

Microsoft®

Caching Architecture Guide for .NET Framework Applications



patterns & practices

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft, MS-DOS, Active Directory, BizTalk, JScript, Visual Basic, Visual C#, Visual Studio, Windows, Windows NT, and Win32 are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

© 2003 Microsoft Corporation. All rights reserved.

Version 1.0

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Contents

Chapter 1

Understanding Caching Concepts	1
Introducing the Problems that Caching Solves	1
Understanding State	2
Understanding the Lifetime of State	3
Understanding the Scope of State	3
Understanding State Staleness	4
Understanding the State Transformation Pipeline	5
Understanding Why Data Should Be Cached	6
Reducing Interprocess Communication	6
Reducing Data Access and Processing	7
Reducing Disk Access Operations	7
Understanding Where Data Should Be Cached	7
Understanding Storage Types	8
Understanding Layered Architecture Elements	8
Introducing Caching Considerations	9
Introducing Format and Access Patterns	9
Introducing Content Loading	10
Introducing Expiration Policies	10
Introducing Security	10
Introducing Management	11
Summary	11

Chapter 2

Understanding Caching Technologies	13
Using the ASP.NET Cache	14
Using Programmatic Caching	14
Using an Output Cache	18
Using the ASP.NET Cache in Non-Web Applications	21
Managing the Cache Object	22
Using Remoting Singleton Caching	23
Using Memory-Mapped Files	24
Using Microsoft SQL Server 2000 or MSDE for Caching	25
Using Static Variables for Caching	26
Using ASP.NET Session State	27
Choosing the Session State Mode	28
Determining What to Cache in the Session Object	29
Implementing Session State	30

Using ASP.NET Client-Side Caching and State	31
Using Hidden Fields	31
Using View State	32
Using Hidden Frames	34
Using Cookies	36
Using Query Strings	37
Using Client-Side Caching Technologies	38
Using Internet Explorer Caching	39
Understanding Internet Explorer Cache Types	39
Determining What to Cache in the Internet Explorer Cache	40
Understanding Benefits and Limitations of the Internet Explorer Cache	41
Implementing Internet Explorer Caching	41
Managing the Internet Explorer Cache	41
Summary	42

Chapter 3

Caching in Distributed Applications 43

Caching in the Layers of .NET-based Applications	43
Caching in the User Services Layer	44
Caching in the Business Services Layer	46
Caching in the Data Services Layer	48
Caching in the Security Aspects	49
Caching in the Operational Management Aspects	50
Selecting a Caching Technology	53
Caching in Browser-based Clients	54
Caching in Smart Clients	55
Caching in .NET Compact Framework Clients	56
Caching in ASP.NET Server Applications	57
Caching in Server Applications	59
Considering Physical Deployment Recommendations	60
Summary	60

Chapter 4

Caching .NET Framework Elements 61

Planning .NET Framework Element Caching	61
Ensuring Thread Safety	61
Cloning	63
Serializing a .NET Class	64
Normalizing Cached Data	66
Choosing a Caching Technology	66
Implementing .NET Framework Element Caching	67
Caching Connection Strings	67
Caching Data Elements	68
Caching XML Schemas	69
Caching Windows Forms Controls	69

Caching Images	71
Caching Configuration Files	73
Caching Security Credentials	73
Summary	74

Chapter 5

Managing the Contents of a Cache 75

Loading a Cache	75
Caching Data Proactively	76
Caching Data Reactively	80
Determining a Cache Expiration Policy	82
Using Expiration Policies	82
Using External Notifications	86
Flushing a Cache	87
Using Explicit Flushing	88
Implementing Scavenging	88
Locating Cached Data	90
Summary	92

Chapter 6

Understanding Advanced Caching Issues 93

Designing a Custom Cache	93
Introducing the Design Goals	93
Introducing the Solution Blueprint	94
Securing a Custom Cache	109
Signing Cache Items	109
Encrypting Cached Items	111
Monitoring a Cache	116
Implementing Performance Counters	116
Monitoring Your Cache Performance	117
Synchronizing Caches in a Server Farm	118
Summary	119

Chapter 7

Appendix 121

Appendix 1: Understanding State Terminology	121
Understanding the Lifetime of State	121
Understanding the Scope of State	122
Appendix 2: Using Caching Samples	124
Implementing a Cache Notification System	124
Implementing an Extended Format Time Expiration Algorithm	126
Appendix 3: Reviewing Performance Data	131
Introducing the Test Scenarios	131
Defining the Computer Configuration and Specifications	131
Presenting the Performance Test Results	132

1

Understanding Caching Concepts

This chapter introduces the concepts of caching. It is important to be familiar with these concepts before trying to understand the technologies and mechanisms you can use to implement caching in your applications.

This chapter contains the following sections:

- “Introducing the Problems that Caching Solves”
- “Understanding State”
- “Understanding Why Data Should Be Cached”
- “Understanding Where Data Should Be Cached”
- “Introducing Caching Considerations”

Introducing the Problems that Caching Solves

When building enterprise-scale distributed applications, architects and developers are faced with many challenges. Caching mechanisms can be used to help you overcome some of these challenges, including:

- **Performance**—Caching techniques are commonly used to improve application performance by storing relevant data as close as possible to the data consumer, thus avoiding repetitive data creation, processing, and transportation.

For example, storing data that does not change, such as a list of countries, in a cache can improve performance by minimizing data access operations and eliminating the need to recreate the same data for each request.

- **Scalability**—The same data, business functionality, and user interface fragments are often required by many users and processes in an application. If this information is processed for each request, valuable resources are wasted recreating the same output. Instead, you can store the results in a cache and reuse them for each request. This improves the scalability of your application because as the user base increases, the demand for server resources for these tasks remains constant.

For example, in a Web application the Web server is required to render the user interface for each user request. You can cache the rendered page in the ASP.NET output cache to be used for future requests, freeing resources to be used for other purposes.

Caching data can also help scale the resources of your database server. By storing frequently used data in a cache, fewer database requests are made, meaning that more users can be served.

- **Availability**—Occasionally the services that provide information to your application may be unavailable. By storing that data in another place, your application may be able to survive system failures such as network latency, Web service problems, or hardware failures.

For example, each time a user requests information from your data store, you can return the information and also cache the results, updating the cache on each request. If the data store then becomes unavailable, you can still service requests using the cached data until the data store comes back online.

To successfully design an application that uses caching, you need to thoroughly understand the caching techniques provided by the Microsoft® .NET Framework and the Microsoft Windows® operating system, and you also need to be able to address questions such as:

- When and why should a custom cache be created?
- Which caching technique provides the best performance and scalability for a specific scenario and configuration?
- Which caching technology complies with the application's requirements for security, monitoring, and management?
- How can the cached data be kept up to date?

It is important to remember that caching isn't something you can just add to your application at any point in the development cycle; the application should be designed with caching in mind. This ensures that the cache can be used during the development of the application to help tune its performance, scalability, and availability.

Now that you have seen the types of issues that caching can help avoid, you are ready to look at the types of information that may be cached. This information is commonly called state.

Understanding State

Before diving into caching technologies and techniques, it is important to have an understanding of state, because caching is merely a framework for state management. Understanding what state is and an awareness of its characteristics, such as lifetime and scope, is important for making better decisions about whether to cache it.

State refers to data, and the status or condition of that data, being used within a system at a certain point in time. That data may be permanently stored in a database, may be held in memory for a short time while a user executes a certain function, or the data may exist for some other defined length of time. It may be shared across a whole organization, it may be specific to an individual user, or it may be available to any grouping in between these extremes.

Understanding the Lifetime of State

The *lifetime* of state is the term used to refer to the time period during which that state is valid, that is, from when it is created to when it is removed. Common lifetime periods include:

- **Permanent state**—persistent data used in an application
- **Process state**—valid only for the duration of a process
- **Session state**—valid for a particular user session
- **Message state**—exists for the processing period of a message

For more details and examples of the lifetime of state, see Chapter 7, “Appendix.”

Understanding the Scope of State

Scope is the term used to refer to the accessibility of an applications state, whether it is the physical scope or the logical scope.

Understanding Physical Scope

Physical scope refers to the physical locations from which the state can be accessed. Common physical scoping levels include:

- **Organization** — state that is accessible from any application within an organization
- **Farm**—state that is accessible from any computer within an application farm
- **Machine**—state that is shared across all applications on a single computer
- **Process**—state that is accessible across multiple AppDomains in a single process
- **AppDomain**—state that is available only inside a single AppDomain

For more details and examples of physical scope, see Chapter 7, “Appendix.”

Understanding Logical Scope

Logical scope refers to the logical locations where the state can be accessed from. Common logical scoping levels include:

- **Application**—state that is valid within a certain application
- **Business process**—state that is valid within a logical business process
- **Role**—state that is available to a subset of the applications’ users
- **User**—state that is available to a single user

For more details and examples of logical scope, see Chapter 7, “Appendix.”

Understanding State Staleness

Cached state is a snapshot of the master state; therefore, its data is potentially stale (obsolete) as soon as it is retrieved by the consuming process. This is because the original data may have been changed by another process. Minimizing the staleness of data and the impact of staleness on your application is one of the tasks involved when caching state.

State *staleness* is defined as the difference between the master state, from which the cached state was created, and the current version of the cached state.

You can define staleness in terms of:

- **Likelihood of changes**—Staleness might increase with time because as time goes by there is an increasing chance that other processes have updated the master data.
- **Relevancy of changes**—It is possible that master state changes will not have an adverse affect on the usability of a process. For example, changing a Web page style does not affect the business process operation itself.

Depending on the use of the state within a system, staleness may, or may not, be an issue.

Understanding State Staleness Tolerance

The effect of the staleness of state on the business process is termed as the *tolerance*. A system can be defined as having no staleness tolerance or some staleness tolerance:

- **No tolerance**—In some scenarios, state tolerance is unacceptable. A good example of this is the caching of short-lived transactional data. When working with data with ACID (atomic, consistent, isolated, durable) characteristics, you cannot accept any tolerance in the staleness of state. For example, if a banking system is using a customer balance to approve a loan, the balance must be guaranteed to be 100 percent accurate and up to date. Using cached data in this instance could result in a loan being approved against the business rules implemented by the bank.
- **Some tolerance**—In some application scenarios, a varying tolerance is acceptable. There are cases where a known and acceptable period for updating the cached items is acceptable (that is, once a day, once a week, or once a month). For example, an online catalog displaying banking products available upon completion of an application form does not necessarily need to be concurrent with all of the products the bank offers. In this situation, a daily or weekly update is sufficient.

When selecting the state to be cached, one of the major considerations is how soon the state will become stale and what effect on the system that staleness will have. The goal is to cache state that either never becomes stale (*static state*) or that has a known period of validity (*semi-static state*). For more information about how you can define the period of validity for semi-static state, see Chapter 5, “Managing the Contents of a Cache.”

Understanding the State Transformation Pipeline

Another attribute of state is its representation during the different stages in the transformation pipeline. While data is in the data store, it is classed as being in its raw format; at the different transformation levels during business logic processing, it is classed as processed; and at the ready-to-display stage, it is classed as rendered data (for example, HTML or Windows Forms controls). Figure 1.1 shows these various stages.

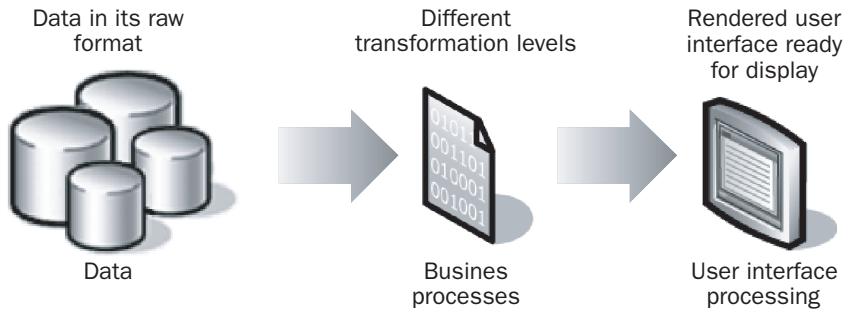


Figure 1.1
The transformation pipeline

Table 1.1 describes and gives examples of the representations of state during the different stages in the transformation pipeline.

Table 1.1: Data representation types

Representation Type	Description	Example
Raw	Data in its raw format.	Dataset reflecting data in a database.
Processed	Data that has gone through business logic processing. At this stage of the pipeline, the same data may undergo several different transformations.	Different representations of the same dataset.
Rendered	Data that is rendered and ready to be displayed in the user interface.	Rendered combo-box Web control containing a list of countries. Rendered Windows Forms TreeView control.

When you plan the caching of state in your application, you need to decide which of the state representation types is the best one to be cached. Use the following guidelines to aid you in this decision:

- Cache raw data when any staleness of the cache can be tolerated by your business logic. Raw data should be cached in:
 - Data access components.
 - Service agents.
- Cache processed data to save processing time and resources within the system. Processed data should be cached in:
 - Business logic components.
 - Service interfaces.
- Cache rendered data when the amount of data to be displayed is large and the rendering time of the control is long (for example, the contents of a large TreeView control). Rendered data should be cached in user interface (UI) components.

For a summarized review of .NET-based application architecture, its common elements, and their roles, see Chapter 3, “Caching in Distributed Applications.”

State is used in one form or another in all types of applications. Because it is time consuming to access state, it is often wise to cache the state to improve overall application performance.

Understanding Why Data Should Be Cached

You use caching to store data that is consumed frequently by an application; you can also use it to store data that is expensive to create, obtain, or transform. There are three main benefits to implementing caching in your applications:

- Reduction of the amount of data that is transferred between processes and computers
- Reduction of the amount of data processing that occurs within the system
- Reduction of the number of disk access operations that occur

The benefits that are important to you vary depending on the type of application that you are developing.

Reducing Interprocess Communication

One result of building distributed applications is that different elements of the application may reside in different processes, either on the same computer or on different computers. For example, an ASP.NET application executes in the `Aspnet_wp.exe` process, while a COM+ server application that it may be using

executes in a different process. This can be less efficient when the application requires a large amount of data to be moved between the processes or when one process is making chatty (that is, numerous small) calls to the other to obtain data.

Making calls between processes requires using remote procedure calls (RPCs) and data serialization, both of which can result in a performance hit. By using a cache to store static or semi-static data in the consuming process, instead of retrieving it each time from the provider process, the RPC overhead decreases and the application's performance improves.

Reducing Data Access and Processing

Another aspect of distributed applications is data processing. Before data is sent to a consumer, there is always some degree of processing required. This may vary from simple database querying to complex operations on the data, such as report generation and data analysis. By storing the resultant processed data in a cache for later reuse, you can save valuable computing power and achieve better application performance and scalability.

Reducing Disk Access Operations

Input/output (I/O) operations are still a major performance hit; loading different XML files, schemas, and configuration files is an expensive operation. By using a cache to store the files in the consuming process instead of reading it each time from the disk, applications can benefit from performance improvements.

These benefits can only truly be realized if you cache your state in an appropriate place in your application architecture.

Understanding Where Data Should Be Cached

Now that you have seen the issues arising in distributed application design, you understand why you should use caching techniques in your systems. A cache is simply a copy of the master data stored in memory or on disk in different transformation levels, as close as possible to the data consumer.

Therefore, in addition to selecting the data to be cached, another major consideration is where it should be cached. The considerations divide into two main categories:

- **Storage types**—where the cache should be physically located
- **Layered architecture elements**—where the cache should be logically located

You have many different options when deciding the physical location and logical location for the cache. The following sections describe some of the options.

Understanding Storage Types

Many caching implementations are available, all of which can be categorized as using either a memory resident cache or a disk resident cache. The following describes these categories:

- **Memory resident cache**—This category contains techniques which implement in-memory temporary data storage. Memory-based caching is usually used when:

- An application is frequently using the same data.
- An application often needs to reacquire the data.

You can reduce the number of expensive disk operations that need to be made by storing the data in memory, and you can minimize the amount of data that needs to be moved between different processes by storing the data in the memory of the consumer process.

- **Disk resident cache**—This category contains caching technologies that use disk-based data storages such as files or databases. Disk based caching is useful when:

- You are handling large amounts of data.
- Data in the application services (for example, a database) may not always be available for reacquisition (for example, in offline scenarios).
- Cached data lifetime must survive process recycles and computer reboots.

You can reduce the overhead of data processing by storing data that has already been transformed or rendered, and you can reduce interprocess communications by storing the data nearer to the consumer.

Understanding Layered Architecture Elements

Each component in your application deals with different types of data and state. This section refers to the application components described in “Application Architecture for .NET: Designing Applications and Services” (in the MSDN Library) because they are representative of most distributed applications.

Figure 1.2 shows a schematic view of the layered architecture’s elements.

When you explore these different elements, you need to address new considerations, including deciding the type of state that should be cached in each element.

For more information about caching considerations of layered architecture elements, see Chapter 3, “Caching in Distributed Applications.”

After you decide that your application architecture can benefit from caching and you decide where you cache state, there are still many considerations before you implement caching.

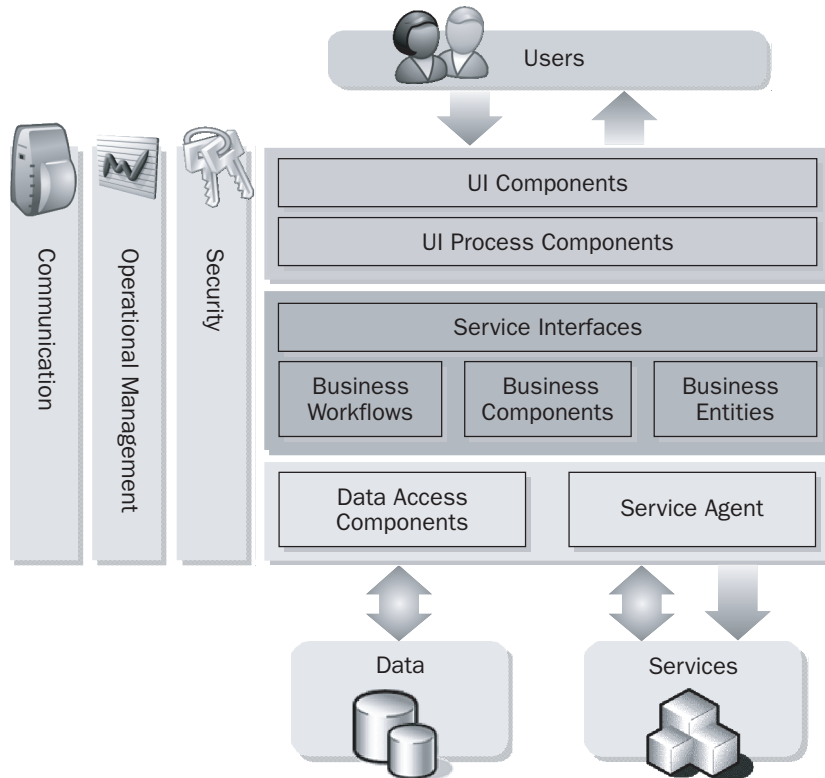


Figure 1.2
Layered architecture elements

Introducing Caching Considerations

In addition to understanding the types of state to cache, and where to cache them, there are several other factors that need to be considered when designing an application and deciding whether state should be cached. These considerations are described in more detail later in this guide, but the topics are introduced here so that you can bear them in mind as you read about the technologies and techniques involved in caching.

Introducing Format and Access Patterns

When you decide whether an object should be cached, you need to consider three main issues regarding data format and access mechanisms:

- **Thread safety**—When the contents of a cache can be accessed from multiple threads, use some form of locking mechanism to protect one thread from interfering with the data being accessed by another thread.

- **Serialization**—When storing an object in a cache and the cache storage serializes data in order to store it, the stored object must support serialization.
- **Normalizing cached data**—When storing state in a cache, make sure that it is stored in a format optimized for its intended usage.

For more information about formatting cached data, see the “Planning .NET Framework Element Caching” section in Chapter 4, “Caching .NET Framework Elements.”

Introducing Content Loading

Before you can use cached data, you must first load data into the cache. There are two methods you can use for loading data:

- **Proactive loading**—When using this method, you retrieve all of the required state, usually when the application or process starts, and then you cache it for the lifetime of the application or the process.
- **Reactive loading**—When using this method, you retrieve data when it is requested by the application and then cache it for future requests.

For more information about content loading, see the “Loading a Cache” section in Chapter 5, “Managing the Contents of a Cache.”

Introducing Expiration Policies

An important aspect of caching state is the way in which you keep it consistent with the master data (for example, the database or files) and other application resources. You can use expiration policies 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, such as a database, file, or other cached items.

For more information about expiration policies, see the “Determining a Cache Expiration Policy” in Chapter 5, “Managing the Contents of a Cache.”

Introducing Security

When caching any type of information, you need to be aware of various potential security threats. Data that is stored in a cache may be accessed or altered by a process that is not permitted access to the master data. This can occur because when the information is held in its permanent store, security mechanisms are in place to protect it. When taking data out of traditional trust boundaries, you must ensure that there are equivalent security mechanisms for the transmission of data to, and the storage of, data in the cache.

For more information about security issues when caching, see the “Securing a Custom Cache” section in Chapter 6, “Understanding Advanced Caching Issues.”

Introducing Management

When you use caching technologies, the maintenance needs of your application increase. During the application deployment, you need to configure things such as the maximum size of the cache and the flushing mechanisms. You also need to ensure that the cache performance is monitored, using some of the techniques made available in Windows and the .NET Framework (for example, event logging and performance counters).

For more information about maintenance issues, see the “Monitoring a Cache” section in Chapter 6, “Understanding Advanced Caching Issues.”

Summary

In this chapter, you have been introduced to some of the problems involved in developing distributed applications and how implementing caching within the application can help alleviate some of these issues. You have also been introduced to some of the considerations that you need to take into account when planning caching mechanisms for different types of applications.

Now that you have an overall understanding of the concepts involved in caching, you are ready to begin learning about the different caching technologies available.

2

Understanding Caching Technologies

Chapter 1 introduced you to the different types of state that can be used within a distributed .NET-based application and provided an overview of why and where to cache some of this state. You also learned about some of the issues you must bear in mind when you design the caching policy for your applications.

This chapter describes the different caching technologies you can use when you build distributed enterprise applications using the Microsoft .NET Framework. Other technologies can be used to cache data, such as NTFS file system caching, Active Directory® directory service caching, and the COM+ Shared Property Manager. However, in most cases, these methods are not recommended and are not described in this chapter.

This chapter describes the different caching technologies and Chapter 3, “Caching in Distributed Applications,” describes how to select the right technology to fit your needs.

This chapter contains the following sections:

- “Using the ASP.NET Cache”
- “Using Remoting Singleton Caching”
- “Using Memory-Mapped Files”
- “Using Microsoft SQL Server 2000 or MSDE for Caching”
- “Using Static Variables for Caching”
- “Using ASP.NET Session State”
- “Using ASP.NET Client Side Caching and State”
- “Using Internet Explorer Caching”

Using the ASP.NET Cache

Storing frequently used data in memory is not a new concept for ASP developers. ASP offers the **Session** and **Application** objects, which enable storing key/value pairs in memory. The **Session** object is used to store per-user data across multiple requests, and the **Application** object is used to store per-application data for use by requests from multiple users.

ASP.NET introduces a new key/value pair object—the **Cache** object. The scope of the ASP.NET cache is the application domain; therefore, you cannot access it from other application domains or processes. The ASP.NET **Cache** object's life span is tied to the application, and the **Cache** object is re-created every time the application restarts, similar to the ASP **Application** object. The main difference between the **Cache** and **Application** objects is that the **Cache** object provides cache-specific features, such as dependencies and expiration policies.

You can programmatically access the **Cache** object and work with it using its properties and methods. This allows you full access to the object and the ability to use advanced caching features. You can also use the ASP.NET cache to store output responses that are transmitted to a Web browser.

Using Programmatic Caching

The **Cache** object is defined in the **System.Web.Caching** namespace. You can get a reference to the **Cache** object by using the **Cache** property of the **HttpContext** class in the **System.Web** namespace or by using the **Cache** property of the **Page** object. You can also access cached information in a user control through the **Cache** property of the **UserControl** class. This is useful particularly when you need to access the cache directly from a user control. In addition to simply storing key/value pairs, the **Cache** object provides additional functionality specifically designed to store transient data, such as .NET Framework objects. Cache features, such as dependencies and expiration policies, also extend the capabilities of the ASP.NET cache.

Using Dependencies and Expirations

When you add an item to the cache, you can define dependency relationships that can force that item to be removed from the cache under specific circumstances. The supported dependencies include the following:

- **File dependency**—Allows you to invalidate a specific cache item when a disk-based file or files change.

The following example shows how to specify a file dependency when adding an item to the cache.

```
CacheDependency cDependency = new  
    CacheDependency(Server.MapPath("authors.xml"));  
Cache.Insert("CachedItem", item, cDependency);
```

- **Key dependency**—Invalidates a specific cache item when another cached item changes. For example, when you cache basic data alongside calculation results on that data, the calculated data should be invalidated when the basic data changes or becomes invalid. As another example, although you cannot directly remove a page from the ASP.NET output cache, you can use a key dependency on the page as a workaround. When the key on which your pages are dependent is removed from cache, your cached pages are removed as well.

The following example shows how to make one cache item dependent on another.

```
// Create a cache entry.
Cache["key1"] = "Value 1";

// Make key2 dependent on key1.
String[] dependencyKey = new String[1];
dependencyKey[0] = "key1";
CacheDependency dependency = new CacheDependency(null, dependencyKey);

Cache.Insert("key2", "Value 2", dependency);
```

- **Time-based expiration**—Invalidates a specific cached item at a predefined time. The time for invalidation can be absolute—such as Monday, December 1, at 18:00—or sliding, which resets the time to expire relative to the current time whenever the cached item is accessed. Examples of both types of time dependency are as follows.

```
/// Absolute expiration
Cache.Insert("CachedItem", item, null, DateTime.Now.AddSeconds(5),
    Cache.NoSlidingExpiration);
/// Sliding expiration
Cache.Insert("CachedItem", item, null, Cache.NoAbsoluteExpiration,
    TimeSpan.FromSeconds(5));
```

Using a duration that's too short limits a cache's usefulness; using a duration that's too long may return stale data and unnecessarily load the cache with pages that are not requested. If using page output caching does not improve performance, your cache duration might be too short. Try fast concurrent page requests to see whether pages are served rapidly from cache.

If you do not have a mechanism, such as dependencies, to invalidate a cached page, keep the duration short. In general, keep your cache duration shorter than the update interval of the data in your page. Usually, if you receive pages that are out of date, your cache duration is too long.

A common question is how to implement a cache database dependency. This requires you to implement a notification mechanism that informs your cache of changes to the original data in the database. For information about using external

notifications, see Chapter 5, “Managing the Contents of a Cache.” For a downloadable code sample that shows how to programmatically implement database dependencies, see Rob Howard’s team page at <http://www.gotdotnet.com/team/rhoward>.

When you need to access a cached item, you must check whether the item is in the cache before you try to use it. Because the ASP.NET cache invalidates cached items based on dependencies, time, or resources, you must always write code to create or retrieve the item you need if it does not currently exist in the cache.

The following example shows how you can do this.

```
string data = (string)Cache["MyItem"];
if (data == null)
{
    data = GetData();
    Cache.Insert("MyItem", data);
}
DoSomethingWithData(data);
```

Check whether data in the cache is valid before using it to avoid using stale items. Use this method when the cost of validating a cached item is significantly less than the cost of regenerating or obtaining the data.

Items added to the cache automatically expire and are removed from the cache if they are not used for a given amount of time. Use the expiration cache feature when the data from which the cached items generate is updated at known intervals.

Dependencies and expirations initiate the removal of items from a cache. Sometimes you want to write code to be called when the removal occurs.

Using Cache Callbacks

You can add a callback method to a cached item to execute when that item is removed from the cache. You can implement such a callback to ensure that the cached item is not removed from the cache or that the cache is updated based on new data.

The following example shows how to define a callback function when adding an item to the cache.

```
CacheItemRemovedCallback onRemove = new
    CacheItemRemovedCallback(this.RemovedCallback);
Cache.Insert("CachedItem",
    item,
    null,
    Cache.NoAbsoluteExpiration,
    Cache.NoSlidingExpiration,
    CacheItemPriority.Default,
    onRemove);

// Implement the function to handle the expiration of the cache.
public void RemovedCallback(string key, object value, CacheItemRemovedReason r)
{
}
```

```
// Test whether the item is expired, and reinsert it into the cache.
if (r == CacheItemRemovedReason.Expired)
{
    // Reinsert it into the cache again.
    CacheItemRemovedCallback onRemove = null;
    onRemove = new CacheItemRemovedCallback(this.RemovedCallback);
    Cache.Insert(key,
                value,
                null,
                Cache.NoAbsoluteExpiration,
                Cache.NoSlidingExpiration,
                CacheItemPriority.Default,
                onRemove);
}
}
```

Notice that in addition to expiration parameters, the **Insert** method also uses a **CacheItemPriority** enumeration.

