is persisted to a BackingStore.
- For situations that need the cache to be highly configurable so that changes to the cache configuration settings will not require application source code changes. Developers first write the code that uses one or more named caches, and then system operators and develop-

2. This is true for the ASP.NET cache in .NET Framework 1.1. Microsoft has tested and does support using the ASP.NET cache for non-Web scenarios for .NET Framework 2.0.

ers can configure each of these named caches differently using the Enterprise Library Configuration Tool.

- When cache items require a combination of expiration settings for absolute time, sliding time, extended time format (e.g., every evening at midnight), file dependency, or never expired. The ASP.NET cache supports absolute time and sliding time expirations; however, it does not support setting both expirations at one time. Only one type of expiration can be set for a particular cache item. The CacheManager supports setting multiple types of cache item expirations at the same time.

The Previous Version of the Caching Application Block

There are some significant differences between the previous version of the Caching Application Block and the Enterprise Library version.

- First and foremost, Enterprise Library's Caching Application Block is thread-safe. The earlier version of the application block could return incorrect data when multiple threads accessed a single cache item in a short period of time.
- The earlier version supported multiple processes sharing a single cache by way of a Singleton object (via the `SingletonCacheStorage`). The Enterprise Library version supports using a cache in a single application domain only.
- The `ICacheStorage` interface has been replaced with the `IBackingStore` interface. Persistent storage in a database is provided through its dependency on the Data Access Application Block. Isolated storage is supported for persistent storage via the new `IsolatedStorageBackingStore`. The application block does not, however, include support for memory-mapped files.
- The earlier version included the scavenging algorithm as a pluggable provider. In the Enterprise Library version, you must modify the application block source code to change the scavenging algorithm.
- The earlier version included encryption as a pluggable provider. The encryption of cache item data in the Enterprise Library version is provided by the Cryptography Application Block.

The Design of the Caching Application Block

The design goals for a custom cache outlined in the *Caching Architecture Guide for .NET Framework Applications* are to:

- Decouple the front-end application interface from the internal implementation of the cache storage and management functions.
- Provide best practices for a high-performance, scalable caching solution.
- Offer support for cache-specific features, such as dependencies and expirations, and enable the use of custom expiration and dependency implementations.
- Allow for support of cache management features such as scavenging.
- Enable extension points for a custom cache storage solution by implementing the storage class interfaces provided.
- Allow the use of custom cache scavenging algorithms by implementing the classes and interfaces provided.

Except for the last bullet point, the design goals for the Caching Application Block are the same. Additionally, the design goals for the Caching Application Block include providing a caching API that is easy to use, easy to maintain, and easy to configure. Furthermore, the caching solution needs to perform efficiently and must be reliable by ensuring that the BackingStore remains intact if an exception occurs while the cache is being accessed.

One of the most important design goals with this version of the Caching Application Block was to ensure that the cache is thread safe, which helps to ensure that the states of the in-memory cache and the BackingStore remain synchronized. The following sections define the primary classes in the Caching Application Block and explain how they are used to accomplish these design goals. Figure 4.1 provides a high-level overview of many of these classes.

`CacheManager`, `CacheManagerFactory`, and `CacheFactory`

The `CacheManager` class lies at the core of the Caching Application Block and provides the application interface for a single cache. In the Caching

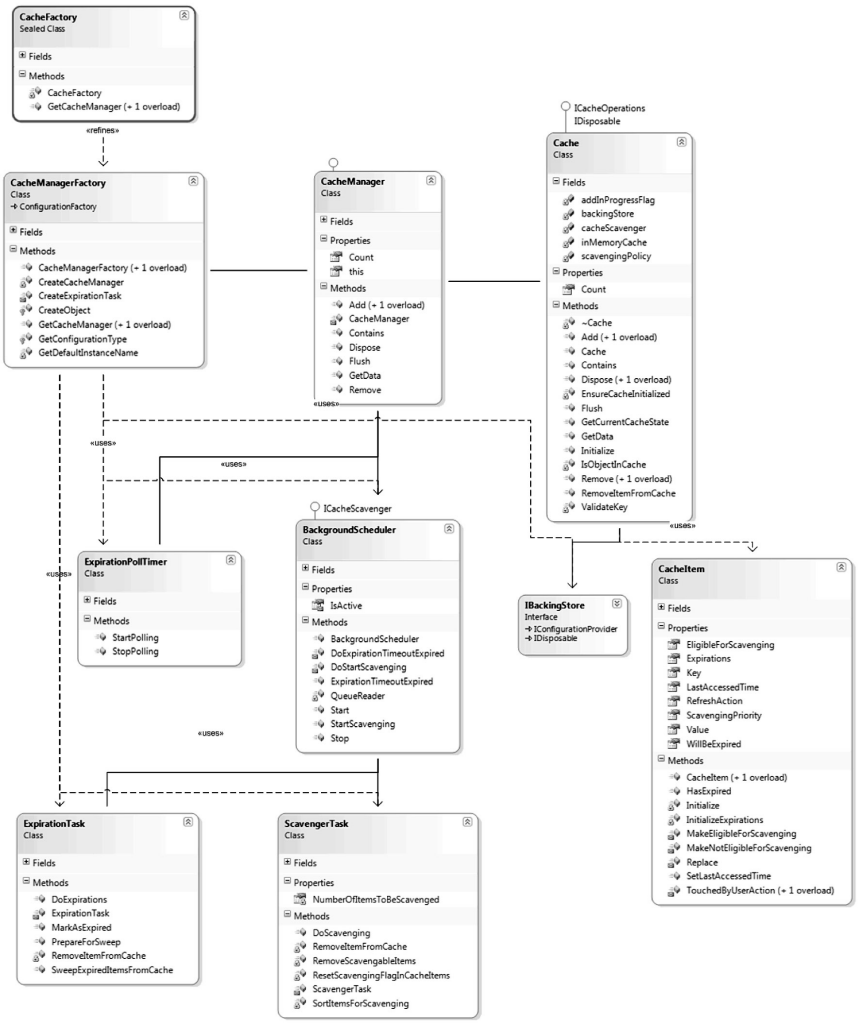

FIGURE 4.1: Design of Enterprise Library's Caching Application Block

Application Block, it is possible to configure and use multiple caches, and thus multiple CacheManagers, in a single application. This is another factor that differentiates the Caching Application Block from the ASP.NET cache; with ASP.NET there can be only one cache. However, it is recommended by the patterns & practices team that when different types of items

(e.g., customer data, countries, zip codes, etc.) are being stored, it is best to store them in different caches—one cache for each type of item. This increases the efficiency of searching when retrieving an item because it reduces the number of items in each cache. In the Caching Application Block, all caching operations occur through the `CacheManager` class. The `CacheManager` class provides all the methods needed to add, retrieve, and remove items from the cache.

All of the Enterprise Library application blocks are designed using the factory design pattern. **Factories** are objects that exist solely to create other objects. In the Caching Application Block, the `CacheManagerFactory` is used to create an instance of a `CacheManager`. The `CacheManagerFactory` uses the features provided by the Configuration Application Block to retrieve configuration information and determine which `CacheManager` should be created. The `CacheManagerFactory` class has two overloads for its `GetCacheManager` method that are used to create a `CacheManager`: one overload takes the name of a `CacheManager` and the other one doesn't take any arguments. The overload that requires an argument will initialize the `CacheManager` with the name that is supplied. The overload that takes no arguments initializes the `CacheManager` that is configured as the `DefaultCacheManager` in the configuration data for this application block. In both cases, the private `CreateCacheManager` method is called to ultimately create the `CacheManager`.

In its `CreateCacheManager` method, the `CacheManagerFactory` first creates a `ScavengingPolicy` and `BackingStore` and uses the instances of these objects to construct the actual Cache object that the `CacheManager` will encapsulate. It then creates instances of the `ExpirationTask` and `ScavengingTask` and uses these instances to create a new `BackgroundScheduler`. The `BackgroundScheduler` is used to initialize the underlying cache.

An `ExpirationPollTimer` is also created, and both the `BackgroundScheduler` and the `ExpirationPollTimer` are started. The instances of the cache, `BackgroundScheduler`, and `ExpirationPollTimer` are passed into the `CacheManager`'s constructor to create the new instance. This new instance is then added to the `HashTable` of `CacheManagers` that the `CacheManagerFactory` manages. Therefore, the `CacheManager` is held in

the scope of the application, and as such, its cache can be accessed from any class or thread concurrently without the need to recreate the `CacheManager` class multiple times.

It is important to note that the `CacheManager` does not hold any state and is simply a front-end interface to the cache. This design allows the `CacheManager` to provide the quickest possible response times to the cache client by performing any operations on the cache metadata after returning the control to the cache client. Because it is such a critical method, I have included the code for the `CreateCacheManager` in Listing 4.1.

