Microsoft

Design and Implementation Guidelines for Web Clients

patterns & practices

Version 1.0

Contents

1

The Presentation Layer

Introduction

Most, if not all, applications require some level of user interaction. In today's distributed applications, the code that manages this user interaction is in the presentation layer.

Most simple presentation layers contain user interface components, such as Microsoft® Windows Forms or ASP.NET Web Forms. These components typically contain code to perform functions such as configuring the visual appearance of controls; accepting and validating user input; and acquiring and rendering data from data access logic components or business components.

The presentation layer can also include user interface process components. User interface process components perform presentation layer tasks that are not directly concerned with user interactions. For example, user interface process components orchestrate the flow of control between forms in the presentation layer and coordinate background tasks such as state management and handling of concurrent user activities.

Design and Implementation Guidelines for Web Clients provides advice on how best to implement logic in the presentation layer of a distributed application. This guide is designed to accompany the User Interface Process Application Block; this application block provides a template implementation for user interface process components. For more information about how and when to use this block, see Chapter 2, "Using Design Patterns in the Presentation Layer," in this guide. For more information about the application block, including download information, see *User Interface Process Application Block Overview* on MSDN® (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/uip.asp*).

How To Use This Guide

This guide addresses specific goals of presentation layer component design. This guide provides prescriptive recommendations and code samples that enable you to use design patterns and Microsoft .NET Framework programming idioms effectively in the presentation layer of your applications. The intent of this guide is to help you increase the portability, maintainability, scalability, security, and overall design quality of your presentation layer code; it does not discuss aesthetic user interface design issues.

The information in this guide is organized into the following chapters:

- This chapter introduces the topics in this guide and provides guidance on basic terminology. It also notes the decisions the authors of this guide assume you have already made.

  Additionally, this chapter summarizes the key messages that are documented in detail elsewhere in this guide and describes how this guide fits into the Microsoft *patterns & practices* family of documentation.
- Chapter 2, "Using Design Patterns in the Presentation Layer," describes how to separate the responsibilities of components in the presentation layers by using common design patterns.

  Design patterns help you get a clean separation between the code that presents data to the user and accepts user interactions and the code that orchestrates the flow of control between forms and handles issues such as state management, data access, and asynchronous behavior in the presentation layer. By partitioning your code in this way, you can reduce the coupling between the various parts of your application, and thereby make your code easier to change and extend as requirements evolve.

  Chapter 2 introduces a template implementation of the key design patterns. The template is included in the User Interface Process Application Block. Chapter 2 describes how to use this block as the basis for your own user interface code, thereby realizing the benefits described in the previous paragraph.
- Chapter 3, "Building Maintainable Web Interfaces with ASP.NET," describes how to make ASP.NET code easier to implement and maintain. This chapter describes how to use custom controls to share specific behavior across multiple controls and how to use common page layouts to ensure there is a common appearance across all the pages in your Web site. This chapter also describes how to use inheritance appropriately and effectively in Web applications in order to reuse controls in the presentation layer.
- Chapter 4, "Managing Data," describes the correct way for components in the user interface (UI) to access, present, update, and validate data input, and how the UI participates in maintaining data integrity.

The first of these topics, "Accessing and Representing Data," compares and contrasts various techniques for accessing data in the presentation layer. This topic describes the best way to represent data in disconnected and streaming applications and how best to use transactions in the presentation layer. This topic also describes the importance of a layered approach to data access and how to use message-driven techniques, data access logic components, and business objects to access data in specific scenarios.

The second topic in the chapter, "Presenting Data Using Formatters, Data Binding, and Paging," describes how to use .NET Framework formatter classes, data binding techniques, and paging to display data to the user.

The third topic in the chapter, "Supporting Data Updates from the Presentation Layer," describes how the presentation layer can participate in updates to data in a back-end data store such as a relational database. This topic describes how to create data maintenance forms that let users create, read, update, and delete data entities either individually or as part of a list. It also describes how to increase productivity by implementing metadata-based forms that are sufficiently flexible to handle data in any structure.

The final topic in the chapter, "Validating Data in the Presentation Layer," describes scenarios where data validation is appropriate in the presentation layer. The presentation layer is the first line of defense against accidental or malicious rogue data from the user. This topic describes how and when to use .NET Framework validator controls to validate the format and content of input data and includes strategies for handling validation failures.

- Chapter 5, "Managing State in Web Applications," describes the types of state used in the presentation layer and offers guidance about how to manage state in applications written for the Web. Correct state management is critical to the scalability and availability of Web applications.

  For Web applications, this chapter discusses the pros and cons of storing per-user session data in the in-process **Session** object, the state server-based **Session** object, or the SQL Server™-based **Session** object. This chapter also discusses how to use cookies, hidden form fields, URL rewiring, and the ASP.NET **ViewState** property as alternative ways to maintain state between pages in an ASP.NET Web application. Finally, it discusses how to use the **Application** object to share state between all users and sessions of an application.

- Chapter 6, "Multithreading and Asynchronous Programming in Web Applications," describes how to increase performance and responsiveness of the code in the presentation layer by using multithreading and asynchronous programming.

  This chapter describes how to use .NET Framework thread pools to simplify multithreaded code in your application. In situations where the thread pool is inappropriate, the chapter describes how to manually create, manage, and synchronize threads in your code.

This chapter also describes how and when to use asynchronous method invocation, by using delegates. Delegates represent method calls on objects; with delegates, you can start methods asynchronously by using the **BeginInvoke** and **EndInvoke** delegate methods.

- Chapter 7, "Globalization and Localization," describes how globalization and localization requirements affect the development of your presentation layers.

  This chapter addresses how the .NET Framework uses cultures to define language-specific and country-specific issues, such as number formats and currency symbols; it also describes how and when it might be useful to define alternative cultures programmatically.

  Additionally, this chapter describes how to format various .NET Framework data types appropriately, according to the current culture. As part of this discussion, it describes how to create custom resource files to hold language-specific string resources and images.

- Appendix A, "Securing and Operating the Presentation Layer," describes how security and manageability considerations affect the design of presentation layer components.

  This appendix reviews the importance of authentication and authorization in the presentation layer and the options available for performing these tasks in various circumstances. Additionally, it describes how to improve the security of communications using Secure Sockets Layer (SSL), Internet Protocol Security (IPSec), and Virtual Private Networking (VPN).

  This appendix also describes operational management issues as they pertain to the presentation layer. It describes how to manage exceptions in the presentation layer; health monitoring; and performance monitoring.

- Appendix B, "How To Samples," provides code samples to illustrate the various techniques described throughout this guide.

The next section defines the types of components that the presentation layer contains; it also describes how this guide fits into the broader family of Microsoft *patterns & practices* guides.

Defining the Presentation Layer

For architectural- and design-level guidance about creating layered, distributed applications, see *Application Architecture for .NET: Designing Applications and Services* on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/distapp.asp*). It defines the application architecture illustrated in Figure 1.1.

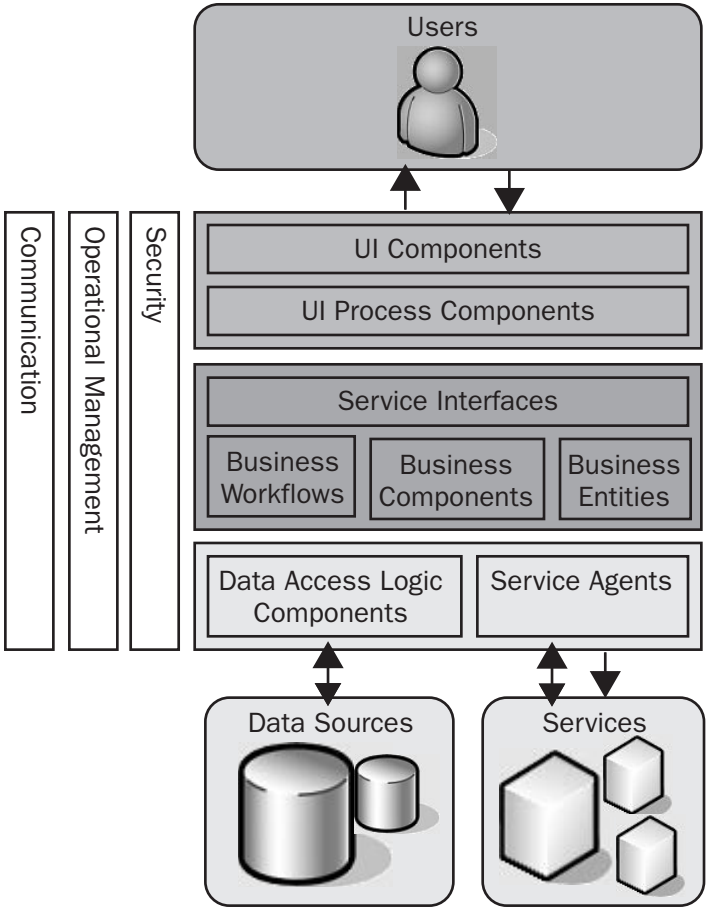


Figure 1.1
Application architecture

The presentation layer includes the following types of software components that perform specific tasks:

- **User interface components**–These components make up the user interface of the application. Users see and interact with these components.
- **User interface process components**–These components orchestrate the user interface elements and control user interaction. Users do not see user interface process components; however, these components perform a vital supportive role to user interface components.

The combination of these two types of components forms the presentation layer of the application. The presentation layer interoperates with the business and data access layers to form the overall solution. The following section outlines the typical responsibility for each kind of component in the presentation layer and explains the benefits for dividing the presentation layer as described.

Defining User Interface Components

User interface components make up the subset of presentation layer components that interact with the user. You can think of it as a layer in itself. They are generally referred to as "views" in presentation design patterns. User interface components are responsible for:

- Acquiring data from the user
- Rendering data to the user

The following characteristics determine other responsibilities for user interface components:

- Validation, input masking, and using appropriate controls for data input
- Managing visual layouts, styles, and the general appearance and navigation of the application
- Encapsulating the affect of globalization and localization
- Formatting data and displaying it in useful visual styles
- Browsing, searching, and organizing displayed data

Common user interface choices include:

- **Console applications**–This approach is suitable for simple utilities that can easily be controlled from a command line.
- **Windows Forms-based applications**–Windows Forms-based user interfaces are the preferred choice for rich client applications that are deployed on desktop, laptop, and tablet PCs.
- **Microsoft Office VBA solutions**–In environments where Microsoft Office applications are used extensively, it can frequently be appropriate to build custom Office-based solutions that allow users to perform business tasks from familiar applications.
- **.NET Compact Framework applications**–The .NET Framework 1.1 includes a subset of classes suitable for building applications for smart devices such as Pocket PCs and Smartphones. This approach can be used to develop rich user interfaces for mobile devices.
- **ASP.NET Web applications**–When an application must be accessible across an intranet or Internet connection, it is a good idea to use ASP.NET to create a Web application hosted in Internet Information Services (IIS). ASP.NET makes it possible to build complex user interfaces that transcend many of the limitations of conventional static HTML and script-based Web solutions.
- **ASP.NET mobile Web applications**–The .NET Framework 1.1 includes ASP.NET-based controls specifically targeted at mobile devices such as Personal Digital Assistants (PDAs) and Wireless Application Protocol (WAP)-enabled mobile phones. These controls are dynamically rendered in a format appropriate to the

device being used to access the application, and therefore make it possible to build Web applications for a broad spectrum of mobile devices.

- **Speech-enabled applications**–The Microsoft .NET Speech SDK makes it possible to build applications that respond to voice input. When combined with Microsoft Speech Server, the .NET Speech SDK makes it possible to build voice-driven telephony solutions with a wide range of applications. Beta 3 of the .NET Speech SDK is currently available. For more information about speech-enabled application development, see the Microsoft Speech Technologies Web site (*http://www.microsoft.com/speech/*).

Regardless of the specific technology used to implement them, user interface components are used to present information to the user and to accept user input, thereby enabling interaction with the business process embodied in the application.

Defining User Interface Process Components

The user interface components described in the preceding section manage data rendering and acquisition with the user, but these responsibilities do not cover the full spectrum of issues that presentation layers must handle.

A user interacts with an application executing use cases. Each use case requires a set of interactions with the user and the business layers to complete. Applications that have use cases involving multiple user interface components, or that have to implement multiple user interfaces, have to decide how to maintain data or state across all user interactions and how the user control flows across multiple user interface components.

User interface process components are responsible for managing interactions between multiple user interactions in a use case. User interface process components are referred to as *application controllers* in design pattern terminology.

User interface process components are responsible for:

- Managing control flow through the user interface components involved in a use case
- Encapsulating how exceptions affect the user process flow
- Separating the conceptual user interaction flow from the implementation or device where it occurs
- Maintaining internal business-related state, usually holding on to one or more business entities that are affected by the user interaction

This means they also manage:

- Accumulating data taken from many UI components to perform a batch update
- Keeping track of the progress of a user in a certain process
- Exposing functionality that user interface components can invoke to receive data they must render to affect the state for the process

To help you separate the tasks performed by user interface process components from the tasks performed by user interface components, follow these guidelines:

- Identify the business process or processes that the user interface process helps to accomplish. Identify how the user sees this as a task.
- Identify the data needed by the business processes. The user process needs to be able to submit this data when necessary.
- Identify additional state you need to maintain throughout the user activity to assist rendering and data capture in the user interface.

The User Interface Process Application Block provides a template implementation for user interface process components. For more information about how to use this block, see Chapter 2, "Using Design Patterns in the Presentation Layer."

Additional Information

For more information about the full range of available *patterns & practices* guides and application blocks, see the *patterns & practices* Web site (*http://www.microsoft.com/resources/practices/*).

The following guides provide guidance and background information about how to implement the other tiers in the recommended application architecture:

- *.NET Data Access Architecture Guide*

  This guide provides guidelines for implementing an ADO.NET-based data access layer in a multi-tiered .NET Framework application. It focuses on a range of common data access tasks and scenarios and presents guidance to help you choose the most appropriate approaches and techniques. For more information about this guide, including download information, see *.NET Data Access Architecture Guide* on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daag.asp*).

- *Designing Data Tier Components and Passing Data Through Tiers*

  This guide covers data access and representation of business data in a distributed application and provides guidance to help you choose the most appropriate way to expose, persist, and pass data through the application tiers. For more information about this guide, including download information, see *Designing Data Tier Components and Passing Data Through Tiers* on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/BOAGag.asp*).

- *Web Service Façade for Legacy Applications*

  This guide defines best practices for interfacing with COM-based applications by using XML Web services created using ASP.NET and the Microsoft .NET Framework. For more information about this guide, including download information, see *Web Service Façade for Legacy Applications* on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/wsfacadelegacyapp.asp*).

Summary

Presentation layer components provide the user interface that users use to interact with the application. Presentation layer components also perform user interface process management to orchestrate those interactions. All applications require a presentation layer of some kind, and when designing a solution, you must consider the architectural issues relating to the presentation layer of your particular application.

The rest of this guide reviews the architectural issues relating to presentation layer development.

2

Using Design Patterns in the Presentation Layer

In This Chapter

This chapter describes how to use presentation layer design patterns to solve common problems in the user interface. It also describes how to use the Microsoft User Interface Process Application Block to implement user interface process components in your application. The chapter includes the following sections:

- Benefits of Using Design Patterns
- Using Design Patterns for the Presentation Layer
- Implementing Design Patterns by Using the User Interface Process Application Block

This chapter describes how to apply design patterns in the presentation layer. The design patterns help you to improve the quality of your implementation code by factoring and organizing components in your presentation layer. Additionally, the design patterns help to increase developer productivity through component reuse and to improve the maintainability of your code. The design patterns also help you identify the components that are affected when making decisions about state management, data access, asynchronous programming, and other areas covered in later chapters in this guide.

Benefits of Using Design Patterns

Most computing problems you will encounter in business applications have already been confronted and solved by someone, somewhere. Design patterns and reusable frameworks based on these solutions help you to overcome the complexity that exists in large applications. The following is a brief summary of the benefits of design patterns and reusable components:

- **Design patterns**–Design patterns provide access to proven methodologies for solving general problems and the ability to use the collective knowledge and experience of the IT community to improve the quality of your own applications. You can use patterns to help you organize code in proven ways to solve well-understood problems.

  There are many proven design patterns that help you solve problems relevant to the presentation layers. Choosing the correct patterns leads to more maintainable code that separates unrelated tasks and functionality. Using these patterns leads to better modularity, higher cohesion, and lower coupling in your application; these are essential characteristics of well-designed systems. The design patterns described in this chapter apply to both Windows Forms-based user interfaces and ASP.NET Web pages.

- **Reusable components**–Reusable components encapsulate functionality that is common across many applications and increase productivity when building your own components following a certain set of design patterns.

  The Microsoft User Interface Process Application Block is a reusable component that helps you build user interfaces based on the Model-View-Controller (MVC) and Application Controller patterns. This block simplifies navigation between related pages or forms in the user interface, and it also lets you take a snapshot of the current state in the application so the user can resume the application at the same stage later. Additionally, the block enables you to get a clean separation between the code that handles user interactions and renders the user interface and the code that performs ancillary tasks; this approach allows you to use the same programming model for Windows Forms applications, Web Forms applications, and mobile applications.

The following section describes how to use design patterns for the presentation layer. Reusable components are covered later in this chapter.

Using Design Patterns for the Presentation Layer

The presentation layer provides a rich source of well-documented and well-understood design patterns. The purpose of design patterns is to:

- Document simple mechanisms that work
- Provide a common vocabulary and taxonomy for developers and architects

- Enable solutions to be described concisely as combinations of patterns
- Enable reuse of architecture, design, and implementation decisions

Appropriate use of patterns reduces the design and development effort required to build your application. Additionally, the adoption of widely used patterns improves maintainability and reduces the risk that an early design decision will have consequences later in the development process or product lifecycle.