Applying Priority to Cache Items

When the server running your ASP.NET application runs low on memory resources, items are removed from cache to free memory in a process known as *scavenging*. When memory is low, the cache determines which items are removed from cache based on priority. You can set the cache item priority when you add the item to the cache. Doing so controls which items scavenging removes first.

For more information about cache item priorities, see “CacheItemPriority Enumeration,” in the MSDN Library.

Flushing a Cache

There is no direct support for automatic flushing of the ASP.NET output cache. One way to do so is to set an external dependency that clears all cached items automatically. For example, the following statement flushes the ASP.NET output cache as soon as it executes.

```
Response.Cache.SetExpires(DateTime.Now)
```

This code flushes the output cache, but the page does not reflect this until the original cache duration completes. For example, if you use the following directive to configure your cache, the cache resets after 10 seconds.

```
<%@ OutputCache Duration="10" VaryByParam="none" %>
```

Flushing the entire output cache is generally not required, and better alternatives are available to your application design. For example, when using the ASP.NET **Cache** object, you can reload your cache items when they become stale, overwriting the existing cache content.

Using an Output Cache

You can use two types of output caching to cache information that is to be transmitted to and displayed in a Web browser: page output caching and page fragment caching. Page output caching caches an entire Web page and is suitable only when the content of that page is fairly static. If parts of the page are changing, you can wrap the static sections as user controls and cache the user controls using page fragment caching.

Using Page Output Caching

Page output caching adds the response of a given request to the ASP.NET cache object. After the page is cached, future requests for that page return the cached information instead of re-creating the entire page. You can implement page output caching by adding the necessary page directives (high-level, declarative implementation) or by using the **HTTPCachePolicy** class in the **System.Web** namespace (low-level, programmatic implementation).

This guide does not discuss the technical details of how page output caching works but concentrates on guidelines and best practices of how to use it correctly. For more information about how to implement page output caching, see “Page Output Caching,” in the MSDN Library.

Determining What to Cache with Page Output Caching

You can use the page output cache to cache a variety of information, including:

- Static pages that do not change often and that are frequently requested by clients.
- Pages with a known update frequency. For example, a page that displays a stock price where that price is updated at given intervals.
- Pages that have several possible outputs based on HTTP parameters, and those possible outputs do not often change—for example, a page that displays weather data based on country and city parameters.
- Results being returned from Web services. The following example shows how you can declaratively cache these results.

```
[WebMethod(CacheDuration=60)]
public string HelloWorld()
{
    return "Hello World";
}
```

Caching these types of output avoids the need to frequently process the same page or results.

Pages with content that varies—for example, based on HTTP parameters—are classed as dynamic, but they can still be stored in the page output cache. This is particularly useful when the range of outputs is small.

Caching Dynamic Pages

You may find yourself designing Web pages that contain dynamic content that is dependent upon input parameters, language, or browser type. ASP.NET lets you cache different versions of these pages based on changing data. By using applicable attributes on dynamically generated pages, you optimize cache usage and get better cache duration control.

You can use the following **OutputCache** attributes to implement this functionality:

- **VaryByParam**—Lets you cache different versions of the same page based on the input parameters sent through the **HTTP GET/POST**.
- **VaryByHeader**—Lets you cache different versions of the page based on the contents of the page header.
- **VaryByCustom**—Lets you customize the way the cache handles page variations by declaring the attribute and overriding the **GetVaryByCustomString** handler.
- **VaryByControl**—Lets you cache different versions of a user control based on the value of properties of ASP objects in the control.

The more specific you are in setting the values of these attributes—for example, supplying more of the HTTP parameters—the better the cache is used because it contains only relevant data. In essence, you are insulating what you are caching from changes in the underlying data. However, as the values become more specific, the cache uses memory less efficiently because the cache keeps more versions of the same page. The ASP.NET cache can erase pages by scavenging when memory becomes low.

Table 2.1 compares response times as the number of cached page versions increases.

Table 2.1: Caching performance

Pages cached	Responses/sec.	Time to first byte	Time to last byte
Less than 1,000	15.37	124.73	643.43
More than 1,200	3.15	2,773.2	3,153.63

Note: These figures are based on sample pages of 250 KB.

For more information about the caching attributes in ASP.NET cache, see “Page Output Caching” and “Caching Multiple Versions of a Page,” in the MSDN Library.

Configuring the Output Cache Location

You can control the location of your output cache by using the **Location** attribute of the **@OutputCache** directive. The **Location** attribute is supported only for page output caching and not for page fragment caching. You can use it to locate the cache on the originating server, the client, or a proxy server. For more information about

the **Location** attribute, see “OutputCacheLocation Enumeration,” in the MSDN Library.

The location of the cache is specified when you are configuring the cache for your application.

Configuring Page Output Caching

You can configure the cache both declaratively (using page directives) and programmatically (using the cache API). The declarative method can meet most of common requirements for configuring a cache, but the cache API includes some additional functionality, such as the ability to register callbacks.

The following example shows how to declaratively configure the cache using page directives in the ASP.NET page header.

```
<%@ OutputCache Duration="20" Location="Server" VaryByParam="state"
VaryByCustom="minorversion" VaryByHeader="Accept-Language"%>
```

The following example shows how to programmatically configure the cache using the cache API.

```
private void Page_Load(object sender, System.EventArgs e)
{
    // Enable page output caching.
    Response.Cache.SetCacheability(HttpCacheability.Server);
    // Set the Duration parameter to 20 seconds.
    Response.Cache.SetExpires(System.DateTime.Now.AddSeconds(20));
    // Set the Header parameter.
    Response.Cache.VaryByHeaders["Accept-Language"] = true;
    // Set the cached parameter to 'state'.
    Response.Cache.VaryByParams["state"] = true;
    // Set the custom parameter to 'minorversion'.
    Response.Cache.SetVaryByCustom("minorversion");
    ...
}
```

Both of these examples result in the same cache configuration.

Using Page Fragment Caching

Page fragment caching involves the caching of a fragment of the page, as opposed to the entire page. Sometimes full page output caching is not feasible—for example, when portions of the page need to be dynamically created for each user request. In such cases, it can be worthwhile to identify portions of the page or controls that do not often change and that take considerable time and server resources to create. After you identify these portions, you can wrap them in a Web Forms user control and cache the control so that these portions of the page don’t need to be recreated each time.

You can implement page fragment caching by adding the necessary directives (high-level, declarative implementation) to the user control or by using metadata attributes in the user control class declaration.

For more information about page fragment caching, see “Caching Portions of an ASP.NET Page,” in the MSDN Library.

Determining What to Cache with Page Fragment Caching

Use page fragment caching when you cannot cache the entire Web page. There are many situations that can benefit from page fragment caching, including:

- Page fragments (controls) that require high server resources to create.
- Sections on a page that contain static data.
- Page fragments that can be used more than once by multiple users.
- Page fragments that multiple pages share, such as menu systems.

Note: ASP.NET version 1.1, part of the Microsoft Visual Studio® .NET 2003 development system, introduces a new **Shared** attribute in the user control's `<%@ OutputCache %>` directive. This attribute allows multiple pages to share a single instance of a cached user control. If you don't specify the **Shared** attribute, each page gets its own instance of the cached control.

Configuring Page Fragment Caching

The following example shows how to implement fragment caching in a Web user control.

```
// Partial caching for 120 seconds
[System.Web.UI.PartialCaching(120)]
public class WebUserControl : System.Web.UI.UserControl
{
    // Your Web control code
}
```

When the page containing the user control is requested, only the user control—not the entire page—is cached.

Using the ASP.NET Cache in Non-Web Applications

The ASP.NET cache object is located in the **System.Web** namespace, and because it is a generic cache implementation, it can be used in any application that references this namespace.

The **System.Web.Caching.Cache** class is a cache of objects. It is accessed either through the static property **System.Web.HttpRuntime.Cache** or through the helper instance properties **System.Web.UI.Page** and **System.Web.HttpContext.Cache**. It is therefore available outside the context of a request. There is only one instance of this object throughout an entire application domain, so the **HttpRuntime.Cache** object can exist in each application domain outside of `Aspnet_wp.exe`.

The following code shows how you can access the ASP.NET cache object from a generic application.

```
HttpRuntime httpRT = new HttpRuntime();  
Cache cache = HttpRuntime.Cache;
```

After you access the cache object for the current request, you can use its members in the usual way.

Managing the Cache Object

ASP.NET supports a host of application performance counters that you can use to monitor the cache object activity. Monitoring these performance counters can help you locate and resolve issues in the performance of your ASP.NET cache. Table 2.2 describes these counters.

Table 2.2: Application performance counters for monitoring a cache

Counter	Description
Cache Total Entries	The number of entries in the cache
Cache Total Hits	The number of hits from the cache
Cache Total Misses	The number of failed cache requests per application
Cache Total Hit Ratio	The ratio of hits to misses for the cache
Cache Total Turnover Rate	The number of additions and removals to the total cache per second
Cache API Entries	The number of entries in the application (programmatic cache)
Cache API Hits	The number of hits from the cache when the cache is accessed through the external cache APIs (programmatic cache)
Cache API Misses	The number of failed requests to the cache when the cache is accessed through the external cache APIs (programmatic cache)
Cache API Hit Ratio	The cache ratio of hits to misses when the cache is accessed through the external cache APIs (programmatic cache)
Cache API Turnover Rate	The number of additions to and removals from the cache per second, when accessed through the external cache APIs (programmatic cache)
Output Cache Entries	The number of entries in the output cache
Output Cache Hits	The number of requests serviced from the output cache
Output Cache Misses	The number of failed output cache requests per application
Output Cache Hit Ratio	The percentage of requests serviced from the output cache
Output Cache Turnover Rate	The number of additions to and removals from the output cache per second

Note that the **Cache API** counters do not track internal usage of the cache by ASP.NET. The **Cache Total** counters track all cache usage.

The **Turnover Rate** counters help determine how effectively the cache is being used. If the turnover is large, the cache is not being used efficiently because cache items are frequently added and removed from the cache. You can also track cache effectiveness by monitoring the **Hit** counters.

For more information about these performance counters, see “Performance Counters for ASP.NET,” in the MSDN Library.

Using Remoting Singleton Caching

Microsoft .NET remoting provides a rich and extensible framework for objects executing in different AppDomains, in different processes, and on different computers to communicate seamlessly with each other. Microsoft .NET remoting singleton objects service multiple clients and share data by storing state information between client invocations.

You can use .NET remoting when you need a custom cache that can be shared across processes in one or several computers. To do this, you must implement a caching service using a singleton object that serves multiple clients using .NET remoting.

To implement a cache mechanism using .NET remoting, ensure that the remote object lease does not expire and that the remoting object is not destroyed by the garbage collector. An object lease is the period of time that a particular object can remain in memory before the .NET remoting system begins to delete it and reclaim the memory. When implementing a remoting singleton cache, override the **InitializeLifetimeService** method of **MarshalByRefObject** to return null. Doing so ensures that the lease never times out and the object associated with it has an infinite lifetime. The following example shows how to cause your object to have infinite lifetime.

```
public class DatasetStore : MarshalByRefObject
{
    // A hash table-based data store
    private Hashtable htStore = new Hashtable();
    //Returns a null lifetime manager so that GC won't collect the object
    public override object InitializeLifetimeService() { return null; }

    // Your custom cache interface
}
```

This code ensures that the garbage collector does not collect the object by returning a null lifetime manager for the object.

You can implement the remoting singleton as a caching mechanism in all layers of a multilayer architecture. However, because of the relatively high development cost of its implementation, you're most likely to choose this solution when a custom cache is needed in the application and data tiers.

You can often use solutions based on Microsoft SQL Server™ instead of remoting singleton caching solutions. It is tempting to choose remoting singleton caching because it is simple to implement, but the poor performance and the lack of scalability it produces mean that this choice is often misguided.

Using Memory-Mapped Files

Memory-mapped files offer a unique memory management feature that allows applications to use pointers to access files on disk—the same way that applications access dynamic memory. With this capability, you can map a view of all or part of a file on disk to a specific range of addresses in your process's address space. After you do so, accessing the content of a memory-mapped file is as simple as dereferencing a pointer in the designated range of addresses.

Both code and data are treated the same way in Windows: Both are represented by pages of memory, and both have those pages backed by a file on disk. The only difference is the type of file that backs them. Code is backed by the executable image, and data is backed by the system pagefile.

Because of the way that memory-mapped files function, they can also provide a mechanism for sharing data between processes. By extending the memory-mapped file capability to include portions of the system pagefile, applications are able to share data that is backed by that pagefile. Each application simply maps a view of the same portion of the pagefile, making the same pages of memory available to each application. Because all processes effectively share the same memory, application performance increases.

You can use these unique capabilities of memory-mapped files to develop an efficient custom cache that can be shared across multiple application domains and processes within the same computer. A custom cache based on memory-mapped files includes the following components:

- **Windows NT service**—Creates the memory-mapped file when it is started and deletes the memory-mapped file when it is stopped. Because each process using the cache needs a pointer to the memory-mapped file, you need a Windows NT® service to create the memory-mapped file and to pass handles to processes that want to use the cache. Alternatively, you can use a named memory-mapped file and get a handle to the memory-mapped file for each process by using the memory-mapped file name.

- **Cache management DLL**—Implements the specific cache functionality, such as:
 - Inserting and removing items from the cache.
 - Flushing the cache using algorithms such as Least Recently Used (LRU) and scavenging. For more information about flushing, see Chapter 5, “Managing the Contents of a Cache.”
 - Validating data to protect it against tampering or spoofing. For more information about security, see Chapter 6, “Understanding Advanced Caching Issues.”

Because a memory-mapped file is a custom cache, it is not limited to a specific layer or technology in your application.

Memory-mapped file caches are not easy to implement because they require the use of complex Win32® application programming interface (API) calls. The .NET Framework does not support memory-mapped files, so any implementations of a memory-mapped file cache run as unmanaged code and do not benefit from any .NET Framework features, including memory management features, such as garbage collection, and security features, such as code access security.

Management functionality for memory-mapped file custom cache needs to be custom developed for your needs. Performance counters can be developed to show cache use, scavenging rate, hit-to-miss ratios, and so on.

Using Microsoft SQL Server 2000 or MSDE for Caching

You can use Microsoft SQL Server 2000, and its scaled down version, Microsoft SQL Server 2000 Desktop Engine (MSDE), to cache large amounts of data. Although SQL Server is not the ideal choice when caching small data items because of its relatively slow performance, it can be a powerful choice when data persistency is critical or when you need to cache a very large amount of data.

Note: In caching terms, SQL Server 2000 and MSDE provide the same capabilities. All references to SQL Server in this section equally apply to MSDE.

If your application requires cached data to persist across process recycles, reboots, and power failures, in-memory cache is not an option. In such cases, you can use a caching mechanism based on a persistent data store, such as SQL Server or the NTFS file system. It also makes sense to use SQL Server to cache smaller data items to gain persistency.

SQL Server 2000 limits you to 8 KB of data when you are accessing **VarChar** and **VarBinary** fields using Transact-SQL or stored procedures. If you need to store larger items in your cache, use an ADO.NET **SQLDataAdapter** object to access **DataSet** and **DataRow** objects.

SQL Server caching is easy to implement by using ADO.NET and the .NET Framework, and it provides a common development model to use with your existing data access components. It provides a robust security model that includes user authentication and authorization and can easily be configured to work across a Web farm using SQL Server replication.

Because the cache service needs to access SQL Server over a network and the data is retrieved using database queries, the data access is relatively slow. Carefully compare the cost of recreating the data versus retrieving it from the database.

SQL Server caching can be useful when you require a persistent cache for large data items and you do not require very fast data retrieval. Carefully consider how much memory you have and how big your data items are. Do not use all of your memory to cache a few large data items because doing so degrades overall server performance. In such cases, caching items in SQL Server can be a good practice. Remember, though, that implementing a SQL Server caching mechanism requires installing, managing, and licensing at least one copy of SQL Server or MSDE.

When you use SQL Server for caching, you can use the SQL Server management tools, which include a host of performance counters that can be used to monitor cache activity. For example, use the **SQLServer:Databases** counters on your cache database table to monitor your cache performance, and use counters such as **Transactions/sec** and **Data File(s) Size (KB)** to monitor usage.

Using Static Variables for Caching

Static variables are often used to save class state, and you can use them to create customized cache objects. To do so, declare your cache data store as a static variable in your custom-developed cache class, and implement interfaces to insert, remove, and access items in the data store.

Using static variables is simple if no special cache capabilities are needed, such as dependencies, expiration, and scavenging. Because the cache is an in-memory system, static variables provide fast, direct access to the cached data. Static variable caching is useful to store state when other mechanisms are not available or are too costly. Use a static variable cache when you do not have another key/value dictionary available or when you need a fast in-process caching mechanism. In ASP.NET applications, use the **Cache** object instead.

You can use static variables to cache large data items, provided that the items do not often change. Because there is no mechanism for invalidating items, obsolete large items can waste valuable memory.

When using a static variable cache, you must ensure that you either implement your own thread safety mechanism or use a .NET Framework object that can be synchronized, for example, a **Hashtable** object.

The following example shows how to implement a static variable cache by using a hash table.

```
static Hashtable mCacheData = new Hashtable();
```

The scope of the static variables can be limited to a class or a module or be defined with project-level scope. If the variable is defined as public in your class, it can be accessed from anywhere in your project code, and the cache will be available throughout the application domain. The lifetime of static variables is directly linked to their scope.

Using ASP.NET Session State

You can use ASP.NET session state (based on the **HttpSessionState** class) to cache per-user session state. It solves many of the limitations that are inherent in ASP session state, including:

- The ASP session state is tied to the ASP hosting process, and as such is recycled when the ASP process is recycled. You can configure ASP.NET session state to avoid this.
- ASP session state has no solution for functioning in a Web server farm. The ASP.NET session state solution does not provide the best results when scalability and availability are important, but it works well for small scalar values, such as authentication information.
- ASP session state depends on the client browser accepting cookies. If cookies are disabled in the client browser, session state cannot be used. ASP.NET session state can be configured not to use cookies.

The ASP.NET session state is much improved. This section describes how and where session state is best used.

ASP.NET session state has three modes of operation:

- **InProc**—Session state is held in the memory space of the `Aspnet_wp.exe` process. This is the default setting. In this setting, the session state is recycled if the process or the application domain is recycled.
- **StateServer**—Session state is serialized and stored in a separate process (`Aspnet_state.exe`); therefore, the state can be stored on a separate computer (a state server).
- **SQLServer**—Session state is serialized and stored in a SQL Server database.

You can specify the mode to be used for your application by using the mode attribute of the `<sessionState>` element of the application configuration file.

The scope of the ASP.NET session state is limited to the user session it is created in. In out-of-process configurations—for example, using **StateServer** or **SQLServer**

mode in a Web farm configuration—the session state can be shared across all servers in the farm. However, this benefit does come at a cost. Performance may decrease because ASP.NET needs to serialize and deserialize the data and because moving the data over the network takes time.

For more information about ASP.NET session state, see “ASP.NET Session State,” in the MSDN Library.

Choosing the Session State Mode

Each mode of operation, **InProc**, **StateServer**, and **SQLServer**, presents different issues to consider when you design your caching policy.

Using InProc Mode

When you use **InProc** mode, you are storing the state in the `Aspnet_wp.exe` process. You cannot use **InProc** mode in a Web garden environment because multiple instances of `Aspnet_wp.exe` are running on the same computer.

InProc is the only mode that supports the **Session_End** event. The **Session_End** event fires when a user’s session times out or ends. You can write code in this event to clean up server resources.

Using StateServer Mode

StateServer mode stores the state in a dedicated process. Because it is an out-of-process system, you must ensure that the objects you cache in it are serializable to enable cross-process transmission.

When using the **Session** object for caching in a Web farm environment, ensure that the `<machineKey>` element in `Web.config` is identical across all of your Web servers. Doing so guarantees that all computers are using the same encryption formats so that all computers can successfully access the stored data. For more information about encryption, see “`<machineKey>` Element,” in the MSDN Library.

For session state to be maintained across servers in a Web farm, the application path of the Web site (for example, `\LM\W3SVC\2`) in the Internet Information Services (IIS) metabase must be identical across all servers in that farm. For more information about this, see article Q325056, “PRB: Session State Is Lost in Web Farm If You Use **SqlServer** or **StateServer** Session Mode,” in the Microsoft Knowledge Base.

Using SQLServer Mode

In **SQLServer** mode, ASP.NET serializes the cache items and stores them in a SQL Server database, so you must ensure that the objects you are using are serializable.

If you specify integrated security in the connection string used to access your session state (for example, `trusted_connection=true` or `integrated security=sspi`), you cannot use impersonation in ASP.NET. For more information about this problem, see

article Q324479, “FIX: ASP.NET SQL Server Session State Impersonation Is Lost Under Load,” in the Microsoft Knowledge Base.

For session state to be maintained across different servers in a Web farm, the application path of the Web site (for example, \LM\W3SVC\2) in the IIS metabase should be identical across all servers in that farm. For details about how to implement this, see article Q325056, “Session State Is Lost in Web Farm If You Use SqlServer or StateServer Session Mode,” in the Microsoft Knowledge Base.

You can configure ASP.NET session state to use SQL Server to store objects. Access to the session variables is slower in this situation than when you use an in-process cache; however, when session data persistency is critical, SQL Server provides a good solution. By default, SQL Server stores session state in the **tempdb** database, which is re-created after a SQL Server computer reboots. You can configure SQL Server to store session data outside of **tempdb** so that it is persisted even across reboots. For information about script files that you can use to configure SQL Server, see article Q311209, “Configure ASP.NET for Persistent SQL Server Session State Management,” in the Microsoft Knowledge Base.

When you use session state in SQL Server mode, you can monitor the SQL Server **tempdb** counters to provide some level of performance monitoring.

Determining What to Cache in the Session Object

You can use the **Session** object to store any of the .NET Framework data types; however, be aware of which sort of data is best suited to each cache mode.

You can use any mode to store .NET Framework basic types (such as **Int**, **Byte**, **String**) because ASP.NET uses an optimized internal method for serializing and deserializing basic data types when using out-of-process modes.

Store complex types (such as **ArrayList**) only in the **InProc** mode, because ASP.NET uses the **BinaryFormatter** to serialize data, which can affect performance. Note that serialization occurs only in the **StateServer** and **SqlServer** out-of-process modes.

For example, the **Session** object is ideal for storing user authentication data during a user session. Because this information is likely to be stored as a .NET Framework basic type (such as, **String**), it can use any of the three modes. However, you must consider the security aspects of storing sensitive data in a cache. The data can be accessed from different pages without requiring the user to re-enter login information.

Try to avoid using session state for large data items because this degrades your application performance. Table 2.3 on the next page compares response times as the size of the cache changes.

Table 2.3: Caching performance

Cache size	Responses per second	Time to first byte	Time to last byte
2 KB	447.38	19.43	19.65
200 KB	16.12	667.95	668.89

Because session state can function as both an in-process and an out-of-process cache, it is suitable for a wide range of solutions. It is simple to use, but it does not support data dependencies, expiration, or scavenging.

Implementing Session State

ASP.NET session state provides a simple interface for adding and removing variables and simple configuration using the Web.config file. Changes to this file are reflected immediately without the need to restart the ASP.NET process.

The following example shows how to implement **SQLServer** mode session caching in the Web.config file.

```
<sessionState
  mode="SQLServer"
  stateConnectionString="tcpip=127.0.0.1:42424"
  sqlConnectionString="data source=127.0.0.1; Integrated Security=SSPI"
  cookieless="false"
  timeout="20"
/>
```

The following example shows how you can use the session object to store and access user credentials.

```
private void SaveSession(string CartID)
{
    Session["ShoppingCartID"] = CartID;
}

private void CheckOut()
{
    string CartID = (string)Session["ShoppingCartID"];
    if(CartID != null)
    {
        // Transfer execution to payment page.
        Server.Transfer("Payment.aspx");
    }
    else
    {
        // Display error message.
    }
}
```

Remember to secure any credentials when caching them. For more information about securing custom caches, see “Securing a Custom Cache,” in Chapter 6, “Understanding Advanced Caching Issues.”

Using ASP.NET Client-Side Caching and State

You can reduce the workload on your server by storing page information using client-side options. This approach has minimal security support but results in faster server performance because the demand on server resources is modest. Because you must send information to the client before it can be stored, there is a practical limit on how much information you can store this way.

The main mechanisms to implement client-side caching are:

- Hidden fields
- View state
- Hidden frames
- Cookies
- Query strings

Each of these mechanisms is useful for caching different types of state in different scenarios.

Using Hidden Fields

You can store page-specific information in a hidden field on your page to maintain the state of that page. For more information about hidden fields, see “Introduction to Web Forms State Management,” in the MSDN Library.

It is best to store only small amounts of frequently changing data in hidden fields because this data is included in the roundtrips to the server on every postback. Storing a large amount of data slows your application because of network traffic load. ASP.NET provides the **HtmlInputHidden** control, which offers hidden field functionality.

Note: If you use hidden fields, you must submit your pages to the server by using the **HTTP POST** method rather than by requesting the page using the **HTTP GET** method.

Benefits and Limitations of Hidden Fields

The benefits of using hidden fields to store client-side state include:

- No server resources are required. The hidden field is stored and read directly from the page.
- Almost all browsers and client devices support forms with hidden fields.

- Hidden fields are simple to implement.
- Hidden fields are good for caching data in Web farm configurations because the data is cached on the client.

The limitations of using hidden fields are:

- The hidden field can be tampered with. The information in this field can be seen if the page output source is viewed directly, creating a potential security issue.
- The hidden field does not support rich structures; it offers only a single value field in which to place information. To store multiple values, you must implement delimited strings and write the code required to parse those strings.
- Page performance decreases when you store large values because hidden fields are stored in the page.

Hidden fields are a good solution for caching small amounts of noncritical, string data for use in Web pages.

Implementing Hidden Fields

The following example shows how to declare a hidden field on a Web page.

```
<input id="HiddenValue" type="hidden" value="Initial Value" runat="server"
NAME="HiddenValue">
```

This code creates a hidden field containing the string data **Initial Value**.

If you cannot use hidden fields because of the potential security risk, consider using view state instead.

Using View State

Web Forms pages and controls all have a **ViewState** property, which is a built-in structure for automatically retaining values among multiple requests for the same page. The view state is internally maintained as a hidden field on the page but is hashed, providing greater security than developer-implemented hidden fields do. For more information about view state, see “Introduction to Web Forms State Management,” in the MSDN Library.

You can use view state to store your own page-specific values across roundtrips when the page posts back to itself. For example, if your application is maintaining user-specific information, such as a user name that is displayed in a welcome message, you can store the information in the **ViewState** property.

Performance of view state varies depending on the type of server control to which it is applied. **Label**, **TextBox**, **CheckBox**, **RadioButton**, and **HyperLink** are server controls that perform well with **ViewState**. **DropDownList**, **ListBox**, **DataGrid**, and **DataList** suffer from poor performance because of their size and the large amounts of data making roundtrips to the server.

In some situations, using view state is not recommended because it can degrade performance. For example:

- Avoid using view state in pages that do not post back to the server, such as logon pages.
- Avoid using view state for large data items, such as **DataSets**, because doing so increases time in roundtrips to the server.
- Avoid using view state when session timeout is required because you cannot time out data stored in the **ViewState** property.

View state is a simple and easy-to-use implementation of hidden fields that provides some security features. Most of the performance considerations, benefits, and limitations of view state and hidden fields are similar.

For more information about **ViewState** performance, see “Maintaining State in a Control” in the MSDN Library.

Benefits and Limitations of View State

The benefits of using view state include:

- No server resources are required because state is contained in a structure in the page code.
- It is simple to implement.
- Pages and control state are automatically retained.
- The values in view state are hashed, compressed, and encoded, thus representing a higher state of security than hidden fields.
- View state is good for caching data in Web farm configurations because the data is cached on the client.

Limitations to using view state include:

- Page loading and posting performance decreases when large values are stored because view state is stored in the page.
- Although view state stores data in a hashed format, it can still be tampered with because it is stored in a hidden field on the page. The information in the hidden field can also be seen if the page output source is viewed directly, creating a potential security risk.

For more information about using view state, see “Saving Web Forms Page Values Using View State,” in the MSDN Library.

Implementing View State

The following example shows how to use the **ViewState** property to store and access user-specific information between page postbacks.

```
public class ViewStateSample : System.Web.UI.Page
{
    private void Page_Load(object sender, System.EventArgs e)
    {
        if (!Page.IsPostBack)
        {
            // Save some data in the ViewState property.
            this.ViewState["EnterTime"] = DateTime.Now.ToString();
            this.ViewState["UserName"] = "John Smith";
            this.ViewState["Country"] = "USA";
        }
    }

    #region Web Form Designer generated code
    ...
    #endregion

    private void btnRefresh_Click(object sender, System.EventArgs e)
    {
        // Get the saved data in the view state and display it.
        this.lblTime.Text = this.ViewState["EnterTime"].ToString();
        this.lblUserName.Text = this.ViewState["UserName"].ToString();
        this.lblCountry.Text = this.ViewState["Country"].ToString();
    }
}
```

When your application runs on a low bandwidth client—for example, a mobile device—you can store the view state on the server by using **Losformatter** and thereby returning slimmer pages with less of a security risk.