LISTING 4.1: The `CreateCacheManager` Method

```
private CacheManager CreateCacheManager(string cacheManagerName)
{
     CacheManager cacheManager =
                    cacheManagers[cacheManagerName] as CacheManager;
     if (cacheManager != null)
     {
          return cacheManager;
     }

     CachingConfigurationView view =
          new CachingConfigurationView(ConfigurationContext);
     CacheManagerData cacheManagerData =
          view.GetCacheManagerData(cacheManagerName);
     CacheCapacityScavengingPolicy scavengingPolicy =
          new CacheCapacityScavengingPolicy(cacheManagerName, view);

     IBackingStore backingStore =
          backingStoreFactory.CreateBackingStore(cacheManagerName);
     Cache cache = new Cache(backingStore, scavengingPolicy);

     ExpirationPollTimer timer = new ExpirationPollTimer();
     ExpirationTask expirationTask = CreateExpirationTask(cache);
     ScavengerTask scavengerTask =
          new ScavengerTask(cacheManagerName, view,
                            scavengingPolicy, cache);
     BackgroundScheduler scheduler =
          new BackgroundScheduler(expirationTask, scavengerTask);
     cache.Initialize(scheduler);

     scheduler.Start();
     timer.StartPolling(new
          TimerCallback(scheduler.ExpirationTimeoutExpired),
          cacheManagerData.ExpirationPollFrequencyInSeconds * 1000);
```

```
     cacheManager = new CacheManager(cache, scheduler, timer);
     cacheManagers.Add(cacheManagerName, cacheManager);
     return cacheManager;
}

[Visual Basic]
Private Function CreateCacheManager(ByVal cacheManagerName As String) _
               As CacheManager

     Dim cacheManager As CacheManager = _
          IIf(TypeOf cacheManagers(cacheManagerName) Is CacheManager,_
               CType(cacheManagers(cacheManagerName), CacheManager), _
               CType(Nothing, CacheManager))
          If Not cacheManager Is Nothing Then
               Return cacheManager
          End If

     Dim view As CachingConfigurationView = _
          New CachingConfigurationView(ConfigurationContext)
     Dim cacheManagerData As CacheManagerData = _
          view.GetCacheManagerData(cacheManagerName)
     Dim scavengingPolicy As CacheCapacityScavengingPolicy = _
          New CacheCapacityScavengingPolicy(cacheManagerName, view)

     Dim backingStore As IBackingStore = _
          backingStoreFactory.CreateBackingStore(cacheManagerName)
     Dim cache As Cache = New Cache(backingStore, scavengingPolicy)

     Dim timer As ExpirationPollTimer = New ExpirationPollTimer()
     Dim expirationTask As ExpirationTask = CreateExpirationTask(cache)
     Dim scavengerTask As ScavengerTask = _
          New ScavengerTask(cacheManagerName, view, _
                            scavengingPolicy, cache)
     Dim scheduler As BackgroundScheduler = _
          New BackgroundScheduler(expirationTask, scavengerTask)
     cache.Initialize(scheduler)

     scheduler.Start()
     timer.StartPolling(New TimerCallback( _
          AddressOf scheduler.ExpirationTimeoutExpired), _
          cacheManagerData.ExpirationPollFrequencyInSeconds * 1000)

     cacheManager = New CacheManager(cache, scheduler, timer)
     cacheManagers.Add(cacheManagerName, cacheManager)
     Return cacheManager
End Function
```

When the internal Cache object is constructed, all data in the BackingStore is loaded into an in-memory representation that is contained in the Cache object. This is the only time that the BackingStore is ever read—when an application makes changes to the cache, the changes are written to both the internal cache and the BackingStore. An application can make requests to the `CacheManager` object to retrieve cached data, add data to the cache, and remove data from the cache, and it should always be synchronized with the BackingStore. Table 4.1 describes the methods that the `CacheManager` class exposes for performing these functions.

Another class, `CacheFactory`, refines the `CacheManagerFactory` class with static methods that simply pass through to an instance of the `CacheManagerFactory` class. This provides a simpler interface for developers because it allows a `CacheManager` to be created without directly having to instantiate a factory class, and it just contains a single method: `GetCacheManager`. `GetCacheManager` contains two overloads: one overload accepts

TABLE 4.1: `CacheManager` **Class**

Method/Property	Description
`Add`	This overloaded method adds new `CacheItem` to cache. If another item already exists with the same key, that item is removed before the new item is added. The `Add` method enables adding items to the cache with or without metadata (expiration policies, scavenging priority, etc.). In the simplest case, the `Add` method just contains a key/value pair. If any failure occurs during this process, the cache will not contain the item being added.
`Count`	Returns the number of items currently in the cache.
`Flush`	Removes all items and metadata from the cache. If an error occurs during the removal, the cache is left unchanged.
`GetData`	Returns the value associated with the given key.
`Item`	Returns the item identified by the provided key.
`Remove`	Removes the given item and its metadata from the cache. If no item exists with that key, this method does nothing.

no arguments and wraps around the `CacheManagerFactory`'s `GetCacheManager` method, and the other overload accepts a string and wraps around the `CacheManagerFactory`'s `GetCacheManager(string)` method. Both the `CacheManagerFactory` and the `CacheFactory` class can be used to obtain a `CacheManager`.

Cache Objects

A **Cache object** receives requests from a `CacheManager` and implements all operations between the BackingStore and the in-memory representation of the cached data. A **cache** is simply a copy of the master data stored in memory or on disk. Therefore, a Cache object simply contains a hash table that holds the in-memory representation of the data; however, that item of data must first be packaged as a `CacheItem` object. A `CacheItem` includes the data itself, together with other information such as the item's key, its priority, a `RefreshAction`, and an array of expiration policies. (All of these classes are explained in detail in the following sections.) The Cache object uses a hash table as a lock to control access to the items in the cache, both from the application and from the `BackgroundScheduler`. It also provides thread safety for the entire Caching Application Block.

When an application adds an item to the cache by calling `CacheManager`'s `Add` method, the `CacheManager` simply forwards the request to the Cache object. If there isn't an item in the in-memory hash table that matches the key for the item being added, the Cache object will first create a dummy cache item and add it to an in-memory hash table. Then, whether the item exists or not, it will use the item found for this key as a snapshot of the item before performing the insert. It then locks the cache item in the in-memory hash table, adds the item to BackingStore, and finally replaces the existing cache item in the in-memory hash table with the new cache item. (In the case where the item was not already in the in-memory hash table, it replaces the dummy item.)

If there is an exception while writing to the BackingStore, it removes the dummy item added to the in-memory hash table and does not continue. The Caching Application Block enforces a strong exception safety guarantee. This means that if an `Add` operation fails, the state of the cache rolls back

to what it was before it tried to add the item. In other words, either an operation is completed successfully or the state of the cache remains unchanged. (This is also true for the `Remove` and `Flush` methods.)

If the number of cached items exceeds a predetermined limit when the item is added, the `BackgroundScheduler` object begins scavenging. When adding an item, the application can use an overload of the `Add` method to specify an array of expiration policies, the scavenging priority, and an object that implements the `ICacheItemRefreshAction` interface. As explained later in this chapter, a `RefreshAction` receives a notification when an item is removed from the cache.

When an application calls the `CacheManager`'s `GetData` method to retrieve an item, the `CacheManager` object forwards the request to the Cache object. If the item is in the cache, it is returned from the Cache's in-memory representation. If it isn't in the cache, the request returns the value null (or Nothing in VB.NET). If the item is expired, the item also returns the value null (or Nothing in VB.NET).

`CacheService` Objects

As described in the section "Custom Cache Detailed Design" of the *Caching Architecture Guide* for .NET Framework Applications, a `CacheManager` object has references to both a `CacheStorage` and a `CacheService` object. The `CacheStorage` object is used for inserting, getting, and removing items from the cache storage. The Caching Application Block implements this design by way of the `BaseBackingStore` class (and classes that inherit from it). The `CacheService` object is designed to manage metadata that may be associated with `CacheItems`. This metadata may include items like expiration policies, priorities, and callbacks. While a single `CacheService` class does not exist in the Caching Application Block, the functionality that such a service is designed to implement does exist by way of the `BackgroundScheduler` and `ExpirationPollTimer` classes.

The `BackgroundScheduler` Class

The `BackgroundScheduler` class is designed to periodically monitor the lifetime of the items in the cache. It is responsible for expiring aging cache items and scavenging lower-priority cache items. When an item expires, the

BackgroundScheduler first removes it and then, optionally, notifies the application that the item was removed. At this point, it is the application's responsibility to refresh the cache as necessary.