Poor design decisions in the presentation layer are particularly expensive and time consuming to resolve. You are most likely to notice poor design decisions when:

- You have to support user interactions of increasing complexity, involving non-trivial relationships between forms and pages in the user interface.
- Existing business processes change and you have to present new or modified functionality to your users.
- You have to port your application to other platforms or make the application accessible to additional client types (such as mobile devices).

By basing your design on frequently used patterns, you can avoid many of the problems associated with these scenarios. The following sections describe the patterns that are applicable in the presentation layer and provide recommendations on when to use each design pattern.

Choosing Design Patterns

Use the following guidelines to help you select and use the appropriate design patterns for your presentation layers:

- Read the following pattern descriptions, and then use Figure 2.1 to understand the reasons for using each design pattern.
- Identify the patterns that address your particular requirements, and then study the design-level and implementation-level descriptions for these patterns.
- Examine the sample implementation for the patterns that are relevant to your requirements. A sample implementation is available for each pattern; it shows how to apply the pattern for .NET Framework applications.
- Evaluate using the User Interface Process Application Block to assist you in implementing the design patterns. For more information, see "Implementing Design Patterns by Using the User Interface Process Application Block" later in this chapter.

The appropriate use of patterns is not always straightforward; patterns provide general-purpose solutions that apply to many different problems. Knowing where to apply a particular pattern can be difficult, especially if the pattern description is particularly abstract or your system requirements documentation is weak.

Ultimately, you will have first-hand experience to help you identify the patterns that are most appropriate for your development team in a particular application scenario and environment. To reduce risk, you are advised to read about as many patterns as you can and experiment with their implementation in test scenarios before you use them on a real project.

Frequently Used Presentation Layer Patterns

The guide *Enterprise Solution Patterns: Using Microsoft .NET* on MSDN (*http://msdn.microsoft.com/practices/type/Patterns/Enterprise/default.asp*) describes how to use patterns in the architecture, design, and implementation of .NET Framework applications. Chapter 3 of the guide focuses on presentation layer patterns and describes patterns that are frequently used in presentation layer design.

The patterns that are most relevant to the presentation layer include:

- Observer
- Page Controller
- Front Controller
- Model-View-Controller (MVC)
- Application Controller

There are some important differences between Web applications and rich-client applications that affect the relevance and suitability of these design patterns for each kind of application. The following section outlines these differences and describes how they influence your choice of design patterns for Web applications and rich-client applications.

Differences in Patterns for Web Applications and Rich-Client Applications

Some patterns that are related to Web presentation seem to be less relevant in the context of rich-client applications because of differences in the programming models exposed to developers, and how state is managed. For example:

- Web applications receive HTTP requests that represent commands from the user. Some of the patterns listed earlier, such as the Front Controller pattern, describe how to direct, interpret, and execute these coarse-grained commands.

  ASP.NET uses, and allows you to implement, the basic patterns for mapping HTTP requests to events and functions in your page components.

  The stateless nature of HTTP, and the fact that a Web application is shared across many users, means you have to think about how and where state is managed across requests. The Application Controller pattern describes how to get state management in Web applications.
- Rich-client applications are generally built on platforms based on in-memory message pumps, such as the message pump built into the Windows operating

system. Messages typically represent low-level user interactions such as button clicks and keyboard entry, and are much finer-grained than Web requests.

.NET Windows Forms encapsulates the underlying message–handling mechanism, and maps Windows messages to 'events' on your form components. It is also possible for business applications to intercept messages and treat them as "business requests," but this approach is uncommon and not recommended.

Rich-client applications are inherently stateful and typically involve interactions with a single user. These two factors frequently mislead designers and developers into treating state management as an afterthought. This can make it difficult to share information between forms and users. The design patterns listed earlier prescribe effective mechanisms for sharing state in rich-client applications.

Driving Factors for Choosing Patterns

Each of the patterns listed earlier in this chapter has its own particular strengths and considerations for specific scenarios.

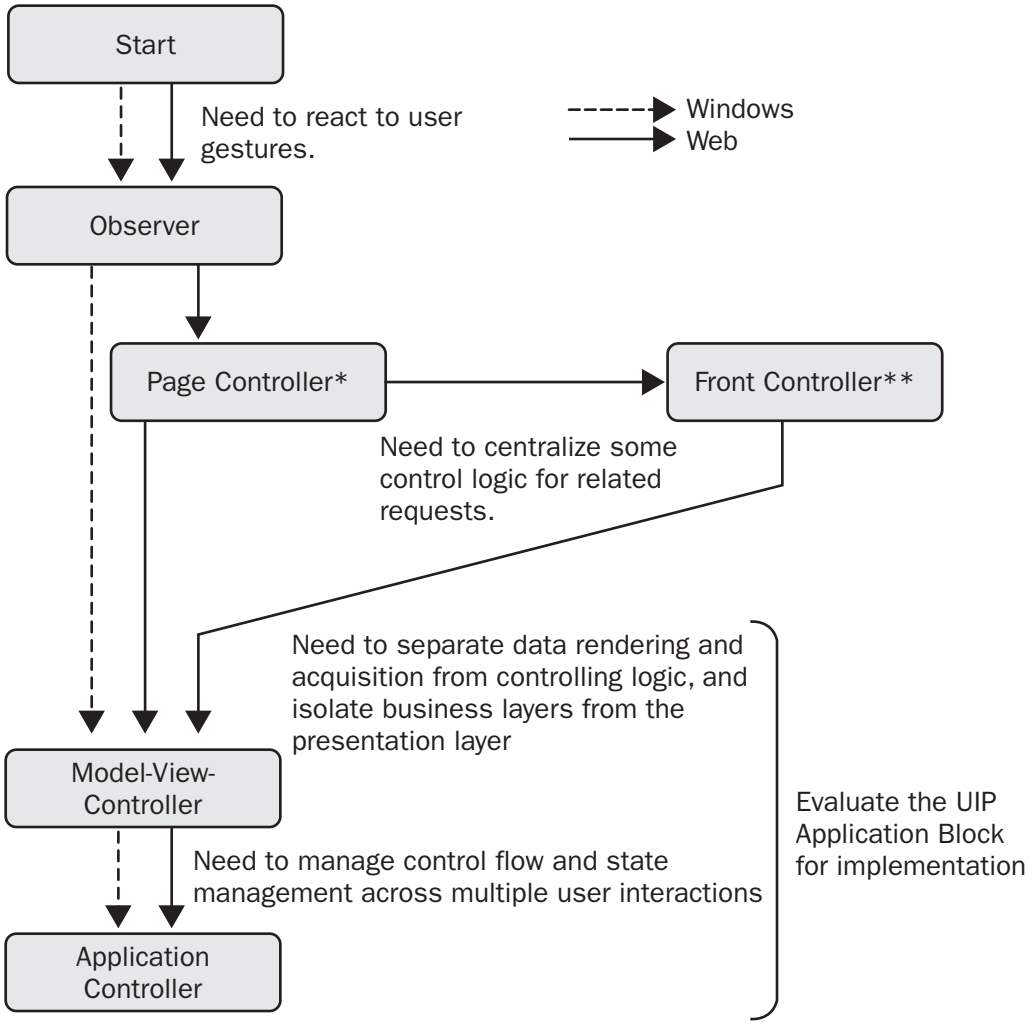
The main factor for choosing among these patterns is to identify how to apportion responsibilities across the various classes and components in the presentation layer. The risk of using a pattern too complicated for the application, team, and environment can be higher complexity and lower productivity.

Use Figure 2.1 on the next page to walk your way through the following descriptions of patterns and determine applicability and effort required.

The following sections describe each of the patterns shown in Figure 2.1. The information for each pattern is organized as follows:

- **Problem**–A brief description of the kind of problem that the pattern solves
- **Solution**–An explanation of how the pattern resolves the stated problem
- **When to use the pattern**–Specific scenarios where the pattern is typically applied
- **Class diagram**–A Unified Modeling Language (UML) class diagram that shows the participating classes and interfaces for the pattern
- **How to use this pattern**–Step-by-step guidance on how to use the pattern
- **References**–Links to further information and samples

This information helps you decide when and how to apply the appropriate design pattern in particular scenarios.



(*) = Encapsulated by the Windows Forms implementation
(**) = Not very relevant in Windows Forms applications

Figure 2.1

Choosing patterns for the presentation layer in .NET Framework applications

Using the Observer Pattern

The Observer pattern provides a mechanism for one object to notify other objects of state changes without being dependent on those other objects.

Problem: You have a simple scenario where you want to react to user actions such as mouse clicks and keyboard entry. In response to these actions, logic executes in your application.

Solution: Use the ASP.NET code-behind class or Windows Forms events to react to user actions and input, and process data accordingly.

When to use the pattern: Use the Observer pattern when you have simple user interfaces where state is not shared across interactions, and where use cases do not drive the flow of control across multiple pages or forms.

Class diagram: Figure 2.2 shows the classes and interfaces that participate in the Observer pattern.

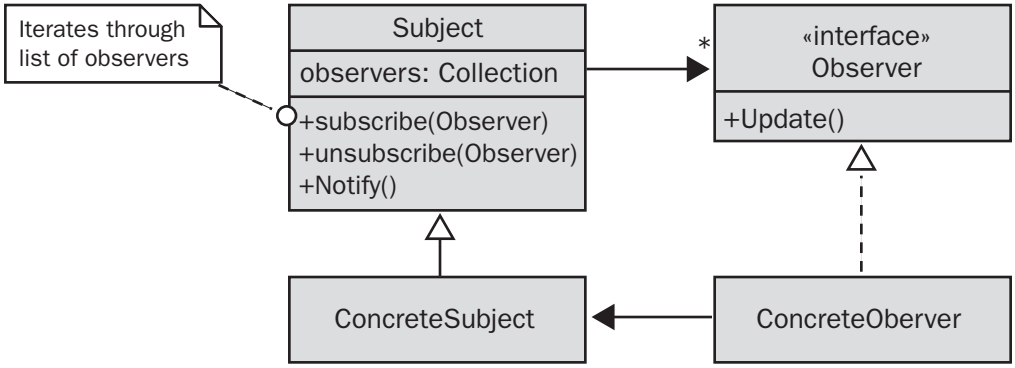


Figure 2.2
The Observer pattern

► **To use the Observer pattern to receive events from user interface components**

1. Decide the user actions or UI events that you have to react to in your application.
2. Evaluate whether you want to use the event handlers provided by ASP.NET pages or Windows Form controls (you typically get these by double-clicking the control in the Microsoft Visual Studio® .NET development system) or add your own handlers.

The following are recommendations for implementing the Observer pattern in the .NET Framework presentation layer code:

- Use Visual Studio .NET support and default handlers as much as possible. For example, to define event handlers in Microsoft Visual C# ® development tool projects, use the Events list in the Properties window in Visual Studio .NET. To define event handlers in Microsoft Visual Basic® .NET development system

projects, use the **Class Name** and **Method Name** list boxes at the top of the code editor window.

- Use standard .NET Framework naming conventions for event handler method names and for any custom delegates and events that you define yourself.
- Do not invoke event handlers from other event handler methods. This quickly leads to unreadable code where it is unclear what triggers what.
- Do not link the default event handlers for specific events to events from other controls or actions. Use your own event handler to make this aggregation clear. For example, do not use the same event handler method to handle both the **Click** and **DoubleClick** events for a **Label** control.
- Use the **+=** syntax to register event handlers in Visual C# code. Use the **Handles controlname.EventName** syntax at the end of event handler method signatures in Visual Basic .NET.

References

- "Observer": *http://msdn.microsoft.com/architecture/patterns/DesObserver/*
- "Implementing the Observer pattern in .NET": *http://msdn.microsoft.com/architecture/patterns/ImpObserverInNET/default.aspx*

Using the Page Controller Pattern

The Page Controller pattern is a variation on the Model-View-Controller (MVC) pattern (described later in this section). The Page Controller pattern is particularly suitable in thin-client applications, where the view and controller are separated; presentation occurs in a client browser, whereas the controller is part of the server-side application.

Problem: You want to separate business layers from the presentation logic in a Web application. You also want to structure the controller components in such a way that you gain reuse and flexibility, while avoiding code duplication between the controller component for each Web page.

Solution: Use a Page Controller component to accept input from the page request, invoke the requested actions on the model, and determine the correct view to use for the resulting page. The Page Controller enables you to separate the dispatching logic from any view-related code.

When you create an ASP.NET Web application using Visual Studio .NET, a Page Controller component is provided automatically for each Web page.

When to use the pattern: Use the Page Controller pattern in Web applications that have simple user interfaces and navigational paths between pages. In these scenarios, the logic in the Page Controller component is simple, and you do not need centralized navigation control or to share state across user interface components.

Class diagram: Figure 2.3 shows the classes that participate in the Page Controller pattern.

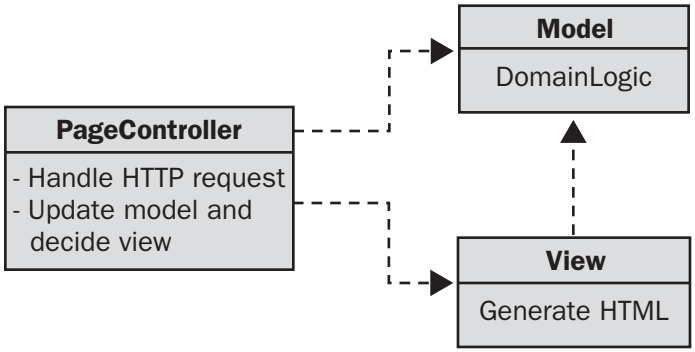


Figure 2.3
The Page Controller pattern

► **To use the Page Controller pattern in your presentation layer**

1. Identify the controller components that you require. This pattern helps mostly with a one-to-one mapping between controller components and Web pages, so typically there will be one controller component per page. Because each Web page is handled by a specific controller, the controllers have to deal with only a limited scope and can remain simple.
2. Evaluate whether you want to write controller code in the ASP.NET code-behind class for the page or in a separate class. The drawback of a separate class is that you will have to access ASP.NET context information by using the **HttpContext.Current** property; this returns an **HttpContext** object that encapsulates all HTTP-specific information about the current HTTP request.

```
// Get HTTP-specific information about the current HTTP request
HttpContext context = HttpContext.Current;
```

The following are recommendations for implementing the Page Controller pattern in ASP.NET applications:

- By default, ASP.NET implements the Page Controller pattern for your application. Every page you create with Visual Studio .NET automatically receives a controller class. This is specified in the **@ Page** tag in the ASPX file.

```
<%@ Page language="c#"
         Codebehind="SimplePage.aspx.cs"
         AutoEventWireup="false"
         Inherits="SimplePage" %>
```

When the user navigates to a URL for an ASP.NET Web page, the Web server analyzes the URL associated with the link and executes the associated ASP.NET page. In effect, the ASP.NET page is the controller for the action taken by the user. The ASP.NET page framework also provides code-behind classes to run controller code. Code-behind classes provide better separation between the controller and the view and also allow you to create a controller base class that incorporates common functionality across all controllers.

- For information about sharing a controller across multiple pages, see Chapter 3, "Building Maintainable Web Interfaces with ASP.NET," in this guide.

References

- "Enterprise Solution Patterns: Page Controller": *http://msdn.microsoft.com/practices/type/Patterns/Enterprise/DesPageController/*
- "Enterprise Solution Patterns: Implementing Page Controller in ASP.NET": *http://msdn.microsoft.com/practices/type/Patterns/Enterprise/ImpPageController/*

Using the Front Controller Pattern

The Front Controller pattern is a variation of the Page Controller pattern. The Page Controller pattern associates a separate controller class with each Web page. It is suitable for simple Web applications where the navigation between Web pages is straightforward. In contrast, the Front Controller pattern is applicable where the page controller classes have complicated logic, are part of a deep inheritance hierarchy, or your application determines the navigation between pages dynamically based on configurable rules.

Problem: You have to enforce consistency on how and where a user's actions and input are handled. You might also have to perform other tasks consistently in several Web pages, such as data retrieval and security checks.

You can reach these goals by using the Page Controller pattern and writing the same code in each code-behind page. However, this approach leads to code duplication in the code-behind classes, and therefore leads to maintenance difficulties in your code.

Alternatively, you can use the Page Controller pattern and create a base class for behavior shared among individual pages. However, this approach can result in complex and inflexible inheritance hierarchies when the number of pages in your application grows.

Solution: Have multiple requests channeled through a single controller class. The controller class provides a centralized location for shared logic and also determines how to transfer control to the appropriate view.

When to use this pattern: Use the Front Controller pattern in Web applications where you want to enforce shared behavior across several pages, or where the flow of control between Web pages is non-trivial.

Class diagram: Figure 2.4 shows the classes and interfaces that participate in the Front Controller pattern.

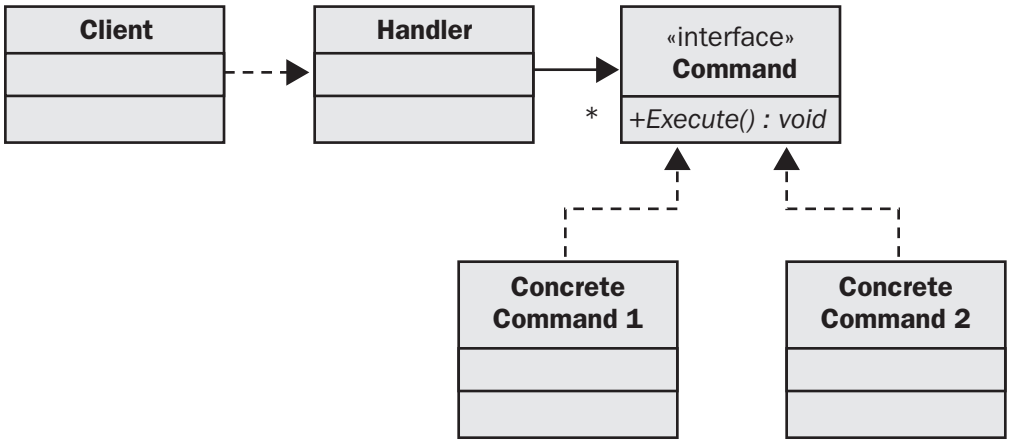


Figure 2.4
The Front Controller pattern

► **To use the Front Controller pattern in Web applications**

1. Identify the commands that you want to handle in a similar fashion across multiple pages. This similarity may be driven by the type of functional work the request invokes, such as updating data in a database or running business components. Alternatively, the similarity might be driven by the policies or aspects associated with the work that will be completed, such as instrumentation, auditing, or authorization.
2. Create a handler class that receives the HTTP Post or Get request from the Web server and retrieves relevant parameters from the request. The handler class uses these parameters to run the appropriate command to handle the request. A common way of implementing this is by using an ASP.NET **HttpHandler**. A code example of how to do this is provided in the "References" section later in this section. The handler should be as efficient as possible and use external resources only when absolutely necessary. You should also consider caching any external resources to increase the handler's performance.