Using Hidden Frames

You can use hidden frames to cache data on the client, avoiding the roundtrips to the server that are inherent in hidden field and view state implementations. You create a hidden frame in your page that loads a Web page into the frame containing your data. The pages in your application can access this data using client-side scripting. This implementation does not require any server resources because the data fields in the hidden frame are stored and read directly from the page.

Hidden frames let you secretly load images for use on other pages within the site. When those images are required, they can be obtained from the client cache rather than from the server.

Benefits and Limitations of Hidden Frames

The benefits of using hidden frames to store client-side state include:

- The ability to cache more than one data field.
- The avoidance of roundtrips of data during postbacks.
- The ability to cache and access data items stored in different hidden forms.
- The ability to access JScript® variable values stored in different frames if they come from the same site.

The limitations of using hidden frames are:

- Functionality of hidden frames is not supported by all browsers. Do not rely on hidden frames to provide data for the essential functionality of your pages.
- The hidden frame can be tampered with, and the information in the page can be seen if the page output source is viewed directly, creating a potential security threat.
- There is no limit to the number of frames (embedded or not) that you can hide. In frames that bring data from the database and that contain several Java applets or images, a large number of frames can negatively affect performance of the first page load.

Hidden frames are useful for storing many items of data and resolve some of the performance issues inherent in hidden field implementations.

Implementing Hidden Frames

The following example shows how to declare a hidden frame in a Web page.

```
<FRAMESET cols="100%,*">
  <FRAMESET rows="100%,*">
    <FRAME src="contents_of_frame1.html">
  </FRAMESET>
  <FRAME src="contents_of_hidden_frame.html">
  <FRAME src="contents_of_hidden_frame.html" frameborder="0" noresize
    scrolling="yes">
  <NOFRAMES>
    <P>This frameset document contains:
    <UL>
      <LI>
        <A href="contents_of_frame1.html" TARGET="_top">
          Some neat contents</A>
      <LI>
        <A href="contents_of_hidden_frame.html" TARGET="_top">
          Some other neat contents</A>
      </UL>
    </NOFRAMES>
  </FRAMESET>
```

You can then use the data stored in this frame in your Web page by using client-side scripting.

Using Cookies

You can use cookies to store small amounts of infrequently changing information on the client. That information can then be included in a request to the server.

Benefits and Limitations of Cookies

Cookies are useful because:

- No server resources are required. The cookie is stored on the client, sent to the server in a request, and then read by the server after the post.
- They are simple to use. The cookie is a lightweight, text-based structure containing simple key/value pairs.
- They support configurable expiration. The cookie can expire when the browser session ends, or it can exist indefinitely on the client computer, subject to the expiration rules on the client.

The use of cookies may be limited by:

- Size restrictions. Most browsers place a 4096-byte limit on the size of a cookie, although support for 8192-byte cookies is becoming more common in the new browser and client-device versions available today.
- User-configured refusal. Some users disable their browser or client device's ability to receive cookies, thereby limiting the use of cookies.
- Security breaches. Cookies can be subject to tampering. Users can manipulate cookies on their computer, which can potentially represent a security compromise or cause a cookie-dependent application to fail.
- Possible expirations. The durability of the cookie on a client computer is subject to cookie expiration processes on the client and to user intervention.

Note: Cookies are often used for personalization, in which content is customized for a known user. In most cases, identification is the issue rather than authentication, so it is usually enough to merely store the user name, account name, or a unique user ID (such as a GUID) in a cookie and use it to access the user personalization infrastructure of a site.

For more information about creating and reading cookies, see “`HttpResponse.Cookies Property`” and “`HttpRequest.Cookies Property`,” in the MSDN Library.

Implementing Cookies

The following example shows how to store and retrieve information that cookies hold.

```

public class CookiesSample : System.Web.UI.Page
{
    private void Page_Load(object sender, System.EventArgs e)
    {
        if (this.Request.Cookies["preferences1"] == null)
        {
            HttpCookie cookie = new HttpCookie("preferences1");
            cookie.Values.Add("ForeColor", "black");
            cookie.Values.Add("BackColor", "beige");
            cookie.Values.Add("FontSize", "8pt");
            cookie.Values.Add("FontName", "Verdana");
            this.Response.AppendCookie(cookie);
        }
    }

    private string getStyle(string key)
    {
        string val = null;
        HttpCookie cookie = this.Request.Cookies["preferences1"];
        if (cookie != null)
        {
            val = cookie.Values[key];
        }
        return val;
    }
}

```

Never rely on cookies for essential functionality in your Web applications, because users can disable them in their browsers.

Using Query Strings

A query string is information sent to the server appended to the end of a page's URL. You can use a query string to submit data back to your page or to another page through the URL.

Query strings provide a simple but limited way of maintaining some types of state information. For example, they make it easy to pass information from one page to another, such as passing a product number to another page, where it is processed.

Note: Query strings are a viable option only when a page is requested through its URL using **HTTP GET**. You cannot read a query string from a page that has been submitted to the server using **HTTP POST**.

Benefits and Limitations of Query Strings

Query strings offer the following benefits:

- No server resources are required. The query string is contained in the HTTP request for a specific URL.
- They provide broad support. Almost all browsers and client devices support passing values in a query string.
- They are simple to implement. ASP.NET provides full support for the query string method, including methods for reading query strings using the **HttpRequest.Params** property.

Query strings do have some limitations; for example:

- The information in the query string is directly visible to the user in the browser user interface. The query values are exposed to the Internet in the URL, so in some cases, security might be an issue.
- Most browsers and client devices impose a 255-character limit on URL length.

Query strings can be very useful in certain circumstances, such as when you are passing parameters to the server to customize the formatting of the data returned.

Implementing Query Strings

The following example shows how to include a query string in a URL.

```
http://www.cache.com/login.asp?user=ronen
```

The following example shows how to access the information at the server.

```
// Check for a query string in a request.  
string user = Request.QueryString["User"];  
if( user != null )  
{  
    // Do something with the user name.  
}
```

Query strings enable you to send string information directly from the client to the server for server-side processing.

Using Client-Side Caching Technologies

Table 2.4 shows recommendations for the use of each client-side caching technology discussed.

Table 2.4: Client-side state technologies

Mechanism	Recommended uses
Hidden fields	To store small amounts of information for a page that posts back to itself or to another page when security is not an issue. You can use a hidden field only on pages that are submitted to the server.
View state	To store small amounts of information for a page that posts back to itself. Using view state provides basic security.
Hidden frames	To cache data items on the client and to avoid the roundtrips of the data to the server.
Cookies	To store small amounts of information on the client when security is not an issue.
Query strings	To transfer small amounts of information from one page to another when security is not an issue. You can use query strings only if you are requesting the same page or another page using a link.

Using Internet Explorer Caching

Microsoft Internet Explorer provides mechanisms for caching pages or page objects on the user's computer. In Internet Explorer, you can cache data on the client rather than on the server, thus reducing server and network load to a minimum.

Internet Explorer caching is not suitable for all situations. In this section, you learn how to best utilize Internet Explorer caching in your applications.

Note: Internet Explorer caching is supported only in an Internet Explorer client environment. If other client browsers are also being used, you must use customization code in your application and provide an alternative caching solution for other types of browser.

Understanding Internet Explorer Cache Types

You can store information in the Internet Explorer cache either by specifying a time that the page should expire from the cache or by using dynamic HTML (DHTML) behaviors:

- You can add an **EXPIRES** directive to the header of objects so that Internet Explorer caches them for the specified time period. When the browser needs to access this data, the expiration date is queried, and if the date is in the future, the cached version of the data is used. If the expiration date has passed, the browser contacts the server for the current version of the data. As a result, any sites that use the **EXPIRES** header perform better.

- You can use persistence through DHTML behaviors that allow you to store information on the client. Doing so reduces the need to query a server database for client-specific information, such as color settings or screen layout preferences, and increases the overall performance of the page. Persistence stores the data hierarchically, making it easier for the Web developer to access.

Note: Users can choose to work offline by selecting **Work Offline** on the **File** menu in Internet Explorer. When **Work Offline** is selected, the system enters a global offline state independent of any current network connection, and content is read exclusively from the cache.

Another method of making Internet Explorer cache pages is by manually setting the page expiration by using the Web pagefile properties in IIS. The site administrator can perform this operation.

► **To set the expiration date manually in IIS**

1. Right-click the file name.
2. Click **Properties**, and then click the **HTTP Headers** tab.
3. Select the **Enable Content Expiration** check box.
4. Set a date in the future, and then click **OK**.

For more information about the Internet Explorer cache, see “Control Your Cache, Speed up Your Site” in the MSDN Library.

Determining What to Cache in the Internet Explorer Cache

You should use the Internet Explorer cache for static page items that do not change often, such as:

- Images and bitmaps on HTML pages.
- Static text objects.
- Page banners and footers, which contain general information or navigation menus that seldom change. Retrieving this data from the Internet Explorer cache gives the user an immediate response while the rest of the page data is fetched from the server.
- Site home pages that rarely change. This page can be stored in the Internet Explorer cache to give the user a prompt response when navigating to the site.
- Client-specific data that uses DHTML persistence. This solution is limited to browsers that support DHTML, such as Internet Explorer 4.0 and later.

Do not use the Internet Explorer cache for dynamic content because the cached copy on the client soon becomes stale.

Understanding Benefits and Limitations of the Internet Explorer Cache

Using the Internet Explorer cache provides the following benefits:

- Reduced network traffic because the client does not need to contact the server for all of the content.
- Facilitation of offline browsing. Internet Explorer cached data is available for offline viewing on the client when it is disconnected from the network.
- The ability to build hierarchical data structures using the XML Document Object Model (DOM), using only DHTML persistence. The page author can use the formatted data structure supported by XML standards to store complex data in the cache, which is easy to navigate and retrieve.

The Internet Explorer cache does have some limitations:

- Cached data expiration time has to be predetermined and cannot depend on server data updates. Internet Explorer currently supports a lazy update scheme, in which cached data loads first. Then Internet Explorer queries the server to check whether the data has been updated. If so, the data is fetched to the Internet Explorer cache to prepare for the next time the client requests the data.
- Internet Explorer cached data is not encrypted. You must avoid storing sensitive data such as credit card numbers in the Internet Explorer cache because anyone with access to the cache folder can view the files and the data cached in them.

The Internet Explorer cache is ideal for improving performance for pages containing large static items.

Implementing Internet Explorer Caching

Add the following header to your page or object header to control how Internet Explorer caches the page.

```
<META HTTP-EQUIV="expires" CONTENT="Tue, 23 Jun 2002 01:46:05 GMT">
```

This example caches the page or object until the date specified by the value attribute.

Managing the Internet Explorer Cache

Internet Explorer lets the user manage the cache size and cache update frequency by changing the settings in the **Internet Explorer Options** dialog box. You can change the cache size to match the amount of free disk space on the client computer.

The Internet Explorer cache can be managed only from the client computer and cannot be controlled from the server.

Summary

This chapter introduced you to various technologies available for implementing caching mechanisms in .NET-based applications. You saw examples of how to implement these mechanisms and learned the benefits and limitations of each.

The next step is to map how to use these technologies in the various application layers and deployment scenarios in distributed applications.

3

Caching in Distributed Applications

The previous chapters of this guide have reviewed the available caching technologies that exist in the Microsoft .NET Framework and Microsoft Windows, and has explored their benefits and limitations. The next step is to map them to the application layers and elements that exist in distributed applications. This helps when you are selecting the appropriate caching technique for a specific layer and element.

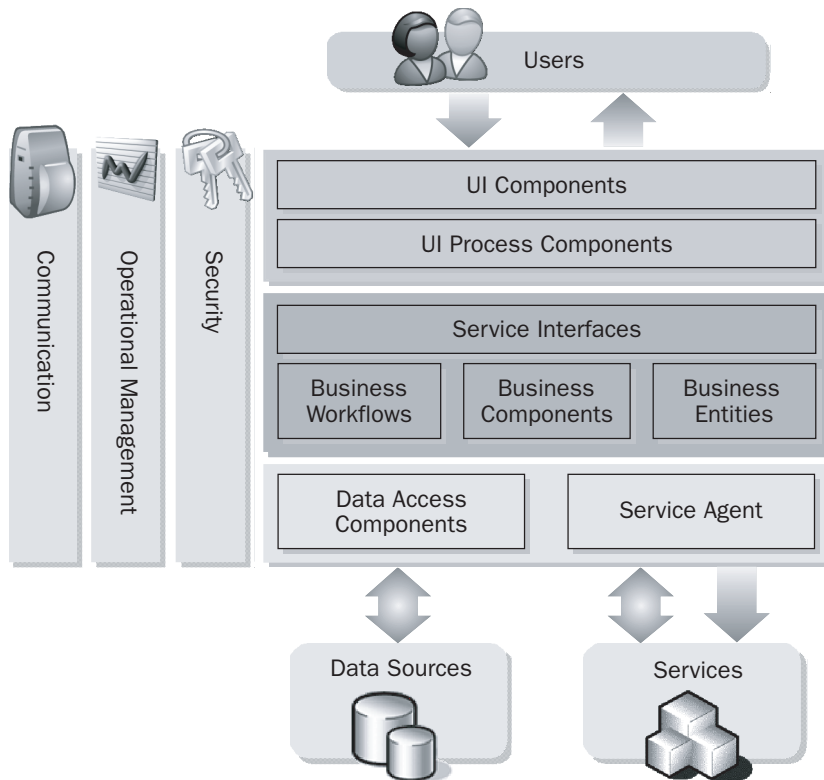
This chapter contains the following sections:

- “Caching in the Layers of .NET-based Applications”
- “Selecting a Caching Technology”
- “Considering Physical Deployment Recommendations”

Caching in the Layers of .NET-based Applications

This section provides a summarized review of .NET-based application architecture, its common elements, and the roles of those elements. You will find this information useful later in the chapter when identifying which caching technologies are most appropriate for each element in a typical .NET-based distributed application and deployment configuration.

Figure 3.1 on the next page shows an abstracted view of .NET-based distributed application architecture containing three logical application layers: user services, business services, and data services. The abstraction of a distributed application into these logical layers provides the means for building an application according to required design goals such as scalability, availability, security, interoperability, and manageability.

**Figure 3.1**

.NET-based distributed application architecture

For a complete overview of the application layers, see “Application Architecture for .NET: Designing Applications and Services” in the MSDN Library.

Caching in the User Services Layer

The user services layer is responsible for enabling the user to interact with the system. This layer includes rich client-based applications using the .NET Framework Windows Forms user interface and Web-based systems using HTML.

The layer comprises of two types of components: user interface (UI) components and UI process components. The next sections describe these components.

Caching in UI Components

Most solutions need to provide some way for users to interact with the application. User interfaces can be implemented using Windows Forms, ASP.NET pages, controls, or any other technology capable of rendering and formatting data for users and acquiring and validating data from users.

You should use UI components to cache data such as:

- ASP.NET pages
- ASP.NET page fragments
- Windows Forms controls with a long render time (for example, TreeView controls)

Because UI personalization data is relevant to a specific UI technology, you may also decide to cache this in the UI components of your application.

Caching these types of data in UI components can improve the performance of your applications.

Caching in UI Process Components

UI process components are used to separate client-side operations, such as authorization, validation, process management, and state management, from the actual user interface. By using these components instead of hard-coding the logic into the user interface, the same components can be reused for multiple user interfaces and applications.

You should use UI process components to cache data such as:

- User credentials
- Business data
- Configuration data
- Metadata

Caching these types of data in UI process components can improve the performance of your applications.

Caching Data in User Services Layer Elements

If you are caching business data in user services layer elements, it is recommended that you store it in the UI process components (as shown in Figure 3.2) rather than in the UI components.

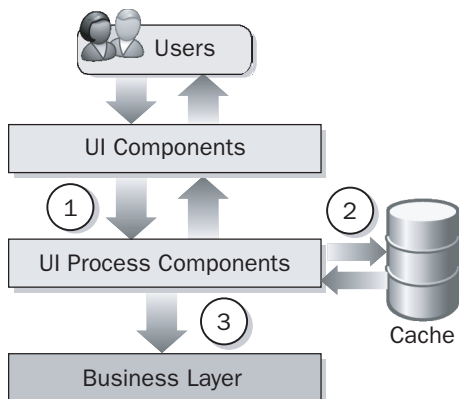


Figure 3.2

Caching business data in the users services layer

The steps that occur when caching data in user service elements are:

1. The UI component elements make a request for data from the UI process components.
2. The UI process component checks if a suitable response is present in the cache, and if so, returns it.
3. If the requested data is not cached or not valid, the UI process components delegate the request to the business layer elements and, if appropriate, place the response in the cache.

This configuration means that you can reuse the same caching technique for multiple user interfaces.

Caching in the Business Services Layer

Business components are usually designed and built to be deployed in Web farm and application farm environments. Because of this, they are designed to not hold state beyond request/response pairs, and are thus termed *stateless components*.

You should strive to build your components in this way, especially in transactional applications or applications that have components running in Web farms or application farms. Caching in the business services layer should be used only to hold certain types of state that you have determined are good caching candidates; for example, holding non-transactional reference data that is usable across many activities from different callers. For more details about the sort of data that should be cached, see Chapter 1, “Understanding Caching Concepts.”

The business services layer encapsulates the core business logic of an application and provides the delivery channels with an easy interface to interact with. The business services functionality is independent from both the underlying data sources and delivery channels.

Caching in Service Interfaces

To expose business logic as a service, you must create service interfaces that support the communication contract (message-based communication, formats, protocols, security, exceptions, and so on) that are needed by its different consumers.

You should use service interface elements to cache data such as service state representing non-transactional business data.

Caching in Business Workflows

Many business processes involve multiple steps that must be performed in the correct order and orchestrated. For example, a retail system would need to calculate the total value of the order, validate the credit card details, process the credit card payment, and arrange delivery of the goods. This process could take an indeterminate amount of time to complete, so the required tasks and the data required to perform them must be managed. Business workflows define and coordinate long

running, multi-step business processes, and they can be implemented using business process management tools such as Microsoft BizTalk® Server Orchestration Designer.

You can store the state being used in a business workflow between the workflow steps to make it available throughout the entire business process. The state can be stored in a durable medium, such as a database or a file on a disk, or it can be stored in memory. The decision of which storage type to use is usually based on the size of the state and the durability requirements.

Caching in Business Components

Business components have the responsibility of implementing an application's business activities; they implement business rules and perform business tasks in a request-response fashion.

You should use business components to cache data such as information in different stages of the transformation pipeline.

Caching in Business Entities

Most applications require data to be passed between components. For example, in a retail application, a list of products must be passed from the data access components to the user interface components so that the product list can be displayed to the users. The data is used to represent real world business entities, such as products or orders. The business entities that are used inside the application are usually data structures such as DataSets, DataReaders, or XML streams, but they could also be implemented using custom object-oriented classes that represent the real world entities your application has to deal with, such as a product or an order.

When caching in business entity elements, you need to consider whether to cache the data in its native format or in the format in which it is used. Based on this consideration, there are two main patterns for caching business entities elements.

Figure 3.3 shows the first pattern, where the business entities are stored in their native format in the cache storage and retrieved as necessary. The main advantage of this pattern is that data stored in its original format and does not need to be translated between different formats.

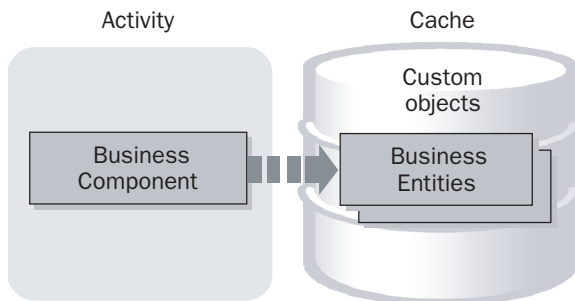


Figure 3.3

Storing business entities in the cache

Figure 3.4 shows the second pattern, where a business entities factory is used to retrieve relevant state. In this pattern, the state is stored in standard formats and translated to its original format when the data is required.

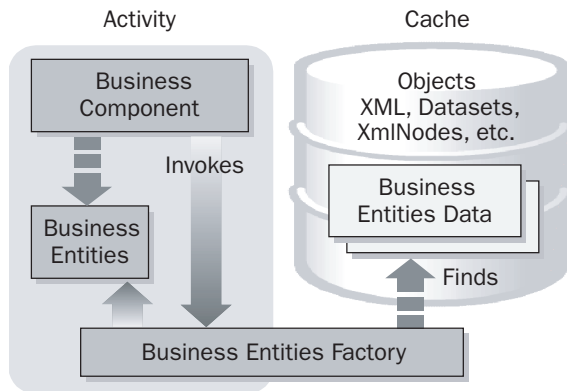


Figure 3.4

Storing data for business entities in the cache

You should store business entities in a cache only when the translation of the data in a business entities factory is expensive.

Caching in the Data Services Layer

The data services layer consists of data access components, data access helpers, and service agents. These elements work together to request and return information from many types of data source to your application. Because of this, they are classic elements in which to cache data.

Caching in Data Access Components

These components are used for centralizing access to data stores. They implement any logic necessary to access data stores thus making the application easier to manage and configure. You should use data access components to cache data such as:

- Raw data returned from queries in a non-transactional context
- Typed dataset schemas

Caching these types of information in data access components can improve the performance of your applications.

Caching in Data Access Helpers

These components are data store specific components used for optimizing access to the underlying data store and to minimize data store related programming errors. For example, a Microsoft SQL Server data access helper should implement the optimized way to connect to the SQL Server, to perform queries, or to execute stored procedures.

You should use data access helpers to cache data such as stored procedure parameters. (For more information, see the `SqlHelperParameterCache` class in “Data Access Application Block” in the MSDN Library.)

Caching in Service Agents

Service agents are components that function as proxies for other application services. These components implement the functionality required for interacting with the services. Figure 3.5 shows how they work.

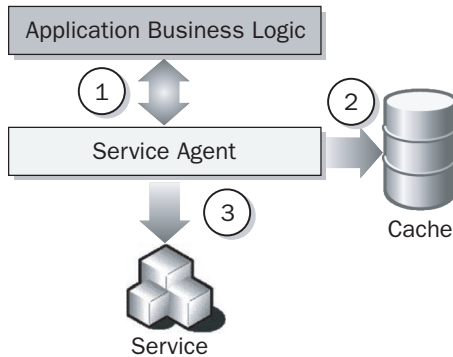


Figure 3.5
Caching process in service agents

The steps that occur when caching data in service agents are:

1. The application business logic invokes the service agent.
2. The service agent checks whether a suitable response (for example, valid data that meets the requested criteria) is present in the cache, and if so, returns it.
3. If the requested data is not cached or not valid, the Web service is invoked, and if appropriate, the requested data is also placed in the cache.

You should use service agents to cache data such as:

- State returned from queries in a non-transactional context
- User credentials

Caching these types of data in service agents can improve the performance of your applications.

Caching in the Security Aspects

Security policy is concerned with authentication, authorization, secure communication, auditing, and profile management, as shown in Figure 3.6 on the next page.

**Figure 3.6***Aspects of security*

Not all of these aspects are relevant to the subject of caching, and this section describes only profile management.

Caching Profile Information

User profiles consist of information about the user that your application can use to customize its behavior. It may have user interface preferences (for example, background colors) and data about the user (for example, the region he or she is in, credit card details, and so on). You may decide to cache profile information for offline application scenarios and to enhance performance.

If the profile information contains sensitive data, you should consider signing, encrypting, or hashing it to ensure that it can't be read and that isn't been tampered with.

Caching in the Operational Management Aspects

Operational management policy is concerned with the ongoing, day-to-day running of an application, and it includes aspects such as exception management, monitoring, metadata, configuration, and service location, as shown in Figure 3.7. Not all of these are relevant to the subject of caching, and this section describes only the relevant aspects.

**Figure 3.7***Aspects of operational management*

For more information about operational management in distributed applications, see Chapter 3, “Security, Operational Management, and Communication Policies,” of “Application Architecture for .NET: Designing Applications and Services” in the MSDN Library.

Caching Configuration Information

Your applications may require configuration data to function technically. Configuration information is generally maintained in .NET configuration files at the user, computer, and application levels. However, you can also use different storage types, such as XML files, SQL Server databases, Active Directory, COM+, and the Windows registry to store your application configuration data. Not all of these are recommended configuration stores, but they do offer adequate functionality.

Frequently accessing configuration information and metadata can cause a performance hit if the configuration information and the metadata are located in a file on a disk or in a database, especially if it is stored remotely. You can cache application-managed configuration information and metadata in memory to prevent this; however, you should ensure that you are not creating a security hole by exposing sensitive information to the wrong application code.

By using expiration policies (such as absolute time and sliding time windows) and dependencies (such as file dependencies), you can keep your cached configuration data up to date with the master configuration data. A good example of this is the way the .NET Framework handles its configuration files (Web.config and Machine.config), which are loaded into memory as soon as they are updated, causing the application to use the new configuration data as soon as it is available.

For more information about caching configuration files, see Chapter 4, “Caching .NET Framework Elements.” For more information about handling cache content, see Chapter 5, “Managing the Contents of a Cache.”

Common caching candidates in the configuration category are:

- Location information that is required to reach remote business layer components, external services, message queuing, and so on.
- Connection data such as connection strings and file names.

For more information about caching connection strings see Chapter 4, “Caching .NET Framework Elements.”

Caching Metadata

You may design your application so that it knows information about itself to make it more flexible to changing runtime conditions. This type of information is known as *metadata*. Designing your application using metadata in certain places can make it more maintainable and lets it adapt to common changing requirements without costly redevelopment or deployment.

Metadata can be stored in multiple places, as described in the preceding “Caching Configuration Information” section. For centralized stores, you can use SQL Server databases or Active Directory. If you want your metadata to be distributed with your assemblies, you can implement it in XML files, or even custom .NET attributes.

A summary of good metadata caching candidates follows, while further discussion on the subject can be found in Chapter 3, “Security, Operational Management, and Communication Policies,” of “Application Architecture for .NET: Designing Applications and Services” in the MSDN Library.

Common caching candidates in the metadata category are:

- **Column headers captions**—Instead of hard-coding column header names into your UI components, you can map a data table or a dataset to header captions stored in a cached metadata repository.
- **Error messages**—By raising error numbers to the UI and translating them into user friendly error messages instead of retrieving them from a database or resource files, you can improve application performance.
- **User assistance text**—This includes short help messages and ToolTip text.
- **Menu hierarchies’ structure**—This includes XML Schema Definitions (XSDs).
- **XML schemas**—This includes XSDs, XML Data Reduced Schemas (XDRs), and Document Type Definitions (DTDs).
- **Extended metadata**—In some situations you may want to provide developers with additional metadata such as:
 - **Logical naming**—In some cases, mapping logical names to physical entities makes programming tasks easier. For example, instead of letting the developer access a certain stored procedure using its physical name (`sp_GetCustomersByName`), a logical name (`Customers`) can be provided and translated to the actual name in the data services layer.
 - **Filter restrictions**—Providing information about which fields in a data table can be used for filtering based on the available stored procedures. For example, you may want to enable filtering only on the first and last name fields in a customers form. By providing filter restrictions metadata, a client application can determine whether to enable filtering.
 - **Sort restrictions**—Providing information about which fields in a data table can be used for sorting that table based on the available stored procedures. For example, you may want to enable sorting only on the first and last name fields in a customers form. By providing sorting restrictions metadata a client application can determine whether to enable sorting or not.

Now that you are aware of the layers within an application’s architecture, you are ready to start deciding the caching technology that is best to be used in each layer.

Selecting a Caching Technology

The previous chapters explored the available caching technologies and set the general frame for further discussion on the subject; this section describes the actual considerations for selecting a caching technology. The matrix for selecting a technology can be very large and sometimes confusing, and there are a large number of variables that you need to consider. To simplify the selection process, this section focuses on only the major considerations and elements in a common .NET-based distributed application.

This section describes the main elements in a distributed application and maps them to the best caching technology based on different requirements such as state scope, durability attributes, and common scenarios in which they are used. The focus is on the following elements:

- Browser-based clients
- Smart clients (based on the .NET Framework or the .NET Compact Framework)
- User services (ASP.NET server applications)
- Business services and data services (enterprise services or non-enterprise services)

Figure 3.8 shows typical elements that exist in a distributed application. Every application has a UI representation, either thin or rich, and every application has server side logic; for example, ASP.NET for Web applications and business logic, which may or may not use enterprise services.