The BackgroundScheduler operates in a worker thread. If a request is made to the BackgroundScheduler, the BackgroundScheduler packages the request as a message and puts it in a message queue instead of immediately executing the requested behavior. This all occurs in the caller's thread. From its own thread, the BackgroundScheduler sequentially removes messages from the queue and then executes the request. The advantage to performing operations serially on a single thread is that it guarantees that the code will run in a single-threaded environment. This makes both the code and its effects simpler to understand.

The ExpirationPollTimer Class

The ExpirationPollTimer triggers the expiration cycle and makes a call to the BackgroundScheduler. The frequency of the timer that regulates how often the BackgroundScheduler should check for expired items can be set through configuration. The unit is in seconds and is determined by the ExpirationPollFrequencyInSecond attribute in the configuration data.

Expiration Policies

An important aspect of caching state is the way in which it is kept consistent with the master data and other application resources. Expiration policies can be used to define the contents of a cache that are invalid based on the amount of time that the data has been in the cache or on notification from another resource. The first type of expiration policy is known as a **time-based expiration** and the second is known as a **notification-based expiration.**

The Caching Application Block's expiration process is performed by the BackgroundScheduler that periodically examines the CacheItems to see if any items have expired. The ExpirationPollFrequencyInSeconds setting for a CacheManager controls how frequently the expiration cycle occurs for that instance of the CacheManager. Expiration is a two-part process. The first part is known as **marking** and the second part is known as **sweeping.** The process is divided into separate tasks to avoid any con-

flicts that can occur if the application is using a cache item that the `BackgroundScheduler` is trying to expire.

- During marking, `BackgroundScheduler` makes a copy of the hash table and examines each cache item in it to see if it can be expired. It locks the item while it is doing this. If an item is eligible for expiration, the `BackgroundScheduler` sets a flag in the cache item.
- During sweeping, the `BackgroundScheduler` reexamines each flagged `CacheItem` to see if it has been accessed since it was flagged. If it has been accessed, the item is kept in the cache. If it hasn't been accessed, it is expired and removed from the cache. A Windows Management Instrumentation (WMI) event occurs when an item expires. WMI events publish management information, like performance counters, about an application so that management systems, like Microsoft Operations Manager, can better manage an application.

The Caching Application Block ships with four expiration policies; three are time-based expirations and one is a notification-based expiration. The time-based expirations are `AbsoluteTime`, `SlidingTime`, and `ExtendedFormatTime`. The notification-based expiration is `FileDependency`. Furthermore, the Caching Application Block provides the capability for adding a custom extension policy to the ones that already exist by creating a new class that implements the `ICacheItemExpiration` interface. This interface, as well as the expiration policies that ship with Enterprise Library, are shown in Figure 4.2.

Time-Based Expirations. Time-based expirations invalidate data based on either relative or absolute time periods. Use time-based expiration when volatile cache items, such as those that have regular data refreshes or those that are valid for only a set amount of time, are stored in a cache. Time-based expiration enables policies to be set that keep items in the cache only as long as their data remains current. For example, if an application displays product information that gets updated in the product catalog once a day at most, the product information can be cached for the time that those products remain constant in the catalog, that is, for a 24-hour period.

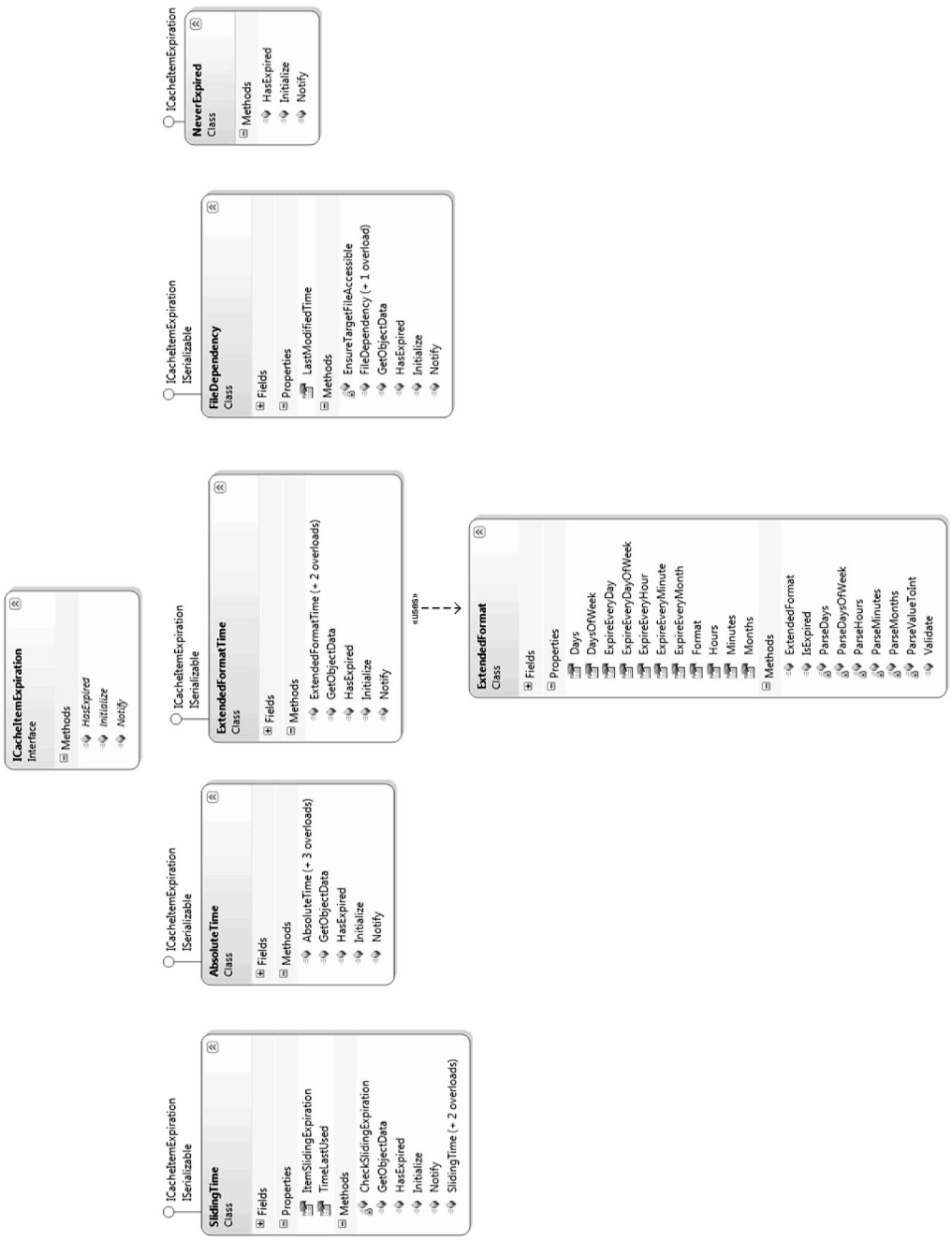

FIGURE 4.2: Expiration Policies in Enterprise Library's Caching Application Block

There are two categories of time-based expiration policies: **absolute** and **sliding.**

- **Absolute time expiration policies** allow the lifetime of an item to be defined by specifying the absolute time for an item to expire. They can use a simple or extended time format. With a simple format absolute time expiration, the lifetime of an item is defined by setting a specific date and time, for example, July 26, 2007 12:00 AM. The Caching Application Block refers to this type of expiration simply as an `AbsoluteTime` expiration. Listing 4.2 shows the code to create an `AbsoluteTime` expiration.

  Absolute time expirations can also be in an extended time format. With an extended time format, the lifetime of an item is defined by specifying expressions such as every minute, every Sunday, expire at 5:15 AM on the 15th of every month, and so on. Extended time format uses this format:

  <Minute> <Hour> <Day of month> <Month> <Day of week>

  where an asterisk (*) is used to represent all periods. Therefore, an expression to indicate that an item should expire at midnight every Saturday can be represented by the string `0 0 * * 6` (0 minutes, 0 hours, every day of the month, every month, on Saturday). Listing 4.3 shows the code to create this `ExtendedFormatTime` expiration expression.
- **Sliding expiration policies** allow the lifetime of an item to be defined by specifying an interval between the last time the item was accessed and the duration until it is expired. For example, it might be desirable to expire a particular cache item if it hasn't been accessed in the last five minutes. The code to create such a `SlidingTime` expiration is shown in Listing 4.4.