The following are recommendations for implementing the Front Controller pattern in ASP.NET pages:

- Use the Front Controller pattern to enforce similar behavior between Web pages but do not try to force the whole application into a single front controller class because this can lead to performance issues.
- If you use the Front Controller pattern to service user interface requests and render the user interface back to the user, you might have to update the URL

through a redirect. Otherwise, the whole application might end up having a single URL, and hyperlinks would not work.

- Carefully test performance of Front Controller implementations because a front controller adds execution overhead to every request it serves.

References

- "Enterprise Solution Patterns: Front Controller": *http://msdn.microsoft.com/practices/type/Patterns/Enterprise/DesFrontController/*
- "Enterprise Solution Patterns: Implementing Front Controller in ASP.NET using **HttpHandler**": *http://msdn.microsoft.com/practices/type/Patterns/Enterprise/ImpFrontControllerInASP/*

Using the Model-View-Controller (MVC) Pattern

The Model-View-Controller (MVC) pattern adds an intermediary to isolate your user interface logic from your business layers.

Problem: You have to separate business layers from the presentation logic. You also have to separate the logic that controls the state and behavior of the user interface from the logic that renders and acquires data from the user.

Solution: Separate the user interface into controllers and views. Use an intermediary to access appropriate business model elements the views require.

When to use the pattern: Use the Model-View-Controller pattern in applications with complex user interfaces that may display or acquire the same data in different ways in different views. This pattern is also appropriate in applications where strict separation of user interface layers and business layers is required.

Class diagram: Figure 2.5 shows the classes that participate in the MVC pattern.

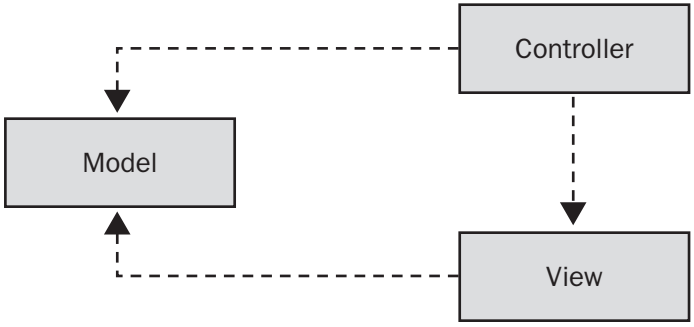


Figure 2.5
The Model-View-Controller(MVC) pattern

► To use the MVC pattern in your presentation layer

1. Identify the components that represent the model. Typically, this is a combination of business entities, business components, data access logic components, and service agents. For information about how to identify the components that apply to your requirements, see Chapter 4, "Managing Data," in this guide.
2. Create the views you have to implement as user interface components, such as Windows Forms, ASP.NET pages, and controls. If you are not using the User Interface Process Application Block, you have to create and hold a reference to the controller object from the view object.
3. Define a controller class and implement methods that the view can call when requesting data or starting an action. These methods will access the model components identified in Step 1.

The following are recommendations for implementing the MVC pattern in Windows Forms and ASP.NET applications:

- Evaluate using the User Interface Process Application Block for your requirements. This block provides a template implementation of the MVC pattern for .NET Framework applications.
- Restrict the view to read-only access to data in the model. If any changes are required in the data, these changes should be performed by the controller instead of by the view. This constraint simplifies application logic by reducing direct coupling between the model and the view.
- Raise model-related events to signify important state changes. This is particularly appropriate in Windows Forms-based applications, where multiple views display data from the same model; the model-related events enable the views to update the data that appears to the user. By raising events in the model, instead of calling methods directly on the views, you preserve the low coupling between the model and the views.
- Implement controllers in a platform-agnostic manner to increase the portability of the controllers regardless of the kinds of views that interact with them. For example, avoid writing ASP.NET-specific or Windows Forms-specific code in the controller class.
- Unit-test the controller classes. Controllers can be easily unit tested, because they expose programmatic functionality instead of providing a user interface.

References

- "Enterprise Solution Patterns: Model-View-Controller": *http://msdn.microsoft.com/practices/type/Patterns/Enterprise/DesMVC/*
- "Enterprise Solution Patterns: Implementing Model-View-Controller in ASP.NET": *http://msdn.microsoft.com/practices/type/Patterns/Enterprise/ImpMVCinASP/*

Using the Application Controller Pattern

The Application Controller pattern manages multiple user interactions by enforcing control flow, providing state management capabilities, and relating views to specific controller classes.

Problem: You have to enforce the flow of control between different views and gain state management across these views.

You can reach these goals by embedding code in the controller and view classes, but this would make it difficult to change navigation and add additional controllers and views in the future.

Solution: Create a separate Application Controller class to control flow through controller logic and views.

When to use the pattern: Use the Application Controller pattern in applications with deterministic flow between views.

Class diagram: Figure 2.6 shows the classes and interfaces that participate in the Application Controller pattern.

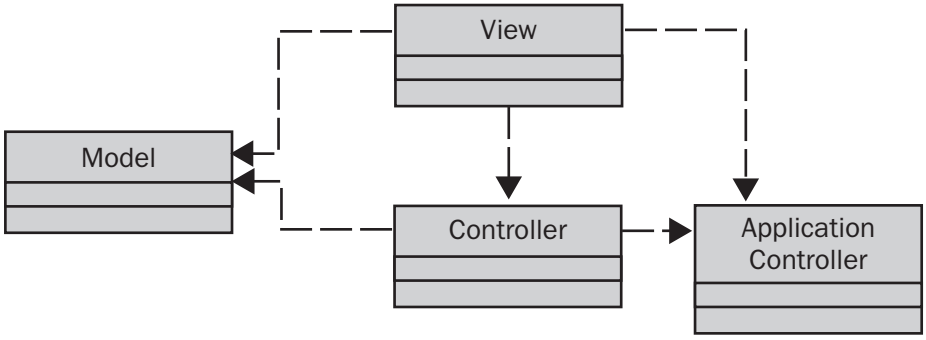


Figure 2.6
The Application Controller pattern

► **To use the Application Controller pattern in your presentation layer**

1. Identify a distinct use case in your application. The use case defines a set of related user activities to fulfil a particular task. Identify the views that participate in the user case and the controllers that are required for the interaction.
2. If you are using the User Interface Process Application Block, configure a navigation graph for each use case; a navigation graph defines the flow between views in the use case. For more information about navigation graphs, see the procedure "To configure the User Interface Process Application Block" later in this chapter.

The following are recommendations for implementing the Application Controller pattern in ASP.NET applications:

- Use the User Interface Process Application Block to implement this pattern in your application. Application Controller functionality is not simple to design or write, especially if you require portability across platforms, if you have to support multiple view types, or if you require state management across the views.
- If you cannot use the User Interface Process Application Block directly in your application, review the documentation and samples that accompany the block; base your design and implementation on the block, where possible.

References

- Application Controller pattern:
  Patterns of Enterprise Applications, Fowler, 2003. ISBN 0321127420
- "User Interface Process Application Block Overview":
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/uip.asp

Implementing Design Patterns by Using the User Interface Process Application Block

After you choose the appropriate design patterns for the presentation layer in your application, the next step is to implement the patterns in code.

The .NET Framework, especially ASP.NET and Windows Forms, provide programming models that implicitly support all the patterns described earlier, except for the Model-View-Controller (MVC) and Application Controller patterns. If you want to use either of these patterns, use the Microsoft User Interface Process Application Block to help you implement these patterns in your application.

The primary goal of most design patterns in the presentation layer is to get a clean separation between the code that renders the user interface and accepts user interactions, and the code that deals with background tasks such as view navigation and state management. Design patterns reinforce the following good practices:

- Do not implement user interface navigation and workflow as part of the user interface views. Otherwise, code becomes complex and difficult to maintain and extend; each piece of user interface code is tightly linked to the steps before and after it, so a small business process change might result in major re-development and re-testing.
- Do not manipulate application and session state directly in the view. Otherwise, code becomes unwieldy because each view is responsible for handing off state to the next step in the application workflow.

To avoid the preceding problems, you should separate your presentation layer into user interface components and user interface process components. User interface components and user interface process components perform distinct tasks:

- User interface components provide the forms and controls that the user interacts with.
- User interface process components synchronize and orchestrate the user interaction with the application, to provide the navigation and workflow required to support the business processes.

For a more detailed discussion of the context of user interface components and user interface process components, see *Application Architecture for .NET: Designing Applications and Services* on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/distapp.asp*).

The User Interface Process Application Block is a flexible, general-purpose, reusable component that you can use in your applications to implement user processes. The block enables you to write complex user interface navigation and workflow processes that you can reuse and that are easy to extend as your application evolves.

This section includes the following topics:

- Design of the User Interface Process Application Block
- Benefits of Using the User Interface Process Application Block
- Building Applications with the User Interface Process Application Block

The User Interface Process Application Block download includes comprehensive documentation on how the block is designed and how to use the block in your applications. The download also includes a sample "shopping cart" ASP.NET Web application that illustrates how to use the block to simplify navigational flow and state management in the presentation layer.

Design of the User Interface Process Application Block

The User Interface Process Application Block helps you build applications based on the Model-View-Controller (MVC) pattern, to get a strict separation between the code that renders the user interface and responds to user interactions, and the code that implements the underlying business logic. The block also implements the Application Controller pattern because it handles control flow and state management between views.

The User Interface Process Application Block has been designed with reusability and extension in mind, to enable you to create controllers that can be used across multiple view types including Windows Forms and ASP.NET applications.

Figure 2.7 shows the major components in the User Interface Process Application Block.

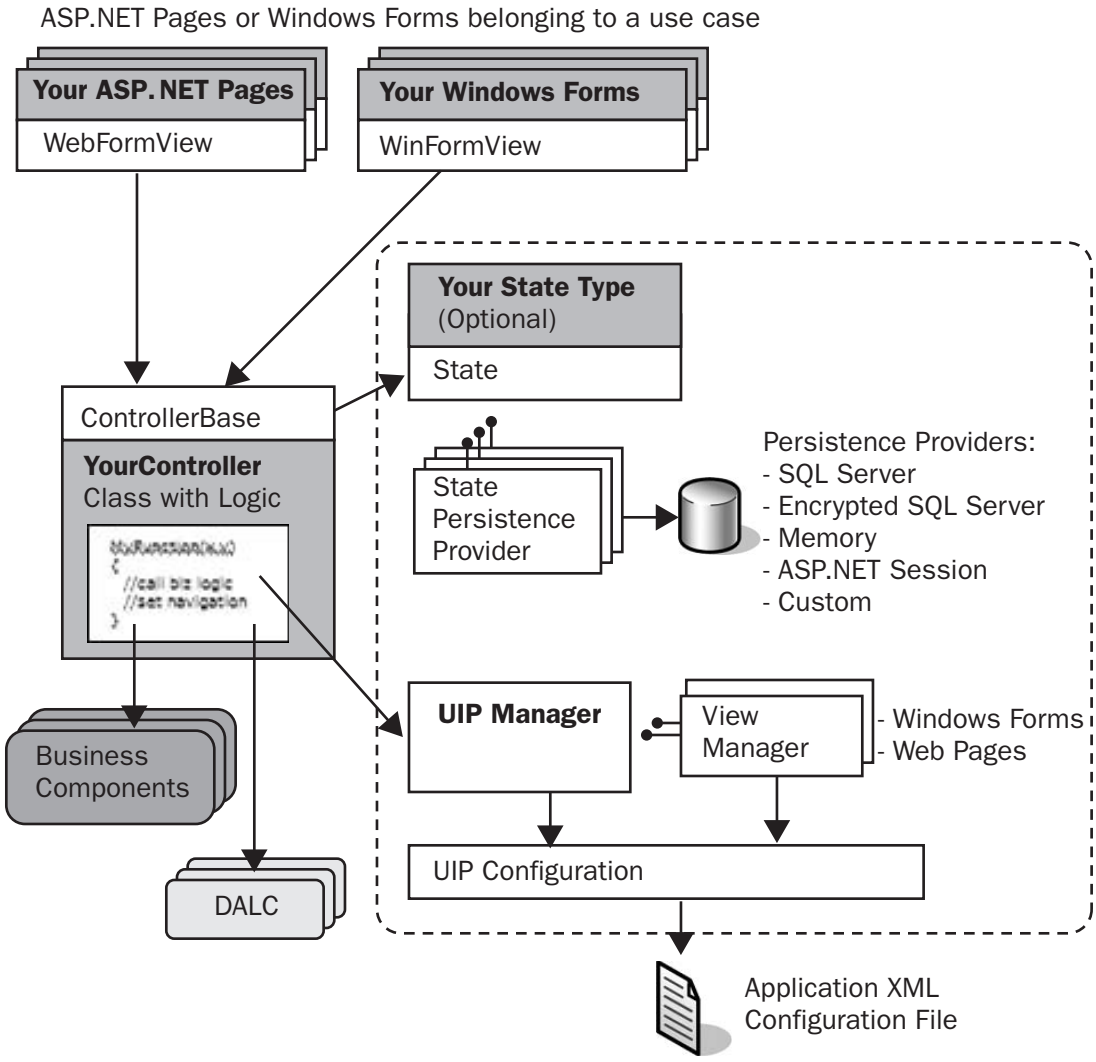


Figure 2.7
Components in the User Interface Process Application Block

The components in the User Interface Process Application Block perform the following tasks:

- **UIPManager**–Manages the user interface process. Dispenses controllers to views, generates new views, coordinates tasks, and provides workflow-based navigation.
- **ControllerBase**–Abstract base class, equivalent to the controller class in the MVC pattern. Its primary purpose is to control the navigation between views, and to act as a façade between the user interface layer and the business layer.

- **State**–Class that represents the state shared across steps in a task. Implements a dictionary-like interface to hold state values. You can inherit from this class to share strong-typed state objects across controllers in a task.
- **SqlServerStatePersistence**, **SecureSqlServerStatePersistence, MemoryStatePersistence**, and **SessionStatePersistence**–These plug-ins manage storing and loading state from multiple places.
- **WinFormView**–Base class for a form in a Windows Forms application. You inherit from this class when developing your Windows forms.
- **WinFormViewManager**–Plug-in that the **UIPManager** uses to transition the user between Windows forms.
- **WebFormView**–Base class for a form in an ASP.NET Web application. You inherit from this class when developing your Web forms.
- **WebFormViewManager**–Plug-in that the **UIPManager** uses to transition the user between Web forms.

The User Interface Process Application Block exposes degrees of freedom in its implementation. There are many ways to extend it, but the common ones are:

- **State type**–You can create strongly-typed state types, by inheriting from the **State** class.
- **State location**–You can store shared state in different places, by implementing the **IStatePersistence** interface. You can write methods to store data held in a **State** object to a durable medium and to retrieve that state when reconstructing a **State** object.
- **Views**–You can assign custom controllers for many component types by implementing the **IView** interface.
- **View transition**–You can handle different styles of visual transition across views by implementing the **IViewManager** interface.

For more information about how to use and customize the block, search for the topics "Customizing State Management" and "Customizing Views and View Management" in the documentation provided with the block.

Benefits of Using the User Interface Process Application Block

The User Interface Process Application Block helps you to:

- Abstract the workflow and navigation from the user interface layer
- Abstract state management from the user interface layer
- Use the same user interface programming model for different types of applications, including Windows-based applications, Web applications, and device applications

- Introduce the concept of a task: a snapshot of an interaction in a use case that can be persisted

Each of these points is described more fully in the following sections. For an example of how to use the block, see "Building Applications with the User Interface Process Application Block" later in this chapter.

Separating Workflow from the User Interface

Workflow is a decision process that results in flowing control, either between Web pages, Windows Forms, or mobile phone screens. These decisions generally depend on the result of tasks performed by back-end systems. For example, some possible actions resulting from trying to authorize a customer's credit card are:

- Ask the customer to use another card because the authorization failed.
- Show the customer the "Thank You" page and ask them if they want to continue to shop.

If a developer writes all the decision-making code and page-redirection logic in the user interface component (for example, in an ASP.NET code-behind class), the developer creates a brittle system. By allowing user interface components to know about each other, they are inextricably linked. In a 10-page application, this is not a huge liability. However, in a 500-page application, this can become a major problem that makes your application difficult to maintain and extend.

The User Interface Process Application Block helps you separate workflow from your user interface component in the following ways:

- The block uses navigation graphs—expressed as XML documents—to define reusable workflow definitions. The navigation graphs and other configuration settings dictate how the workflow proceeds.
- The block delegates all workflow responsibilities to two components: your controller class and the standard **UIPManager** class. The controller and the **UIPManager** cooperate as follows, to control the application workflow:
  - The controller decides the next step in the workflow based on business logic such as whether the customer credit card was authorized satisfactorily.
  - The **UIPManager** uses the results provided by the controller to determine the appropriate view to present the next step in the workflow to the user.

By abstracting workflow in this way, the block enables you to create extremely flexible presentation layers. Flexible presentation layers include views that are independent of each other, business logic that is centralized and reusable, and multiple output modalities that can share most of the same code except the actual user interface elements themselves.