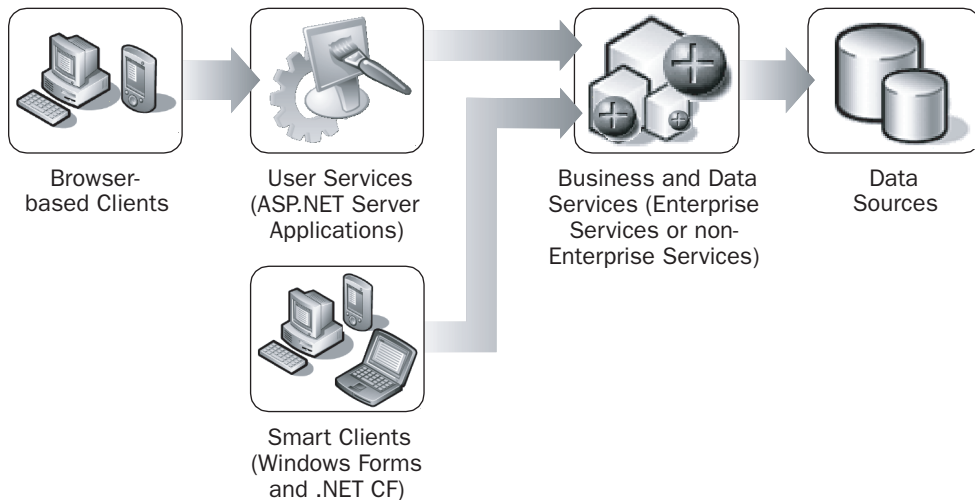


Figure 3.8
Common elements in a distributed application

Note: Although the .NET Framework and the Windows platform provide many caching, design, and deployment options, this section concentrates on the solutions that most closely suit distributed applications.

Caching in Browser-based Clients

Chapter 2 contains an in-depth review of the options available for caching state in browser-based scenarios. The main issues that arise when caching state in browsers are that the technology options available are very limited and the caching functionality varies between different browser types. Also, when caching any information on a client, you need to be sure that security is not compromised.

When you consider whether to cache state in a browser, you should consider the following:

- Does caching state in the client create a security risk?
- Do all of the target browsers support the chosen caching technology?

The following tables compare the scope, durability, and scenario usage of browser-based caching technologies. Table 3.1 compares the scope of different technologies available for caching data for browser-based clients.

Table 3.1: Scope of browser-based caching technologies

Technology	Single page	Multiple page (same window)	Multiple windows (same window)
ViewState	✓	Additional implementation required.	
Cookies	✓	✓	✓
Hidden frames	✓	✓	

Table 3.2 compares the durability of different technologies available for caching data for browser-based clients.

Table 3.2: Durability of browser-based caching technologies

Technology	None	Survives browser shutdown	Survives reboots
ViewState	✓		
Cookies	Subject to expiration rules in client.	Subject to expiration rules in client.	Subject to expiration rules in client.
Hidden frames	✓		

Table 3.3 shows commonly used scenarios for each of these technologies.

Table 3.3: Browser-based caching scenarios

Technology	Scenario
ViewState	Small amounts of data used between requests of the same page.
Cookies	Small amounts of data used between requests of the same application.
Hidden frames	Relatively large amounts of data accessed only from a client-side script.

Use the information in the preceding tables to select the appropriate caching technology for your browser-based clients.

Note: Caching state in a browser results in the cached information being stored as clear text. When storing sensitive data in this way, you should consider encrypting the data to avoid tampering.

Caching in Smart Clients

The following tables focus on client-side applications based on the .NET Framework Windows Forms technology. Table 3.4 compares the scope of different technologies available for caching data in Windows Forms applications.

Table 3.4: Scope of smart client caching technologies

Technology	Single AppDomain	Computer	Organization
Static variables	✓		
Memory-mapped files		✓	
Remoting singleton			✓
SQL Server 2000 / MSDE			✓

Table 3.5 compares the durability of different technologies available for caching data in Windows Forms applications.

Table 3.5: Durability of smart client caching technologies

Technology	None	Survives process recycles	Survives reboots
Static variables	✓		
Memory-mapped files	✓		
Remoting singleton	✓		
SQL Server 2000 / MSDE		✓	✓

Table 3.6 shows commonly used scenarios for each of these technologies.

Table 3.6: Smart client caching scenarios

Technology	Scenario
Static variable	In-process, high performance caching.
Memory-mapped files	Computer-wide, high performance caching.
Remoting singleton	Organization-wide caching (not high performance).
SQL Server 2000 / MSDE	Any application which requires high durability.

Use the information in the preceding tables to select the appropriate caching technology for your smart clients.

Caching in .NET Compact Framework Clients

Although .NET Compact Framework applications have not been thoroughly described so far in this guide, the caching mechanisms and recommendations for them are simply a subset of those available to other types of applications. This section focuses on client-side applications built using the .NET Compact Framework. Table 3.7 compares the scope of different technologies available for caching data in .NET Compact Framework applications.

Table 3.7: Scope of .NET Compact Framework caching technologies

Technology	Single AppDomain	Computer	Organization
SQL Server CE		✓	Not applicable
Static variables	✓		
File system		✓	

Table 3.8 compares the durability of different technologies available for caching data in .NET Compact Framework applications.

Table 3.8: Durability of .NET Compact Framework caching technologies

Technology	None	Survives process recycles	Survives reboots
SQL Server CE		✓	✓
Static variables	✓		
File system		✓	✓

Table 3.9 shows commonly used scenarios for each of these technologies.

Table 3.9: .NET Compact Framework caching scenarios

Technology	Scenario
SQL Server CE	Any scope (within the device) which requires high durability.
Static variables	In process, high performance caching.
File system	Any scope that requires high durability.

Use the information in the preceding tables to select the appropriate caching technology for your .NET Compact Framework clients.

Caching in ASP.NET Server Applications

This section focuses on ASP.NET server-side applications. Table 3.10 compares the scope of different technologies available for caching data in ASP.NET server applications.

Table 3.10: Scope of ASP.NET server-side caching technologies

Technology	User	Single AppDomain	Computer	Organization
ASP.NET Cache		✓		
ASP.NET Session object (InProc)	✓	Not applicable	Not applicable	Not applicable
ASP.NET Session object (StateServer)	✓	Not applicable	Not applicable	Not applicable
ASP.NET Session object (SQLServer)	✓	Not applicable	Not applicable	Not applicable
ViewState	✓	Not applicable	Not applicable	Not applicable
Static variables		✓		
Memory-mapped files			✓	
Remoting singleton				✓
SQL Server 2000 / MSDE				✓

Table 3.11 compares the durability of different technologies available for caching data in ASP.NET server applications.

Table 3.11: Durability of ASP.NET server-side caching technologies

Technology	None	Survives process recycles	Survives reboots
ASP.NET Cache	✓		
ASP.NET Session object (InProc)	✓		
ASP.NET Session object (StateServer)		✓ (survives aspnet_wp process recycles)	
ASP.NET Session object (SQLServer)		✓ (survives aspnet_wp process recycles)	Possible (for more information, see article Q311209, "HOW TO: Configure ASP.NET for Persistent SQL Server Session State Management," in the Microsoft Knowledge Base).
ViewState	Not Applicable	Not Applicable	Not Applicable
Static variables	✓		
Memory-mapped files	✓		
Remoting singleton		Good for surviving the applications process recycle (not the remoting process recycle).	
SQL Server 2000 / MSDE			✓

Table 3.12 shows commonly used scenarios for each of these technologies.

Table 3.12: ASP.NET server-side caching scenarios

Technology	Scenario
ASP.NET Cache	In process, high performance caching. Good for scenarios that require specific caching features.
ASP.NET Session object (InProc)	User session scope cache. Good for small amounts of session data that require high performance.
ASP.NET Session object (StateServer)	User session scope cache. Good for sharing session data in a Web farm scenario where SQL Server is not available.

Technology	Scenario
ASP.NET Session object (SQLServer)	User session scope cache. Good for sharing session data in a Web farm scenario where SQL Server is available.
ViewState	Good for request scope scenarios.
Static variables	In process, high performance cache.
Memory-mapped files	Computer-wide, high performance cache.
Remoting singleton	Organization-wide cache (not high performance).
SQL Server 2000 / MSDE	Can be used for any scope requirement which demands high durability.

Use the information in the preceding tables to select the appropriate caching technology for your ASP.NET server-side applications.

Caching in Server Applications

This section focuses on server-side applications, including those that use .NET Enterprise Services and those that do not. Table 3.13 compares the scope of different technologies available for caching data in server applications.

Table 3.13: Scope of server-side caching technologies

Technology	Single AppDomain	Computer	Organization
ASPNET Cache	✓		
Static variables	✓		
Memory-mapped files		✓	
Remoting singleton			✓
SQL Server 2000 / MSDE			✓

Table 3.14 compares the durability of different technologies available for caching data in server applications.

Table 3.14: Durability of server-side caching technologies

Technology	None	Survives process recycles	Survives reboots
ASPNET Cache	✓		
Static variables	✓		
Memory-mapped files	✓		
Remoting singleton	✓		
SQL Server 2000 / MSDE			✓

Table 3.15 shows commonly used scenarios for each of these technologies.

Table 3.15: Server-side caching scenarios

Technology	Scenario
ASP.NET Cache	In process, high performance caching. Good for scenarios which require specific caching features.
Static variables	In process, high performance caching.
Memory-mapped files	Computer-wide high performance caching.
Remoting singleton	Organization-wide cache (not high performance).
SQL Server 2000 / MSDE	Can be used for any scope requirement which requires high durability.

Use the information in the preceding tables to select the appropriate caching technology for your server-side applications.

Considering Physical Deployment Recommendations

You should consider the following guidelines when designing your cache:

- Select a technology without direct association to the current physical deployment environment because this may change during the lifetime of the application.
- If possible, use only one type of cache for the different layers. This way only one caching mechanism requires maintenance.
- Because most of the mechanisms support a key-value storage type, it is recommended that a naming convention is used. For example, cached data in the presentation layer could have the prefix of “UI_”, and so on. Using a naming convention makes changing physical deployment easier.

Again, considering the caching policy in the design phase of an application can greatly simplify the physical deployment phase.

Summary

In this chapter, you have learned about the application layers in a .NET-based distributed application, and you have seen how the different caching technologies available best suit different layers.

You are now ready to consider how to cache the many elements of a .NET-based application.

4

Caching .NET Framework Elements

Preceding chapters explained caching, including available caching technologies and the appropriate locations for the different types of cached data. In this chapter, you learn about how to make your own classes cacheable, what issues to consider when you implement caching, and what types of Microsoft .NET Framework elements can be cached in a distributed application.

This chapter contains the following sections:

- “Planning .NET Framework Element Caching”
- “Implementing .NET Framework Element Caching”

Planning .NET Framework Element Caching

You must consider various coding implementation issues when planning to cache .NET Framework elements in your applications. Such issues include:

- Ensuring thread safety
- Cloning
- Serializing
- Normalizing cached data
- Choosing a caching technology

Some of these issues can result in inconsistent data, whereas some simply influence the application’s efficiency.

Ensuring Thread Safety

When you write multithreaded applications, you must ensure that threads interact properly. For example, when a program or routine can be called from multiple threads, you must ensure that the activities of one thread do not corrupt information required by another. A routine is considered thread safe when it can be called from multiple programming threads without unwanted interaction among them.

When you are caching data types that are type safe, you can eliminate the risk that one thread will interfere and modify data elements of another thread through coordinated access to shared data, thus circumventing potential data race situations.

If your code is not thread safe, the following can occur:

- A cached item is read from the cache on one thread and then updated on another, resulting in the use of inconsistent data.
- A cached item is flushed from the cache by one thread while a client using another thread tries to read the item from cache.
- Two or more threads concurrently update a cached item with different values.
- Remove requests result in an item being removed from the cache while it is being read by another thread.

You can avoid these problems by ensuring that your code is thread safe, allowing only synchronized access to the cached data. You can do so either by implementing locking in the caching mechanism or by using the **ReaderWriterLock** class in the cache items.

Implementing Thread Safety Using the Cache

You can implement thread safety by using the following methods when using a cache:

- **Action-level locking**—You can implement thread safety in the Insert, Update, Get, and Delete cache interfaces. In this scenario, the cache automatically locks an item when it is added to the cache or updated.
- **Cache-level locking**—You can provide **Lock** and **Unlock** methods that the client code uses when thread safety is needed. The client needs to call the **Lock** method before updating the cached item and call **Unlock** afterward.

To find out whether a specific cache technology supports synchronization and thread safety, see “Choosing a Caching Technology,” later in this chapter.

Implementing Thread Safety in the Cached Items

If your chosen caching mechanism does not provide thread safety, you can implement thread safety directly in your cached objects by using the **ReaderWriterLock** class from the **System.Threading** namespace in the .NET Framework. For more information about this class, see “ReaderWriterLock Class,” in the MSDN Library.

The following example shows how you can use the **ReaderWriterLock** class to implement thread safety in cached items.

```
class Resource {  
    ReaderWriterLock rwl = new ReaderWriterLock();  
    Object m_data;      // Your class data
```

```

public void SetData(Object data) {
    rwl.AcquireWriterLock(Timeout.Infinite);
    // Update your object data.
    rwl.ReleaseWriterLock();
}

public Object GetData() {
    // Thread locks if other thread has writer lock
    rwl.AcquireReaderLock(Timeout.Infinite);
    // Get your object data...
    rwl.ReleaseReaderLock();
    return m_data;
}
}

```

Ensuring thread safety of cached items guarantees synchronized access of the items and avoids such issues as dirty reads and multiple writes.

For more information about thread synchronization, see “Safe Thread Synchronization” in the MSDN Magazine.

Cloning

Another solution to thread safety and synchronization issues is to use cloning. Instead of obtaining a reference to the original copy of your object when retrieving cached items, your cache can clone the object and return a reference to the copy.

To do so, your cached objects must implement the **ICloneable** interface. This interface contains one member, **Clone()**, which returns a copy of the current object. Take care to implement a deep copy of your object, copying all class members and all objects recursively referenced by your object.

If you design your cache to include interfaces for receiving both a cloned object and a reference to the original cached object, you will find new possibilities for resolving synchronization issues in your application:

- If your object is not thread safe, you can use cloning to create a copy of your cached data, update it, and then update the cache with the new object. This technique solves synchronization issues when data is updated. In this scenario, updating the cached item is performed as an atomic action, locking the access to the object during the update.
- One process can be responsible for updating an object while another process uses a cloned copy of the object for read-only purposes. This solves the problem of synchronizing read-after-write operations on your cache because the read object is merely a copy of the cached object, and the original object can be updated at the same time.

- Write-after-write operations on the cached object should still be synchronized, either by the cache or by the object, as described in the “Ensuring Thread Safety” section earlier in this chapter.

Figure 4.1 shows how cloning can solve synchronization problems when there is one writer but many users.

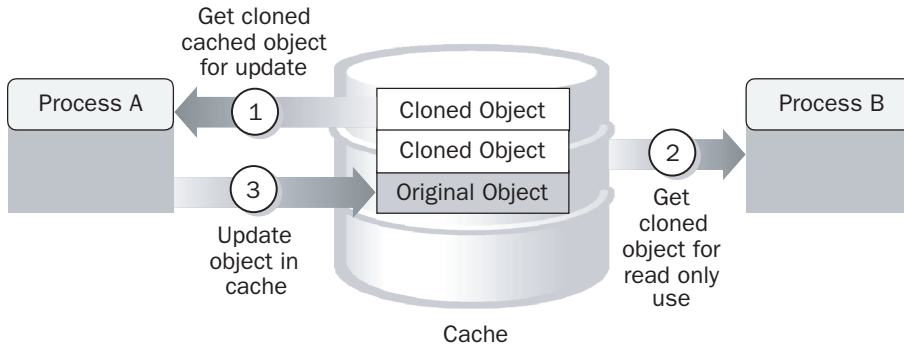


Figure 4.1
Cloning cached objects

Figure 4.1 shows the following steps in a sample cloning process:

1. Process A gets a clone of a cached object and updates the cloned object.
2. Process B gets a read-only clone of the cached object.
3. Process A updates the object in cache from its cloned copy without affecting the copy process that B is using.

The next time process B requests the object, it receives the new version of the object that process A updated.

Note: An alternative solution to synchronization issues is for process A to create a new object instead of cloning an existing cached object and then to update the cache with the new object.

Serializing a .NET Class

When sharing a cache between several processes or across an application farm, the cached items must be serialized when being inserted into the cache and deserialized when retrieved. An example of this is when you are storing ASP.NET session state in a SQL Server database. For more details, see the “Using ASP.NET Session State,” section in Chapter 2, “Understanding Caching Technologies.”

Because memory cannot be shared between two processes or between two computers, your cached object must be converted into a format such as XML or a binary representation to transport it between the memory locations. Consider this fact

when designing your .NET classes, and if you plan to use a caching technology that requires objects to be serialized, you must make your .NET class serializable. You can do so by using the **Serializable** attribute or by custom serializing your classes by implementing the **ISerializable** interface.

Note: To see whether a specific caching technology requires serialization support, see “Choosing a Caching Technology,” later in this chapter.

The following example shows how to use the **Serializable** attribute to make a class serializable.

```
[Serializable]
public class MyClass {
    ...
}
```

For more information about serializing a .NET class by using the **Serializable** attribute, see “Basic Serialization” in the MSDN Library.

The following example shows how to custom serialize a .NET class by implementing the **ISerializable** interface.

```
public class MyClass1: ISerializable
{
    public int size;
    public String shape;

    //Deserialization constructor
    public MyClass1 (SerializationInfo info, StreamingContext context) {
        size = (int)info.GetValue("size", typeof(int));
        shape = (String)info.GetValue("shape", typeof(string));
    }

    //Serialization function
    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("size", size);
        info.AddValue("shape", shape);
    }
}
```

For more information about custom serialization, see “Custom Serialization” in the MSDN Library.

It is recommended that you implement **ISerializable** instead of using the **Serializable** attribute because **ISerializable** does not use reflection and results in better performance.

For more information about serializing a .NET class, see “Run-time Serialization, Part 2” and “Run-time Serialization, Part 3” in the MSDN Magazine.

Normalizing Cached Data

When you are caching data, store it in the format in which clients use it; this is known as *normalizing* cached data. Normalizing cached data is a process in which an object is constructed once and then accessed as needed rather than being constructed anew each time it is accessed. For example, if you are caching a polygon for display in your application’s user interface, and the polygon details are saved in a database as a set of points, do not cache these points in the user interface. Instead, construct the polygon class, and cache that object.

By definition, normalizing cached data involves transforming your cached data to the format that is needed for later use. When you normalize your cached data, you ensure that:

- You can easily retrieve and use data from the cache; no formatting or deserialization is required.
- Repetitive transformation of the data each time it is retrieved from the cache is avoided.

Note: This definition of normalization differs from the classic data processing definition of the term, which is concerned with saving the data in the most efficient and memory-saving form. When you consider normalizing cached data, your first priority is usually performance rather than memory.

Choosing a Caching Technology

To choose an appropriate caching technology for your application, consider which caching technologies require serialization support in cache items and which provide synchronization mechanisms. Table 4.1 compares the available caching technologies and their support for serialization and synchronization.

After you consider the issues involved when caching .NET Framework elements in your applications, you can begin to think about what elements to cache and where to cache them.

Table 4.1: Caching technologies

Technology	Requires serialization support in items	Provides synchronization mechanism
ASP.NET		
Programmatic data cache	No	Yes
Page output cache	Not applicable	Not applicable
Page fragment cache	Not applicable	Not applicable
Session state — InProc	No	No
Session state — StateServer	Yes	No
Session state — SQLServer	Yes	No
View state	Yes	Not applicable
Hidden fields	Yes	Not applicable
Cookies	Yes	Not applicable
Query strings	Not applicable	Not applicable
Internet Explorer cache	Not applicable	Not applicable
Remoting singleton	Yes	Custom implementation required
Memory mapped files	No	No
SQL Server	Yes	Yes
Static variable	No	Custom implementation required

Implementing .NET Framework Element Caching

Some of the .NET Framework elements are excellent candidates for caching. In this section, you will learn about some of these elements and where they should be cached.

Caching Connection Strings

Database connection strings are difficult to cache because of the security issues involved in storing credentials that are used to access your database—for example, in *Machine.config* or *Web.config*—because the .NET Framework automatically caches the configuration files. However, because the configuration files are stored as clear ASCII text files, encrypt your connection strings before entering them in the configuration files and then decrypt them in your application before using them.

For more information about securely caching connection strings in your applications, see the guidelines for using Data Protection API (DPAPI) in the “Storing Database Connection Strings Securely” section of “Building Secure ASP.NET Applications” in the MSDN Library.

For more information about using DPAPI encryptions, see the following articles in “Building Secure ASP.NET Applications,” in the MSDN Library:

- “How To: Create a DPAPI Library”
- “How To: Use DPAPI (Machine Store) from ASP.NET”
- “How To: Use DPAPI (User Store) from ASP.NET with Enterprise Services”

These How To articles contain step-by-step instructions to help you learn how to use DPAPI encryption in your ASP.NET applications.

Caching Data Elements

Because of the relatively high performance costs of opening a connection to a database and querying the data stored in it, data elements are excellent caching candidates and are commonly cached. In this section, you learn whether or not to cache **DataSet** and **DataReader** objects.

Caching DataSet Objects

DataSet objects are excellent caching candidates because:

- They are serializable and as such can be stored in caches either in the same process or in another process.
- They can be updated without needing to reinsert them into the cache. Because you cache a reference to your **DataSet**, your application can update the data without the reference in the cache needing to change.
- They store formatted data that is easily read and parsed by the client.

Caching frequently used **DataSet** objects in your applications often results in improved performance.

Caching DataReader Objects

Never cache **DataReader** objects. Because a **DataReader** object holds an open connection to the database, caching the object extends the lifetime of the connection, affecting other users of the database. Also, because the **DataReader** is a forward-only stream of data, after a client has read the information, the information cannot be accessed again. Caching it would be futile.

Caching **DataReader** objects disastrously affects the scalability of your applications. You may hold connections open and eventually cache all available connections, making the database unusable until the connections are closed. Never cache **DataReader** objects no matter what caching technology you are using.

Also do not cache **SqlDataAdapter** objects because they use **DataReader** objects in their underlying data access method.

Caching XML Schemas

The .NET Framework includes the **XmlSchemaCollection** class that can help cache XML-Data Reduced (XDR) and XML Schema Definition language (XSD) schemas in memory instead of accessing them from a file or a URL. This makes the class a good caching candidate.

The **XmlSchemaCollection** class has methods for adding schemas to the collection, checking for the existence of a schema in the collection, and retrieving a schema from the collection. You can also use the **XmlSchemaCollection** to validate an XML document against XSD schema.

The **XmlSchemaCollection** improves performance by storing schemas in the cached collection so that they do not need to be loaded into memory each time validation occurs. For more information about the **XmlSchemaCollection**, see “**XmlSchemaCollection as a Schema Cache**,” in the MSDN Library.

Caching Windows Forms Controls

Caching Windows Forms controls in your user interface can improve responsiveness of the user interface. Because Windows Forms controls that include structured data require you to populate them with the necessary data before displaying them to the user, applications may appear sluggish. However, because it is the data loading that takes a long time, caching the populated version of these controls improves performance.

A static variable cache is the natural choice for caching Windows Forms controls because it has the following attributes:

- **Application scope**—Your Windows Forms controls are used within the scope of your Windows-based application, which matches the scope of your cache.
- **Simple implementation**—A static variable cache is an ideal choice for a simple caching implementation if you do not need advanced features such as dependencies, expirations, or scavenging.
- **Good performance**—Because the static variable cache is held in the application’s memory space, all caching calls are within the same process space, resulting in good performance.

You can use Windows Forms controls in only one page at a time. Because the cache holds a reference to the control, you have only one copy of the control to use at any given time. It does not make sense to clone Windows Forms controls because most of the work is in populating your control, and cloning effectively creates a new control. An exception to this is if you need to concurrently use the same class in

several forms in your application. In this situation, you can clone several instances of the control to be used concurrently by all forms requiring the control.

The following Windows Forms controls are classic examples of good caching candidates:

- **DataGrid**—Displays ADO.NET data in a scrollable grid
- **TreeView**—Contains a Nodes collection of **TreeNode** objects that implement **ICloneable**
- **ListView**—Contains an Items collection of **ListViewItem** objects that implement **ICloneable**

Because these controls often require complex processing before being rendered, caching them can improve application performance. Cache controls that take a long time to populate with data or that have complex initialization (that is, constructor) code. You may also decide to cache controls that are used in multiple forms in your application user interface.

The following example shows how to create a cache based on a hash table.

```
static Hashtable m_CachedControls = new Hashtable();
```

The following example shows how to create a **TreeView** control and insert it into the cache that has been created.

```
private void btnCreate_Click(object sender, System.EventArgs e)
{
    // Create the TreeView control and cache it.
    TreeView m_treeView = new TreeView();
    m_treeView.Name = "tvCached";
    m_treeView.Dock = System.Windows.Forms.DockStyle.Left;
    m_CachedControls.Add(m_treeView.Name, m_treeView);
}
```

The following example shows how to obtain the control from cache and bind it to a form

```
private void GetAndBindControl(Form frm)
{
    frm.Controls.Add((Control)m_CachedControls["tvCached"]);
    frm.Show();
}
```

These code samples are all you need to implement Windows Forms control caching in your application.

Caching Images

Because images are relatively large data items, they are good caching candidates. An image can be loaded from various locations, including:

- A local disk drive
- A network drive on a local area network (LAN)
- A remote network location or URL

Loading from local drives is faster than the other options; however, images cached in memory load extremely quickly and improve your application's performance and the user's experience.

Before you decide where to cache an image, consider how the image is used in the application. There are two common scenarios for image usage:

- Images displayed to the user in the presentation tier of an application
- Images used in the business tier, such as an image processing application in which images are used by different image processing algorithms

Following the rule that data should be stored as closely as possible to where it is used, the caching solution for each of these scenarios is unique.

Caching Presentation Tier Images

Two types of presentation tier caching exist: Web client caching and rich client caching. They use similar caching principles are similar, but you should choose different caching technologies.

Caching Images in a Web Client

When caching images in a Web client, do the following:

- Use the ASP.NET cache as your caching technology.
- Explicitly manage the caching of images on the page. Images contained on a page are loaded using separate HTTP requests, and they are not automatically cached when using ASP.NET page caching.
- Use file dependencies on the cached image file to ensure that the image is invalidated when the data changes.
- Consider your cache expiration time, taking into account:
 - Whether your images are static (never change) or dynamic
 - Whether your images change at known intervals
 - Whether your image refresh time is dependent on other cache keys or on disk files

For more information about specifying cache expiration times for images in Web applications, see "Using Dependencies and Expirations" in Chapter 2, "Understanding Caching Technologies."

Caching Images in a Rich Client

When caching images in a rich client, do the following:

- Use the static variable cache as your caching technology. Static variables give you both good performance and simple development.
- Cache only if the source of the images is outside your application assembly. It is common practice to store images as statically linked resources in the application assembly, in which case the images are loaded into memory with the assembly, and caching is not necessary.
- Cache images that are being retrieved from a database store or from a remote computer on the local disk when the images are first read, and then read them from the disk to service future requests.

Figure 4.2 shows how you can cache remotely stored images on a local disk for future use.

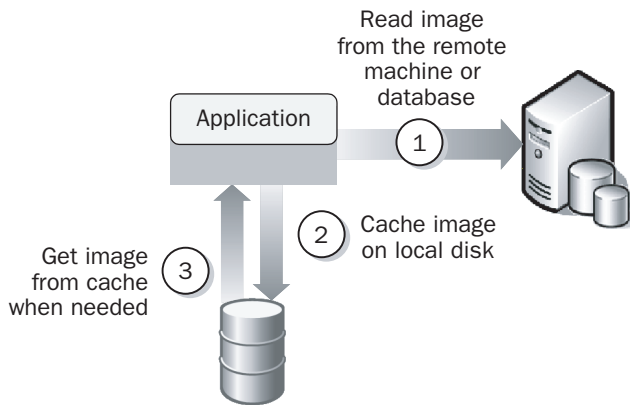


Figure 4.2

Caching image files from a database

Caching images in the presentation tier helps increase the user-perceived performance of your application.

Caching Business Tier Images

When caching images in the business tier, do the following:

- If all clients that are using the cached image are in the same process or AppDomain, use either the ASP.NET cache object or a static variable cache as your caching mechanism. Both of these methods facilitate simple programming models, but both have limitations:
 - Static variables do not offer any cache-specific features such as dependencies, expiration, and scavenging.

- ASP.NET caching requires you to install ASP.NET on your servers in the business tier.
- If your cached image needs to be accessed from different processes, implementing a memory-mapped file cache. Memory-mapped file caching offers very good performance at the cost of complicated implementation.

Caching images in either the presentation tier or the business tier of your applications can enhance the performance because it reduces the time needed to acquire and render image.

Caching Configuration Files

The .NET Framework automatically caches .NET configuration files (for example, `Machine.config` and `Web.config`) by using the internal .NET caching mechanisms. Because of this, you can use the .NET configuration files without implementing any caching mechanism yourself. Although the configuration files are cached by the .NET Framework, you should remember the following facts when working with them in an application that caches data:

- Changes to the `Global.asax` and `Web.config` files restart the `AppDomain`, which resets your ASP.NET cache, application state, and session state (if using **InProc** mode).
- Changes to the `Machine.config` file restarts the ASP.NET worker process (`Aspnet_wp.exe`).

Therefore, you should not scan configuration files for viruses. Doing so restarts the application each time.

Caching Security Credentials

The .NET Framework includes a **CredentialCache** class that you can use to cache multiple security credentials. This is useful when you are accessing multiple network resources that require different credentials, such as Web sites or Web services from different vendors. For example, in the .NET domain, your application may be required to authenticate with different service providers, whether they publish Web services that your application uses or have Web sites accessed by your application that require you to authenticate upon entering.