LISTING 4.2: Creating an `AbsoluteTime` Expiration

```
[C#]
DateTime expiryTime = new DateTime(2007, 7, 26, 0, 0, 0);
AbsoluteTime absExpiryTime = new AbsoluteTime(expiryTime);

[Visual Basic]
Dim expiryTime As DateTime = New DateTime(2007, 7, 26, 0, 0, 0)
Dim absExpiryTime As AbsoluteTime = New AbsoluteTime(expiryTime)
```

LISTING 4.3: Creating an `ExtendedFormatTime` Expiration

```
[C#]
ExtendedFormatTime expireTime = new ExtendedFormatTime("0 0 * * 6");

[Visual Basic]
Dim expireTime As ExtendedFormatTime = _
     New ExtendedFormatTime("0 0 * * 6")
```

LISTING 4.4: Creating a `SlidingTime` Expiration

```
[C#]
TimeSpan expiryTime = new TimeSpan(0, 5, 0);
SlidingTime slideExpireTime = new SlidingTime(expiryTime);

[Visual Basic]
Dim expiryTime As TimeSpan = New TimeSpan(0, 5, 0)
Dim slideExpireTime As SlidingTime = New SlidingTime(expiryTime)
```

Notification-Based Expirations. Notification-based expirations invalidate data based on instructions from an internal or external source. Notification-based expirations define the validity of a cached item based on the properties of an application resource, such as a file, a folder, or any other type of data source. If a dependency changes, the cached item is invalidated and removed from the cache.

The Caching Application Block ships with one notification-based expiration: the `FileDependency` expiration. With the `FileDependency` expiration, the item expires after a specific file has been modified. For example, a cache item can be set to expire if an XML file that contains product information has been modified. Listing 4.5 shows how to create a `FileDependency` expiration.

LISTING 4.5: Creating a `FileDependency` Expiration

```
[C#]
FileDependency expireNotice = new FileDependency("ProductInfo.xml");
productsCache.Add(myProduct.ProductID, myProduct,
CacheItemPriority.Normal, null, expireNotice);

[Visual Basic]
Dim expireNotice As FileDependency = New FileDependency("Products.XML")
```

```
productsCache.Add(myProduct.ProductID, myProduct, _
CacheItemPriority.Normal, Nothing, expireNotice)
```

Creating a Custom Expiration Policy

More often than not, the master data source for an enterprise application is a database and not a file. Therefore, a useful notification-based expiration would be one that expires a cached item based on modifications that are made to a database table instead of a file. While no such notification-based expiration policy ships with Enterprise Library, there is an example of a similar type of expiration that extends the ASP.NET cache capabilities. It was developed by Rob Howard and can be found at www.gotdotnet.com/team/rhoward. It is called the `SqlDependency Expiration` and can be found under the example for ASP.NET Cache Invalidation on Database Change. I have taken the ideas behind this expiration policy and refactored it to not only work with Enterprise Library's Caching Application Block, but to also take advantage of the Data Access Application Block's data transparency features (explained in more detail in Chapter 3).

Fortunately, the designers of the Caching Application Block foresaw that there might be a need for other expiration policies other than the ones that shipped with Enterprise Library and allowed the addition of custom expiration policies as an extension point. All that is required of a custom expiration policy is to implement the `ICacheItemExpiration` interface. This interface contains three methods: `Initialize`, `HasExpired`, and `Notify`. Listing 4.6 shows the `HasExpired` method for a custom `DatabaseDependency` expiration policy. This is the most interesting part of this expiration policy, because it is the check to determine if any data in the table has been modified and signifies to the `BackgroundScheduler` that this item has expired if it has. The `Notify` and `Initialize` methods for this expiration policy do nothing.

LISTING 4.6: `HasExpired` **Method for the** `DatabaseDependency` **Expiration Policy**

```
[C#]
public bool HasExpired()
{
     bool bRetVal = false;
     try
```

```
        {
            DateTime currentLastWriteTime = DateTime.MinValue;
            Database db =
                DatabaseFactory.CreateDatabase
                (dependencyDatabaseInstance);

            IDataReader dataReader =
                db.ExecuteReader
                ("GetLastNotificationDate", dependencyTableName);

            if( dataReader.Read())
                currentLastWriteTime = dataReader.IsDBNull(0) ?
                                       DateTime.MinValue :
                                       dataReader.GetDateTime( 0 );
            dataReader.Close();
            if (lastModifiedTime.Equals(DateTime.MinValue))
            {
                lastModifiedTime = currentLastWriteTime;
            }

            if (lastModifiedTime.Equals(currentLastWriteTime) == false)
            {
                lastModifiedTime = currentLastWriteTime;
                bRetVal = true;
            }
        }
        catch (Exception e)
        {
            throw new ApplicationException(String.Format("{0}: {1}",
                      SR.ExceptionInvalidDatabaseNotificationInfo
                      (dependencyTableName),e.Message), e);
        }
        return bRetVal;

    }

    [Visual Basic]
    Public Function HasExpired() As Boolean
        Dim bRetVal As Boolean = False
        Try
            Dim currentLastWriteTime As DateTime = DateTime.MinValue
            Dim db As Database = _
                DatabaseFactory.CreateDatabase _
                    (dependencyDatabaseInstance)

            Dim dataReader As IDataReader = _
                db.ExecuteReader _
                 ("GetLastNotificationDate", dependencyTableName)

            If dataReader.Read() Then
```

```
                currentLastWriteTime = IIf(dataReader.IsDBNull(0), _
                                           DateTime.MinValue, _
                                           dataReader.GetDateTime(0))
           End If
           dataReader.Close()
           If lastModifiedTime.Equals(DateTime.MinValue) Then
                lastModifiedTime = currentLastWriteTime
           End If

           If lastModifiedTime.Equals(currentLastWriteTime) = False Then
                lastModifiedTime = currentLastWriteTime
                bRetVal = True
           End If
      Catch e As Exception
           Throw New ApplicationException(String.Format("{0}: {1}", _
                     SR.ExceptionInvalidDatabaseNotificationInfo _
                     (dependencyTableName),e.Message), e)
      End Try
      Return bRetVal
End Function
```

After the `DatabaseDependency` expiration policy has been created, using it with the Caching Application Block is just as easy as using the other expiration policies (see Listing 4.7).

LISTING 4.7: Setting a `DatabaseDependency` Expiration Policy for a `CacheItem`

```
[C#]
//Monitor the Products table in the Northwind DB instance.
DatabaseDependency expireNotice =
          new DatabaseDependency("Northwind", "Products");
productsCache.Add(myProduct.ProductID, myProduct,
          CacheItemPriority.Normal, null, expireNotice);

[Visual Basic]
'Monitor the Products table in the Northwind DB instance.
Dim expireNotice As DatabaseDependency = _
          New DatabaseDependency ("Northwind", "Products");
productsCache.Add(myProduct.ProductID, myProduct, _
          CacheItemPriority.Normal, Nothing, expireNotice)
```

The `ICacheItemRefreshAction` Interface (aka `CacheItemRemovedCallback`)

The `ICacheItemRefreshAction` is a bit of a misnomer. The Caching Application Block does an excellent job of keeping similar terminology and design to what it laid out in the *Caching Architecture Guide for .NET Frame-*

work Applications; however, it seems to deviate on this one item. The delegate that is described as the `CacheItemRemovedCallback` in the *Caching Architecture Guide for .NET Framework Applications* is known as the `ICacheItemRefreshAction` interface in the Caching Application Block.

During the development of the Caching Application Block, the responsibility for this delegate changed. Originally, callbacks were only designed for expirations, and the purpose of the callback was solely to allow the owner of that item to refresh it in the cache. However, as development progressed, the requirement surfaced that callbacks were needed for removals and scavengings too, but the name was never changed. So, even though the name implies that an implementation of this interface should *refresh* a cached item, it is not necessary to do so. Rather, the `ICacheItemRefreshAction` interface just defines the contract that must be implemented so that an object will be notified when an item is removed from cache. It is then up to that implementation to determine what action should occur.

It is important to note that the implementing class of an `ICacheItemRefreshAction` must be serializable. Take care when implementing this interface not to create an object that maintains too much state about its environment, because all portions of its environment will be serialized as well, possibly creating a huge object graph. Figure 4.3 illustrates the `ICacheItemRefreshAction` interface as well as the enumeration that is passed to the `Refresh` method, which lists the possible values for why an item may have

FIGURE 4.3: The `ICacheItemRefreshAction` Interface and Reasons for Removing an Item from Cache

been removed from the cache. This enumeration is named `CacheItemRemovedReason`.

As Figure 4.3 illustrates, there is only one method that must be developed to implement the `ICacheItemRefreshAction` interface. This is the `Refresh` method. Listing 4.8 provides an example of an implementation that does not refresh the cache, but instead leverages the Logging and Instrumentation Application Block to log the fact that an item was removed from the cache.