Enhancing Portability

Many applications provide multiple user interfaces to support different client types. For example, an e-commerce application might offer a Web-based user interface to customers and a Windows-based user interfaces to administrative, sales, and support staff. Additionally, as applications evolve, you frequently have to add support for additional client types, such as Personal Digital Assistants (PDAs) and mobile phones.

Although you can make each user interface share common appearance and usage models, their implementation and delivery mechanisms differ significantly. However, despite the user interface differences, the business processes used by the consumers of the application are frequently common across all clients. The solution is to define a user interface process component, as shown in Figure 2.8.

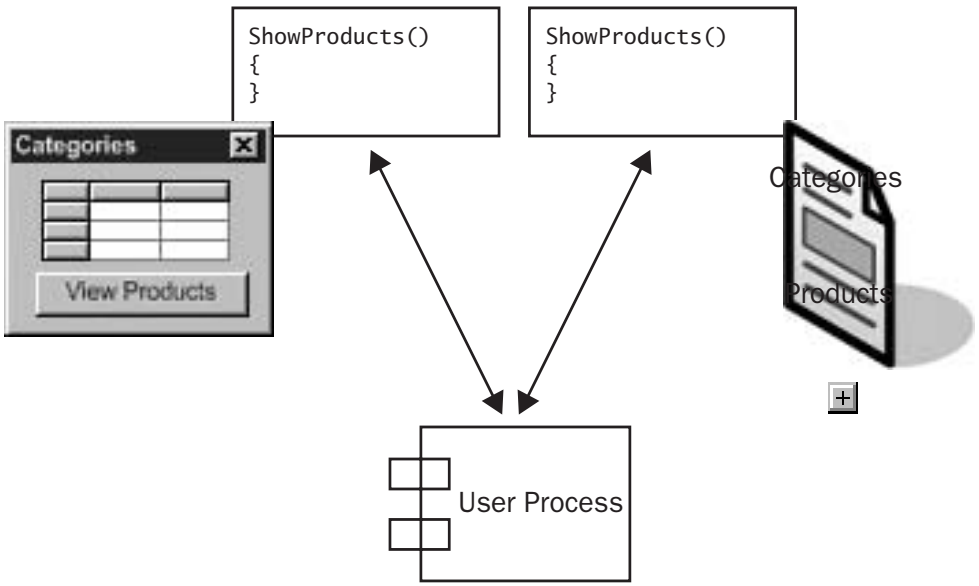


Figure 2.8
Supporting multiple user interfaces

The User Interface Process Application Block helps you build more portable applications in the following ways:

- The block abstracts state management, allowing both Windows-based and Web applications to use the same state management mechanisms. When you migrate an application, you do not have to rework your state management mechanisms.
- The block encourages views to never hold state, except state that might be directly involved in generating the user interface elements themselves.

- The block manages state communication between views by centralizing state in a single entity.
- The block delegates all business-logic and workflow control to the controller and **UIPManager**.

The block encapsulates all issues that are not directly related to rendering the user interface and intercepting user actions; these tasks are performed by the user interface components. If you have to migrate your application to a new platform, for example migrating a Windows Forms-based application to a Web application, all you have to re-implement is the user interface components; the code in the user interface process components can be reused without modification.

Abstracting State Management from the User Interface

In classic applications, state is frequently stored directly in the user interface component. This results in complex code to store, access, and manipulate that state. It also means that the state is dependent on the user interface type; this makes it difficult to transfer code between application types.

The following problems typically occur if you store state in the user interface component:

- In Windows Forms applications, a form might have many member variables that hold state during a user interaction with that form. In most non-trivial scenarios, you have to access the state from other forms in the application. Developers have to write a great deal of extraneous code to capture and transfer that state, using one or all of the following techniques:
  - Externalize the state to a structure or stateful class
  - Use member variables in the business classes underlying the user interface
- In Web applications, passing state between views is especially chaotic. For example:
  - Query strings produce dependencies between views. Both views must know the name of the passed item, and the underlying type of the information.
  - Hidden form fields introduce the same problem as query strings.
  - Cookies experience the same problem as query strings and hidden form fields; additionally, they are unreliable because some users completely deny cookies.
  - The **Session**, **Application**, and **Cache** types in the .NET Framework class library offer better type-preservation, but they also intimately tie the user interface to ASP.NET.

The User Interface Process Application Block simplifies state management by:

- Allowing long-running user interaction state to be easily persisted, so that a user interaction can be abandoned and resumed, possibly even using a different user interface. For example, a customer can add some items to a shopping cart using

the Web-based user interface, and then call a sales representative later to complete the order.

- Centralizing user process state, making it easier to achieve consistency in user interfaces with multiple related elements or windows. Centralized user process state also makes it easier to allow users to perform multiple activities at the same time.
- Abstracting state location and lifetime, to enhance the robustness of the user interface component; views know where to find state, but they do not have to know where state is stored or how it is passed to other views.
- Providing the **IStatePersistence** interface for persisting **State** objects. The flexibility offered by separating persistence behavior from the **State** data means that developers are free to optimize persistence without affecting the consumers of **State** objects.

The preceding benefits illustrate how the User Interface Process Application Block can simplify the development of presentation layers that require user interface process functionality.

Building Applications with the User Interface Process Application Block

There are two steps to building applications with the User Interface Process Application Block:

- Gather requirements for your application. The first step in this task is to separate the application into distinct use cases. For each use case, you must define the workflow and state management requirements.
- Implement views and controller classes as required and write a configuration file to configure the User Interface Process Application Block for use in your application.

Gathering Requirements

The following steps help you identify the requirements for your application, and they help you decide how to use the User Interface Process Application Block to assist you in implementation:

1. Separate the application into use cases.
2. Design the navigation graph for each use case.
3. Verify the existing functionality in the User Interface Process Application Block satisfies your requirements.
4. Determine state persistence for each task.
5. Design the controller classes.
6. Design the views.

The following sections describe each of these steps.

Separate the Application into Use Cases

Applications are typically composed of many use cases. For example, an online retailing application might have use cases to purchase items, add stock to the inventory, and remove low-selling stock from the inventory.

Each use case involves a particular sequence of interactions with the user. Separating use cases correctly gives you better reusability, because your use cases will be encapsulated and you will almost be able to call them as a function.

Design the Navigation Graph for Each Use Case

Decide the control flow that each use case requires and draw a navigational graph that depicts the use case as a visual flow between views.

For example, Figure 2.9 shows a navigation graph for the UIProcessQuickstarts_Store sample application in the User Interface Process Application Block download.

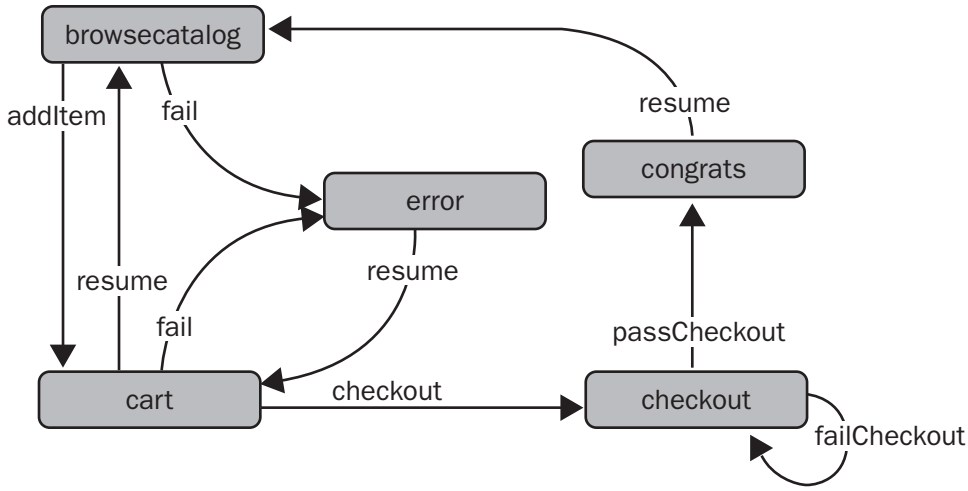


Figure 2.9
Sample navigation graph

The navigation graph in Figure 2.9 identifies logical views in the application, such as "browsecatalog" and "cart." The navigation graph also defines the navigational steps that cause a transition from one view to another. For example, the "addItem" navigational step causes a transition from the "browsecatalog" view to the "cart" view, and the "resume" navigational step causes a transition back again.

Verify that the Existing Functionality in the User Interface Process Application Block Satisfies your Requirements

The User Interface Process Application Block includes common implementations for state persistence, state type, views, and view managers. Review your requirements to evaluate whether the block provides sufficient functionality in the following areas:

- State location (the block provides built-in support for storing state data in SQL Server, secure SQL Server, session, and memory)
- State type (by default, the block uses loose-typed dictionary-like state storage)
- View management (the block provides built-in support for Windows Forms navigation and Web forms navigation)

For more information about how to customize these aspects of behavior in the block, see the User Interface Process Application Block documentation.

Determine State Persistence for Each Task

In the terminology used in the User Interface Process Application Block documentation, a *task* is a running instance of a process. For example, a user can start a task for an "Add Payment Method" process. A task encapsulates all conversation state between the application and the user.

Each use case might have a different lifetime for its tasks. For example, you may want the user to be able to resume a task he or she started earlier in a different Web session or after closing a Windows application. You also have to determine whether the task can be associated with only a single user (for example, a checkout in a retail site) or multiple users (for example, a support call being transferred across members of a helpdesk team).

Table 2.1 describes the attributes of the four state persistence providers available in the User Interface Process Application Block. Use this table to help you decide the state persistence provider that best suits your requirements.

Table 2.1: Attributes of State Persistence Providers

State Provider Provider	Flexibility in assigning a task to different users	Ability for task to span session or application lifetime	Supported for Windows-based applications	Supported for Web applications
SQL Server	✓	✓	✓	✓
Secure SQL Server	✓	✓	✓	✓
Session				✓
Memory			✓	

Design the Controller Classes

The User Interface Process Application Block defines a reusable class named **ControllerBase**; this acts as the base class for your application-specific controller classes.

In your controller classes, you must write properties and methods that interact with the business layers to retrieve data, update data, and perform business-related processing. Declare some of these properties and methods public; this enables views to render data or pass updated data to the controller when necessary. Do not allow views to modify state directly, because this violates the MVC principle of separation between the model and the view.

For example, the UIProcessQuickstarts_Store sample application defines a **StoreController** class that has methods such as **IsUserValid**, **AddToCart**, and **CheckoutOrder**.

Design the Views

The final step is to design the views for your application. Typically, you implement each view as a Windows form or an ASP.NET Web form. Each view is responsible for displaying a particular user interface, retrieving data from a controller and rendering it on the form, and invoking other methods on the controller to instigate business-related processing.

Building Components

After you gather the requirements, implement the views and controllers that make up the use case and configure the User Interface Process Application Block to tie them together.

The first step is to add a reference to the block in your application.

► **To reference the User Interface Process Application Block**

1. Open Visual Studio .NET, and create a Windows-based application or ASP.NET Web application, as appropriate.
2. In Solution Explorer, right-click your project, and then add a reference to the **Microsoft.ApplicationBlocks.UIProcess.dll** assembly. The location of this assembly depends on how you installed it.
   - If you installed the assembly in the global assembly cache, you can reference the assembly from this location.
   - Otherwise, you must reference the assembly from its installation folder, such as C:\Program Files\Microsoft Application Blocks for .NET\User Interface Process\Code\CS\Microsoft.ApplicationBlocks.UIProcess\bin\Debug.

3. In each source file that uses classes from the block, add a **using** statement to reference the **Microsoft.ApplicationBlocks.UIProcess** namespace. All User Interface Process Application Block types are located in this namespace.

```
using Microsoft.ApplicationBlocks.UIProcess;
```

► To create a view

1. If you are building a Windows-based application, view the source code for the Windows form class. In the form class definition, remove the base class **System.Windows.Forms.Form** and replace it with the **WinFormView** base class.

```
public class MyWindowsForm : WinFormView
```

   If you are building an ASP.NET Web application, view the source code for the Web form class. In the form class definition, remove the base class **System.Web.UI.Page** and replace it with **WebFormView**.

```
public class MyWebForm : WebFormView
```

2. Create a private property "getter" to return a reference to the controller object for your application.

   Use the static **Controller** property, defined in the **WinFormView** or **WebFormView** base class, to access the controller object. Cast the controller object into the actual type of the controller class for your application.

```
private StoreController StoreController
{
  get{ return (StoreController)this.Controller; }
}
```

3. Add code to your view class to perform tasks such as retrieving data from the database, saving data to the database, and moving on to the next form in the process. These tasks are implemented by methods in the controller class; to instigate these tasks from the view, invoke the appropriate methods on this controller object.

   The following example invokes a method named **AddToCart** on the controller to add a quantity of a particular product to the user's shopping cart.

```
StoreController.AddToCart(productID, quantity);
```

4. Add code to your view class to retrieve state information from the controller object when required. The controller object has a public **State** property that enables you to get and set named items of state, and it facilitates state management for both Windows-based and Web applications.

The following example gets the value of a state item named **CustomerNameState**.

```
customerNameLbl.Text = StoreController.State["CustomerNameState"].ToString();
```

The controller object encapsulates all implementation details about state management. The code in your view class is unconcerned about where the state comes from or where it is stored.

► To create a controller class

1. Create a new class using Visual Studio .NET.
2. Make this class inherit from **ControllerBase**.

```
public class StoreController : ControllerBase
{
  // ...
}
```

3. Add a constructor that takes a **State** parameter and calls the corresponding constructor in the base class.

```
public class StoreController : ControllerBase
{
    public StoreController( State controllerState )
        : base( controllerState ){}
    //...
}
```

4. Write helper properties in your controller class to help state management.

   In the following example, the **Cart** property gets and sets a state item named **CartState**; it is a hash table containing the user's shopping cart.

```
private Hashtable Cart
{
  get
  {
    Hashtable cart = (Hashtable)State["CartState"];
    if (cart == null)
    {
      cart = new Hashtable();
      State["CartState"] = cart;
    }
    return cart;
  }
  set
  {
    State["CartState"] = value;
  }
}
```

5. Write public methods in your controller class to provide controlled access to state items.

   In the following example, the **AddToCart** method adds a quantity of a particular product to the user's shopping cart.

```
public void AddToCart(int productID, int quantity)
{
  if (Cart.Contains(productID))
  {
    // Product ID is already in cart, so just increment quantity
    int existingQuantity = (int)Cart[productID];
    Cart[productID] = existingQuantity + quantity;
  }
  else
  {
    // Product ID is not in cart, so insert it
    Cart[productID] = quantity;
  }
}
```

6. Write public methods in your controller class to change the current step in the user interface process. To change the current step, set the **NavigateState** property on the **State** object and call the **Navigate** method. The **UIPManager** checks its configuration to determine the view that is expected to show on that outcome, and then it uses the **ViewManager** of the current **NavigationGraph** to transition to that view.

   The following methods show how to stop and resume the current shopping task.

```
public void StopShopping()
{
  // Proceed to next view
  State.NavigateValue = "stop";
  Navigate();
}

public void ResumeShopping()
{
  // Proceed to next view
  State.NavigateValue = "resume";
  Navigate();
}
```

   The following method shows how to complete checkout. The method validates the user's credit card details, and then it proceeds to the "failCheckout" or "successCheckout" navigation step accordingly.

```
public void CompleteCheckout( string name, string addr, string ccnum )
{
  if ( … some credit card validation code … )
```

```
    {
      State.NavigateValue = "failCheckout";
    }
    else
    {
      State.NavigateValue = "successCheckout";
    }
    Navigate();
  }
```

In each of these examples, notice that the controller code does not contain any hard-coded links to specific views. Instead, the association between navigation steps and views is specified declaratively in the navigation graph. Therefore, the controller class has no direct dependencies on the view classes or on the sequence of views in each use case.

► To configure the User Interface Process Application Block

This section describes how to configure the User Interface Process Application Block to define the flow of control between views, and to specify state management information in your application.

For complete information about how to configure the block, search for "Creating the Configuration File" in the documentation provided with the block.

1. Open the configuration file for your application:
   - For ASP.NET Web applications, the configuration file is Web.config in your ASP.NET project.
   - For Windows Forms applications, create a configuration file named ApplicationName.exe.config in the same folder as your application executable file The configuration file must have a root element named **<configuration>**.
2. In the configuration file, define a new configuration section in the **<configSections>** element as follows.

```
<configuration>
  …
  <configSections>
    <section name="uipConfiguration"
             type="Microsoft.ApplicationBlocks.UIProcess.UIPConfigHandler,
                   Microsoft.ApplicationBlocks.UIProcess" />
  </configSections>
  …
</configuration>
```

3. Create a new configuration section named **<uipConfiguration>**. This is used to define the classes to be used by the application for view management, state management, and controllers; the views available to the user; and the navigation graphs that define the workflow path through those views.