The **NetworkCredential** class represents credentials for password-based authentication schemes, such as basic, digest, NTLM, and Kerberos authentication. Use the **NetworkCredential** class to store your authentication data when caching security credentials because the authentication data is stored formatted, ready to use in your application.

Summary

This chapter helps you plan the caching of .NET Framework elements and covers the options available for doing so.

After you decide what to cache, where to cache it, and how to cache it, you must consider how to get the data into the cache and how to manage it once it's there.

5

Managing the Contents of a Cache

So far in this guide, you have seen what data to cache and where to cache it. This chapter describes other issues concerning caching management, including how to load data into a cache and how to manage the data inside the cache.

This chapter contains the following sections:

- “Loading a Cache”
- “Determining a Cache Expiration Policy”
- “Flushing a Cache”
- “Locating Cached Data”

Loading a Cache

Data can be loaded into a cache using various methods. This section covers the different options for populating the cache and how to select a method for loading the data.

When determining the data acquisition method for your cache, consider how much of the data you want to acquire and when you want to load it. For example, you may decide to load data into the cache when the application initializes or to acquire the data only when it is requested. Table 5.1 shows the options available for loading a cache.

Table 5.1: Cache loading options

Loading type	Synchronous methods	Asynchronous methods
Proactive loading	Not recommended	Asynchronous pull loading Notification-based loading
Reactive loading	Synchronous pull loading	Not applicable

Each of these methods has its uses in different application scenarios.

Caching Data Proactively

You can cache data proactively to retrieve all of the required state for an application or a process, usually when the application or process starts, and cache it for the lifetime of the application or process.

Advantages of Proactive Loading

Because you can guarantee that the data has been loaded into the cache, in theory there is no need to check whether the state exists in the cache; however, check whether an item exists in the cache before retrieving it, anyway, because the cache may have been flushed.

Your application performance improves because cache operations are optimized when loading state into the cache proactively, and application response times improve because all of the data is cached.

Disadvantages of Proactive Loading

Proactive loading does not result in the most optimized system because a large amount of the state is cached, even though you may not need it all. For example, an application may contain 100 processes, each of which may require a few items in the cache. If a user launches this application but activates only one process, hundreds of items are needlessly cached.

Proactive caching may result in an implementation more complex than traditional techniques, in which each item is retrieved synchronously in a well-known program flow. Using proactive caching requires working with several threads, so synchronizing them with the application main thread, keeping track of their status, and handling exceptions in an asynchronous programming model can be difficult.

Recommendations for Proactive Loading

If you do not use proactive loading correctly, applications may initialize slowly. When you implement proactive caching, load as much state as possible when the application initializes or when each process initializes. You should use an asynchronous programming model to load the state on a background thread.

Proactive caching is recommended in situations in which:

- You are using static or semistatic state that has known update periods. If you use it in other scenarios, the state might expire before it is used.
- You are using state with a known lifetime.
- You are using state of a known size. If you use proactive cache data loading when you don't know the size of the data, you might exhaust system resources. You must not try to use resources that you do not have.

- You have problematic resources, such as a slow database, a slow network, or unreliable Web services. You can use this technique to retrieve all of the state proactively, cache it, and work against the cache as much as possible.

In these situations, proactive loading can improve application performance.

Using Asynchronous Pull Loading

In *asynchronous pull loading*, data is loaded into the cache for later use. It is a proactive process, based on expected not actual usage.

When using asynchronous pull loading, none of the requests for data perform worse than any other because the state is retrieved into the cache proactively, not as a response to a specific request. Consider using this technique at startup for applications with problems such as network latencies or service access difficulties.

Service agent caches are usually good candidates for asynchronous pull loading, especially if the data comes from many services and needs to be consolidated for consumption. For example, in a news feed application the data may be consolidated from many news agencies.

Figure 5.1 shows how to implement asynchronous pull loading in service agent elements.

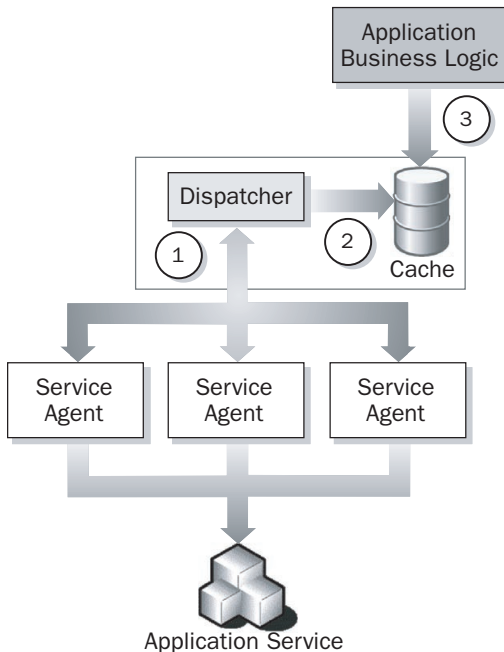


Figure 5.1

Asynchronous pull loading

As Figure 5.1 shows, the steps for asynchronous pull loading are:

1. Based on a schedule or other type of notification, a dispatcher queries application services for their data.
2. Return values from the service agents are placed in the cache, and possibly aggregated with other data.
3. The application queries the cache for subsets of data and uses them.

Asynchronous pull loading ensures that the data is loaded into the cache independently of the application flow and is available for use whenever the application requests it.

Using Notification-Based Loading

In *notification-based loading*, the application's services notify the application or the cache of state changes. This technique is tightly coupled with the underlying application services.

When using this technique, you have two options for retrieving the state from the application's services into the cache:

- Receiving a notification from the application's services and then pull loading the data to populate the cache
- Receiving the state as part of a notification and populating the cache

Notification-based loading is generally more complex to implement than pull loading; however, it guarantees that new data is loaded only when necessary.

Advantages of Notification-Based Loading

A primary reason to use notification-based loading is that it results in minimal state staleness, so it makes it much easier for you to cache state and items that do not tolerate state becoming stale.

Because the application services handle the notifications, you do not need to implement a refresh mechanism in the application. This helps reduce the management overhead of the application.

You can use notification-based loading in Web farm scenarios, in which more than one cache needs to be notified of changes. Notification-based loading can ensure that multiple caches remain synchronized.

Disadvantages of Notification-Based Loading

The major disadvantage of this technique is that many services do not implement notification mechanisms. For example, Web services and legacy systems do not support notifications.

The application services layer may need more maintenance than when using other loading mechanisms. For example, triggers in SQL Server used for notifications need to be maintained.

Implementing notification-based loading can be more complex than using pull loading techniques.

Recommendations for Notification-Based Loading

Implementing notification-based loading usually involves the *publish-subscribe* (pub-sub) design pattern. In pub-sub applications, data published by a publisher entity is sent to all subscriber entities that subscribe to that data. Use the pub-sub pattern to receive notifications of state changes. For more information about using notifications, see “Using Expiration Policies” later in this chapter.

Using SQL Server 2000 Notification Services

Microsoft SQL Server 2000 Notification Services is a powerful notification application platform. It helps developers build centralized pub-sub applications and deploy them on a large scale. These applications are often used to send notification messages to end users and applications.

You can use this technology to monitor services, including Web services and legacy systems, for changes to data, and then to notify applications of the changes. The information updates can be delivered to various devices, including mobile devices such as cellular phones and Pocket PCs.

For more information about Notification Services, see the SQL Server 2000 Notification Services Web site.

Figure 5.2 shows one way to use Notification Services in your caching systems.

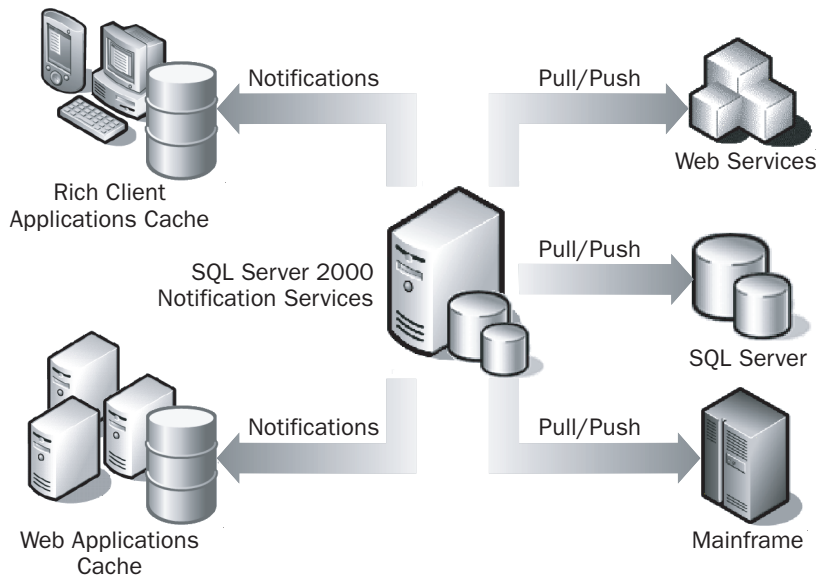


Figure 5.2

Implementing a pub-sub system

For a sample application design that demonstrates how to use Notification Services as a push mechanism for cached items, see Chapter 7, “Appendix.”

Caching Data Reactively

You can cache data reactively to retrieve data as it is requested by the application and cache it for future requests.

Advantages of Reactive Loading

Because you are not loading large amounts of data when the application initializes, your system resources are not misused.

This method results in an optimized caching system because you are storing only requested items.

Disadvantages of Reactive Loading

Performance might decrease when any piece of data is requested the first time because it must be loaded from the source not retrieved from the cache.

You need to check whether an item exists in the cache before you can use it. Implementing this checking in every service agent can result in excessive conditional logic in your code.

Recommendations for Reactive Loading

When you implement reactive caching, load the state only when it is requested using a synchronous programming model. For more information about using a synchronous programming model, see the next section, “Using Synchronous Pull Loading.”

Reactive caching is recommended in situations in which:

- You are using large amounts of state and do not have adequate resources to cache all of the state for the entire application.
- You are using reliable and responsive resources, such as a database, network, or Web service that will not impede application stability and performance.

In these situations, reactive loading can improve performance.

Using Synchronous Pull Loading

In *synchronous pull loading*, the data is loaded into the cache when the application requires the data. As such, it is a reactive loading method.

When using synchronous pull loading, the first request for the data decreases performance because the data has to be retrieved from the application service. Therefore, synchronous pull loading is best used when you need to get a specific piece of state rather than when you want to cache all of the state for the entire application or process.

You may want to use synchronous pull loading only when your system is running in a steady state, so that, for example, your application won't suffer from unexpected network latencies.

Advantages of Synchronous Pull Loading

It is relatively easy to implement synchronous pull loading because all of the loading code is written within the application and none is needed in the underlying application services. For example, you do not need to create or maintain SQL Server triggers or Windows Management Instrumentation (WMI) event listeners.

The synchronous pull loading implementation is independent of the underlying services. Synchronous pull loading does not rely on a specific application services technology, so it can be used to retrieve data from services such as Web services and legacy systems.

Disadvantages of Synchronous Pull Loading

The major problem that occurs when using pull loading is state staleness. Because no mechanism exists to notify the cache of data changes in the underlying application services, data changes in the application services might not be reflected in the cached data. For example, you may be caching flight departure information in your application. If a flight departure time changes and that change is not reflected in the cache, you may miss your flight.

Recommendations for Synchronous Pull Loading

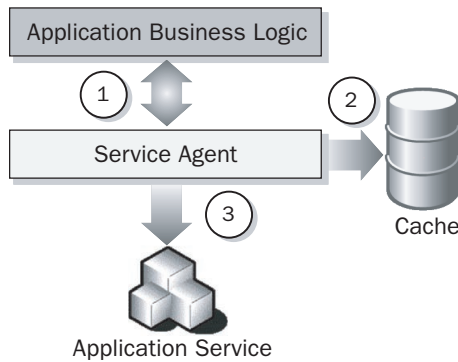
Use synchronous pull loading for populating the cache when:

- The state retrieved from the different application services can have known and acceptable degrees of staleness.
- The state can be refreshed at known intervals. For example, cached stock quotes in a delayed stock ticker application can be refreshed once every 20 minutes.
- Working with third-party services, such as Web services or legacy systems, or while using services that don't implement notification mechanisms.

Figure 5.3 on the next page shows how you can implement synchronous pull loading in a service agent element.

As Figure 5.3 shows, the steps for synchronous pull loading are:

1. The application business logic calls the service agent and requests the data.
2. The service agent checks whether a suitable response is available in the cache—that is, it looks for the item in the cache and the expiration policies of any relevant items in it.
3. If the data is available in the cache, it is used. If not, the application service is called to return the data, and the service agent places the returned data in the cache.

**Figure 5.3***Synchronous pull loading*

Note: It is recommended that you separate service agents that use caching from those that do not. Doing so avoids having excess conditional logic in the service agent code.

In addition to reloading cache items to keep them valid, you may also want to implement an expiration policy to invalidate cached items.

Determining a Cache Expiration Policy

Preceding sections describe how to keep cache items valid by reloading the cache items using pull or push loading techniques. In this section, you learn about the different ways to maintain the validity of the data and items in the cache by using time-based or notification-based expiration.

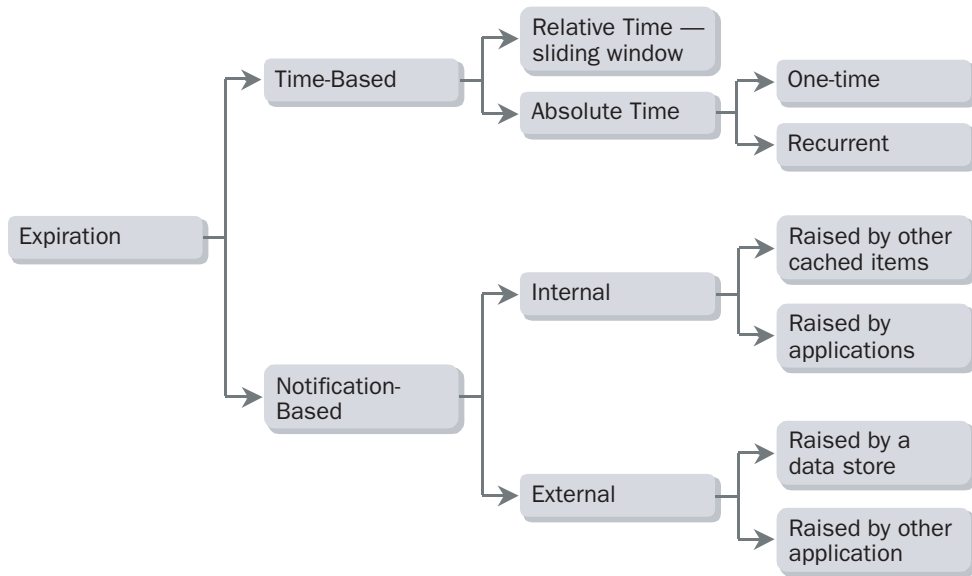
Using Expiration Policies

There are two categories of expiration policy:

- **Time-based expirations**—Invalidate data based on either relative or absolute time periods
- **Notification-based expirations**—Invalidate data based on instructions from an internal or external source

Figure 5.4 shows the different dependencies that can be used to define the validity of cached items by using expiration policies.

Each type of expiration policy has its advantages and disadvantages.

**Figure 5.4***Policy dependencies*

Using Time-Based Expirations

You should 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 you to set policies that keep items in the cache only as long as their data remains current. For example, if you are writing an application that tracks currency exchange rates by obtaining the data from a frequently updated Web site, you can cache the currency rates for the time that those rates remain constant on the source Web site. In this situation, you would set an expiration policy that is based on the frequency of the Web site updates—for example, once a day or every 20 minutes.

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

- **Absolute**—Absolute expiration policies allow you to define the lifetime of an item by specifying the absolute time for an item to expire. There are two types:
 - **Simple**—You define the lifetime of an item by setting a specific date and time for the item to expire.
 - **Extended**—You define the lifetime of an item by specifying expressions such as every minute, every Sunday, expire at 5:15 AM on the 15th of every month, and so on.

The following example shows how you can implement absolute time expiration.

```
internal bool CheckSimpleAbsoluteExpiration(DateTime nowDateTime,
                                           DateTime absoluteExpiration)
{
    bool hasExpired = false;

    //Check expiration.
    if(nowDateTime.Ticks > absoluteExpiration.Ticks)
    {
        hasExpired = true;
    }
    else
    {
        hasExpired = false;
    }
    return hasExpired;
}
```

- **Sliding**—Sliding expiration policies allow you to define the lifetime of an item by specifying the interval between the item being accessed and the policy defining it as expired.

The following example shows how you can implement sliding time expiration.

```
internal bool CheckSlidingExpiration(DateTime nowDateTime,
                                     DateTime lastUsed,
                                     TimeSpan slidingExpiration)
{
    bool hasExpired = false;

    //Check expiration.
    if(nowDateTime.Ticks > (lastUsed.Ticks + slidingExpiration.Ticks))
    {
        hasExpired = true;
    }
    else
    {
        hasExpired = false;
    }
    return hasExpired;
}
```

For more information about implementing extended-format time expirations, see Chapter 7, “Appendix.”

Using Notification-Based Expirations

You can use notification-based expiration to 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.

Common sources of dependencies include:

- File changes
- WMI events
- Business logic operations

You can implement file dependencies by using the **FileSystemWatcher** class, in the **System.IO** namespace. The following code demonstrates how to create a file dependency class for invalidating cached items.

```
namespace Microsoft.Samples.Caching.Cs
{
    public class FileDependency
    {
        private System.IO.FileSystemWatcher m_fileSystemWatcher;

        public delegate void FileChange(object sender,
                                         System.IO.FileSystemEventArgs e);
        //The OnFileChange event is fired when the file changes.
        public event FileChange OnFileChange;

        public FileDependency(string fullFileName)
        {
            //Validate file.
            System.IO.FileInfo fileInfo = new System.IO.FileInfo(fullFileName);
            if (!fileInfo.Exists)
                throw new System.IO.FileNotFoundException();

            //Get path from full file name.
            string path = System.IO.Path.GetDirectoryName(fullFileName);
            //Get file name from full file name.
            string fileName = System.IO.Path.GetFileName(fullFileName);
            //Initialize new FileSystemWatcher.
            m_fileSystemWatcher = new System.IO.FileSystemWatcher();

            m_fileSystemWatcher.Path = path;
            m_fileSystemWatcher.Filter = fileName;
            m_fileSystemWatcher.EnableRaisingEvents = true;
            this.m_fileSystemWatcher.Changed += new
                System.IO.FileSystemEventHandler(this.fileSystemWatcher_Changed);
        }

        private void fileSystemWatcher_Changed(object sender,
                                                System.IO.FileSystemEventArgs e)
        {
            OnFileChange(sender, e);
        }
    }
}
```

Whenever the specified file (or folder) changes, the **OnFileChange** event executes, and your custom code can be used to invalidate a cached item.

Using External Notifications

In some situations, external sources used in your application provide notifications of state changes. You can handle those notifications in two ways:

- Create a dependency directly from the cache to the service.
- Create an internal pub-sub to obtain external notification of changes and notify the internal subscribers for this event.

Figure 5.5 illustrates how to create a dependency directly from the cache to the service.

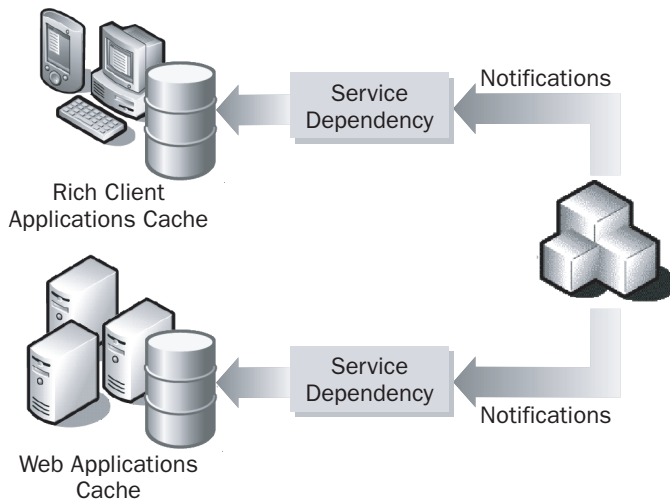


Figure 5.5

Creating a dependency for external service notification

Figure 5.6 shows how to create an internal pub-sub to obtain external notification of changes and notify the internal subscribers for this event.

When you build applications with a large number of event subscribers, you should handle external notifications in a pub-sub way to improve scalability and availability.

If your application's data is stored in a SQL Server database, you can use triggers to implement a simple notification mechanism for your application's cache. The main advantage of using triggers is that your application is immediately notified of database changes.

One disadvantage of using this technique is that it is not optimized for many subscribers or for many events, which can lead to a decrease in database performance and can create scalability issues. Another disadvantage is that triggers are not easy

to implement. In the database, you must write stored procedures containing the code to run when the trigger executes and write custom listeners in the application to receive the notifications.

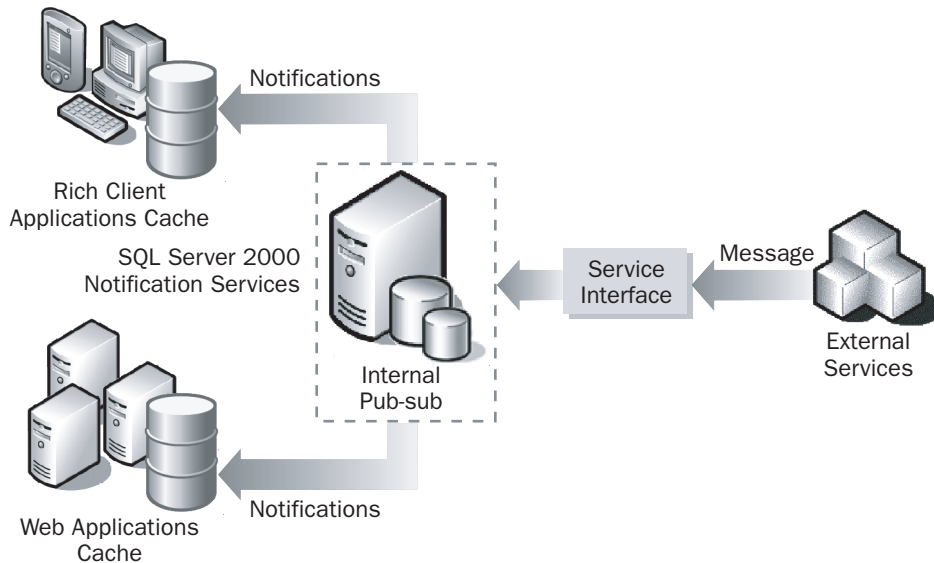


Figure 5.6
Using internal pub-sub for external notifications

For more information and sample code showing how to implement dependencies using SQL Server triggers, see Rob Howard's team page at <http://www.gotdotnet.com/team/rhoward>.

Expirations work well for removing invalid items from a cache; however, this may not always be enough to ensure an efficiently managed cache.

Flushing a Cache

Flushing allows you to manage cached items to ensure that storage, memory, and other resources are used efficiently. Flushing is different from expiration in that you may decide in some cases to flush valid cache items to make space for more frequently used items, whereas expiration policies are used to remove invalid items.

There are two categories of flushing techniques: explicit flushing and scavenging. Scavenging can be implemented to flush items based on when they were last used, how often they have been used, or using priorities that you assign in your application.

Figure 5.7 shows the two categories of flushing techniques.

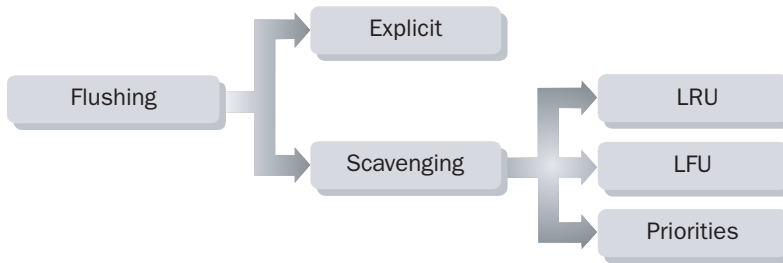


Figure 5.7

Flushing techniques

Explicit flushing requires that you write the code to determine when the item should be flushed and the code to flush it. With scavenging, you write an algorithm for the cache to use to determine what items can be flushed.

Using Explicit Flushing

Different scenarios require that cache stores be explicitly flushed by instructing the cache to clear its content. You can implement explicit flushing either manually or programmatically. For example, if cached data becomes obsolete or damaged, there may be an immediate need to clear the contents of the cache. In this situation, a system administrator must be able to explicitly flush the cache.

None of the custom stores—for example, SQL Server and static variables—support scavenging, so you must explicitly implement any flushing techniques that you require.

Implementing Scavenging

You can use a scavenging algorithm to automatically remove seldom used or unimportant items from the cache when system memory or some other resource becomes scarce.

Typically, scavenging is activated when storage resources become scarce, but it can also be activated to save computing resources—for example, to reduce the time and CPU cycles required to look up specific cached items.

The following code shows how to initiate a process based on the amount of memory available.

```

using System;
using System.Management;

namespace Microsoft.Samples.Caching.Cs.WMIDependency
{
    public delegate void WmiMonitoringEventHandler(object sender, object newValue);
  
```

```

public class WMIMemoryDependency : IDisposable
{
    string path = @"\\\" + \"localhost\" + @\"\\root\\cimv2\";
    private int m_MemoryLimit = 0;
    private ManagementEventWatcher memoryEventWatcher;
    public event WmiMonitoringEventHandler MemoryLimitReached;

    public WMIMemoryDependency(int memoryLimit)
    {
        m_MemoryLimit = memoryLimit;
        StartEvent();
    }
    private void StartEvent()
    {
        ManagementScope scope = new ManagementScope(path);
        scope.Options.EnablePrivileges = true;
        WqlEventQuery eventQuery = new WqlEventQuery();

        eventQuery.QueryString = \"SELECT * FROM __InstanceOperationEvent \" +
            \"WITHIN 30 WHERE TargetInstance ISA \" +
            \"'Win32_PerfRawData_PerfOS_Memory'\";
        memoryEventWatcher = new ManagementEventWatcher(scope, eventQuery);
        memoryEventWatcher.EventArrived += new
            EventArrivedEventHandler(MemoryEventArrived);
        memoryEventWatcher.Start();
    }

    private void MemoryEventArrived(object o, EventArrivedEventArgs e)
    {
        try
        {
            ManagementBaseObject eventArg = null;

            eventArg = (ManagementBaseObject)(e.NewEvent[\"TargetInstance\"]);
            string memory =
                eventArg.Properties[\"AvailableKBytes\"].Value.ToString();
            Int64 freeMemory = Convert.ToInt64(memory);
            if (freeMemory <= m_MemoryLimit)
            {
                if (MemoryLimitReached != null)
                    MemoryLimitReached(this, freeMemory);
            }
        }
        catch (Exception ex) { throw(ex); }
    }

    void IDisposable.Dispose()
    {
        if (memoryEventWatcher != null)
            memoryEventWatcher.Stop();
    }
}

```

Flushing items from memory can be initiated using one of three common algorithms:

- **Least Recently Used algorithm**—The Least Recently Used (LRU) algorithm flushes the items that have not been used for the longest period of time.

The following pseudocode shows how the LRU algorithm works.

```
Sort_cache_items_by_last_used_datetime;
While (%cache_size_from_available_memory >
max_cache_size%_from_available_memory)
{
    item = peek_first_item;
    Remove_item_from_cache( item );
}
```

- **Least Frequently Used algorithm**—The Least Frequently Used (LFU) algorithm flushes the items that have been used least frequently since they were loaded. The main disadvantage of this algorithm is that it may keep obsolete items infinitely in the cache.

The following pseudocode shows how the LFU algorithm works.

```
Sort_cache_items_by_usage_count;
While (%cache_size_from_available_memory >
max_cache_size%_from_available_memory)
{
    item = peek_first_item;
    Remove_item_from_cache( item );
}
```

- **Priority algorithm**—The priority algorithm instructs the cache to assign certain items priority over other items when it scavenges. Unlike in the other algorithms, the application code rather than the cache engine assigns priorities.

You have seen how to load a cache and how to remove items from a cache for different reasons. The last main planning task is to determine how to access the information stored in the cache.

Locating Cached Data

In many cases, a cache exposes an API to enable you to load and retrieve items based on some kind of identifier—for example, a key, a tag, or an index. If you want your cache to be generic, it must support storage of many types of items; therefore, it must expose an API that allows applications to access any of the items it stores.

When you are caching many different types of items, create multiple caches, one for each type of item. This increases the efficiency of data searching because it reduces the number of items in each cache. Furthermore, in rare cases, you may not know exactly what cached items your application will retrieve. For example, you may

have an asynchronously pull loaded cache that obtains news feeds from many news agencies and places them in a SQL Server database. When the application retrieves cached data, it may simply request news items related to Microsoft. In this scenario, you must implement some identification method based on the cached data attributes, such as keywords, which allows this type of search to be used.