LISTING 4.8: Implementing the `ICacheItemRefreshAction` Interface

```
[C#]
public class LoggingRefreshAction : ICacheItemRefreshAction
{
     public void Refresh(string key,
                         object expiredValue,
                         CacheItemRemovedReason removalReason)
     {
          // Log that the item has been removed from cache.
          Logger.Write(String.Format("The {0} with the key {1} was
               removed from the cache for the following reason: {2}",
               expiredValue.GetType().Name, key,
               removalReason.ToString()), Category.General,
               Priority.Normal);
     }
}


[Visual Basic]
Public Class LoggingRefreshAction : Inherits ICacheItemRefreshAction
     Public Sub Refresh(ByVal key As String, _
                        ByVal expiredValue As Object, _
                        ByVal removalReason As CacheItemRemovedReason)

          'Log that the item has been removed from cache.
          Logger.Write(String.Format("The {0} with the key {1} was" & _
                         " removed from the cache for the following " & _
                         "reason: {2}", expiredValue.GetType().Name, key, _
                         removalReason.ToString()), Category.General, _
                         Priority.Normal)
     End Sub
End Class
```

CacheStorage

The *Caching Architecture Guide for .NET Framework Applications* defines the third major component of a custom cache triad to be `CacheStorage`. The `CacheStorage` implementation separates the cache functionality from the cache data store. The Caching Application Block implements this design and provides an extension point to the block with the `IBackingStore` interface and the `BaseBackingStore` abstract base class. This interface defines the contract that must be implemented by all BackingStores.

Implementers of this method are responsible for interacting with their underlying persistence mechanisms to store and retrieve `CacheItems`. All methods must guarantee Weak Exception Safety—that operations must complete entirely, or they must completely clean up from the failure and leave the cache in a consistent state. The mandatory cleanup process will remove all traces of the item that caused the failure, causing that item to be expunged from the cache entirely.

The abstract `BaseBackingStore` class, which implements the `IBackingStore` interface, is provided to facilitate the creation of BackingStores. This class contains implementations of common policies and utilities that can be used by all BackingStores. Table 4.2 lists the `BaseBackingStore`'s methods and properties. All methods other than the `Add`, `CurrentCacheManager`, and `Load` methods are abstract and must therefore be overridden by a concrete BackingStore.

The concrete cache storage classes that are included with the Caching Application Block are the `NullBackingStore`, the `IsolatedStorageBackingStore`, and the `DataBackingStore`.

- The `NullBackingStore` class simply retains the cached items in memory.
- The `IsolatedStorageBackingStore` class stores cache items in domain-specific isolated storage and is configured to use a named isolated storage.
- The `DataBackingStore` class uses a database as its BackingStore and leverages the Data Access Application Block to connect to and perform database operations against a database.

TABLE 4.2: `BaseBackingStore` **Methods and Properties**

Method/Property	Description
`Add`	Is responsible for adding a `CacheItem` to the BackingStore. This operation must be successful even if an item with the same key already exists. This method must also meet the exception safety guarantee and make sure that all traces of the new or old item are gone if the add fails in any way.
`CurrentCacheManager`	Gets the current name of the `CacheManager` using this instance.
`Load`	Loads all `CacheItems` from the underlying database.
`AddNewItem`	A protected method that adds a new item to the persistence store.
`Count`	The number of objects stored in the BackingStore.
`Flush`	Flushes all `CacheItems` from the BackingStore. This method must meet the Strong Exception Safety guarantee.
`LoadDataFromStore`	A protected method that is responsible for loading items from the underlying persistence store.
`Remove`	An overloaded method that removes an item with the given key from the BackingStore.
`RemoveOldItem`	A protected method that removes existing items stored in the persistence store with the same key as the new item.
`UpdateLastAccessedTime`	An overloaded protected method that updates the last accessed time for a cache item referenced by this unique storage key.

The Caching Application Block communicates with all BackingStores through the `IBackingStore` interface. Figure 4.4 shows the relationship between the `IBackingStore`, `BaseBackingStore`, `DataBackingStore`, `IsolatedStorageBackingStore`, and `NullBackingStore` classes.

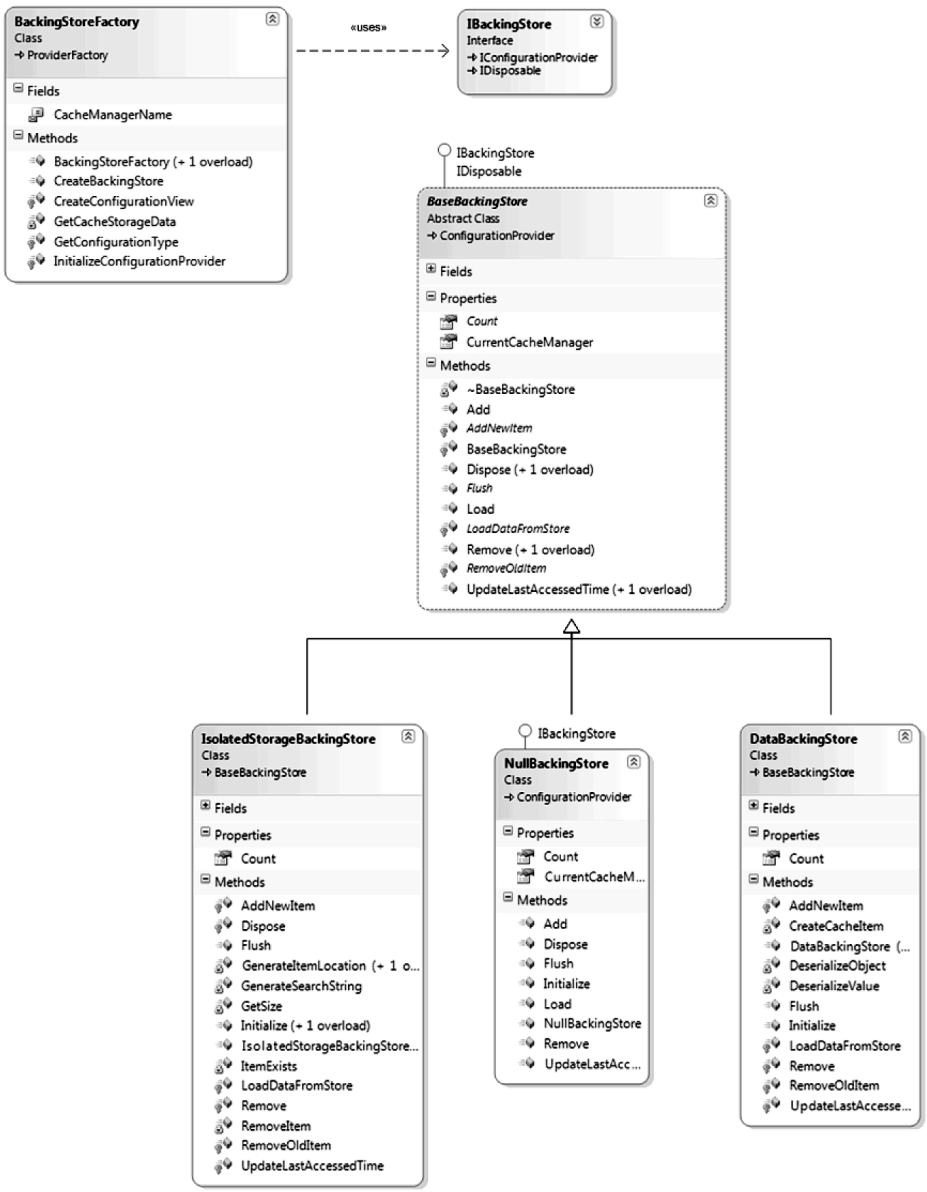

FIGURE 4.4: Available BackingStores in Enterprise Library's Caching Application Block

Memory-Resident Cache (`NullBackingStore`)

A memory-resident cache contains techniques that implement in-memory temporary data storage. Memory-based caching is usually used when an application is frequently using the same data or an application often needs to reacquire the data. By default, the Caching Application Block stores items only in memory by way of a `NullBackingStore`. The `NullBackingStore` doesn't persist cached items; cached data exists only in memory. This means that cached data will not live past application restarts; that is, the cached items will be refreshed from the original data source when the application restarts.

Disk-Resident Cache

A disk-resident cache contains technologies that use disk-based data storages, such as files or databases. Disk-based caching is useful when large amounts of data need to be handled, the data in the application services may not always be available for reacquisition, or the cached data must survive process recycles and computer reboots. Both the overhead associated with data processing and interprocess communications can be reduced by storing data that has already been transformed or rendered nearer to the data consumer.