The structure of the **<uipConfiguration>** section is defined in the UIPConfigSchema.xsd XML Schema document, which is included in the block. This schema specifies which elements and attributes are required and which are optional, how many times each may occur, and in what order they must appear. If your **<uipConfiguration>** section does not adhere to the schema, an exception will occur at run time.

```
<configuration>
  …
  <uipConfiguration enableStateCache="true">
    <objectTypes>
      … identify the relevant classes in the application …
    </objectTypes>
    <views>
      … identify the views available to the user …
    </views>
    <navigationGraph>
      … define the navigation graphs …
    </navigationGraph>
  </uipConfiguration>
  …
</configuration>
```

4. Specify the details in the **<objectTypes>** element to identify the relevant classes in your application. You must provide the following information for **<objectTypes>**.

```
<objectTypes>

  <iViewManager
    name="name of view manager class"
    type="type of view manager class"/>

  <state
    name="name of state management class"
    type="type of state management class"/>

  <controller
    name="name of controller class"
    type="type of controller class"/>

  <statePersistenceProvider
    name="name of state persistence provider class"
    type="type of state persistence provider class"
    connectionString="connection string to data source for state management"/>

</objectTypes>
```

The following example shows the **<objectTypes>** element in the configuration file for the UIProcessQuickstarts_Store sample application in the User Interface Process Application Block download.

```
<objectTypes>

  <iViewManager
    name="WebFormViewManager"
    type="Microsoft.ApplicationBlocks.UIProcess.WebFormViewManager,
          Microsoft.ApplicationBlocks.UIProcess, Version=1.0.1.0,
          Culture=neutral, PublicKeyToken=null"/>

  <state
    name="State"
    type="Microsoft.ApplicationBlocks.UIProcess.State,
          Microsoft.ApplicationBlocks.UIProcess, Version=1.0.1.0,
          Culture=neutral, PublicKeyToken=null"/>

  <controller
    name="StoreController"
    type="UIProcessQuickstarts_Store.StoreController,
          UIProcessQuickstarts_Store.Common, Version=1.0.1.0,
          Culture=neutral, PublicKeyToken=null" />

  <controller
    name="SurveyController"
    type="UIProcessQuickstarts_Store.SurveyController,
          UIProcessQuickstarts_Store.Common, Version=1.0.1.0,
          Culture=neutral, PublicKeyToken=null" />

  <statePersistenceProvider
    name="SqlServerPersistState"
    type="Microsoft.ApplicationBlocks.UIProcess.SqlServerPersistState,
          Microsoft.ApplicationBlocks.UIProcess, Version=1.0.1.0,
          Culture=neutral, PublicKeyToken=null"
    connectionString="Data Source=localhost;Initial Catalog=UIPState;
                      user id=UIP;password=U1PrOc3ss"/>

</objectTypes>
```

5. Specify the details in the **<views>** element to identify the views available in your application. You must provide the following information for **<views>**.

```
<views>

  <view
    name="logical name of view"
    type="ASP.NET file for view"
    controller="name of controller class for this view"/>

  … plus additional <view> elements, for the other views in the application …

</views>
```

The following example shows the **<views>** element in the configuration file for the UIProcessQuickstarts_Store sample application in the User Interface Process Application Block download.

```
<views>

  <view
    name="cart"
    type="cart.aspx"
    controller="StoreController" />

  <view
    name="browsecatalog"
    type="browsecatalog.aspx"
    controller="StoreController" />

  <view
    name="error"
    type="uipError.aspx"
    controller=»StoreController» />

  <view
    name="congratulations"
    type="congratulations.aspx"
    controller="StoreController" />

  <view
    name="checkout"
    type="checkout.aspx"
    controller="StoreController" />

  <view
    name="survey"
    type="survey.aspx"
    controller="SurveyController" />

</views>
```

6. Specify the details in the **<navigationGraph>** element, as follows:
   - **<navigationGraph>** has attributes that specify the name of the process, the starting view, the **ViewManager** type, the shared state type, and the state persistence provider for this process.
   - **<navigationGraph>** also has **<node>** elements that identify the nodes in the navigation graph.

   The structure of the **<navigationGraph>** is as follows.

```
<navigationGraph
    name="name of this navigation graph"
    startView="logical name of first view in this navigation graph"
    iViewManager="name of the view manager"
```

```
    state="name of the state type"
    statePersist="name of the state persistence mechanism">

  <node view="name of view'>
    <navigateTo navigateValue="navigation step"
                view="view to navigate to, for this value of navigateValue"/>
    … plus additional <navigateTo> elements, for other navigational steps …
  </node>

  … plus additional <node> elements, for other views …

</navigationGraph>
```

The following example shows the navigation graph for the UIProcessQuickstarts_Store sample application provided in the User Interface Process Application Block download.

```
<navigationGraph
    iViewManager="WinFormViewManager"
    name="Shopping"
    state="State"
    statePersist="SqlServerPersistState"
    startView="cart">

  <node view="cart">
    <navigateTo navigateValue="resume"   view="browsecatalog" />
    <navigateTo navigateValue="checkout" view="checkout" />
    <navigateTo navigateValue="fail"     view="error" />
    <navigateTo navigateValue="stop"     view="cart" />
  </node>
  <node view="browsecatalog">
    <navigateTo navigateValue="addItem" view="cart" />
    <navigateTo navigateValue="fail"    view="error" />
  </node>
  <node view="error">
    <navigateTo navigateValue="resume"  view="cart" />
  </node>
  <node view="checkout">
    <navigateTo navigateValue="successCheckout"  view="congratulations" />
    <navigateTo navigateValue="failCheckout"     view="checkout" />
  </node>
  <node view="congratulations">
    <navigateTo navigateValue="resume" view="cart" />
  </node>

</navigationGraph>
```

The navigation graph configuration maps closely to the flow of control identified in the requirements and it clearly exposes the major options for user interface process design.

Summary

This chapter described how to use design patterns to solve common problems in the presentation layer. The .NET Framework provides built-in support for most of these patterns in Windows Forms applications and ASP.NET Web applications. Additionally, the Microsoft User Interface Process Application Block provides support for the Model-View-Controller and Application Controller patterns to enable you to decouple user interface code from user interface process code. You can use this block as a starting point in your own applications to help you implement best practice presentation layer solutions.

3

Building Maintainable Web Interfaces with ASP.NET

In This Chapter

This chapter describes how to create reusable and maintainable Web interfaces with ASP.NET. The chapter includes the following sections:

- Creating New Web Server Controls
- Defining Common Page Layouts

These techniques can simplify ASP.NET Web application development and can help you get consistency across the various pages in your Web application.

Creating New Web Server Controls

ASP.NET provides a wide range of controls that you can use to build your Web applications. However, situations might occur that require you to use specialized controls or to share specific behavior across multiple controls. ASP.NET supports the following two mechanisms for creating new Web server controls:

- Web user controls
- Web custom controls

Web user controls are generally more suitable for static layout, and Web custom controls are more suitable for dynamic layout. The following sections describe how to create and use Web user controls and Web custom controls; they also provide recommendations on when to use each approach.

Creating and Using Web User Controls

This section includes the following topics:

- Overview of Web User Controls
- Usage Scenarios for Web User Controls
- Creating Web User Controls
- Adding Web User Controls to an ASP.NET Web Page
- Defining Shared Code-Behind Files for Web User Controls

This section provides guidance on how and where to use Web user controls and includes code samples to illustrate key points.

Overview of Web User Controls

A Web user control is a collection of related controls and associated code; it provides a reusable unit that you can add to any ASP.NET Web pages that require this functionality.

In implementation terms, Web user controls are similar to ASPX pages except for the following differences:

- Web user controls have an .ascx file name extension instead of .aspx. The .ascx file name extension prevents attempts to execute the Web user control as if it were a standalone ASP.NET Web page.
- Web user controls do not have elements such as **<html>**, **<head>** or **<body>**. These elements are provided by the ASPX Web page that hosts the Web user control.
- Web user controls can have code-behind files, just like regular ASP.NET Web pages. The class in the code-behind file for a Web user control must inherit from **System.Web.UI.UserControl**, whereas ASP.NET Web pages inherit from **System.Web.UI.Page**.

To use a Web user control in an ASP.NET Web page, you must add the .ascx file and its code-behind file to the project. In other words, Web user controls are reused at the source-code level instead of being compiled into reusable assembly files.

When you add a Web user control to an ASP.NET page, the Web user control appears as a placeholder glyph on the page. The constituent controls that make up the Web user control do not appear individually; this means you cannot set their properties directly in the Properties window.

Usage Scenarios for Web User Controls

Web user controls are useful in the following scenarios:

- You have a collection of related controls that you want to use in several ASP.NET Web pages. You can use the rapid control development features of the Visual

Studio .NET Designer to design a Web user control that groups these related controls together.

- You want to partition an ASP.NET Web page into separate sections so you can define different caching strategies for some parts of the page. For example, if part of the Web page requires you to perform time-consuming database operations, you might want to cache that part of the Web page for a relatively long period of time to improve performance. Web user controls give you the ability to partition Web pages in this fashion, by including an **@ OutputCache** directive in the .ascx file as shown in the following example.

```
<%@ OutputCache Duration="120" VaryByParam="None" %>
```

- You do not have to add the Web user control to the Toolbox. Web user controls cannot be added to the Toolbox because they are not compiled into discrete assemblies. Instead you must copy the Web user control into each application that requires it; this can be disadvantageous if the Web user control is used in many applications.
- When developers add a Web user control to their ASP.NET Web page, they do not require design-time access to the properties on the individual controls that make up the Web user control.

If these scenarios do not match your requirements, Web custom controls might be more appropriate. For more information, see "Usage Scenarios for Web Custom Controls" later in this chapter.

Creating Web User Controls

The easiest way to create a Web user control is to use the visual support provided by Visual Studio .NET.

► **To create a Web user control**

1. Open Visual Studio .NET and create an ASP.NET Web application.
2. In Solution Explorer, right-click the project name. On the shortcut menu, point to **Add**, and then click **Add Web User Control**.
3. Drag controls from the Toolbox onto the Designer to create the visual appearance that you want for your Web user control. By default, the Web user control is in flow layout mode; this means controls are arranged from top-to-bottom and from left-to-right. If you want absolute positioning, first add a **Grid Layout Panel** to your Web user control.
4. Use the Properties window to set the design-time properties for the constituent controls in your Web user control.

5. Add code to the code-behind class, as follows:
   - Declare public or protected instance variables corresponding to the constituent controls.
   - Initialize the constituent controls in the **Page_Load** method.
   - Handle events from the constituent controls locally to encapsulate the overall behavior of the Web user control and to maximize its usefulness and reusability by host ASP.NET Web pages.
   - Add public methods and properties to enable external code (such as the host ASP.NET Web page) to configure the appearance of the constituent controls and to get and set their values.

It is also possible to manually create an .ascx file and code-behind files without using Visual Studio .NET.

► **To manually create an .ascx file and code-behind files**

1. Create a file with an .ascx file name extension.
2. Add a **@ Control** directive to the .ascx file, as shown in the following example. For more information about the permitted attributes, see **@ Control directive** in Visual Studio .NET Help.

```
<%@ Control Language="cs" AutoEventWireup="false"
    Codebehind="MyWebUserControl.ascx.cs" Inherits="MyProj.MyWebUserControl"
    TargetSchema="http://schemas.microsoft.com/intellisense/ie5"%>
```

3. Add controls to the .ascx file in the same way that you add controls to an ASP.NET Web page. Do not define **<html>**, **<head>**, **<body>**, or **<form>** elements, or a **<!DOCTYPE>** directive.
4. Create the code-behind file with a file name extension, such as .ascx.cs.
5. Define a code-behind class that inherits from **System.Web.UI.UserControl**. Implement the class in the same way that you implement ASP.NET Web pages.

You can convert an existing ASP.NET Web page into a Web user control so that you can reuse it on other pages.

► **To convert an ASP.NET Web page into a Web user control**

1. Change the file name extension of the ASP.NET Web page file from .aspx to .ascx.
2. Change the file name extension of the ASP.NET code-behind file (for example, .ascx.cs or .ascx.vb).
3. Remove the **<html>**, **<head>**, **<body>**, and **<form>** elements from the .ascx file. Also remove the **<!DOCTYPE>** directive.
4. Change the **@ Page** directive to a **@ Control** directive in the .ascx file. Also change the **Codebehind** attribute to refer to the control's code-behind class file (for example, .ascx.cs or .ascx vb).

5. Remove any attributes that are not supported in the **@ Control** directive. For a full list of attributes supported in each directive, see **@ Page directive** and **@ Control directive** in Visual Studio .NET Help.
6. Modify the code-behind class so that it inherits from **System.Web.UI.UserControl** instead of **System.Web.UI.Page**.

After you create a Web user control, you can add it to ASP.NET Web pages (or to other Web user controls) as described in the next section.

Adding Web User Controls to an ASP.NET Web Page

The easiest way to add a Web user control to an ASP.NET Web page is by using the drag-and-drop capabilities of Visual Studio .NET. In this approach, the Web user control must be implemented in the same language as the host ASP.NET Web page.

► **To add a Web user control to an ASP.NET Web page**

1. Open Visual Studio .NET and create an ASP.NET Web application.
2. Add the .ascx file and code-behind file for the Web user control to the project. The Web user control must be implemented in the same language as the host ASP.NET Web page.
3. Drag the .ascx file from Solution Explorer onto the Designer. The Designer displays a glyph to represent the Web user control, as shown in Figure 3.1. The constituent controls that make up the Web user control do not appear individually, so you cannot edit their properties or visual appearance in the Windows Forms Designer. If you require this capability, consider using Web custom controls instead. For more information, see "Creating and Using Web Custom Controls" later in this chapter.

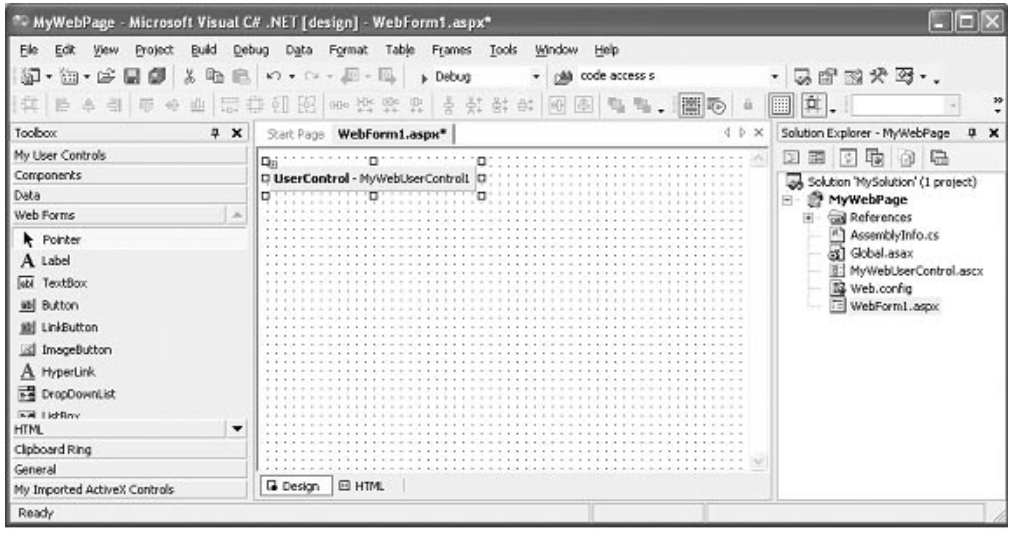


Figure 3.1
Appearance of a Web user control in an ASP.NET Web page

4. View the HTML markup for the ASP.NET page and notice the **@ Register** directive. Visual Studio .NET adds this directive when you drag a Web user control onto the Web page. The **@ Register** directive associates a tag prefix and tag name with the .ascx source file for the Web user control.

```
<%@ Register TagPrefix="uc1"
             TagName="MyWebUserControl"
             Src="MyWebUserControl.ascx" %>
```

   Visual Studio .NET also adds an element to represent the instance of the Web user control on your ASP.NET Web page. The tag prefix and tag name (for example, **uc1:MyWebUserControl**) identify the type of the Web user control. The **id** attribute (for example, **MyWebUserControl1**) identifies a particular instance.

```
<uc1:MyWebUserControl id="MyWebUserControl1" runat="server">
</uc1:MyWebUserControl>
```

5. Add code to the code-behind file for the ASP.NET Web page as follows:
   - Declare a public or protected instance variable corresponding to the Web user control instance. The variable name must be the same as the **id** attribute of the Web user control as shown in the following example.

```
protected MyWebUserControl MyWebUserControl1;
```

   - Write code to interact with the Web user control as required. If the Web user control has public fields for its constituent controls, you can access these controls directly from the ASP.NET Web page. Otherwise, you have to use the methods and properties defined on the Web user control itself.
6. Build the ASP.NET Web project. The compiler generates a single assembly that contains the compiled code-behind classes for the ASP.NET Web page and the Web user control.

It is also possible to manually add a Web user control by writing code in the HTML file and the code-behind files for the ASP.NET. In this approach, the Web user control and the ASP.NET Web page are compiled separately; therefore, they can be implemented in different languages.

► **To manually add a Web user control**

1. Copy the .ascx file and code-behind file for the Web user control into an appropriate folder, so that the files can be accessed by the ASP.NET Web page.
2. Open the .aspx file for the ASP.NET page and add a **@ Register** directive as follows. The **@ Register** directive associates a tag prefix and tag name with the .ascx source file for the Web user control.

```
<%@ Register TagPrefix="uc1"
             TagName="MyWebUserControl"
             Src="MyWebUserControl.ascx" %>
```

3. Add an element to represent the instance of the Web user control on your ASP.NET Web page as shown in the following example.

```
<uc1:MyWebUserControl id="MyWebUserControl1" runat="server">
</uc1:MyWebUserControl>
```

4. Add code to the code-behind file for the ASP.NET Web page as described earlier.
5. Compile the source files for the Web user control and the ASP.NET Web page:
   - If the source files are written in the same language, you can compile them into a single DLL assembly as shown in the following example.

     ```
     csc /t:library WebForm1.aspx.cs MyWebUserControl.ascx.cs ...
     ```

   - If the source files are written in different languages, you must compile them separately using the appropriate compiler. The following example compiles the Web user control into its own DLL, and then it compiles the ASP.NET Web page by referencing that DLL.

     ```
     csc /t:library MyWebUserControl.ascx.cs ...
     vbc /t:library WebForm1.aspx.vb /r:MyWebUserControl.ascx.dll
     ```

To summarize this section on adding Web user controls to an ASP.NET Web page, the easiest approach is to use the drag-and-drop capabilities provided by Visual Studio .NET. The main reason for not using Visual Studio .NET is because you want to use a Web user control that is written in a different language.