Figure 5.8 shows how data is commonly requested, using either a known key or a set of attributes.

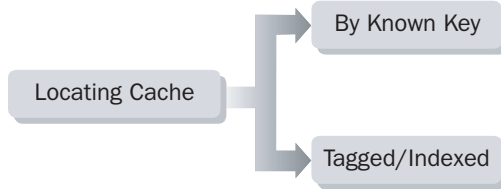


Figure 5.8

Locating cached data

Whether you are implementing a known key or a data attribute-based cache searching mechanism, use the following guidelines for locating an item:

- If there is only one primary key or identifier, use key-value pair collections such as hash table or hybrid dictionary.
- If there are multiple potential keys, create one hash table for each identifier, as Figure 5.9 shows. However, doing so can make it harder to remove items from the cache because you need information from all of the hash tables that refer to the cache item.

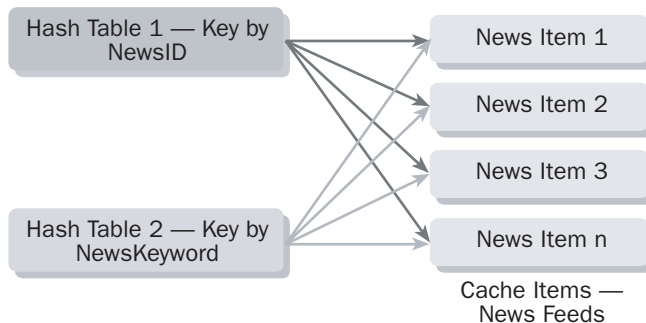


Figure 5.9

Using multiple hash tables

Be careful not to over-engineer your indexing strategy. If you require complex indexing, aggregation, or querying language capabilities, use a relational database management system as your caching mechanism.

Summary

This chapter explains the options for loading data into a cache and managing the data after it is in the cache. However, you must consider other issues when planning and designing a cache, including security, monitoring, and synchronizing caches in server farms.

6

Understanding Advanced Caching Issues

Now that you have seen what caching technologies are available and how to use them in Microsoft .NET-based applications, you have an overall picture of how to implement caching. In this chapter, you learn about some of the advanced issues related to the subject.

This chapter contains the following sections:

- “Designing a Custom Cache”
- “Securing a Custom Cache”
- “Monitoring a Cache”
- “Synchronizing Caches in a Server Farm”

Designing a Custom Cache

This section of the guide describes the design and implementation of a custom cache framework. The cache design provides a simple yet extensible framework for creating a custom cache. It includes the design goals and a solution blueprint for a generic cache solution. It can easily be used as a building block within your own .NET-based application.

Introducing the Design Goals

The cache is designed 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.

- Provide support for cache specific features, such as dependencies and expirations, and enables you to write your own expiration and dependency implementations.
- Provide support for cache management features such as scavenging.
- Enable you to design your own cache storage solution by implementing the storage class interfaces provided.
- Enable you to design your own cache scavenging algorithms by implementing the classes and interfaces provided.

This design should enable you to create reusable cache mechanisms for your .NET distributed applications.

Introducing the Solution Blueprint

The solution blueprint details the design of a cache that meets the goals described in the preceding section. This section describes the main cache components, deployment scenarios, custom cache design, custom cache configuration, and custom cache use cases.

Introducing the Main Cache Components

The design of the custom cache includes the cache blocks shown in Figure 6.1.

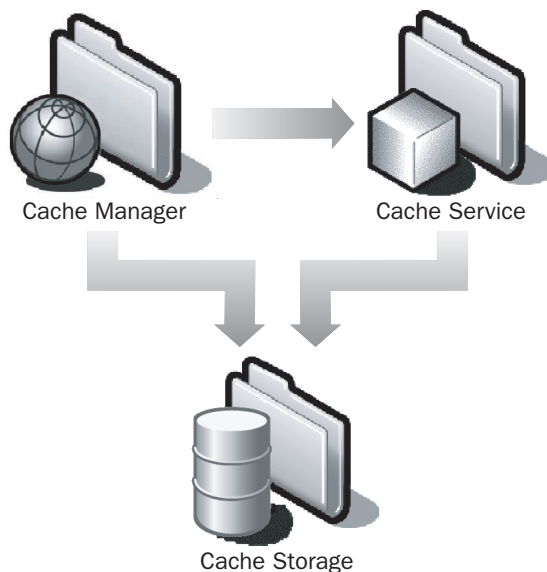


Figure 6.1

Custom cache block diagram

These components communicate together to provide the overall caching system. The following list describes each of these cache blocks:

- **Cache manager**—The cache manager provides the application interface to the cache itself. Classes and interfaces included in this package provide the interfaces required for adding, retrieving, and removing items and metadata to and from the cache.
- **Cache service**—The cache service is responsible for managing cache metadata. The cache service can be deployed either in the same AppDomain as the cache manager or in a different process, depending on the required storage and its scope.
- **Cache storage**—The cache storage separates the cache data store from the cache functional implementation. The cache storage handles insertion, retrieval, and deletion of cached items to and from the cache data store.

Understanding Deployment Scenarios

One of the design goals of the custom cache is to enable different types of cache storages, such as static variables, memory-mapped files, and Microsoft SQL Server. To enable optimal use of these types of storages, different deployment methods are required.

Three deployment scenarios that you can use for your custom cache are:

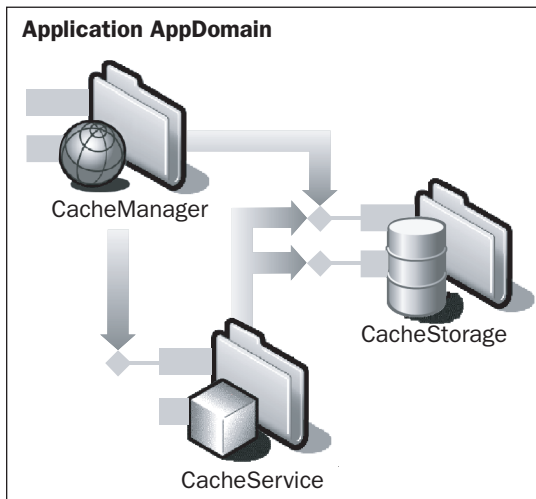
- AppDomain scope deployment
- Machine scope deployment
- Application farm scope deployment

The next sections describe each of these scenarios.

Using AppDomain Scope Deployment

When using a static variable cache, you should deploy the three blocks in the same AppDomain as the application as shown in Figure 6.2 on the next page.

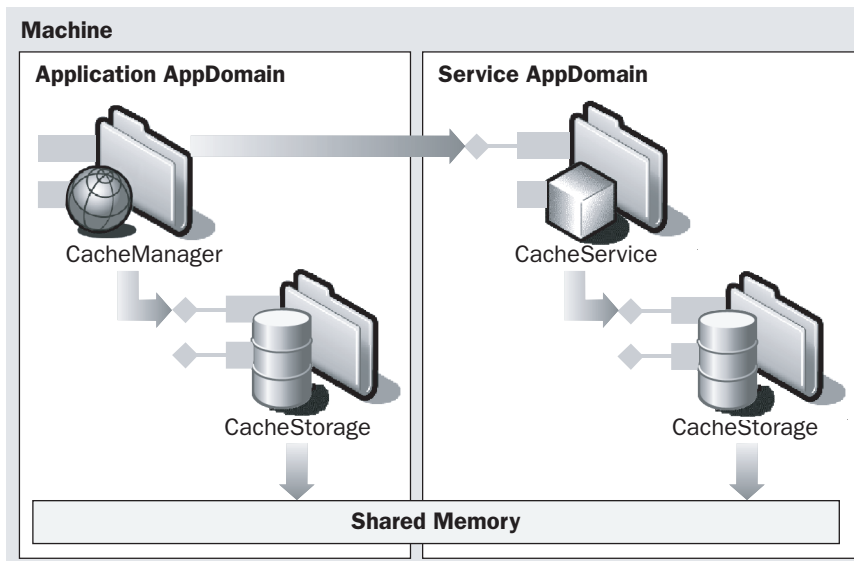
Because the storage is accessed only from threads executing in its AppDomain, this deployment option provides you with the optimal performance.

**Figure 6.2**

AppDomain scope deployment

Using Machine Scope Deployment

Storage with a machine scope requires a different deployment method. Because the storage may be shared across multiple applications that reside in different AppDomains on the same computer, the **CacheManager** is deployed in each application's AppDomain while the **CacheService** is deployed in a single, separate AppDomain. To enable access to the shared storage, a **CacheStorage** proxy is deployed in each application AppDomain. This is shown in Figure 6.3.

**Figure 6.3**

Machine scope deployment

In this configuration, every application can access the storage directly while the metadata is managed separately from a single location.

Using Application Farm Scope Deployment

In this deployment scenario the storage is implemented as a SQL Server durable storage system as shown in Figure 6.4.

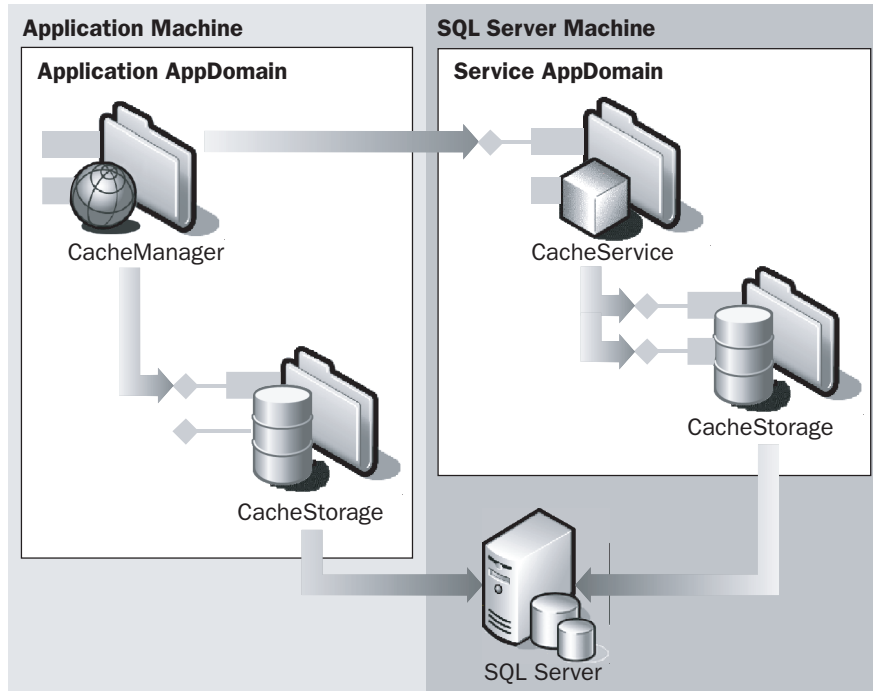


Figure 6.4
Application farm scope deployment

In application farm scope deployment, the **CacheManager** is deployed in each application's AppDomain, and the **CacheService** is deployed on the same computer as the SQL Server. To enable access to the shared storage, a **CacheStorage** proxy is deployed in each application's AppDomain.

Understanding the Custom Cache Detailed Design

This section describes the detailed design of the custom cache starting with the class diagram and then looking at the classes and interfaces that constitute the custom cache. Finally, the cache use cases are overviewed with sequence diagrams.

Figure 6.5 on the next page shows the three main classes used in a custom cache and the functionality that each one provides.

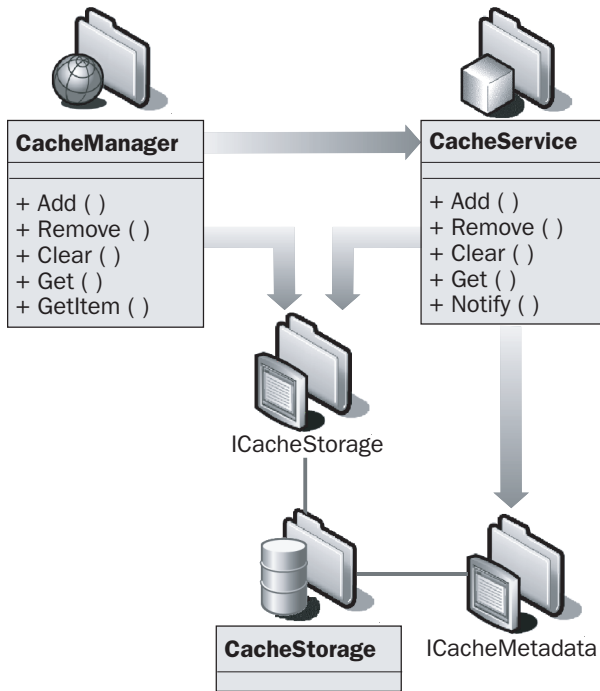


Figure 6.5
Class diagram

The next sections describe each of these classes in detail.

Using the CacheManager Class

The **CacheManager** class has references to each of the other classes. It references the:

- **CacheStorage** for inserting, getting, and removing items from the cache storage.
- **CacheService** for registering cache item metadata.

The definition of this class is shown in Figure 6.6.

For each cache operation, the **CacheManager** first synchronously calls the **CacheStorage** to update the cache data store. After that, it asynchronously calls the **CacheService** to update the cached items' expirations, priority, and callback metadata in the cache metadata store. This design provides 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.

The **CacheManager** is a static variable within the scope of the application, and as such, the 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.



CacheManager
<pre> + Add (in key : string, in value : object, in expirations : ICACHEItemExpiration[], in priority : CACHEItemPriority, in onRemoveCallback : CACHEItemRemoveCallback) : void + Remove (in key : string) : void + Clear () : void + Get (in key : string) : object + GetItem (in key : string) : CACHEItem </pre>

Figure 6.6

The CacheManager class

The **CacheManager** class implements the following default interface:

- **Add**—The **Add** method inserts an item into the cache. The **Add** method enables you to add items to the cache with or without metadata. In the simplest case, you can use the **Add** method with a key/value pair.
- **Remove**—The **Remove** method removes a specific item and its metadata from the cache identified by the provided key.
- **Clear**—The **Clear** method clears all items and their metadata from the cache.
- **Get**—The **Get** method retrieves a specific item from the cache identified by the provided key. This method returns an object reference to that item in the cache.
- **GetItem**—The **GetItem** method returns a specific **CACHEItem** object identified by the provided key. This object includes the metadata associated with the cached item alongside the cached item value.

The **CacheManager** class provided in the custom cache does not need to be changed regardless of the storage technology you choose and your deployment model.

Using the CacheService Class

The **CacheService** class is responsible for managing the metadata associated with cached items. This metadata may include expiration policies on cached items, cached item priorities, and callbacks.

The definition of this class is shown in Figure 6.7 on the next page.

The **CacheService** class is implemented as a singleton, meaning that only one instance of this class can exist for the cache. The instantiation of the **CacheService** may change, depending on the scope of your cache data store and the deployment scenario. If the scope of your cache data store is an AppDomain (for example, when using static variable based cache storage systems) the **CacheService** class can exist

within the AppDomain of your application and cache. On the other hand, if your cache store is shared across multiple AppDomains or processes (for example, when using memory-mapped files or SQL Server based cache storage systems), the **CacheService** class needs to be instantiated by a separate process and it is accessible from all processes using the cache. Note that each AppDomain or process using the cache has its own instance of the **CacheManager** class regardless of the deployment scenario.



CacheService
<pre>+ Add (in key : string, in expirations : ICACHEItemExpiration[], in priority : CacheItemPriority, in onRemoveCallback : CacheItemRemoveCallback) : void + Remove (in key : string) : void + Clear () : void + Get (in key : string) : CacheItem + Notify (in key : string) : void</pre>

Figure 6.7

The CacheService class

The **CacheService** implements a default asynchronous remoting interface, which includes the following methods:

- **Add**—The **Add** method has the following cache items metadata arguments: key, expiration, priority, and callback. These arguments are used by the **CacheService** to invalidate expired items and to notify the calling application when items are invalidated.
- **Remove**—The **Remove** method removes a specific item's metadata from the cache service identified by the provided key.
- **Clear**—The **Clear** method clears all metadata from the **CacheService**.
- **Get**—The **Get** method returns a specified **CacheItem** object identified by the provided key. This object includes the metadata associated with the cached item.
- **Notify**—The **Notify** method is used by the **CacheManager** to indicate that a cached item was accessed. This notification is passed by the **CacheService** to the expiration object associated with a cached item. For example, this notification can be used to manage sliding expiration implementations where the expiration time is reset every time the cached item is accessed.

These methods provide all of the functionality that you need to implement a class to handle the metadata associated with cached items.

Using the CacheStorage Class

The **CacheStorage** implementation separates the cache functionality from the cache data store. In the custom cache, three cache data stores are presented:

- **Static variables**—This is the simplest form of cache data store. The static variable data store is an in-process data store that can be used only within the scope of the application domain. For more information about static variables based caching, see Chapter 2, “Understanding Caching Technologies.”
- **Memory-mapped files**—The memory-mapped files data store is a fast access memory data store with cross-process capabilities. A cache based on memory-mapped files is limited to the scope of the computer on which it is running. For more information about caching using memory-mapped files, see Chapter 2, “Understanding Caching Technologies.”
- **SQL Server**—The SQL Server cache data store is the only data store that provides data persistency, meaning that your cached data is not lost if the data store computer fails. The scope of a SQL Server-based cache is the server farm. For more information about SQL Server-based caching, see Chapter 2, “Understanding Caching Technologies.”

Every cache storage implementation implements the following interfaces:

- **ICacheStorage**
- **ICacheMetadata**

The next sections describe each of these interfaces.

Using the ICacheStorage Interface

The **ICacheStorage** interface is used to store only the cached data. The interface definition of this is shown in Figure 6.8.

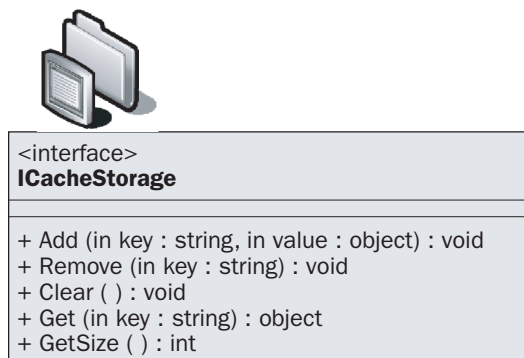


Figure 6.8

ICacheStorage interface

The **ICacheStorage** has the following interface:

- **Add**—The **Add** method inserts a key/value pair into the storage.
- **Remove**—The **Remove** method takes a key as input and removes the stored item value identified by this key from the storage.
- **Clear**—The **Clear** method clears all items from the storage.
- **Get**—The **Get** method takes a cached item's key as input and returns the stored item value identified by this key from the storage.
- **GetSize**—The **GetSize** method is used by the **IScavengingAlgorithm** implementation.

These methods are used by the **CacheManager** to store cache items.

Using the **ICacheMetadata** Interface

The **CacheStorage** can also implement the **ICacheMetadata** interface. The interface definition of this is shown in Figure 6.9. This is used by the **CacheService** to provide persistency for the metadata. This interface should be implemented only by storages that are persistent, such as SQL Server-based storage systems. In such cases, the metadata has to be persisted alongside the cached items.



```
<interface>
ICacheMetadata

+ Add (in key : string, in expirations : ICacheItemExpiration[], in priority
  : CacheItemPriority) : void
+ Remove (in key : string) : void
+ Clear ( ) : void
```

Figure 6.9

ICacheMetadata interface

The **ICacheMetadata** interface includes the following methods:

- **Add**—The **Add** method inserts a cached item's metadata to the persistent store.
- **Remove**—The **Remove** method removes an item's metadata from the persistent storage. This method is used by the **CacheService** when an item is removed from cache.
- **Clear**—The **Clear** method clears all metadata information from the storage.
- **Get**—The **Get** method retrieves all metadata from the storage. This method is used by the **CacheService** to initialize cache metadata from the persistent store after a power fail or process recycle.

You need to implement this interface only when you are using persistent storage systems, for example, SQL Server.

Using the CacheItem Class

The **CacheItem** class is a wrapper class around a cached item's value and metadata. The class declaration is shown in Figure 6.10.



CacheItem
+ Key : string + Value : object + Expirations : ICACHEITEMExpiration[] + Priority : CacheItemPriority

Figure 6.10

CacheItem class

An object of this type is returned by the **CacheManager** class and the **CacheService** class when the item's metadata is requested using the **GetItem** method.

Using the ICACHEITEMExpiration

This interface needs to be implemented by your expiration class. The interface definition is shown in Figure 6.11.



<interface> ICACHEITEMExpiration
+ Key : string + HasExpired () : bool + Notify () : void <<signal>> + ItemDependencyChange (in key : string)

Figure 6.11

ICACHEITEMExpiration interface

The **ICACHEITEMExpiration** interface has the following members:

- **HasExpired**—This method is periodically called by the **CacheService** to check whether a cached item has expired. The service scans the cache metadata for each cache item that has an expiration attached to it and checks whether it has expired. If the item has expired, the service removes the item from cache.

- **Notify**—This method is used by the **CacheService** to notify the expiration object attached to a cached item that that item has been accessed. For example, this is useful for sliding expiration schemes where the expiration time resets every time the item is accessed.
- **OnChange**—This event is listened to by the **CacheService** that, in response, invalidates the cached item and removes it from the cache.

You can use these methods to manage the expiration of your cached items.

Using the **IScavengingAlgorithm** Interface

This interface needs to be implemented by your scavenging algorithm class. The interface definition of this is shown in Figure 6.12.



```
<interface>
IScavengingAlgorithm

+ Add (in key : string, in priority : CacheItemPriority) : void
+ Remove (in key : string) : void
+ Clear ( ) : void
+ Scavenge ( ) : void
+ Notify (in key : string) : void
```

Figure 6.12

IScavengingAlgorithm interface

The **IScavengingAlgorithm** interface includes the following methods:

- **Add**—The **Add** method inserts a new key with its priority metadata to the scavenger class.
- **Remove**—The **Remove** method removes item metadata from the scavenger class.
- **Clear**—The **Clear** method clears all metadata information from the scavenger class.
- **Scavenge**—The **Scavenge** method is called by the **CacheService** and is used to run the scavenging algorithm that removes items from the cache to free cache storage resources.
- **Notify**—The **Notify** method is called by the **CacheService** to notify the scavenging algorithm object that a cached item was accessed. The scavenging algorithm uses this information when performing the scavenging process to decide which items should be removed from cache based on the scavenging policy (LRU, LFU, or any other implementation).

These methods are used by the **CacheStorage** class to execute the scavenging process when cache storage resources become scarce.

Using Additional Support Classes

In addition to the classes described so far, there are several other classes supporting the design, as shown in Figure 6.13.

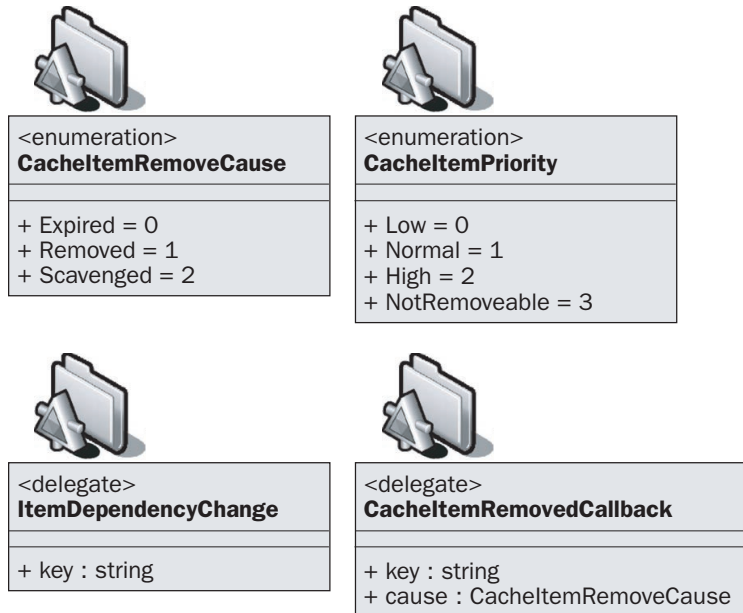


Figure 6.13
Supporting classes

These classes include delegates and enumerations.

Configuring a Custom Cache

As part of the custom cache implementation, there is an option to use custom configuration settings to customize the initialization and behavior of your custom cache. The explanations for the purpose and function of each field in the configuration file appear in the file, as shown in the following configuration sections.

```
<configuration>
  <configSections>
    <section name="CacheManagerSettings"
      type="Microsoft.ApplicationBlocks.Cache.CacheConfigurationHandler,
        Microsoft.ApplicationBlocks.Cache" />
  </configSections>
  <CacheManagerSettings>
```

```
<!-- STORAGE SETTINGS
    Use StorageInfo to set the assembly and class which implement
    the storage interfaces for the cache.

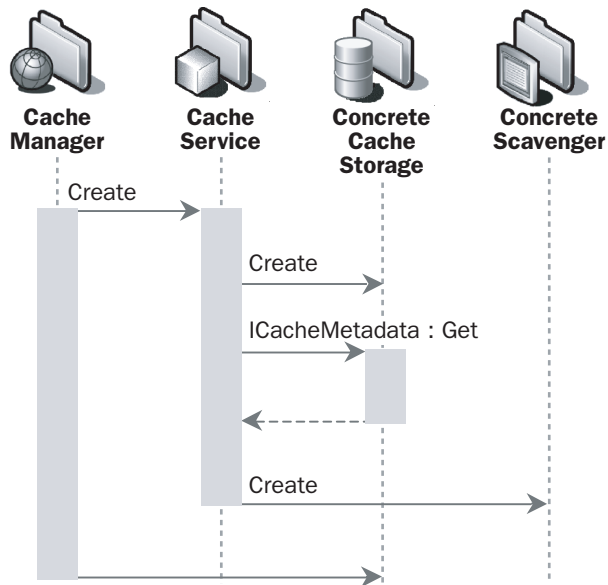
    Modes: InProc, OutProc, SqlServer
-->
<StorageInfo
    AssemblyName="Microsoft.ApplicationBlocks.Cache"
    ClassName="Microsoft.ApplicationBlocks.Cache.Storagees.
        SingletonCacheStorage"
    Mode="InProc"
    ConnectionString="Data Source=localhost;Database=cacheab;UserID
        =sa;password="
    ServiceConnectionString="tcpip=127.0.0.1:8282"
/>
<!-- SCAVENGING SETTINGS
    Use the ScavengingAlgorithm to set a class that will be executed
    when is performed.
-->
<ScavengingInfo
    AssemblyName="Microsoft.ApplicationBlocks.Cache"
    ClassName="Microsoft.ApplicationBlocks.Cache.Scavenging.
        LruScavenging"
    UtilizationForScavenging="80"
/>
<!-- EXPIRATION SETTINGS
    Use the ExpirationCheckInterval to change the interval to check for
    Cache items expiration. The value attribute is represented in
    seconds.
-->
<ExpirationInfo
    Interval="10"
/>
</CacheManagerSettings>
</configuration>
```

Because this is an XML file, it is easy to view and update without recompiling or redistributing sections of your caching system.

Understanding Custom Cache Use Cases

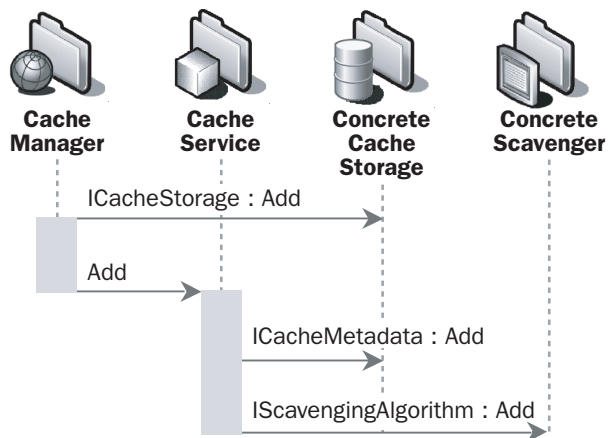
This section presents the major use cases of a custom cache.

The use case shown in Figure 6.14 details the sequence of events when the cache is initialized.

**Figure 6.14**

Cache initialization use case

The use case shown in Figure 6.15 details the sequence of events when an item is added to the cache.

**Figure 6.15**

Add item use case

The use case shown in Figure 6.16 details the sequence of events when an item is removed from the cache.

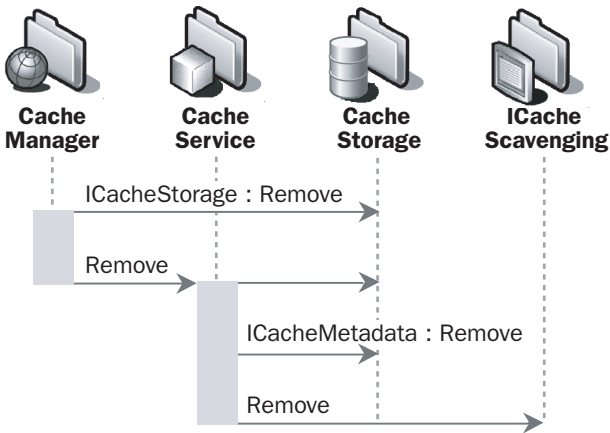


Figure 6.16
Remove item use case

The use case shown in Figure 6.17 details the sequence of events when an item is retrieved from the cache.