If a `CacheManager` has been configured to use a persistent BackingStore, the Caching Application Block will load the cache contents from the BackingStore when the cache is first created. After the initial load, the BackingStore is updated after each operation on the in-memory cache. However, the BackingStore is never read from again (unless the cache is disposed and recreated, for example, on application restart). It is also important to note that while an application can use more than one CacheManager, the Caching Application Block does not support the use of the same persistent BackingStore location and partition name by multiple CacheManagers in an application. For example, configuring an application with two CacheManagers that both leverage isolated storage and have a partition name of *ProductCache* will most likely cause data corruption.

In its original state, the Caching Application Block supports two types of persistent BackingStores, each of which is suited to particular situations: isolated storage and data cache storage. Additionally, you can also extend

the Caching Application Block to support additional types of BackingStores, including custom cache storage.

Isolated Storage. Isolated storage is a data storage mechanism that provides isolation and safety by defining standardized ways of associating code with saved data. When an application stores data in a file without leveraging isolated storage, the file name and storage location must be carefully chosen to minimize the possibility that the storage location will be known to another application and, therefore, vulnerable to corruption. Without a standard system in place to manage these problems, developing ad hoc techniques that minimize storage conflicts can be complex and the results can be unreliable.

With isolated storage, data is always isolated by user and by assembly. Credentials such as the origin or the strong name of the assembly determine assembly identity. Data can also be isolated by application domain using similar credentials. Because of the obstacles that must be overcome to isolate by user, isolated storage is rarely used for server applications; it is, however, a good choice for smart client applications.

When using isolated storage, you don't need to write any code to determine unique paths to specify safe locations in the file system, and data is protected from other applications that only have isolated storage access. "Hard-coded" information that indicates where an application's storage area is located is unnecessary.

When configured to use isolated storage, the Caching Application Block isolates the BackingStore by the cache instance name, the user name, the assembly, and the application domain. The data compartment is an abstraction, not a specific storage location; it consists of one or more isolated storage files, called **stores,** which contain the actual directory locations where data is stored. For example, a smart client application might have a data compartment associated with it, and a directory in the file system would implement the store that actually preserves the data for that application. For the developer, the location of the data compartment is transparent.[3]

3. Introductory formation about isolated storage is from http://msdn.microsoft.com/library/en-us/cpguide/html/cpconintroductiontoisolatedstorage.asp.

The decision whether or not to use isolated storage must be weighed very carefully. The general rule is that it usually makes sense for smart client applications where the cache needs to survive application restarts, but it does not generally make sense for server applications. The following are some other scenarios for using isolated storage.

- **Downloaded controls.** Managed code controls downloaded from the Internet are not allowed to write to the hard drive through normal I/O classes, but they can use isolated storage to persist users' settings and application states.
- **Persistent Web application storage.** Web applications are also prevented from using I/O classes. These programs can use isolated storage for the same purposes as downloaded components.
- **Shared component storage.** Components that are shared between applications can use isolated storage to provide controlled access to data stores.
- **Roaming.** Applications can also use isolated storage with roaming user profiles. This allows a user's isolated stores to roam with the profile.

Data Cache Storage. The data cache storage is a disk-resident storage mechanism that allows an application to leverage the Data Access Application Block to store cached data in a database. The Data Access Application Block BackingStore option is suitable for server applications where each application domain has its own cache and access to a database. Currently, the Caching Application Block includes a script to create the required database schema for Microsoft SQL Server and has only been tested against Microsoft SQL Server databases. Other database types, like Oracle and DB2, can certainly be used as BackingStores; however, you must first port the SQL script to support that database type.

It is important to note that each `CacheManager` object that is running in a single application domain must use a different portion of the database. A **partition** for a data cache store is defined as a combination of the application name and the cache instance name. Therefore, two separate and distinct applications cannot have the same application name and cache

instance name where both leverage the same Data Access Application Block configuration. For example, two distinct applications that are both configured to use the `DataBackingStore`, both named Northwind, and both have `CacheManagers` named *ProductCache* will be seen as sharing a `CacheManager` across application domains and is not supported by the Caching Application Block. Rather, every application that leverages the data cache store should have its own instance and partition.

It is possible, however, to have the same application run in multiple processes (for example, the application is deployed on multiple computers in a Web farm). There are three possible ways to configure the Caching Application Block for this circumstance.

- **Partitioned caches.** All instances of the application use the same database instance, but each instance of the application uses a different database partition. In this scenario, each `CacheManager` operates independently. Although they share the same BackingStore database instance, each `CacheManager` persists the cache data to a different partition. In effect, there is one cache for each application instance. When an application restarts, each `CacheManager` loads its data from its own partition in the BackingStore.
- **Shared partition.** All instances of the application use the same database instance and the same database partition, and all `CacheManagers` can read from and write to the cache. Each instance of an application operates against a unique in-memory cache. When an application creates a `CacheManager`, the `CacheManager` populates the in-memory cache with the data in the BackingStore. This means that if an application creates a `CacheManager` when it starts, and if all of the application instances are started at the same time, each in-memory cache will be loaded with identical data. Because the applications are using the same partition, each application instance does not require additional storage in the BackingStore.

  After the `CacheManagers` are created, the in-memory cache contents are determined by the application instance using the cache. How an instance of the application uses the cache can vary from one instance to another as requests are routed to different servers. Differ-

ent instances of an executing application can have in-memory caches with different contents. As an application adds and removes items, the contents of the in-memory cache change. The in-memory cache contents also change when the `CacheManager` removes or scavenges expired items.

As the in-memory cache changes, the `CacheManager` updates the BackingStore to reflect these changes. This is risky, though, because the BackingStore does not notify `CacheManager` instances when its contents have changed. Therefore, when one application instance changes the BackingStore contents, the other application instances will have in-memory caches that don't match the BackingStore data. This means that after an application restarts, the in-memory cache can have contents that are different from the contents it contained before the application restarted.

- **Single writer.** All instances of the application use the same database instance and the same database partition, and only one `CacheManager` can write to the cache. All `CacheManager`s can read from the cache. In this scenario, only one instance of the application writes to the cache. All other application instances can only read from the cache.

  The instance of the application that writes to the cache is the master. The in-memory cache of the master is always identical to the data in the BackingStore. The in-memory cache in each application instance is populated with data from the BackingStore at the time the `CacheManager` is created. The application instances that can only read data from the cache receive a snapshot of the data. However, this is rarely wise because the application instances don't have the ability to refresh their caches; therefore, their caches become stale and shrink as items expire.

Custom Cache Storage. As previously mentioned, Enterprise Library's Caching Application Block has been designed to allow another extension point by adding and using custom BackingStores in addition to the BackingStores that ship with the application block. There are two ways that the Caching Application Block can be extended with a BackingStore: by creat-

ing and adding a custom BackingStore, or by creating and adding a new BackingStore with all the design-time features as the ones that ship with the Caching Application Block.

The simplest approach from a development perspective is to simply create a new custom BackingStore that inherits from the abstract `BaseBackingStore` class. Be sure that the implementation guarantees the BackingStore will remain intact and functional if an exception occurs during any operation that accesses it. For example, if the application tries to add an item to the cache and there is already an item in the cache with the same name, an exception may be thrown. The implementation should remove the older item from both the BackingStore and the in-memory representation, and then it should throw an exception to the application.

Because of the way the Cache object operates, any BackingStore is guaranteed to be called in a single-threaded manner. This means that custom BackingStore implementations do not need to be overly concerned with thread safety. Furthermore, custom configuration information for a custom BackingStore can be retrieved through an `Extensions` collection that the Caching Application Block provides for all custom BackingStores. When the BackingStore is initialized, it can retrieve its necessary configuration information via this collection.

For example, imagine that a new BackingStore was needed in an enterprise that leveraged an XML file on the server instead of a database. A quick way to provide this type of functionality and use it with the Caching Application Block is to create a new class that derives from the `BaseBackingStore` class and overrides the abstract methods. This class would need to read configuration information to determine the name of the file that it should use as the BackingStore, and it might also need a partition name to avoid collisions between applications that might choose the same XML file name. Listing 4.9 illustrates how you can use the `Extensions` collection to get the configuration information for just such a custom BackingStore.