Defining Shared Code-Behind Files for Web User Controls

As stated earlier, a Web user control typically comprises two files:

- An .ascx file; this file defines the visual appearance of the Web user control.
- A code-behind file; this file provides the functionality of the Web user control.

This division of responsibility between visual appearance and functionality, coupled with the fact that Web user controls are shipped as source files instead of as compiled assemblies, enables you to create shared code-behind files that can be used by several .ascx files. Figure 3.2 on the next page illustrates this technique.

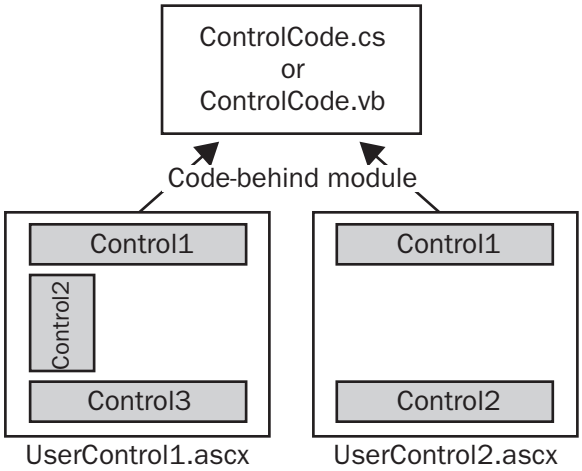


Figure 3.2
Using shared code-behind files for Web user controls

Shared code-behind files are useful if you have to define several different user interfaces for the same Web user control. Each user interface is represented by a separate .ascx file; each .ascx file defines its own set of constituent controls as appropriate. The .ascx files are linked to the same code-behind file to reuse the functionality of the code-behind file.

In the code-behind class, you can define instance variables representing the superset of controls that appear in any .ascx file linked to the code-behind file. The following example shows a sample code-behind class for the scenario depicted in Figure 3.2.

```
public class ControlCode : System.Web.UI.UserControl
{
  protected Control1Type Control1;
  protected Control2Type Control2;
  protected Control3Type Control3;
  ...
}
```

If some of these controls are absent in certain Web user controls, the corresponding instance variable contains a null reference. Therefore, you must look for null references whenever you try to access controls in your code as shown in the following example.

```
public class ControlCode : System.Web.UI.UserControl
{
  ...
  public void MyMethod1()
  {
    if (Control1 != null)
      ...
  }
}
```

If there are wide differences between the constituent controls defined in each .ascx file, the code-behind class can become unwieldy because it has to cope with all the different user interfaces. Shared code-behind files work best when the .ascx files contain broadly similar sets of controls.

Creating and Using Web Custom Controls

The previous section described how to create Web user controls as a way of reusing portions of user interface code in different ASP.NET Web pages. Web custom controls offer an alternative approach to get the same result.

This section includes the following topics:

- Overview of Web Custom Controls
- Usage Scenarios for Web Custom Controls
- Creating Web Custom Controls
- Adding Web Custom Controls to an ASP.NET Web Page
- Defining Alternative User Interfaces for Web Custom Controls

This section describes how Web custom controls differ from Web user controls and provides guidance on how and where to use Web custom controls. Code samples are included to illustrate key points.

Overview of Web Custom Controls

A Web custom control is a compiled component that encapsulates user interface and related functionality into reusable packages. Web custom controls are very different from Web user controls:

- Web user controls are written using the same programming model as ASP.NET Web pages and are ideal for rapid application development. A Web user control is saved as an .ascx file and is reused by adding the .ascx file to each project that requires it.
- Web custom controls are compiled components that rely on object-oriented programming features such as inheritance, polymorphism, and encapsulation. A Web custom control is compiled into an assembly and is reused by referencing the assembly in each project that requires it.

You can add Web custom controls to the Toolbox in Visual Studio .NET to simplify reuse of the control at development time. You can also insert Web custom controls into the global assembly cache to share the control between applications and to simplify accessibility of the control at run time.

Usage Scenarios for Web Custom Controls

Web custom controls are useful in the following scenarios:

- There is an existing ASP.NET Web server control that meets most of your requirements, but you have to add some additional features.
- None of the existing ASP.NET Web server controls meet your requirements. You can create a completely new Web server control in this scenario.
- You have a collection of related controls that you want to use in several ASP.NET Web pages. There are two approaches available:
  - Create a Web user control, as described earlier in this chapter. The benefit of this approach is simplicity at design time, because you can visually drag constituent controls from the Toolbox. The disadvantage is limited configurability when you use the control, because you cannot modify properties directly in the Properties window.
  - Create a composite Web custom control. This entails writing a class that inherits directly or indirectly from **System.Web.UI.Control**, and creates the constituent controls programmatically. The benefit of this approach is flexibility when you use the control, because you can add it to the Toolbox and access properties directly in the Properties window. The disadvantage is complexity at design time, because you must create the control programmatically instead of using ASP.NET-like drag-and-drop techniques.

The following section describes how to create Web custom controls in each of the scenarios described earlier.

Creating Web Custom Controls

You can define Web custom controls to extend the functionality of an existing ASP.NET Web server control.

► **To create a Web custom control**

1. Create a class library project in Visual Studio .NET and add a reference to the **System.Web.dll** assembly.
2. Add a **using** statement (**Imports** in Visual Basic .NET) in your code to import the **System.Web.UI.WebControls** namespace.
3. Define a class that inherits from the ASP.NET Web server control of your choice.
4. Define methods, properties, and events as necessary in the new class.
5. Annotate your class with **[Description]** attributes to provide design-time information about the class and its members. The **[Description]** attribute is defined in the **System.ComponentModel** namespace.

The following example illustrates these points. The example defines a customized text box control that can detect and remove space characters: the **CountSpaces**

property counts the number of spaces in the text, the **RemoveSpaces** method removes all space characters, and the **TextChangedWithSpaces** event is generated (instead of **TextChanged**) if the text changes between postbacks and the new text contains spaces.

```
using System.Web.UI.WebControls;   // For various Web UI classes
using System.ComponentModel;       // For the [Description] attribute

namespace MyCustomControls
{
  [Description("Space-aware TextBox control")]
  public class MyTextBoxControl : TextBox
  {
    [Description("Fires when the text has been changed, and it contains spaces")]
    public event EventHandler TextChangedWithSpaces;

    [Description("Count of spaces in the text")]
    public int CountSpaces
    {
      get
      {
        int count = 0;
        foreach (char c in this.Text)
          if (c == ' ') count++;
        return count;
      }
    }

    [Description("Remove all spaces from the text")]
    public void RemoveSpaces()
    {
      this.Text = this.Text.Replace(" ", "");
    }

    protected override void OnTextChanged(EventArgs e)
    {
      if (this.Text.IndexOf(" ") != -1)
        this.TextChangedWithSpaces(this, e);
      else
        base.OnTextChanged(e);
    }
  }
}
```

If none of the existing Web server controls provide a suitable baseline for your new control, you can create a completely new Web server control that is not based on an existing control.

► To create a completely new Web server control

1. Create a class library project in Visual Studio .NET and add a reference to the **System.Web.dll** assembly.

2. Add a **using** statement (**Imports** in Visual Basic .NET) in your code to import the **System.Web.UI.WebControls** namespace.
3. Define a new class for your control. If your control renders a user interface, inherit from **WebControl**; otherwise, inherit from **Control**.
4. Optionally implement the following interfaces, as necessary:
   - **INamingContainer**–This is a marker interface that makes sure the constituent controls have unique IDs.
   - **IPostBackDataHandler**–This interface indicates that the control must examine form data that is posted back to the server by the client. This interface allows a control to determine whether its state should be altered as a result of the postback, and to raise the appropriate events.
   - **IPostBackEventHandler**–If a control captures client-side postback events, and handles the events or raises server-side events, the control must implement the **IPostbackEventHandler** interface.
5. Define methods, properties, and events as necessary in the new class. For example, if your new control has a user interface, you must write code to render the user interface.

For an example that illustrates these techniques, see "Developing a Simple ASP.NET Server Control" in Visual Studio .NET Help. For more information about how to render a user interface in a Web server control, see "Rendering an ASP.NET Server Control" in Visual Studio .NET Help.

In some situations, you may find it appropriate to develop a composite Web custom control that is an amalgamation of several existing controls.

► **To develop a composite Web custom control**

1. Create a class library project in Visual Studio .NET and add a reference to the **System.Web.dll** assembly.
2. Add a **using** statement (**Imports** in Visual Basic .NET) in your code to import the **System.Web.UI.WebControls** namespace.
3. Define a new class that inherits from **Control**.
4. Implement the **INamingContainer** marker interface to make sure that the constituent controls have unique IDs in the hierarchical tree of controls on a Web page.
5. Override the **CreateChildComponents** method in your class. In this method, create instances of the constituent controls and add them to the **Controls** collection.
6. Define methods, properties, and events as necessary in the new class.

For more information about composite controls, see "Developing a Composite Control" in Visual Studio .NET Help. For an example of how to implement custom controls, see "Composite Server Control Sample" in Visual Studio .NET Help.

Adding Web Custom Controls to an ASP.NET Web Page

This section describes how to use a Web custom control in ASP.NET Web pages.

If you intend to use the control in only a small number of ASP.NET Web pages, you can copy the control's DLL assembly into the subfolder for each ASP.NET Web page. If you intend to use the Web custom control in many ASP.NET Web pages, consider adding the control to the global assembly cache to simplify reuse.

The easiest way to reuse a Web custom control is through the Toolbox.

► **To add a control to the Toolbox**

1. Right-click the Toolbox, and then click **Add Tab**. Enter an appropriate name for the new tab.
2. Right-click in the new tab, and then click **Add/Remove Items**.
3. In the **Customize Toolbox** dialog box, click **Browse**. Locate the DLL assembly for your Web custom control, and then click **OK**. Verify that the control appears in the list of controls in the **Customize Toolbox** dialog box.
4. In the **Customize Toolbox** dialog box, click **OK**. Verify that the control appears in the Toolbox.

When you drag a Web custom control from the Toolbox onto an ASP.NET Web page, the Designer displays the actual visual interface for the Web custom control (in contrast to Web user controls, where a placeholder glyph appears instead).

Figure 3.3 shows a Web page that contains various controls, including the custom text box control described earlier.

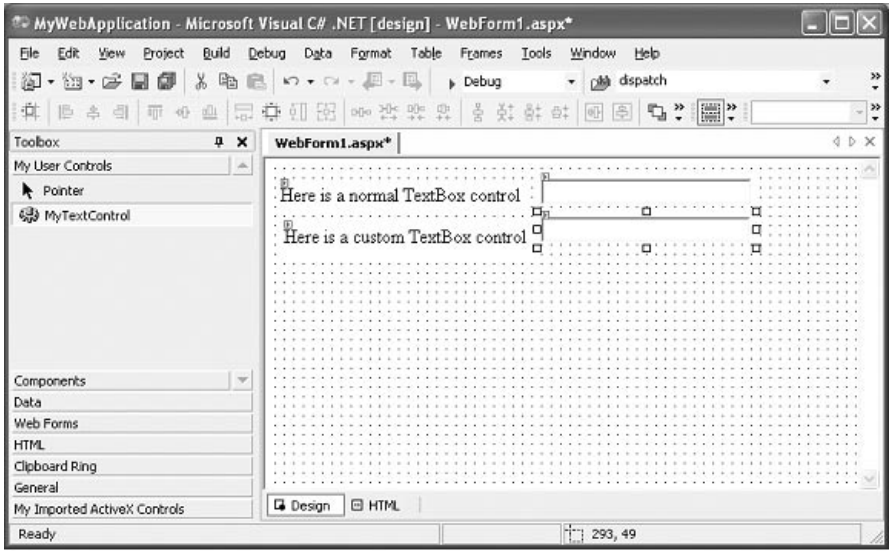


Figure 3.3
Appearance of a Web custom control in an ASP.NET Web page

You have full design-time access to the properties and events of the Web custom control, as shown in Figure 3.4 and Figure 3.5.

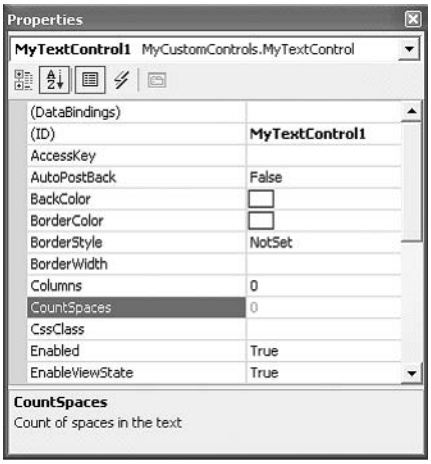


Figure 3.4
Design-time access to the properties of a Web custom control

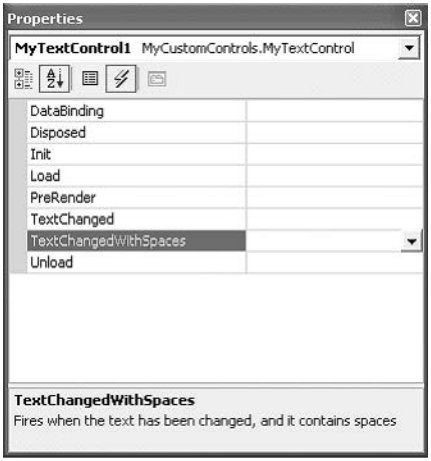


Figure 3.5
Design-time access to the events of a Web custom control

Figures 3.4 and 3.5 illustrate that Web custom controls are easier to use than Web user controls, because you can use them just like built-in ASP.NET Web server controls in the Designer.

Defining Alternative User Interfaces for Web Custom Controls

If you have to provide alternative user interfaces for a Web custom control, you can use a technique known as *skinning* to get this effect.

Skinning is similar to the shared code-behind technique discussed in "Defining Shared Code-Behind Files for Web User Controls" earlier in this chapter. Skinning involves defining a Web custom control that has no user interface elements of its own. At run time, load a Web user control to provide the interface (or skin) for the Web custom control.

The Web user control provides only the user interface; it contains no code. The Web user control must define a set of constituent controls with well known names. The Web custom control uses the well known names to obtain references to the constituent controls, and it wires up its own methods to handle control events or assigns its own instance members to the constituent controls.

With this technique, you can mix and match Web custom controls and Web user controls. Different Web user controls can implement subsets of the constituent controls defined in the Web custom control and can use different layouts. The Web custom control must make sure that a constituent control exists before accessing it.

Defining Common Page Layouts

This section describes how to define a common page layout for pages in a Web application. A consistent look makes it easier for users to navigate and use your Web application, and it can simplify application maintenance. It also increases branding and user recognition; these are important factors if your application provides services that compete directly with those of other companies.

The features of a common page layout typically include elements such as headers, footers, menus, and navigation controls. With ASP 3.0, common HTML or script was included into pages using server-side includes. ASP.NET provides a variety of approaches, including:

- Using a common set of controls
- Using customizable regions
- Using page inheritance

The following sections discuss these approaches, including the benefits and disadvantages of each.

Using a Common Set of Controls

Using a common set of controls provides a simple mechanism for making sure there is a consistent look across the pages in a Web application. You must create the individual controls that make up your common layout and put them in a shared

location. Each ASPX page references the common set of controls in addition to its custom content.

This is an improvement over ASP server-side includes and ensures that all application pages display the same elementary controls. However, there is no concept of a shared layout; you must define the location of each control separately in each page. Figure 3.6 illustrates two ASP.NET pages that implement a common page layout through the disciplined use of a common set of controls.

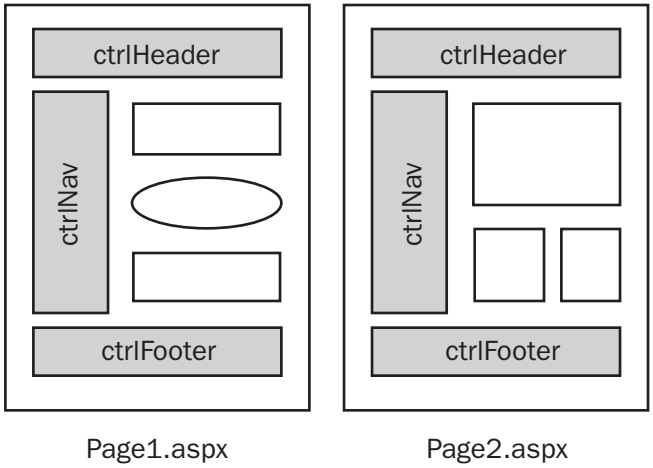


Figure 3.6
Using a common set of controls

The requirement to work with each page individually makes this approach suitable only for small ASP.NET applications that have a manageable number of pages and where the overhead of a more elaborate approach would be unwarranted.

Using Customizable Regions

In this approach, you define the common controls and their layout; you also identify a region in the common layout that holds the custom content for each page. There are two frequently used techniques to implement this model, master page and master Web user control.

This section includes the following topics:

- Using a Master Page
- Using a Master Web User Control

Using a Master Page

The master page approach requires you to create a single page that you return in response to every user request. The master page defines the common page content and layout of your application. Additionally, this master page defines the region that loads the appropriate contents to satisfy a specific user request.

The content specific to each request is typically defined as a Web user control. Your application identifies the appropriate content to load for each user request using a key; this key is specified in the query string or a hidden field of the user request.

The master page approach makes it easy for you to get a common layout for the pages in your Web application; you have to develop only the individual Web user controls for each of the unique pages. Figure 3.7 illustrates a master page (MasterPage.aspx) that displays custom content based on an ID specified in a query string or hidden field.