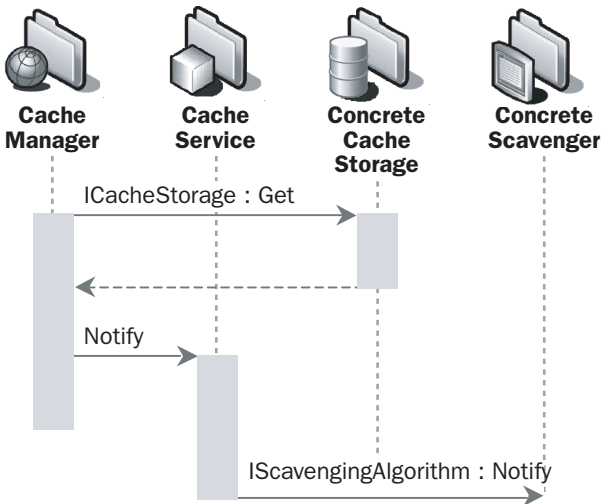
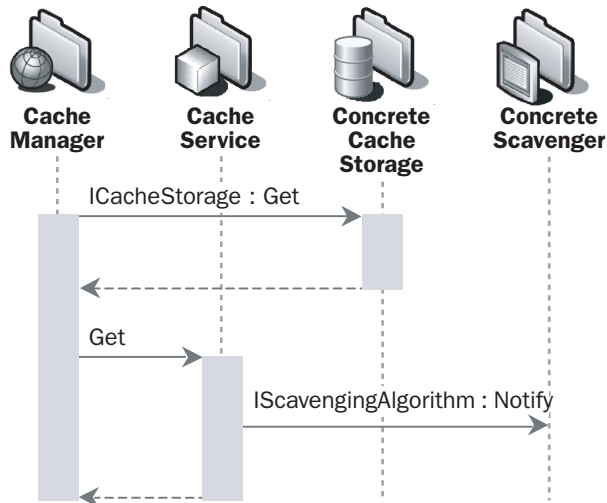


Figure 6.17
Get item use case

The use case shown in Figure 6.18 details the sequence of events when an item with metadata is retrieved from the cache.

**Figure 6.18***Get item and metadata use case*

After you design and implement a cache for your application, you may decide that the data in there is sensitive to your organization. If this is the case, you should add some code to secure the contents, just as you would secure them in their natural storage mechanism.

Securing a Custom Cache

This section of the guide describes how to secure your cached items against data tampering and data spoofing. You should ensure that any data that is secured in its original format is also secured when being transmitted to and from the cache and when stored inside the cache.

This section describes two methods of signing and verifying cache items to prevent tampering, and also how to encrypt data to prevent spoofing.

Signing Cache Items

You can use cache item signing to ensure that your cached data is not tampered with. To do this, all cache clients share a secret key that is used to compute the hash code of the cached item. This hash code is stored in the cache alongside the cached item. When a client retrieves the item from cache, it re-computes the hash using the secret key and if both versions of the hash code are not identical, the cached item has been tampered with.

The following code samples show how to sign and verify your cache items using different signing techniques.

Implementing MACTripleDES Data Signing and Verifying

Use the following code to sign a cache item.

```
public byte[] SignDataMAC( byte[] macKey, string item )
{
    //Encode the data to be hashed
    byte[] hashedData = Encoding.ASCII.GetBytes( item );
    //Compute the hash
    MACTripleDES mac = new MACTripleDES ( macKey );
    byte[] hash = mac.ComputeHash( hashedData, 0, hashedData.Length );
    return hash;
}
```

Use the following code to verify a MACTripleDES signed item.

```
public bool VerifySignDataMAC( byte[] macKey,
    byte[] cachedElementHash,
    string cachedElement )
{
    byte[] hashedData = Encoding.ASCII.GetBytes( cachedElement );

    //Compute the hash
    MACTripleDES mac = new MACTripleDES( macKey );
    byte[] newHash = mac.ComputeHash( hashedData, 0, hashedData.Length );

    //Compare the hashes
    for( int i = 0; i < 8; i++ )
    {
        if( newHash[ i ] != cachedElementHash[ i ] ) return false;
    }
    return true;
}
```

Using this code, you can ensure that your application is aware of any tampering that occurs to items stored in your cache.

Implementing XML Data Signing and Verifying

Use the following code to sign a cache item.

```
public string SignDataXml( RSA key, XmlNode xmlNode )
{
    // Create the SignedXml message.
    SignedXml signedXml = new SignedXml();
    signedXml.SigningKey = key;
    // Create a data object to hold the data to sign.
    DataObject dataObject = new DataObject();
    dataObject.Data = xmlNode.ChildNodes;
    dataObject.Id = "SignedCacheElement";
}
```

```

// Add the data object to the signature.
signedXml.AddObject( dataObject );

// Create a reference to be able to package everything into the message.
Reference reference = new Reference();
reference.Uri = "#SignedCacheElement";

// Add it to the message.
signedXml.AddReference( reference );

// Add a KeyInfo.
KeyInfo keyInfo = new KeyInfo();
keyInfo.AddClause( new RSAKeyValue( key ) );
signedXml.KeyInfo = keyInfo;

// Compute the signature.
signedXml.ComputeSignature();
return signedXml.GetXml().OuterXml;
}

```

Use the following code to verify a SignedXML signed item.

```

public bool VerifySignDataXml( string cachedData )
{
    XmlDocument cachedDoc = new XmlDocument();
    cachedDoc.LoadXml( cachedData );

    //Create a signedXml
    SignedXml signedXml = new SignedXml();

    //Load the xml document into the signed xml instance
    signedXml.LoadXml( (XmlElement)cachedDoc.ChildNodes[0] );

    //Check the signature
    if(!signedXml.CheckSignature()) return false;
    else return true;
}

```

Using this code, you can ensure that your application is aware of any tampering that occurs to items stored in your cache.

Encrypting Cached Items

You can use encryption to protect your cached data against data spoofing. Before inserting your item into the cache, you encrypt it using a secret key shared between all cache clients. When a client gets a cached item, it decrypts the item using that same key prior to using it.

The following code samples show how to encrypt and decrypt your cached data.

Implementing General Type Definitions

The following types are used in the encryption and decryption code.

```
#region P/Invoke structures
[StructLayout(LayoutKind.Sequential, CharSet=CharSet.Unicode)]
internal struct DATA_BLOB
{
    public int cbData;
    public IntPtr pbData;
}

[StructLayout(LayoutKind.Sequential, CharSet=CharSet.Unicode)]
internal struct CRYPTPROTECT_PROMPTSTRUCT
{
    public int cbSize;
    public int dwPromptFlags;
    public IntPtr hwndApp;
    public String szPrompt;
}
#endregion

#region External methods
[DllImport("Crypt32.dll", SetLastError=true, CharSet=CharSet.Auto)]
private static extern bool CryptProtectData(
    ref DATA_BLOB pDataIn,
    String szDataDescr,
    ref DATA_BLOB pOptionalEntropy,
    IntPtr pvReserved,
    ref CRYPTPROTECT_PROMPTSTRUCT pPromptStruct,
    int dwFlags,
    ref DATA_BLOB pDataOut);

[DllImport("Crypt32.dll", SetLastError=true, CharSet=CharSet.Auto)]
private static extern bool CryptUnprotectData(
    ref DATA_BLOB pDataIn,
    String szDataDescr,
    ref DATA_BLOB pOptionalEntropy,
    IntPtr pvReserved,
    ref CRYPTPROTECT_PROMPTSTRUCT pPromptStruct,
    int dwFlags,
    ref DATA_BLOB pDataOut);

[DllImport("kernel32.dll", CharSet=CharSet.Auto)]
private unsafe static extern int FormatMessage(int dwFlags,
    ref IntPtr lpSource,
    int dwMessageId,
    int dwLanguageId,
    ref String lpBuffer,
    int nSize,
    IntPtr *Arguments);
#endregion
```

```
#region Constants
public enum Store {Machine = 1, User};
static private IntPtr NullPtr = ((IntPtr)((int)(0)));
private const int CRYPTPROTECT_UI_FORBIDDEN = 0x1;
private const int CRYPTPROTECT_LOCAL_MACHINE = 0x4;
#endregion
```

These types must be declared for the encryption and decryption code to function.

Implementing Item Data Encryption

Use the following code to encrypt your data before storing it in the cache.

```
public byte[] Encrypt(byte[] plainText, byte[] optionalEntropy)
{
    bool retVal = false;
    DATA_BLOB plainTextBlob = new DATA_BLOB();
    DATA_BLOB cipherTextBlob = new DATA_BLOB();
    DATA_BLOB entropyBlob = new DATA_BLOB();

    CRYPTPROTECT_PROMPTSTRUCT prompt = new CRYPTPROTECT_PROMPTSTRUCT();
    prompt.cbSize = Marshal.SizeOf(typeof(CRYPTPROTECT_PROMPTSTRUCT));
    prompt.dwPromptFlags = 0;
    prompt.hwndApp = NullPtr;
    prompt.szPrompt = null;

    int dwFlags;
    try
    {
        try
        {
            int bytesSize = plainText.Length;
            plainTextBlob.pbData = Marshal.AllocHGlobal(bytesSize);
            if(IntPtr.Zero == plainTextBlob.pbData)
                throw new Exception("Unable to allocate plaintext buffer.");
            plainTextBlob.cbData = bytesSize;
            Marshal.Copy(plainText, 0, plainTextBlob.pbData, bytesSize);
        }
        catch(Exception ex)
        {
            throw new Exception("Exception marshalling data. " + ex.Message);
        }
        dwFlags = CRYPTPROTECT_LOCAL_MACHINE|CRYPTPROTECT_UI_FORBIDDEN;
        //Check to see if the entropy is null
        if(null == optionalEntropy)
        {
            //Allocate something
            optionalEntropy = new byte[0];
        }
        try
        {
            int bytesSize = optionalEntropy.Length;
            entropyBlob.pbData = Marshal.AllocHGlobal(optionalEntropy.Length);
            if(IntPtr.Zero == entropyBlob.pbData)
                throw new Exception("Unable to allocate entropy data buffer.");
        }
    }
}
```

```
        Marshal.Copy(optionalEntropy, 0, entropyBlob.pbData, bytesSize);
        entropyBlob.cbData = bytesSize;
    }
    catch(Exception ex)
    {
        throw new Exception("Exception entropy marshalling data. " + ex.Message);
    }
    retVal = CryptProtectData( ref plainTextBlob, "", ref entropyBlob,
        IntPtr.Zero, ref prompt, dwFlags, ref cipherTextBlob);
    if(false == retVal) throw new Exception("Encryption failed.");
}
catch(Exception ex)
{
    throw new Exception("Exception encrypting. " + ex.Message);
}
byte[] cipherText = new byte[cipherTextBlob.cbData];
Marshal.Copy(cipherTextBlob.pbData, cipherText, 0, cipherTextBlob.cbData);
return cipherText;
}
```

This code ensures that the data stored in the cache cannot be read without access to the shared key.

Implementing Item Data Decryption

Use the following code to decrypt your data before using it in the client.

```
public byte[] Decrypt(byte[] cipherText, byte[] optionalEntropy)
{
    bool retVal = false;
    DATA_BLOB plainTextBlob = new DATA_BLOB();
    DATA_BLOB cipherBlob = new DATA_BLOB();
    CRYPTPROTECT_PROMPTSTRUCT prompt = new CRYPTPROTECT_PROMPTSTRUCT();
    prompt.cbSize = Marshal.SizeOf(typeof(CRYPTPROTECT_PROMPTSTRUCT));
    prompt.dwPromptFlags = 0;
    prompt.hwndApp = IntPtr.Zero;
    prompt.szPrompt = null;

    try
    {
        try
        {
            int cipherTextSize = cipherText.Length;
            cipherBlob.pbData = Marshal.AllocHGlobal(cipherTextSize);
            if(IntPtr.Zero == cipherBlob.pbData)
                throw new Exception("Unable to allocate cipherText buffer.");
            cipherBlob.cbData = cipherTextSize;
            Marshal.Copy(cipherText, 0, cipherBlob.pbData, cipherBlob.cbData);
        }
        catch(Exception ex)
        {
            throw new Exception("Exception marshalling data. " + ex.Message);
        }
    }
}
```



```

    }
    DATA_BLOB entropyBlob = new DATA_BLOB();
    int dwFlags;

    dwFlags = CRYPTPROTECT_LOCAL_MACHINE|CRYPTPROTECT_UI_FORBIDDEN;
    //Check to see if the entropy is null
    if(null == optionalEntropy)
    {
        //Allocate something
        optionalEntropy = new byte[0];
    }
    try
    {
        int bytesSize = optionalEntropy.Length;
        entropyBlob.pbData = Marshal.AllocHGlobal(bytesSize);
        if(IntPtr.Zero == entropyBlob.pbData)
            throw new Exception("Unable to allocate entropy buffer.");
        entropyBlob.cbData = bytesSize;
        Marshal.Copy(optionalEntropy, 0, entropyBlob.pbData, bytesSize);
    }
    catch(Exception ex)
    {
        throw new Exception("Exception entropy marshalling data. " + ex.Message);
    }
    retVal = CryptUnprotectData(ref cipherBlob, null, ref
                                entropyBlob, IntPtr.Zero, ref prompt,
                                dwFlags, ref plainTextBlob);
    if(false == retVal) throw new Exception("Decryption failed.");
    //Free the blob and entropy.
    if(IntPtr.Zero != cipherBlob.pbData)
        Marshal.FreeHGlobal(cipherBlob.pbData);
    if(IntPtr.Zero != entropyBlob.pbData)
        Marshal.FreeHGlobal(entropyBlob.pbData);
}
catch(Exception ex)
{
    throw new Exception("Exception decrypting. " + ex.Message);
}
byte[] plainText = new byte[plainTextBlob.cbData];
Marshal.Copy(plainTextBlob.pbData, plainText, 0, plainTextBlob.cbData);
return plainText;
}

```

This code uses the shared key to decrypt the data that is encrypted in the cache.

After you implement your cache, and optionally secure it, you are ready to start using it. As with all pieces of software, you should monitor the performance of your cache and be prepared to tune any aspects that do not behave as expected.

Monitoring a Cache

Monitoring your cache usage and performance can help you understand whether your cache is performing as expected and helps you to fine tune your cache solution. This section of the guide describes which cache parameters should be monitored and what to look for when monitoring your cache.

Implementing Performance Counters

Implementing performance counters in your custom cache is the most effective way to monitor it. This section explains which performance counters to implement in your custom cache and what to look for when monitoring these counters.

Using the Total Cache Size Counter

This counter indicates the overall size of your cache in bytes. As the size of your cache increases, you often require the removal of unused items. This can be implemented by a process referred to as scavenging, which performs remove operations on the cache. This locks items in the cache and may result in degrading performance while the scavenging algorithm is running.

Using the Total Cache Entries Counter

This counter indicates the overall number of items in your cache. This performance counter on its own does not provide enough information to reach any conclusion regarding your cache performance. However, when combined with other counters, it can provide valuable information.

Using the Cache Hit/Miss Rate Counter

A cache hit occurs when you request an item from the cache and that item is available and returned to you. A cache miss occurs when you request an item from the cache and that item is not available. Obviously, your cache hit rate should be as high as possible and cache miss rate as low as possible. The higher your hit rate, the more effective your cache is performing because the cache can serve more of the items that your application needs. These counters indicate whether your application is using the cache effectively. If you observe low hit rates or high miss rates, consider the following points to improve your cache hit/miss ratio:

- Check your cache loading technique. If required items are not readily available in the cache, you may be using an inappropriate loading mechanism. For more information about the mechanisms available, see Chapter 5, “Managing the Contents of a Cache”.
- Increase the expiration time of your cached items. This retains items in cache for a longer period, resulting in an increased hit rate.

After you tune the caching mechanism to improve the hit rate, you should continue to monitor these figures for any unexpected changes.

Using the Cache Turnover Counter

The cache turnover rate refers to the number of insertions and deletions of items from the cache per second. A high cache turnover rate, together with a low cache hit rate, indicates that items are added and removed from cache frequently but are seldom used from cache, resulting in inefficient use of the cache.

Using the Cache Insert Time Counter

The cache insert time refers to the time it takes the cache to store your item and its metadata in the cache data store. This counter should be as low as possible for your cache to be more responsive to the application. High cache insert times adversely affect the performance of your application and defy the purpose for which caching is considered in the first place.

High cache insert times can indicate a problem with your cache implementation. You should note that the cache insert time should be constant regardless of the number of items in cache.

Using the Cache Retrieve Time Counter

The cache retrieve time refers to the time it takes the cache to get an item from the cache data store and return it to the client. Note that this time is equally important in cache misses where the requested item is not available in cache. This number should be as low as possible for improved performance of your application. High cache retrieve times limit the usefulness of your cache because they degrade your application's performance.

Cache retrieve times should remain constant regardless of the number of items in your cache.

Note: To implement performance counters in your custom cache, use the `System.Diagnostics.PerformanceCounter` class in the .NET Framework. For samples, see “How To: Diagnostics” in the MSDN Library.

Monitoring Your Cache Performance

After you implement performance counters into your custom cache, you can use the Windows performance monitor application (Perfmon) to view and analyze your cache performance data.

You will usually decide to monitor your cache when it is not delivering the expected performance. The following steps present a recommended scenario for monitoring your cache performance.

Note: Chapter 7, “Appendix,” includes cache performance data and graphs that were collected and analyzed under varying cache loads.

► **To monitor cache performance**

1. Monitor the cache insert and retrieve times under different cache loads (for example, number of items and size of cache) to identify where your performance problem is coming from.
2. Check your cache hit/miss ratio. If this is low, it indicates that items are rarely in cache when you need them. Possible causes for this include:
 - Your cache loading technique is not effective.
 - Your maximum allowed cache size is too small, causing frequent scavenging operations, which results in cached items being removed to free up memory.
3. Check your cache turnover rate. If this is high, it indicates that items are inserted and removed from cache at a high rate. Possible causes for this include:
 - Your maximum allowed cache size is too small, causing frequent scavenging operations which result in cached items being removed to free up memory.
 - Faulty application design, resulting in improper use of the cache.

Regular monitoring of your cache should highlight any changes in data use and any bottlenecks that these might introduce. This is the main management task associated with the post-deployment phase of using a caching system.

Synchronizing Caches in a Server Farm

A common problem for distributed applications developers is how you synchronize cached data between all servers in the farm. Generally speaking, if you have a situation in which your cache needs to be synchronized in your server farm, it almost always means that your original design is faulty. You should design your application with clustering in mind and avoid such situations in the first place.

However, if you have one of those rare situations where such synchronization is absolutely required, you should use file dependencies to invalidate the cache when the information in the main data store changes.

► **To create file dependencies for cache synchronization**

1. Create a database trigger that is activated when a record in your data store is changed.
2. Implement this trigger to create an empty file in the file system to be used for notification. This file should be placed either on the computer running SQL Server, a Storage Area Network (SAN), or another central server.
3. Use Application Center replication services to activate a service that copies the file from the central server to all disks in the server farm.
4. Make the creation of the file on each server trigger a dependency event to expire the cached item in the ASP.NET cache on each of the servers in the farm.

This method has several advantages, including:

- Using the file system is very efficient because the operating system is doing a lot of disk caching anyway.
- It is very flexible and easy to fine tune. For example, you can publish the files at timed intervals (instead of on every change) to invalidate several cache items at the same time.

However, because replicating a file across the server farm can take time, it is inefficient in cases where the cached data changes every few seconds.

Summary

This chapter has described the design details for a custom cache framework. You can use this solution as a basis for your own caching systems. It has also introduced how you can secure the contents of a cache and the items as they are transmitted to and from the cache, and it has described methods for monitoring a deployed caching solution.

7

Appendix

This chapter includes the following sections:

- “Appendix 1: Understanding State Terminology”
- “Appendix 2: Using Caching Samples”
- “Appendix 3: Reviewing Performance Data”

Appendix 1: Understanding State Terminology

This appendix provides descriptions and examples of the following terms associated with state, as overviewed in Chapter 1, “Understanding Caching Concepts”:

- *Lifetime of state*
- *Scope of state*, including:
 - *Physical scope*
 - *Logical scope*

State refers to data, and the condition of that data, being used in a system at a certain point in time. The lifetime and scope of state need to be taken into account when you are caching state, because the cache must be accessible for the same period of time and from the same locations as the original state.

Understanding the Lifetime of State

The *lifetime of state* refers to the time during which that state is valid—that is, from when it is created to when it is removed. Table 7.1 on the next page shows examples of lifetime durations.

Table 7.1: State lifetime

Lifetime of state	Duration of validity	Example
Permanent	Always valid. Data that is used across the application and exists beyond the lifetime of the application and its processes.	Data stored in a database, file system, or any other durable medium.
Process (atomic transaction)	Valid only during the lifetime of that transaction.	Product selection process for adding items to a shopping basket in an online store.
Process (long-running transaction)	Valid across all the messages involved in the transaction.	Purchase data for an order used during a checkout conversation between client and server.
Session	Specific to a particular user in a particular session and is valid only during that interactive user session.	The shopping basket created during an e-commerce session.
Message	The information that is passed between communicating services. Valid between creating the data and sending it on the one side and between receiving the message and processing the message data on the other.	Purchase order document submitted to a Web service from a client browser.

In addition to considering the duration of the validity of the state, you also need to consider where the state can be accessed from.

Understanding the Scope of State

Scope of state refers to the accessibility of the applications state, whether it is the physical locations or the logical locations.

Understanding Physical Scope

Physical scope refers to the physical locations from which the state can be accessed. Table 7.2 shows examples of physical scope.

Table 7.2: Physical scope

Physical scope	Locations of accessibility	Example
Organization	Can be accessed from any application in an organization	Active Directory information, Exchange Server Global Address List (GAL)
Farm	Shared between computers within an application farm	Hit count on a Web site
Machine	Shared amongst all applications or processes of an application on one computer	Registry data
Process	Shared across multiple <i>AppDomains</i> in the same process	Authentication tokens
AppDomain	Accessible only within the boundaries of an application domain	<i>AppDomain</i> parameters

Understanding Logical Scope

Logical scope refers to the logical locations from which the state can be accessed. Table 7.3 shows examples of logical scope.

Table 7.3: Logical scope

Logical scope	Locations of accessibility	Example
Application	Can be accessed only within a certain application	Product catalog for online site, which may be different from the catalog available to business-to-business partners
Business process	Available to a logical business process	Checkout process from an online shop
Role	Available to a subset of users/callers of an application	Members of the management role accessing employee salary information
User	Available to only one logged user	Personal data, news page, and session page

Understanding your application's state and its characteristics, such as lifetime and scope, is important for planning your caching policies and mechanisms.

Appendix 2: Using Caching Samples

The following samples demonstrate aspects of caching that preceding chapters describe. They show advanced details of ways to load and expire data in your cache.

The first sample shows how to implement a cache notification system for expiring data in a cache and loading new data into a cache using SQL Server 2000 Notification Services. This is one of the loading systems discussed in Chapter 5, “Managing the Contents of a Cache.”

The second sample shows how to develop an extended format time expiration using Microsoft Visual C#® .NET development tool. You can use this code to create a recurring absolute time expiration policy. This was introduced in Chapter 5, “Managing the Contents of a Cache.”

Implementing a Cache Notification System

This sample supplements the earlier discussion of SQL Server 2000 Notification Services. It implements a flight schedule Web application that uses data from an XML Web service. The flow of the application is demonstrated in Figure 7.1.

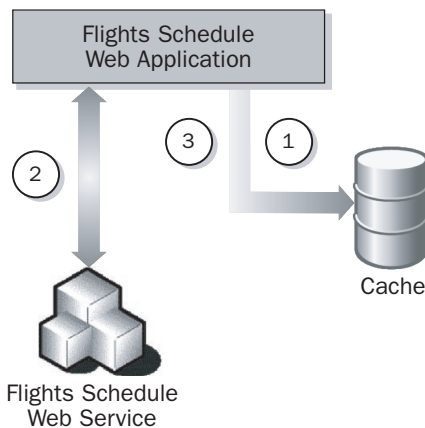


Figure 7.1

Flight schedule application flow

The flow of data within the applications follows these steps:

1. The application looks for flight schedule data in the cache. If it's there, the data is returned and displayed.
2. If the data is not in the cache, the application retrieves the data from the Web service.
3. The data is stored in the cache for future requests.

The problem with this scenario is that if the flight schedule in the Web service changes, the change is not reflected in the cache, and the Web application displays obsolete data.

To solve this problem, you can implement a notification mechanism using SQL Server Notification Services to notify the cache of flight schedule changes. The flow of the application after implementing this mechanism is shown in Figure 7.2.

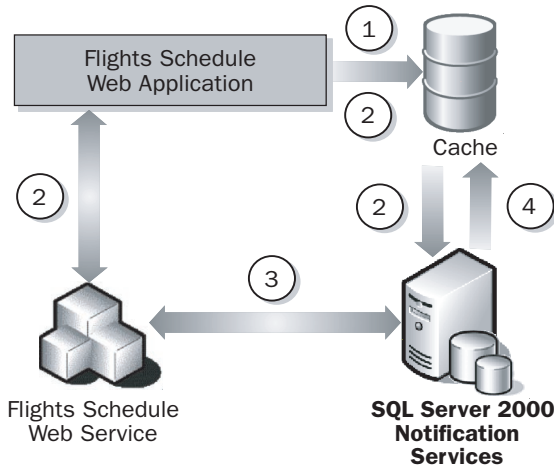


Figure 7.2

Flight schedule application flow when using SQL Server 2000 Notification Services

The flow of data in the application now follows a different set of steps:

1. The application looks for the flight schedule data in the cache. If it's there, the data is returned and displayed.
2. If the data is not in the cache, the Web application retrieves the data from the Web service and instructs the cache to store the data. The cache stores the data and subscribes to a **FlightsNotification** event in Notification Services.
3. Notification Services monitors the Web service for flight schedule changes.
4. When a change occurs in the Web service, Notification Services informs the cache, which invalidates the obsolete item and then removes it.

Using Notification Services ensures that any changes to schedules are reflected in the Web application in a timely manner.

The full code sample for the flight schedule application can be downloaded from the Patterns and Practices Caching Architecture Guide workspace at GotDotNet.

For more information about Notification Services, see the SQL Server 2000 Notification Services Web site at <http://www.microsoft.com/sql/ns/>.