LISTING 4.9: `Initialize` **Method for a Custom BackingStore**

```
[C#]
public override void Initialize(ConfigurationView configurationView)
{
     ArgumentValidation.CheckForNullReference
           (configurationView, "configurationView");
     ArgumentValidation.CheckExpectedType
```

```
            (configurationView, typeof (CachingConfigurationView));

      CachingConfigurationView cachingConfigurationView =
            (CachingConfigurationView) configurationView;

      CustomCacheStorageData  customConfiguration =
            (CustomCacheStorageData)
                cachingConfigurationView.GetCacheStorageDataForCacheManager
                (CurrentCacheManager);

      partitionName = customConfiguration.Extensions["PartitionName"];
      xmlFileName = String.Format("{0}.{1}",
           customConfiguration.Extensions["XmlFileName"],partitionName);

      if (customConfiguration.StorageEncryption != null)
      {
           StorageEncryptionFactory encryptionFactory = new
                StorageEncryptionFactory
                (cachingConfigurationView.ConfigurationContext);

           encryptionProvider =
                encryptionFactory.CreateSymmetricProvider
                (CurrentCacheManager);
      }
}

[Visual Basic]
Public Overrides Sub Initialize _
     (ByVal configurationView As ConfigurationView)

     ArgumentValidation.CheckForNullReference _
           (configurationView, "configurationView")
     ArgumentValidation.CheckExpectedType _
           (configurationView, GetType(CachingConfigurationView))

     Dim cachingConfigurationView As CachingConfigurationView = _
          CType(configurationView, CachingConfigurationView)

     Dim customConfiguration As CustomCacheStorageData = _
          (CustomCacheStorageData) _
          cachingConfigurationView.GetCacheStorageDataForCacheManager _
          (CurrentCacheManager)

     partitionName = customConfiguration.Extensions("PartitionName")
     xmlFileName = String.Format("{0}.{1}", _
          customConfiguration.Extensions("XmlFileName"),partitionName)

     If Not customConfiguration.StorageEncryption Is Nothing Then
          Dim encryptionFactory As StorageEncryptionFactory = _
```

```
                New StorageEncryptionFactory _
                      (cachingConfigurationView.ConfigurationContext)

            encryptionProvider = _
                 encryptionFactory.CreateSymmetricProvider _
                 (CurrentCacheManager)
        End If
End Sub
```

Implementing this method as well as the abstract methods defined by the `BaseBackingStore` class makes it easy to create new custom BackingStores. This is a nice feature of the Caching Application Block because it does not tie an enterprise into just using the BackingStores that ship with the application block, and it doesn't require a massive amount of development to create a new one.

One of the worrisome aspects of using this approach, however, is that the configuration information will not be strongly typed. That is, there is a greater possibility that an error can be made when entering the configuration information for a custom BackingStore than there is when adding configuration information for one of the BackingStores that ship with the Caching Application Block. The Enterprise Library Configuration Tool will not perform any validation on the name or the values of the items that are stored in the `Extensions` collection. For example, an administrator may accidentally transpose some of the characters for the name of the `XmlFileName` item by accidentally typing *XlmFileName.* Such an error would not be caught at design time, but rather it would be caught at runtime when an attempt to retrieve the `XmlFileName` item takes place in the `Initialize` method.

To avoid these types of errors and provide a more user-friendly experience for configuring a custom BackingStore, a new BackingStore can be created and used that has all the design-time features as the ones that ship with the Caching Application Block. To create such a BackingStore, a few more steps are required than with the kind of custom BackingStore just discussed.

The first task is to create a data transfer object that is responsible for housing the configuration information needed by the new BackingStore. By expanding the previous example and creating a custom BackingStore that

uses an XML file, the custom BackingStore will need to retain information about the name of the XML file and the partition name. Therefore, a data transfer object must be created that encapsulates this information. This object will be used by design-time classes for setting the configuration information, and it will be used by the custom BackingStore to initialize itself in its `Initialize` method. Listing 4.10 shows two properties of an `xmlFileCacheStorageData` class that let the file name and partition name be set and retrieved.

LISTING 4.10: Properties for the `XmlFileCacheStorageData` Class

```
[C#]
[XmlAttribute("xmlFileName")]
public string XmlFileName
{
     get { return xmlFileName; }
     set { xmlFileName = value; }
}

[XmlAttribute("partitionName")]
public string PartitionName
{
     get { return partitionName; }
     set { partitionName = value; }
}

[Visual Basic]
<XmlAttribute("xmlFileName")> _
Public Property XmlFileName() As String
     Get
          Return xmlFileName
     End Get
     Set
          xmlFileName = Value
     End Set
End Property

<XmlAttribute("partitionName")> _
Public Property PartitionName() As String
     Get
          Return partitionName
     End Get
     Set
          partitionName = Value
     End Set
End Property
```

The second task that must be completed is the same as the task that needed to be completed for the previous type of custom BackingStore; that is, to create a class that inherits from the abstract `BaseBackingStore` class and override the abstract methods. The only difference between this class and the one created for the previous type is that in this class' `Initialize` method, the data transfer object that was just created will be used instead of the `CustomCacheStorageData` class. Listing 4.11 shows what the revised `Initialize` method would look like.

LISTING 4.11: `Initialize` **Method for the** `XmlFileBackingStore`

```
[C#]
public override void Initialize(ConfigurationView configurationView)
{
     ArgumentValidation.CheckForNullReference
           (configurationView, "configurationView");
     ArgumentValidation.CheckExpectedType
           (configurationView, typeof (CachingConfigurationView));

     CachingConfigurationView cachingConfigurationView =
           (CachingConfigurationView) configurationView;

     xmlFileCacheStorageData xmlFileConfiguration =
               (xmlFileCacheStorageData)
               cachingConfigurationView.GetCacheStorageDataForCacheManager
               (CurrentCacheManager);

     partitionName = xmlFileConfiguration.PartitionName;
     xmlFileName = String.Format("{0}.{1}",
               xmlFileConfiguration.XmlFileName,
               xmlFileConfiguration.PartitionName);

     if (xmlFileConfiguration.StorageEncryption != null)
     {
          StorageEncryptionFactory encryptionFactory = new
               StorageEncryptionFactory
               (cachingConfigurationView.ConfigurationContext);

          encryptionProvider =
               encryptionFactory.CreateSymmetricProvider
               (CurrentCacheManager);
     }
}

[Visual Basic]
Public Overrides Sub Initialize _
```

```
 (ByVal configurationView As ConfigurationView)

 ArgumentValidation.CheckForNullReference _
      (configurationView, "configurationView")
ArgumentValidation.CheckExpectedType _
      (configurationView, GetType(CachingConfigurationView))

Dim cachingConfigurationView As CachingConfigurationView = _
     CType(configurationView, CachingConfigurationView)

Dim xmlFileConfiguration As xmlFileCacheStorageData = CType _
      (cachingConfigurationView.GetCacheStorageDataForCacheManager _
      (CurrentCacheManager), xmlFileCacheStorageData)

Dim partitionName = xmlFileConfiguration.PartitionName
Dim xmlFileName = String.Format("{0}.{1}", _
     xmlFileConfiguration.XmlFileName, _
     xmlFileConfiguration.PartitionName)

If Not xmlFileConfiguration.StorageEncryption Is Nothing Then
     Dim encryptionFactory As StorageEncryptionFactory = New _
          StorageEncryptionFactory _
           (cachingConfigurationView.ConfigurationContext)

     encryptionProvider = _
     encryptionFactory.CreateSymmetricProvider _
           (CurrentCacheManager)
End If
End Sub
```

The last step is to create the design-time classes that allow the Enterprise Library Configuration Tool to present the user-friendly interface that makes configuring a new BackingStore easier and less error-prone. This step is not absolutely necessary; a new BackingStore can still be used even if design-time classes for it do not exist. In that case, however, the configuration information for it needs to be entered and modified manually, and the benefits with respect to validating configuration information will not be realized.

Three tasks must be performed to create the design-time classes needed to configure a new BackingStore.

1. Create a `ConfigurationNode` for the new BackingStore.
2. Create a `ConfigurationDesignManager` for the new BackingStore.

3. Modify the AssemblyInfo file so the Enterprise Library Configuration Tool can recognize the design-time features for the new BackingStore.

Chapter 2 provides much more detail about how and why these classes need to be created. The next few paragraphs document the specific steps needed to create the design-time interface for the `XmlFileBackingStore`.

The first task is to create a new `ConfigurationNode` that provides a user with the ability to add and modify the configuration properties of the `XmlFileBackingStore`. The specific properties that need to be exposed are the `FileName` and the `PartitionName`. The Caching Application Block's design-time assembly provides an abstract base class named `CacheStorageNode` that makes it easier to create a `ConfigurationNode` for a BackingStore. Listing 4.12 shows the `XmlFileCacheStorageNode` class that is derived from the `CacheStorageNode` base class.