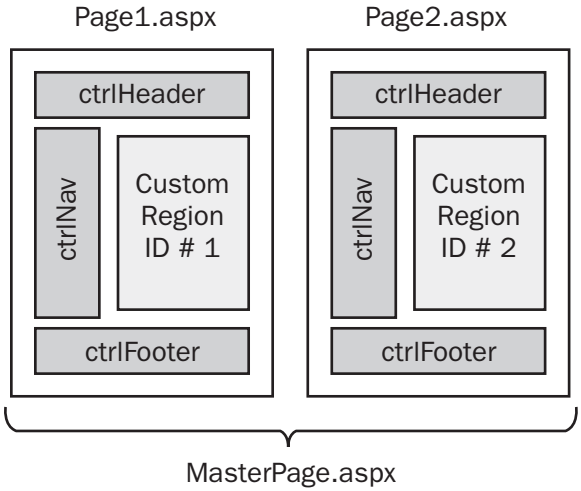


Figure 3.7
Using a master page

The main disadvantage of the master page approach is that you cannot use the built-in authorization mechanism provided by ASP.NET to restrict access to each page, because the client always requests the same page. Therefore, you must implement your own authorization mechanism. Additionally, because the client request contains the content key, it leaves your application susceptible to attack.

Overall, the master page approach works best when you have a common layout for your whole application and there are no restrictions on the content that users are permitted to access.

The IBuySpy ASP.NET example application, accessible at *http://www.asp.net /IBS_Store/*, uses the master page approach and includes an example of how to secure individual pages.

Using a Master Web User Control

The master Web user control approach requires you to create a separate .aspx file for every page in your Web application. You also create a Web user control that defines the common page layout and contains an updateable region, such as a table cell, where the dynamic content is inserted. You add the Web user control to every .aspx file that requires the common layout.

Each ASPX page is responsible for defining the content to display in its custom region. For example, a page can set a property on the master Web user control to refer to another Web user control that contains the custom content for that page.

Figure 3.8 illustrates two pages that contain a master Web user control. On each page, the customizable region of the master Web user control is configured to show a different Web user control.

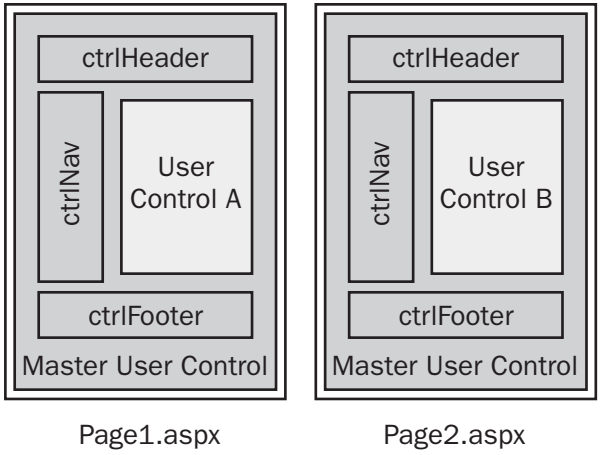


Figure 3.8
Using a master Web user control

The master Web user control approach is best used on small- to medium-sized Web applications, where the cost of developing a more extensive architecture is not warranted. Each Web page corresponds to a separate .aspx file, so you can use the built-in authorization mechanisms of ASP.NET on a page-by-page basis. The main overhead of the master Web user control approach is the requirement to develop the Web user controls that provide the custom content for each page.

Using Page Inheritance

All ASP.NET Web pages compile to classes that inherit from the common base class **System.Web.UI.Page**. You can take advantage of this support for inheritance as a means to implement a common page layout in your ASP.NET application.

► **To use page inheritance**

1. Define a class derived from **System.Web.UI.Page**.
2. Add controls to the page to create the common layout for all pages that will inherit from this page. Also add a container control, such as a **Panel**, where derived classes can generate their own content.

Figure 3.9 illustrates the use of page inheritance to implement a common page layout.

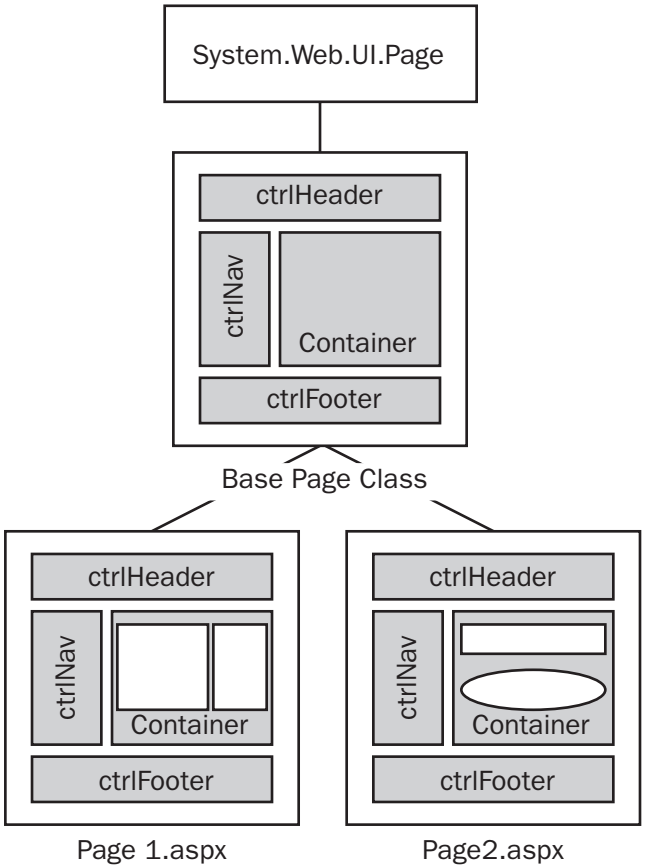


Figure 3.9
Using page inheritance

The main advantage of using page inheritance is that it enables you to implement common behavior and common layout; it also provides a convenient way of defining shared helper routines. Also, because each Web page corresponds to a separate .aspx file, you can use the built-in authorization mechanisms of ASP.NET on a page-by-page basis.

The main disadvantage of using page inheritance is increased complexity compared to the other options discussed in this section.

Summary

This chapter described how to create Web user controls and Web custom controls to encapsulate recurring controls in your ASP.NET Web applications.

The choice between using Web user controls or Web custom controls involves a trade-off between simplicity at development time versus flexibility at usage time. Web user controls are easier to develop because they use familiar ASP.NET-like drag-and-drop techniques, whereas Web custom controls must be created programmatically. In contrast, Web custom controls offer more flexibility at usage time because they provide full access to the properties, events, and methods in the Visual Studio .NET Designer.

This chapter also discussed several techniques for creating common page layouts in a Web application. Master Web pages are appropriate if all the pages in the Web application have the same layout and access restrictions. Master Web user controls are more appropriate if you have different access restrictions for the pages in the Web application. Page inheritance is useful if you want to define shared application logic across the pages in the Web application.

4

Managing Data

In This Chapter

This chapter describes how to manage data in the presentation layer. The chapter includes the following sections:

- Accessing and Representing Data
- Presenting Data Using Formatters, Data Binding, and Paging
- Supporting Data Updates from the Presentation Layer
- Validating Data in the Presentation Layer

Most business applications involve some degree of data processing. Typically, the presentation layer must display data retrieved from a data store, such as a relational database, and accept user input of data. This chapter describes best practices for accessing, presenting, updating, and validating input of data in the presentation layer.

For recommendations specific to data access, see the *.NET Data Access Architecture Guide* on MSDN (*http://msdn.microsoft.com/library/en-us/dnbda/html/daag.asp*).

Accessing and Representing Data

You must take into account a number of considerations when designing the data access functionality of an application. These considerations include:

- Choosing the representation format for data passed between application layers
- Working with transactions in the presentation layers
- Determining which layers should access data

This section addresses each of these considerations.

Choosing the Representation Format for Data Passed Between Application Layers

Data can be represented in a number of formats as it is passed internally between the components and layers of a distributed application. The following formats are generally used to represent data in distributed Microsoft .NET Framework applications:

- Data set
- Typed data set
- Data reader
- XML
- Custom "business entity" objects

It is a good idea to choose the most appropriate data representation format for your requirements, and use it consistently throughout your application. For recommendations about choosing representations for your business data and for passing data across tiers, see *Designing Data Tier Components and Passing Data Through Tiers* on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/boagag.asp*)

There are two main ways to access data from the presentation layer:

- **Using disconnected data**–In this scenario, you pass data such as data sets, custom objects, or XML documents to the presentation layer. These objects and documents may or may not represent data in a database, and they do not imply any connection to a data store.
- **Using streaming data**–In this scenario, you use an object such as a data reader (**SqlDataReader**, **OracleDataReader**, or **OleDbDataReader**) to access streaming data in a data store, typically in read-only, forward-only manner.

The following sections describe how to use disconnected data and streaming data in the presentation layer.

Using Disconnected Data

The term "disconnected data" refers to data that you retrieve from a database and then you close the database connection or leave the scope of the current transaction. After you close the database connection or exit the transaction, you continue to use the data even though you have no current connection to the database.

It is a good idea to use disconnected data structures in your user interface layers in any scenario where you have to handle, populate, or modify data outside the scope of a database access or transaction, or if you cannot receive data from a direct connection to the database from the physical tier that contains the presentation layers.

When using disconnected data, you have to:

- Choose the type of object you will use to represent the data. Possible representation formats include XML, data sets, and custom "business entity" objects.
- Determine how you will manage concurrency. For example, you must create a policy for dealing with overlapping updates to data in the data store by different users.
- Decide how you will retain access to the data, especially in Web scenarios. Possible solutions include the use of session state to store per-user data, or application state to store per-application data.

These issues are addressed in the guide described earlier, *Designing Data Tier Components and Passing Data Through Tiers*. The rest of this section discusses these issues from the perspective of the presentation layers.

Using Streaming Data

The term "streaming data" refers to data that you obtain through a data reader. You use the data reader to pull information from the database in a read-only, forward-only manner while a connection is kept open.

Use streaming data from the user interface in the following scenarios:

- You are consuming data in a read-only, forward only manner.
- You require access to the data in a physical tier that can access the database directly.
- You are outside the scope of a transaction.

Streaming data is typically more appropriate in Web applications than in smart-client applications, because Web applications are more likely to have access to the database server. A typical scenario is for a Web application to use a data reader to populate controls with read-only data from the data store.

Note: Data readers implement the **System.Collections.IEnumerable** interface; a subset of .NET Framework user interface controls can use this for data binding. In contrast, data sets implement **System.Collections.IList**; this allows data sets to be data bound to a wider range of user interface controls. For more information, see "Presenting Data Using Formatters, Data Binding, and Paging" later in this section.

Working with Transactions in the Presentation Layer

This section describes how to manage transactions in the presentation layer. There are two distinctly different kinds of transactions:

- **Atomic transactions**–Atomic transactions are intended to encapsulate small, brief units of work that occupy few resources and complete quickly. A typical example is the transfer of money; this locks the accounts being updated for a

short time while they are updated, and then releases the locks after the updates are complete.

Atomic transactions have just two potential outcomes—success or failure—and provide ACID (atomicity, consistency, isolation, and durability) guarantees.

- **Business transactions**–Business transactions encapsulate operations that can last several minutes, hours, days, or even weeks. An example is the exchange of documents between two businesses that have to follow a particular protocol. The documents may require manual processing or authorization; these operations could take a lot more time than atomic transactions. In these circumstances, it is not advisable to keep resources locked for the duration of the transaction, so an alternative strategy has to be used instead.

  Business transactions may have many potential outcomes, including compensation activities to handle various transaction failure scenarios.

In distributed systems (unlike client-server applications), the presentation layer should not initiate, participate in, or vote on atomic transactions for the following reasons:

- Atomic transactions typically represent a business operation that should be handled by a business component. The business component should be isolated from how data is displayed to the user.
- If you initiate an atomic transaction in the presentation layers, the physical computer where the presentation layers are deployed becomes part of the transaction. This means that an extra node and set of communication links are required to coordinate the transaction; this adds failure points and may add security risks because of the extra channels involved.
- If you initiate or vote on atomic transactions in the presentation layers, you risk exposing a situation that requires user interaction between the transaction initiation and its completion. During this time span, all resource managers participating in the transaction have to keep locks to provide ACID guarantees; scalability is drastically reduced because other activities have contention on these locks.

To prevent presentation layers from initiating, participating in, or voting on atomic transactions, follow these guidelines:

- Do not use the **Transaction** attribute on ASP.NET pages.
- If your controller classes are hosted in COM+ applications, they must not have the **Supported**, **Required**, or **Requires New**, transactional attributes.

The implication for distributed systems of not initiating transactions in the presentation layers is that all data in the presentation layers exists outside the scope of transactions; this implies the data might be stale. For information about mechanisms for managing data staleness and optimistic concurrency, see the guide described earlier, *Designing Data Tier Components and Passing Data Through Tiers*.

Determining Which Layers Should Access Data

When designing a layered application, you sometimes have to choose between "strict layering" and "loose layering":

- Strict layering means a component only interacts with components in the same layer, or with components in the layer directly beneath it.
- Loose layering means a component is allowed to interact with components in any layer, not just those in the layer directly beneath it.

The choice between strict layering and loose layering arises because of potential tradeoffs between the maintainability that strict layering provides by letting you change and extend the behavior in the future, and the productivity gain that you can get by letting a layer access layers other than the one directly beneath it.

For data access, the data source itself should be accessed only by data access logic components or data access helper components in the data access layer. The main design decision is whether to allow the presentation layer to access the data access layer directly or force all data access requests to pass through the business layer. There are three different approaches to consider:

- Using message-based business processes
- Invoking business objects
- Invoking data access logic components directly from the presentation layer

Note: This section describes the relationship between the logical user interface layers and other logical layers. It does not describe how to distribute these layers in a multi-tiered environment.

Using Message-based Business Processes

In the message-based approach to data access, data is accessed by exchanging messages between the user interface process components in the presentation layer and business workflows in the business layer. The reliance on purely message-based communication means that it is the easiest way for some applications to convert smart-client user interfaces to offline mode.

The processes-based approach is shown in Figure 4.1 on the next page. In the illustration, the solid lines represent requests for data and the dashed lines represent the returned data.

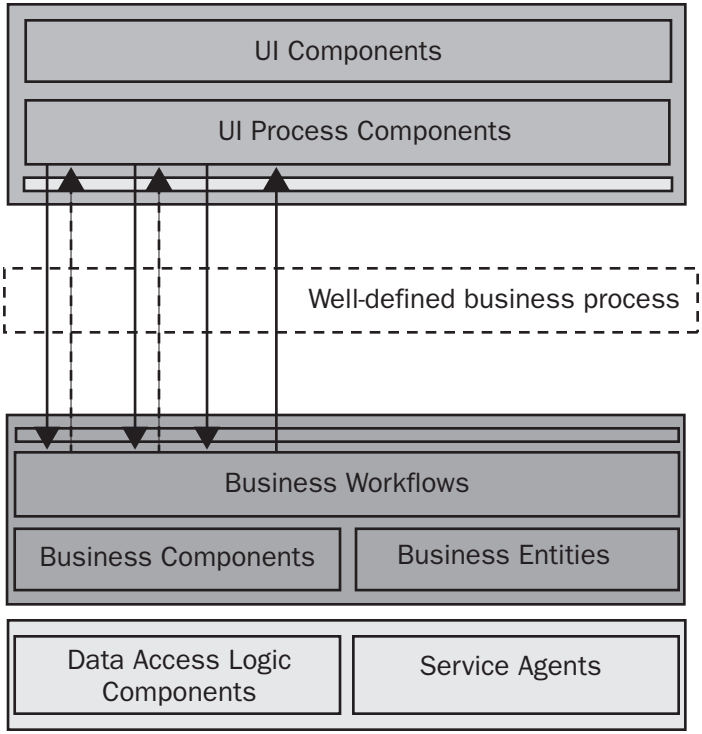


Figure 4.1
Using message-based business processes for data access

This approach is not straightforward and relies on a careful analysis of your requirements.

► **To access data by using message-based business processes**

1. Analyze your application use cases to determine the data flow back and forth.
2. Define or use an existing business process as an exchange of messages.
3. Design coarse-grained messages or Web services for each data flow. Coarse-grained communication increases efficiency and establishes a document-based information exchange that can be reused by other clients (not necessarily user interfaces) using the same business process.
4. Write code in your user interface process layer controllers to invoke the service interfaces that access the business workflow or business components.

This approach may be cumbersome if the presentation and business layers are built together as part of the same application. Also, if the business process was not originally designed to be consumed from a user interface, it might rely on messages being sent to your presentation layers. For example, your user interface might have to react to incoming messages or Web service calls. Such notification architecture for

your presentation layer necessitates additional infrastructure and code and is outside the scope of this guide. You can build notification as part of your application, or rely on external infrastructures, such as e-mail, MSN® Messenger, or Windows Messenger.

Invoking Business Objects

The invocation of business objects is probably the most used approach when business logic exposed by an application is designed to service the user interface. For example, the presentation layer can invoke business components through .NET Framework remoting or Web services to retrieve data, and then invoke other business components. This approach does not rely on business workflows to control the process.

Figure 4.2 shows how to invoke business objects from the presentation layer.

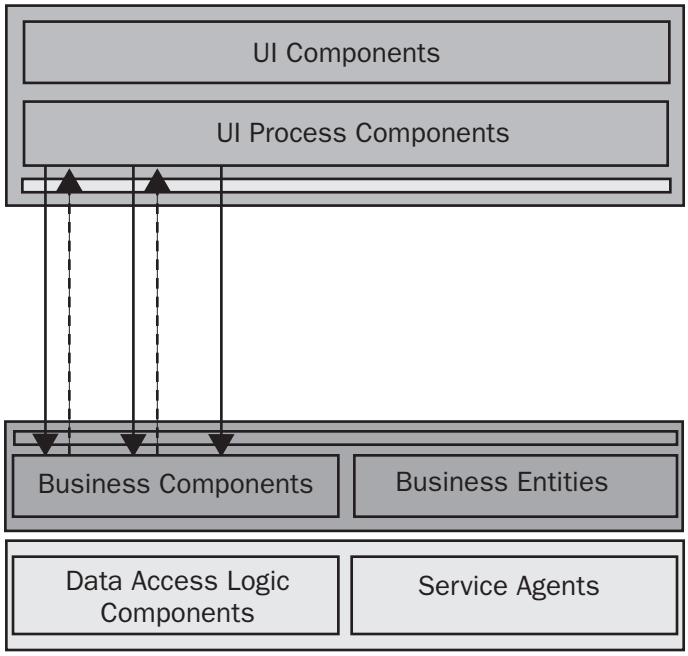


Figure 4.2
Invoking business objects from the presentation layer

► **To access data by invoking business objects**

1. Design methods in your business components that acquire the reference data the user interface requires and return it to the presentation layer. Write coarse-grained methods that return a set of data all at the same time, such as a whole **Order** object or a data set. The use of course-grained methods increases communication efficiency. Optimistic concurrency management is also simplified, because related data is acquired in one call and is therefore easier to time stamp.