Implementing an Extended Format Time Expiration Algorithm

The following sample shows an implementation of absolute time, recurrent expiration algorithm. The extended format expiration algorithm allows you to define the lifetime of an item by specifying the expiration policy as a recurrent absolute time. It also enables you to define the lifetime of an item by specifying the day of week or the day of month when an item should expire.

```
using System;
```

```
/*
```

```
    minute      - 0-59
    hour        - 0-23
    day of month - 1-31
    month       - 1-12
    day of week  - 0-7 (Sunday is 0 or 7)
    wildcards   - * means run every
```

```
Examples:
```

```
5 * * * *      - run 5th minute of every hour
5,31 * * * *   - run every 5th and 31st minute of every hour
1 21 * * *     - run every minute of the 21st hour of every day
1 15,21 * * *- run every minute of the 15th and the 21st hour of every day
31 15 * * *    - run 3:31 PM every day
7 4 * * 6      - run Saturday 4:07 AM
15 21 4 7 *    - run 9:15 PM on 4 July
```

```
*/
```

```
namespace ExtendedFormat
```

```
{
```

```
    class Test
```

```
    {
```

```
        /// <summary>
```

```
        /// The main entry point for the application
```

```
        /// </summary>
```

```
        [STAThread]
```

```
        static void Main(string[] args)
```

```
        {
```

```
            Console.WriteLine(
```

```
                ExtendedFormat.IsExtendedExpired(
```

```
                    "0 15,22 * * *",
```

```
                    new DateTime( 2002, 4, 22, 16, 0, 0 ),
```

```
                    new DateTime( 2002, 4, 22, 22, 0, 0 )));
```

```
        }
```

```
    }
```

```
    public class ExtendedFormat
```

```
    {
```

```

/// <summary>
/// Test the extended format with a given date.
/// </summary>
/// <param name="format">The extended format string</param>
/// <param name="getTime">The time when the item has been
                                refreshed</param>
/// <param name="nowTime">DateTime.Now, or the date to test
                                with</param>
/// <returns></returns>
public static bool IsExtendedExpired( string format,
                                     DateTime getTime,
                                     DateTime nowTime )
{
    //Validate arguments
    if( format == null )
        throw new ArgumentNullException( "format" );

    string[] fmtarr = format.Split( ' ' );
    if( fmtarr.Length != 5 )
        throw new ArgumentException( "Invalid format (" +
                                     format + ")", "format");

    return ( ValidateMinute( fmtarr[ 0 ], getTime, nowTime ) &&
             ValidateHour( fmtarr[ 1 ], getTime, nowTime ) &&
             ValidateDayOfMonth( fmtarr[ 2 ], getTime, nowTime ) &&
             ValidateMonth( fmtarr[ 3 ], getTime, nowTime ) &&
             ValidateDayOfWeek( fmtarr[ 4 ], getTime, nowTime ) );
}

public static bool ValidateMinute( string formatItem,
                                  DateTime getTime,
                                  DateTime nowTime )
{
    if( formatItem.IndexOf( "*" ) != -1 )
    {
        return true;
    }
    else
    {
        TimeSpan ts;
        int minute;
        string[] minutes = formatItem.Split( ',' );
        foreach( string sminutes in minutes )
        {
            minute = int.Parse( sminutes );
            ts = nowTime.Subtract( getTime );
            if( ts.TotalMinutes >= 60 )
            {
                return true;
            }
        }
        else
        {

```

```
        if( ( getTime.Minute < minute && nowTime.Minute >= minute ) ||
            ( getTime.Hour != nowTime.Hour && nowTime.Minute >= minute ) ||
            ( getTime.Hour != nowTime.Hour && getTime.Minute < minute ) )
        {
            return true;
        }
    }
    return false;
}

public static bool ValidateHour( string formatItem,
                                DateTime getTime,
                                DateTime nowTime )
{
    if( formatItem.IndexOf( "*" ) != -1 )
    {
        return true;
    }
    else
    {
        TimeSpan ts;
        int hour;
        string[] hours = formatItem.Split( ',' );
        foreach( string shour in hours )
        {
            hour = int.Parse( shour );
            ts = nowTime.Subtract( getTime );
            if( ts.TotalHours >= 24 )
            {
                return true;
            }
            else
            if( ( getTime.Hour < hour && nowTime.Hour >= hour ) ||
                ( nowTime.Day != getTime.Day && nowTime.Hour >= hour ) ||
                ( getTime.Day != nowTime.Day && getTime.Hour < hour ) )
            {
                return true;
            }
        }
        return false;
    }
}

public static bool ValidateDayOfMonth( string formatItem,
                                        DateTime getTime,
                                        DateTime nowTime )
{
    if( formatItem.IndexOf( "*" ) != -1 )
    {
        return true;
    }
    else
    {

```

```

        TimeSpan ts;
        int day;
        string[] days = formatItem.Split( ',' );
        foreach( string sday in days )
        {
            day = int.Parse( sday );
            ts = nowTime.Subtract( getTime );
            if( ts.TotalDays >= 30 )
            {
                return true;
            }
            else
            {
                if( ( getTime.Day < day && nowTime.Day >= day ) ||
                    ( getTime.Month != nowTime.Month && nowTime.Day >= day ) ||
                    ( getTime.Month != nowTime.Month && getTime.Day < day ))
                {
                    return true;
                }
            }
        }
        return false;
    }
}

public static bool ValidateMonth( string formatItem,
                                DateTime getTime,
                                DateTime nowTime )
{
    if( formatItem.IndexOf( "*" ) != -1 )
    {
        return true;
    }
    else
    {
        TimeSpan ts;
        int month;
        string[] months = formatItem.Split( ',' );
        foreach( string smonth in months )
        {
            month = int.Parse( smonth );
            ts = nowTime.Subtract( getTime );
            if( ts.TotalDays >= 365 )
            {
                return true;
            }
            else
            {
                if( ( getTime.Month < month && nowTime.Month >= month ) ||
                    ( nowTime.Year != getTime.Year && nowTime.Month >= month ) ||
                    ( getTime.Year != nowTime.Year && getTime.Month < month ))
                {
                    return true;
                }
            }
        }
    }
}

```

```
        }
    }
    return false;
}

public static bool ValidateDayOfWeek( string formatItem,
                                     DateTime getTime,
                                     DateTime nowTime )
{
    if( formatItem.IndexOf( "*" ) != -1 )
    {
        return true;
    }
    else
    {
        DateTime dt;
        DayOfWeek weekday;
        TimeSpan ts;

        string[] days = formatItem.Split( ',' );
        foreach( string sday in days )
        {
            weekday = (DayOfWeek)Enum.Parse(
                typeof(DayOfWeek), sday, true );
            ts = nowTime.Subtract( getTime );
            if( ts.TotalDays >= 7 )
            {
                return true;
            }
            else
            {
                dt = nowTime;
                for( int i = 0; getTime < dt; i++ )
                {
                    dt = dt.AddDays( -1 );
                    if( dt.DayOfWeek == weekday )
                    {
                        return true;
                    }
                }
                return false;
            }
        }
        return false;
    }
}

}
```

This code allows you to specify your time expirations in recurring absolute terms, such as at midnight on the first day of every month. This can be useful for a caching mechanism that requires expiration policies other than standard absolute or sliding expirations.

Appendix 3: Reviewing Performance Data

This appendix contains performance data comparing the performance of the different caching technologies discussed in Chapter 2, “Understanding Caching Technologies.”

After you narrow down your choice of caching mechanism based on the scope, lifetime, and location required, you can use these performance testing results to further guide you in choosing the most appropriate cache technology for your application. To use this data, you must know the approximate size of the items you will be caching and the approximate user load that your application will need to support.

Introducing the Test Scenarios

Each technology was tested and performance monitored in two different ways:

- **Changing the size of the cached items**—Firstly, the time taken to insert items into the cache and retrieve them from the cache and the number of requests that could be handled per second was monitored for items of various sizes. These results show you which caching technology is best to use for small and large cache items. When the cache storage resides in a different process than the application, the size of the cached items can affect performance. When the data is retrieved or stored it needs to be serialized, and serialization is a relatively time and resource consuming operation.
- **Increasing the number of concurrent requests**—In the second test, the time taken per request and the number of requests that could be handled per second was monitored as the number of users was increased. These results show you which caching technology is best to use for different expected user loads.

Workload can affect the performance of an application. Because each cache storage technology handles multiple requests differently (for example, using different locking techniques), the technology you choose can affect the scalability of your application.

Each of these test scenarios was carried out for both retrieving items from the cache and storing items in the cache.

Defining the Computer Configuration and Specifications

The tests were carried out in a lab using a classic three-tier architecture: a database server, a Web server, and multiple client computers. The network configuration is shown in Figure 7.3 on the next page.

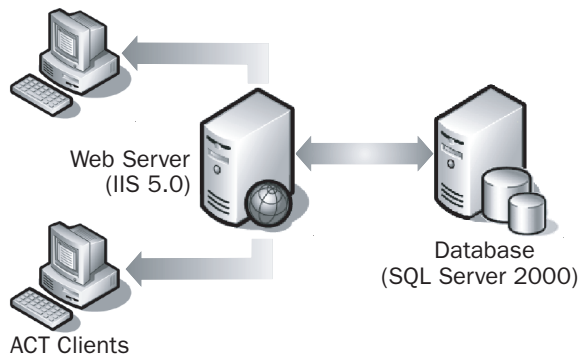


Figure 7.3
Test bed configuration

All computers were running Windows 2000 Advanced Server with Service Pack 2. The database server was using SQL Server 2000 and the Web server was using Internet Information Service 5.0. Table 7.4 shows a summary of the computer specification for each computer in the test system.

Table 7.4: Computer specifications

Computer	Type and CPU	Number of CPUs	Memory	Disk
Client	Dell PowerEdge 2550, 1GHz	2	256 MB	16.9 GB
Web server	Compaq ProLiant DL380, 2GHZ	2	1 GB	16.9 GB
Database server	Compaq ProLiant DL380, 2GHZ	2	1 GB	16.9 GB

The following graphs results were obtained running the described tests on these computers.

Presenting the Performance Test Results

This section presents the results of the performance testing. Because all of the caching technologies were similarly tested some of the results are obvious.

All of the tests noted that the ASP.NET view state doesn't perform well; this is because the data is transported back and forth between the client and the server. Another obvious result is that the in-process technologies, such as the ASP.NET cache and static variables, provide the best performance. This is because the cache itself exists within the same process as the application so no data serialization is required.

The rest of the results shown in the following graphs vary, but you can see that SQL Server 2000 and remoting singletons perform similarly. This is because they are both out-of-process technologies that require serialization.

The only technology that behaves significantly differently is the memory mapped files. This is because the memory-mapped file technology is un-managed and is implemented by streaming data to and from specified memory locations. Memory-mapped files are neither an in-process technology nor an out-of-process technology.

Testing Cached Items of Different Sizes

This test compared the performance of various caching technologies when retrieving items from the cache and storing items in the cache using different sizes of cache items.

Retrieving Items from the Cache

The in-process technologies, such as the ASP.NET cache and static variables, produce the best performance when retrieving large items from the cache. This is expected because these technologies don't require serialization. Memory-mapped files produce acceptable results for items up to about 10 MB in size, as do the ASP.NET session, SQL Server 2000, and remoting singletons for up to about 200 KB items. View state shows the worst performance, taking about twice the time of the other technologies to retrieve even very small items.

Figure 7.4 shows the effect of the item size on the retrieval request execution time.

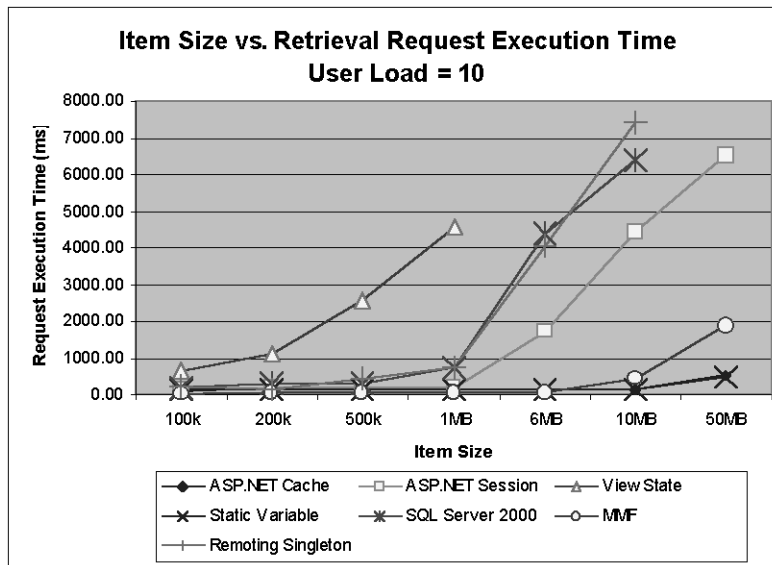


Figure 7.4

Item size versus retrieval request execution time

As the size of the items increases, the number of retrieval requests that the cache can handle per second stays relatively constant for the in-process technologies, but it decreases for the others. In this scenario, the ASP.NET session cache produces consistently better results for items less than 6 MB in size than the memory-mapped

file cache. The other technologies do not perform well with anything other than very small items.

Figure 7.5 shows the effect of the item size on the number of retrieval requests serviced per second.

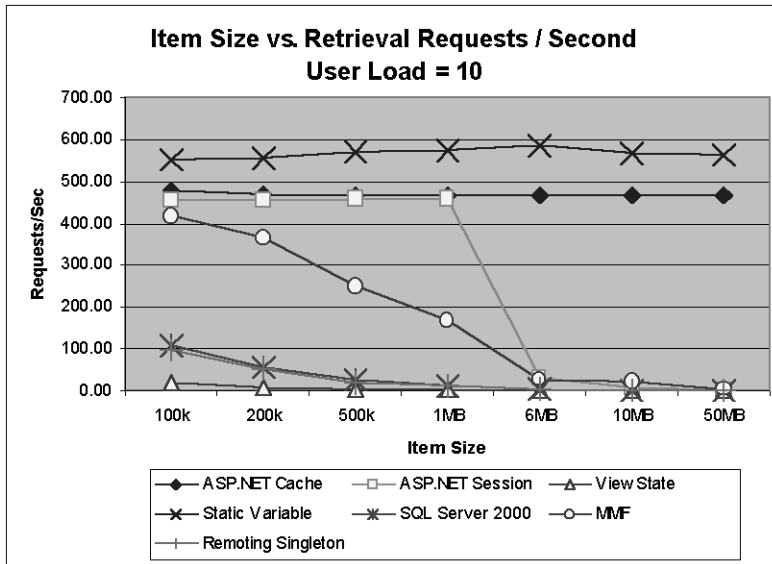


Figure 7.5

Item size versus retrieval requests per second

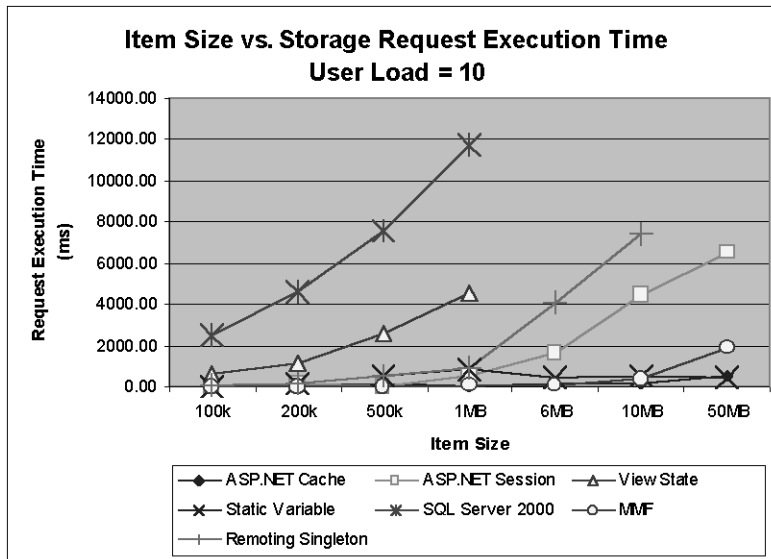
These figures show that ASP.NET session and static variables produce consistently good results for retrieving all sizes of items from the cache.

Storing Items in the Cache

The results for the time taken to store increasingly sized items in the cache are similar to those for retrieving items for all technologies except for SQL Server 2000, which is relatively slower. This is because when you retrieve data, SQL Server automatically stores frequently used tables in memory to minimize input/output operations and to improve performance. So although the retrieval of SQL Server items is similar to the other technologies, the storage of those items is significantly slower due to the need for SQL Server to do more than simply store the data, for example, to create indexes.

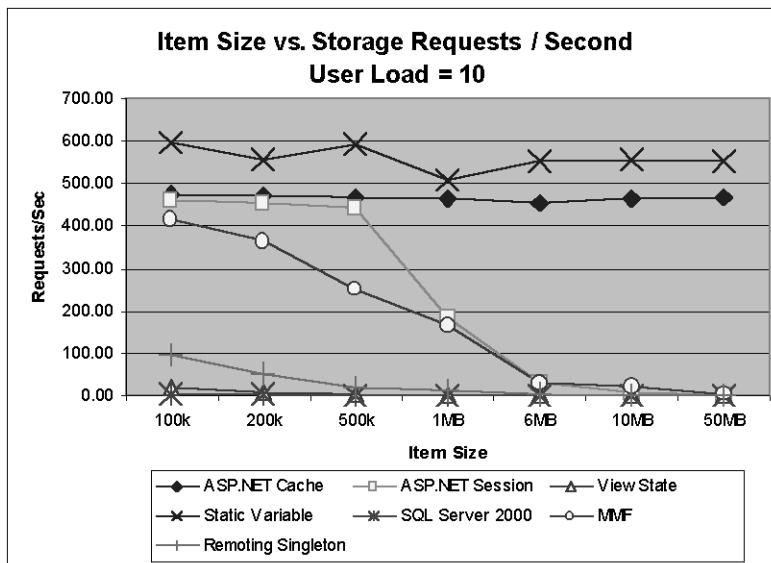
Figure 7.6 shows the effect of the item size on the storage request execution time.

The results for the effect of item size on the number of storage requests that can be handled per second are very similar to the results for the retrieval test, and the reasoning behind those results is the same.

**Figure 7.6**

Item size versus storage request execution time

Figure 7.7 shows the effect of the item size on the number of storage requests serviced per second.

**Figure 7.7**

Item size versus storage requests per second

From all of these results you can deduce that the best performing caches are based on the in-process technologies, and the worst are based on out-of-process technologies. However, the data presented here can help you make the correct choice of technology based on the size of the items you need to cache alongside all the other constraints discussed in Chapter 2, “Understanding Caching Technologies.”

Testing Increasing Numbers of Concurrent Requests

This test compared the performance of various caching technologies when retrieving items from the cache and storing items in the cache under different user loads.

Retrieving Items from the Cache

This test found that the time taken to retrieve 200 KB items from the cache did not vary significantly for any of the technologies with a user load of up to about 10 users. However, beyond this, it was found that the in-process technologies (for example, the ASP.NET cache and static variables), the ASP.NET session, and memory-mapped files produced a consistently good performance, whereas SQL Server 2000 and remoting singleton did not perform so well. View state showed unacceptable results for any user load greater than about 10 users.

Figure 7.8 shows the effect of the user load on the retrieval request execution time.

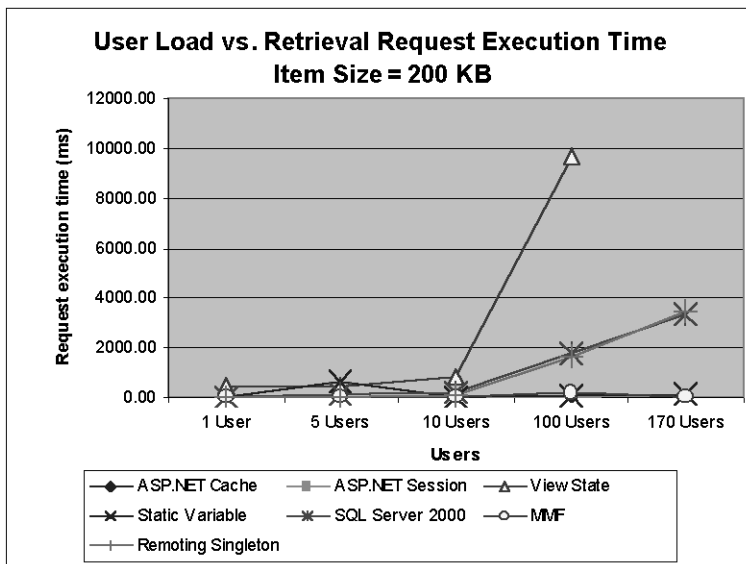


Figure 7.8

User load versus retrieval execution time

As the number of the users increases, the number of retrieval requests that the cache handles per second grows until a bottleneck is hit and then the number of requests per second flattens out. For the in-process technologies, you can see that the amount of requests per second at the bottleneck (about 100 users) is much higher than the out-of-process ones.

Figure 7.9 shows the effect of the user load on the number of retrieval requests serviced per second.

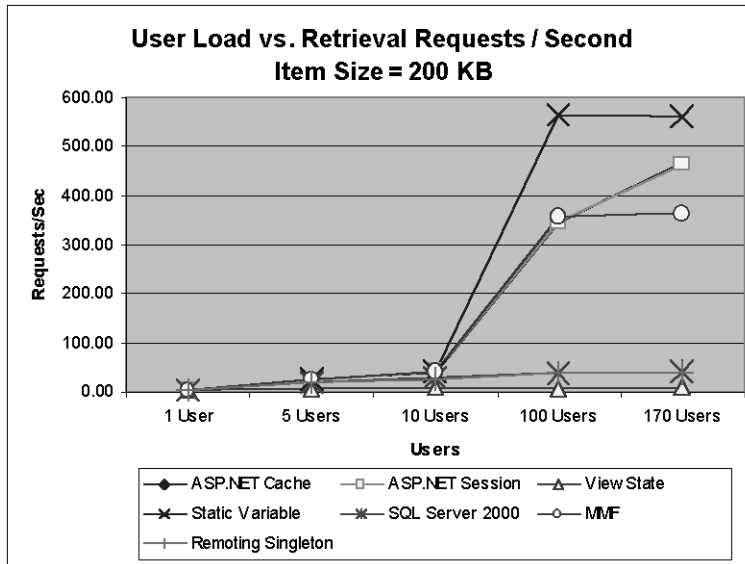


Figure 7.9

User load versus retrieval requests per second

The static variable cache produces the best retrieval results for increasing user load, as it did for increasing item sizes.

Storing Items in the Cache

The results for the time taken to store items in the cache as the user load increased are similar to those for retrieving items for all technologies except for SQL Server 2000, which is relatively slower. This is because when you are storing data, SQL Server has to index the inserted data in the required tables and to physically store the data on the disk. These types of operations take a relatively significant amount of time to complete.

Figure 7.10 on the next page shows the effect of the user load on the storage request execution time.

The results for the effect of user load on the number of storage requests that can be handled per second are very similar to the results for the retrieval test, and the reasoning behind those results is the same.

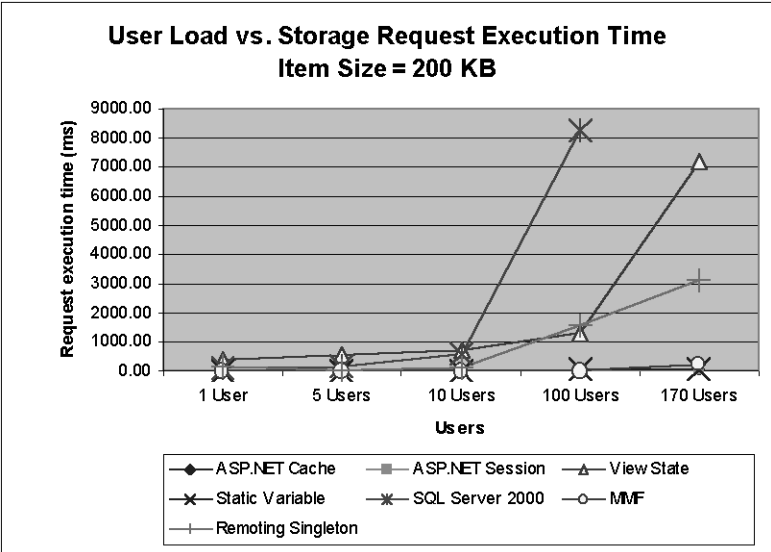


Figure 7.10
User load versus storage request execution time

Figure 7.11 shows the effect of the user load on the number of storage requests serviced per second.

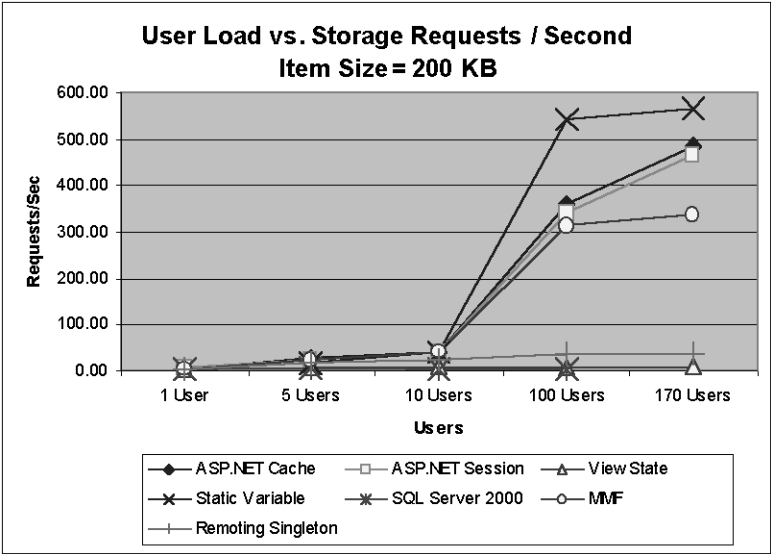


Figure 7.11
User load versus storage requests per second

The performance tests for differing user loads show the same overall results as the differing item size tests: that the in-process caching technologies perform significantly better than out-of-process systems. Use the data presented here along with your own parameters, estimated user load and item size data, to select the appropriate caching technology for your application within the technology constraints discussed in Chapter 2, “Understanding Caching Technologies.”

Microsoft®

patterns & practices



Proven practices for predictable results

Patterns & practices are Microsoft's recommendations for architects, software developers, and IT professionals responsible for delivering and managing enterprise systems on the Microsoft platform. Patterns & practices are available for both IT infrastructure and software development topics.

Patterns & practices are based on real-world experiences that go far beyond white papers to help enterprise IT pros and developers quickly deliver sound solutions. This technical guidance is reviewed and approved by Microsoft engineering teams, consultants, Product Support Services, and by partners and customers. Organizations around the world have used patterns & practices to:

Reduce project cost

- Exploit Microsoft's engineering efforts to save time and money on projects
- Follow Microsoft's recommendations to lower project risks and achieve predictable outcomes

Increase confidence in solutions

- Build solutions on Microsoft's proven recommendations for total confidence and predictable results
- Provide guidance that is thoroughly tested and supported by PSS, not just samples, but production quality recommendations and code

Deliver strategic IT advantage

- Gain practical advice for solving business and IT problems today, while preparing companies to take full advantage of future Microsoft technologies.

To learn more about *patterns & practices* visit: msdn.microsoft.com/practices

To purchase *patterns & practices* guides visit: shop.microsoft.com/practices

patterns & practices

Proven practices for predictable results

patterns & practices

Proven practices for predictable results

Patterns & practices are available for both IT infrastructure and software development topics. There are four types of patterns & practices available:

Reference Architectures

Reference Architectures are IT system-level architectures that address the business requirements, operational requirements, and technical constraints for commonly occurring scenarios. Reference Architectures focus on planning the architecture of IT systems and are most useful for architects.

Reference Building Blocks

Reference Building Blocks are re-usable sub-systems designs that address common technical challenges across a wide range of scenarios. Many include tested reference implementations to accelerate development.

Reference Building Blocks focus on the design and implementation of sub-systems and are most useful for designers and implementors.

Operational Practices

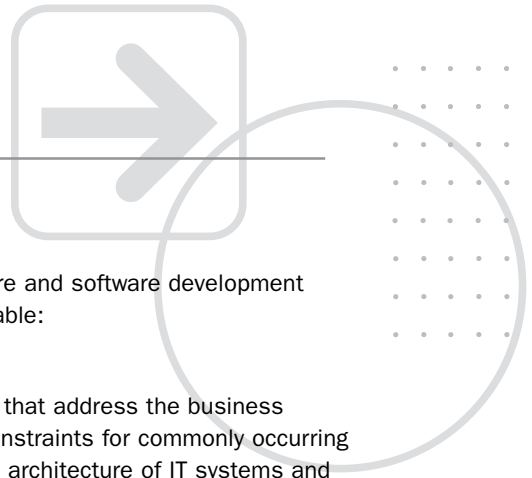
Operational Practices provide guidance for deploying and managing solutions in a production environment and are based on the Microsoft Operations Framework. Operational Practices focus on critical tasks and procedures and are most useful for production support personnel.

Patterns

Patterns are documented proven practices that enable re-use of experience gained from solving similar problems in the past. Patterns are useful to anyone responsible for determining the approach to architecture, design, implementation, or operations problems.

To learn more about *patterns & practices* visit: msdn.microsoft.com/practices

To purchase *patterns & practices* guides visit: shop.microsoft.com/practices



patterns & practices current titles



December 2002

Reference Architectures

Microsoft Systems Architecture—Enterprise Data Center 2007 pages
Microsoft Systems Architecture—Internet Data Center 397 pages
Application Architecture for .NET: Designing Applications and Services 127 pages
Microsoft SQL Server 2000 High Availability Series: Volume 1: Planning 92 pages
Microsoft SQL Server 2000 High Availability Series: Volume 2: Deployment 128 pages
Enterprise Notification Reference Architecture for Exchange 2000 Server 224 pages
Microsoft Content Integration Pack for Content Management Server 2001
and SharePoint Portal Server 2001 124 pages
UNIX Application Migration Guide 694 pages
Microsoft Active Directory Branch Office Guide: Volume 1: Planning 88 pages
Microsoft Active Directory Branch Office Series Volume 2: Deployment and
Operations 195 pages
Microsoft Exchange 2000 Server Hosting Series Volume 1: Planning 227 pages
Microsoft Exchange 2000 Server Hosting Series Volume 2: Deployment 135 pages
Microsoft Exchange 2000 Server Upgrade Series Volume 1: Planning 306 pages
Microsoft Exchange 2000 Server Upgrade Series Volume 2: Deployment 166 pages

Reference Building Blocks

Data Access Application Block for .NET 279 pages
.NET Data Access Architecture Guide 60 pages
Designing Data Tier Components and Passing Data Through Tiers 70 pages
Exception Management Application Block for .NET 307 pages
Exception Management in .NET 35 pages
Monitoring in .NET Distributed Application Design 40 pages
Microsoft .NET/COM Migration and Interoperability 35 pages
Production Debugging for .NET-Connected Applications 176 pages
Authentication in ASP.NET: .NET Security Guidance 58 pages
Building Secure ASP.NET Applications: Authentication, Authorization, and
Secure Communication 608 pages

Operational Practices

Security Operations Guide for Exchange 2000 Server 136 pages
Security Operations for Microsoft Windows 2000 Server 188 pages
Microsoft Exchange 2000 Server Operations Guide 113 pages
Microsoft SQL Server 2000 Operations Guide 170 pages
Deploying .NET Applications: Lifecycle Guide 142 pages
Team Development with Visual Studio .NET and Visual SourceSafe 74 pages
Backup and Restore for Internet Data Center 294 pages

For current list of titles visit: msdn.microsoft.com/practices

To purchase *patterns & practices* guides visit: shop.microsoft.com/practices