LISTING 4.12: `XmlFileCacheStorageNode` **Class**

```
[C#]
public class XmlFileCacheStorageNode : CacheStorageNode
{
     xmlFileCacheStorageData xmlFileCacheStorageData;

     public XmlFileCacheStorageNode():
          this(new xmlFileCacheStorageData(SR.XmlFileCacheStorage))
     {
     }

     [Browsable(false)]
     public override string Type
     {
          get { return xmlFileCacheStorageData.TypeName; }
     }

     [Required]
     [SRDescription(SR.Keys.FileNameDescription)]
     [SRCategory(SR.Keys.CategoryGeneral)]
     public string FileName
     {
          get { return xmlFileCacheStorageData.XmlFileName; }
          set { xmlFileCacheStorageData.XmlFileName = value; }
     }
```

```
        [Required]
        [SRDescription(SR.Keys.FilePartitionNameDesciption)]
        [SRCategory(SR.Keys.CategoryGeneral)]
        public string PartitionName
        {
            get { return xmlFileCacheStorageData.PartitionName; }
            set { xmlFileCacheStorageData.PartitionName = value; }
        }
    }

    [Visual Basic]
    Public Class XmlFileCacheStorageNode : Inherits CacheStorageNode
        Private xmlFileCacheStorageData As xmlFileCacheStorageData

        Public Sub New()
            Me.New(New xmlFileCacheStorageData(SR.XmlFileCacheStorage))
        End Sub

        <Browsable(False)> _
        Public Overrides ReadOnly Property Type() As String
            Get
                Return xmlFileCacheStorageData.TypeName
            End Get
        End Property

        <Required,  _
            SRDescription(SR.Keys.FileNameDescription), _
            SRCategory(SR.Keys.CategoryGeneral)> _
        Public Property FileName() As String
            Get
                Return xmlFileCacheStorageData.XmlFileName
            End Get
            Set
                xmlFileCacheStorageData.XmlFileName = Value
            End Set
        End Property

        <Required, _
            SRDescription(SR.Keys.FilePartitionNameDesciption), _
            SRCategory(SR.Keys.CategoryGeneral)> _
        Public Property PartitionName() As String
            Get
                Return xmlFileCacheStorageData.PartitionName
            End Get
            Set
                xmlFileCacheStorageData.PartitionName = Value
            End Set
        End Property
    End Class
```

A new class that implements the `IConfigurationDesignManager` interface is needed to register the new `XmlFileCacheStorageNode` and associate menu items and commands with it. An `XmlIncludeType` also needs to be added so the Configuration Application Block knows how to create the `xmlFileCacheStorageData` type. Listing 4.13 shows the `RegisterIncludeTypes` and `RegisterNodeTypes` methods that are called from the virtual `Register` method for the new `xmlFileBackingStoreConfigurationDesignManager`.

LISTING 4.13: Registration Methods for the `xmlFileBackingStoreConfigurationDesignManager`

```
[C#]
private static void RegisterXmlIncludeTypes
     (IServiceProvider serviceProvider)
{
     IXmlIncludeTypeService xmlIncludeTypeService =
          serviceProvider.GetService(typeof(IXmlIncludeTypeService))
          as IXmlIncludeTypeService;

     xmlIncludeTypeService.AddXmlIncludeType
          (CacheManagerSettings.SectionName,
          typeof(xmlFileCacheStorageData));
}

private static void RegisterNodeTypes(IServiceProvider serviceProvider)
{
     INodeCreationService nodeCreationService =
          ServiceHelper.GetNodeCreationService(serviceProvider);

     Type nodeType = typeof(XmlFileCacheStorageNode);

     NodeCreationEntry entry =
          NodeCreationEntry.CreateNodeCreationEntryNoMultiples
          (new AddChildNodeCommand(serviceProvider, nodeType), nodeType,
          typeof(xmlFileCacheStorageData), SR.XmlFileCacheStorage);

          nodeCreationService.AddNodeCreationEntry(entry);
}

[Visual Basic]
Private Shared Sub RegisterXmlIncludeTypes _
      (ByVal serviceProvider As IServiceProvider)

     Dim xmlIncludeTypeService As IXmlIncludeTypeService = _
          IIf(TypeOf serviceProvider.GetService _
```

```
            (GetType(IXmlIncludeTypeService)) Is IXmlIncludeTypeService, _
            CType(serviceProvider.GetService _
             (GetType(IXmlIncludeTypeService)), IXmlIncludeTypeService), _
            CType(Nothing, IXmlIncludeTypeService))

        xmlIncludeTypeService.AddXmlIncludeType _
       (CacheManagerSettings.SectionName, GetType(xmlFileCacheStorageData))
End Sub

Private Shared Sub RegisterNodeTypes _
           (ByVal serviceProvider As IServiceProvider)

        Dim nodeCreationService As INodeCreationService = _
             ServiceHelper.GetNodeCreationService(serviceProvider)

        Dim nodeType As Type = GetType(XmlFileCacheStorageNode)

        Dim entry As NodeCreationEntry = _
             NodeCreationEntry.CreateNodeCreationEntryNoMultiples _
             (New AddChildNodeCommand(serviceProvider, nodeType), _
                  nodeType, GetType(xmlFileCacheStorageData), _
                  SR.XmlFileCacheStorage)

        nodeCreationService.AddNodeCreationEntry(entry)
End Sub
```

Lastly, a new assembly attribute needs to be added to the `AssemblyInfo.cs` (or `vb`) file because the Enterprise Library Configuration Tool looks for this to determine whether it should load a new `ConfigurationDesignManager`. Listing 4.14 shows the part of the `AssemblyInfo` file that sets the `ConfigurationDesignManagerAttribute` to the `xmlFileBackingStoreConfigurationDesignManager`.

LISTING 4.14: Assembly Attribute for the `xmlFileBackingStoreConfigurationDesignManager`

```
[C#]
[assembly :
     ConfigurationDesignManager(
     typeof(xmlFileBackingStoreConfigurationDesignManager))
]

[Visual Basic]
<assembly : _
     ConfigurationDesignManager( _
     GetType(xmlFileBackingStoreConfigurationDesignManager))
>
```

Once this assembly has been compiled and deployed so the Enterprise Library Configuration Tool can access it, you can add and configure the `XmlFileBackingStore` just as easily as any of the `BackingStores` that ship with Enterprise Library's Caching Application Block. (You'll see how to do this for all of the BackingStores a little later in the chapter.) Figure 4.5 shows the list of options for adding a `BackingStore` in the Enterprise Library Configuration Tool once this new assembly has been deployed.

Encrypting Cached Data

It is often important to ensure that data that must be secured in its original format is also secured when being transmitted to and from the cache and when stored inside the cache. Data that is stored in a cache may be accessed or altered by a process that isn't permitted access to the master data. The `DataBackingStore`, `IsolatedStorageBackingStore`, and custom BackingStores allow cache item data to be encrypted before it is persisted to storage; however, the `NullBackingStore` does not.

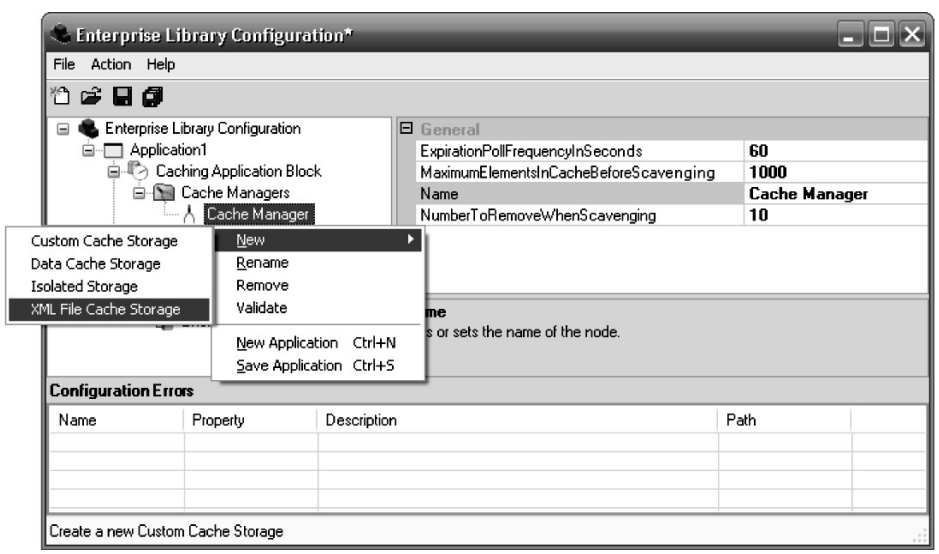

FIGURE 4.5: Available BackingStores Now Include the `XmlFileBackingStore`

Methods to prevent tampering of cache items usually include signing and verifying the items, and spoofing can be prevented by encrypting the cache items. The Caching Application Block does not offer any functionality in the way of signing data; however, it does offer the option to encrypt the cache data that is written to persistent storage. The Caching Application Block uses the Cryptography Application Block to create a symmetric encryption algorithm provider that you can use to encrypt cached data. You'll learn how to configure a BackingStore to use a symmetric encryption algorithm later in this chapter. More detailed information about using symmetric encryption algorithms in Enterprise Library can be found in Chapter 8.