2. Design methods in your business components that take coarse grained sets of data, encapsulate transactions, and update data.

This approach is appropriate for most applications and enforces strict layering in the application. However, if the methods exposed by the business components are just wrappers for methods provided by data access logic components and provide no additional business logic, you might prefer to bypass the business layer and access the data access logic components directly from the presentation layer.

Invoking Data Access Logic Components Directly from the Presentation Layer

It is a good idea to allow your presentation layer components to directly access the data access layer when:

- You do not mind tightly coupling your data access semantics with your presentation semantics. This coupling involves joint maintenance of presentation layer changes and data schema changes. Evaluate this option if you have not encapsulated all data access and entity-specific business logic into business entity components.
- Your physical deployment places the data access layer and presentation layer components together; in this scenario you can retrieve data in streaming formats (such as a data reader) from data access logic components, and bind the data directly to user interface elements for performance. If your presentation layer components and data access logic components are deployed on different servers, this functionality is not available.

There are two variations of this approach:

- Invoke data access logic components from user interface process "controller" components.
- Invoke data access logic components from user interface components.

The following sections describe how and when to use each technique.

Invoking Data Access Logic Components from User Interface Process Controllers

If you do not require the maintainability and growth capabilities afforded by strict layering, you might decide to acquire data, and maybe even update it directly, using the data access logic layer from the controller functions in the user interface process layer. This is a useful approach in small data-intensive applications that have predictable areas of growth.

The user interface process layer uses data access logic component methods to read reference data that is to be displayed to the user, and it uses the layering style described in the previous section for operations that might require additional business transaction logic at some future point. Figure 4.3 shows this approach.

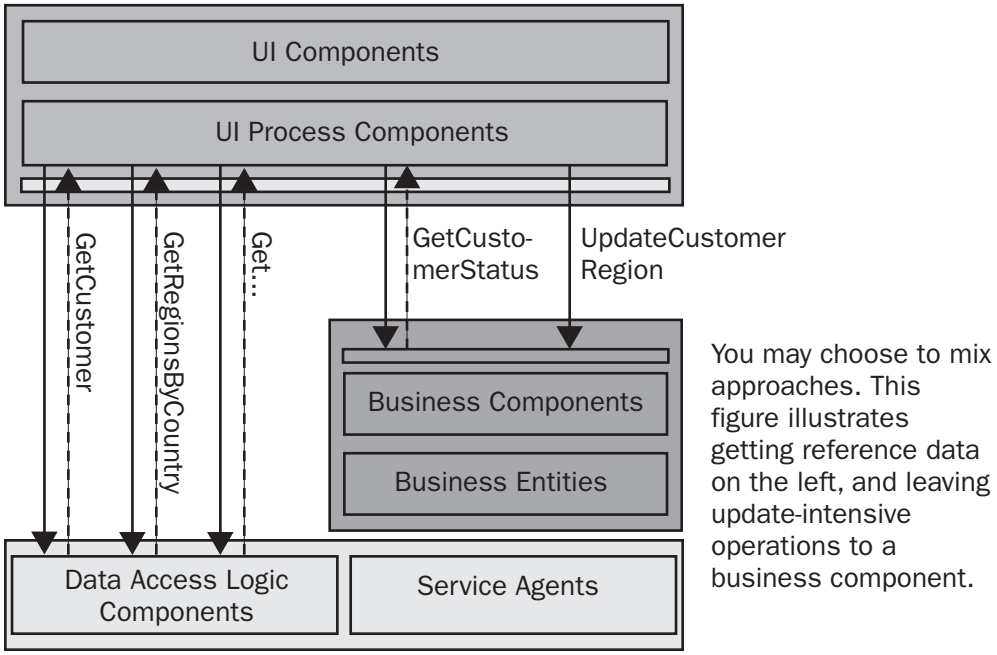


Figure 4.3
Invoking data access logic components from the user interface process layer

► **To invoke data access logic components from the user interface process layer**

1. Decide the kinds of operations that will be permitted from the user interface process layer to the data access layer. For example, you might choose to allow only operations that read data, or you might also allow operations that update the data.
2. Choose which methods or use cases will have direct access to the data access layer. It is a good idea to choose read methods that are unlikely to require data aggregation or some other type of logic in the future, and write methods that are unlikely to grow into business transactions with more implications than a data write.

Consider the following issues when deciding whether to allow the user interface process layer to directly invoke the data access layer:

- You have to expose the data access logic to the user interface process layer. In smart client scenarios, or Web scenarios with remote application tiers, this means configuring .NET Framework remoting or Web services to enable remote access from the user interface process components. Consider the security and maintainability implications of exposing such fine-grained operations. Keep in mind that it may not add complexity to make remote calls from the user interface process layer with controller functions if business components already require remote access.

- Business components functions that return data typically do so in disconnected formats (such as a data set), whereas data access logic component frequently return streaming data (through a data reader). If you originally use data access logic components to get data as a data reader, and then you have to upgrade your logic to use a business component instead, you will have to change your presentation layer code to use data sets (or another disconnected format). To avoid costly code rewrites, plan ahead and return disconnected data for complex queries that might evolve into more complex operations.

Invoking Data Access Logic Components from User Interface Components

It is a good idea to access the data access layer directly from the user interface components (forms, pages, controls) only in specific cases where:

- You have to encapsulate logic for accessing the data in the user interface component.
- The user interface component requires read-only access to reference data, and the use case embodied in the user interface process is agnostic to the data.
- The data, or its source, does not vary depending on the use case.
- Bypassing the controller functions does not negatively affect maintainability in the long term.

For example, you can develop a custom **Countries** list box control that knows how to populate itself with country data from a data access logic component. The benefit of this approach is that the developer building the user interface does not have to know how to retrieve or load the data. The drawback is the tight coupling that this introduces between the user interface controls and data design and the affect it may have on scalability if multiple controls get their data independently from a remote tier. It is not recommended to use this approach for update operations because this would be equivalent of allowing views to update model data in the Model-View-Controller (MVC) pattern.

Figure 4.4 shows user interface components directly using data access logic components.

A slightly different design with a potential for better maintainability and scalability is to encapsulate functions in the data-intensive controls that are specifically intended to be controller function helpers, so that the user interface process controller methods can invoke them when appropriate. For example, the **Countries** list box control can have a method that invokes a data access logic component and places the data in the state of the current user interface process, the control, or both. This way the developer has more control over what data is displayed and when the queries are performed.

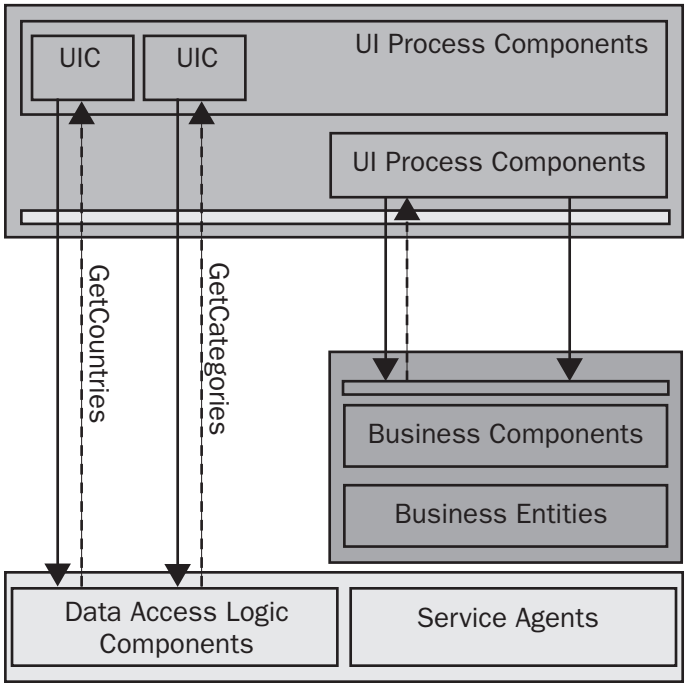


Figure 4.4
Invoking data access logic components from user interface components

Presenting Data Using Formatters, Data Binding, and Paging

One of the main functions of the presentation layer is to present data to the user. There are a number of architectural considerations for data presentation that you must take into account. They are:

- What format should you use to display the data the user?
- How should you bind user interface controls to data in the data source?
- What pagination strategy should you use if there is a large amount of data to display to the user?

This section addresses each of these issues.

Formatting Data

You frequently have to format data for display. For example, you might want to display the database value "1.2300000" as "1.23." The .NET Framework provides several format specifiers that you can use to format strings in your application. Other basic types must be converted to strings using their **ToString** method before formatting can be applied. The format specifiers include:

- **Numeric**–The .NET Framework provides many standard numeric format strings, such as currency, scientific notation, and hexadecimal, for formatting numbers. For a complete list of the numeric format strings, see "Standard Numeric Format Strings" in the *.NET Framework Developer's Guide* on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconstandardnumericformatstrings.asp*).
- **Date and Time**–When displaying date and time information to a user, you frequently want to display a simpler representation than the complete contents of the **DateTime** data type. For a complete list of the date and time format strings, see "Standard DateTime Format Strings" in the *.NET Framework Developer's Guide* on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconstandarddatetimeformatstrings.asp*).
- **Enumeration**–When you have an enumeration, you can use **ToString** to create a numeric, hexadecimal, or string representation of the enumeration value. For more information, see "Enumeration Format Strings" in the *.NET Framework Developer's Guide* on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconenumerationformatstrings.asp*).
- **Custom**–If none of the built-in format strings fully meet the formatting functionality your application requires, you can create a base type that accepts a custom format string or create a format provider class to provide formatting for an existing type. For more information, see "Customizing Format Strings" in the *.NET Framework Developer's Guide* on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcustomizingformatstrings.asp*).

  For an example of how to define a custom formatter to format business entity objects, see "How to: Define a Formatter for Business Entity Objects" in Appendix B of this guide.

With each of the format specifiers, you can supply a culture to localize the string format. For more information, see "Formatting for Different Cultures" in the *.NET Framework Developer's Guide* on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconformattingfordifferentcultures.asp*).

Data Binding

Web Forms and Windows Forms allow you to display information by binding controls to a source of data.

However, because of the nature of Web Forms and the architecture of Web programming, there are some significant differences between data binding in Web Forms and Windows Forms. The most significant difference is that the data-binding architecture of Web Forms does not perform updates—that is, it does not write data from the control back to the data source; you must perform this logic.

You can bind to any data source that implements the **IEnumerable** interface; this includes collection objects, data reader objects, data set objects, **DataView** objects, and **DataTable** objects. All these objects (except data readers**)** also implement the **IList** interface; this supports data binding to a wider range of Windows Forms controls.

This difference is because of the type of scenario each object type is designed for. Data set and **DataTable** objects provide a rich, disconnected structure suited to both Windows Forms applications and Web applications. Data readers, on the other hand, are optimized for Web applications that require optimized forward-only data access.

For an example of how to perform data binding, see "How to: Perform Data Binding in ASP.NET Web Forms" in Appendix B of this guide.

Paging Data

When you have to retrieve a large amount of data, it is a good idea to consider using data paging techniques to avoid scalability problems. Generally, follow the simple rule of not retrieving any more data than you require at any one time. For example, if you have to display 1,000 rows of data in a grid with 20 rows per page, implement data retrieval logic that retrieves 20 rows at a time.

Data paging techniques help to reduce the size of data sets, and to avoid expensive and unnecessary heap allocations that are not reclaimed until the process is recycled. For more information about data paging, see the *.NET Data Access Architecture Guide* on MSDN (*http://msdn.microsoft.com/library/en-us/dnbda/html/daag.asp*).

Supporting Data Updates from the Presentation Layer

In addition to viewing data, many applications must allow users to make updates, insertions, or deletions. There are a number of considerations to keep in mind when implementing data update functionality in the presentation layer. These considerations include the following:

- Is it appropriate to perform batched updates?
- How should you implement optimistic concurrency?
- Do you have to define data maintenance forms to support CRUD (Create, Read, Update, Delete) operations?

This section addresses each of these issues.

Batching Updates

The purpose of batching updates is to improve performance, scalability, and integrity of data. This technique groups related operations and submits them as a unit, so that they occur in one network roundtrip, or so that they can be encapsulated in a transaction.

To batch updates you can use two techniques:

- Make changes to data in a data set, and then bind the data set to a **DataAdapter** object in the data access logic components.
- Store data for your changes in your custom business objects, and invoke the appropriate data access logic methods from a business object.

The first technique is easier to implement, but it does not offer much flexibility in how the resulting changes are sent back. For example, the data set has to be bound to a **DataAdapter** that is specific to a database connection.

Using Optimistic Concurrency

When using optimistic concurrency, a row in the data source is not locked when a user reads it. Because the row of data is not locked, other users can read or update the row after the original user reads it. When the original user tries to update the row, the system must check whether the data has been modified by another user in the intervening period.

There are various techniques for identifying whether the data has been modified. For example, you can use timestamps to indicate the last-modification time for the row. Another approach is to keep a copy of the original data for the row and compare it against the current data for the row when you perform an update.

For guidance on implementing optimistic concurrency strategies, see the section titled "Using Optimistic Concurrency" in *Designing Data Tier Components and Passing Data Through Tiers* on MSDN (*http://msdn.microsoft.com/library/en-us/dnbda/html/BOAGag.asp*).

Designing Data Maintenance Forms to Support Create, Read, Update, and Delete Operations

Many applications require Create, Read, Update, and Delete (CRUD) forms to allow administrators and other users to perform day-to-day data maintenance tasks. However, data maintenance is only a part of the user interface of the application; most applications also provide forms to support specific business use cases and to perform reporting tasks.

Note: If a use case for updating data works mostly with other services, instead of being just an update, it is not likely to be built as a distinct CRUD form. For example, checkout pages in an e-commerce site are generally not designed as CRUD forms.

Data Maintenance Principles

The primary motivation for designing CRUD forms is to maintain simple data related to business entities. Typically, the design and implementation of CRUD forms is strongly driven by the relational data storage design of the application.

One of the key assumptions of CRUD data maintenance forms is that the actions are relatively predictable on the data of the application; this can lead to some optimizations. For example, you can assume that adding a product category generally results in a new row in a table of product categories.

Data maintenance operations are also constrained to simple entities that are normalized, pretty much in the same way as the database. This predictability leads to opportunities to use caching, and thereby reduce roundtrips to remote servers.

Typically, many data maintenance forms are built around the following base elements:

- The business entity that you want to maintain
- A mechanism to display a list of business entities, where the user selects one from the list
- A mechanism to view, edit, or delete a single business entity

The business entity you are maintaining does not have to be complex. In fact, maintaining complex business entities typically involves departing from conventional CRUD mechanisms. The business entities you deal with can be expected to have the following set of characteristics:

- A set of attributes that together represent the business entity data
- A set of validation rules for the individual data members in the business entity, and for the business entity as a whole
- A set of fields that identify the business entity, such as a primary key
- Fields that are references to other business entities, such as a **RegionID** field in a customer entity.

To display a set of business entities in a control, such as a **DataGrid** or a **DataList**, you typically have to consider the following questions:

- What business entities are being displayed? Business entities are typically filtered in some way, or they are loaded on specific user gestures such as paging through a large set of entities.
- What attributes of the business entities have to be displayed?
- What actions are permitted on the business entities (such as editing and deleting) and what action triggers them (for example, right-clicking on a **DataList**, selecting a shortcut menu option, and clicking a hyperlinked attribute)?

To display and allow data entry on a business entity, consider the following questions:

- What attributes are shown in "new" or "edit" modes?
- How do you make sure there is integrity in new or edited business entities?
- What controls are used to display the attributes, and how do they proactively enforce integrity? For example, you can use a drop-down list to select the region of a customer, or a calendar control for picking a date. You also require extra information, such as default values, and how the reference data for drop-down lists is retrieved.

Using Different Visual Styles for Data Maintenance

There are many visual styles for data maintenance. The most common are:

- **Implementing separate forms for the list and entity display**–Create separate forms to display a list of business entities and a single business entity. For example, you might display a form that shows a list of all geographical regions to the user. When the user selects a specific region, display a new form that shows the details for the selected region.
- **Implement list and details in the same form**–Create a single master-detail form. The master-section of the form displays a list of all the business entities. When the user selects one of the entities, its details are displayed in the detail-section of the form.
- **Implement inline editing in grids or specialized controls**–Create a form that contains grids or other specialized controls to allow inline editing of the data.

When deciding which style to use, consider issues such as usability and the complexity of the business entity being maintained. Simple business entities (for example, reference data consisting of an ID and a small set of fields) can be represented in a grid, whereas more complex entities generally have separate forms for the display. Practical issues such as development effort and maintainability are important factors. Consistency is also an important consideration, because it increases usability and therefore reduces training costs.

The following sections describe each of the techniques listed earlier in this section. If you examine the diagrams closely, you will see that each technique uses the same functionality in the controllers and performs the same interaction with the server. Therefore, if you implement these techniques correctly, they can all be equally scalable. However, you also have to consider the impact of the user interface on how the data is consumed. For example, if you show the details for customers just below a list of all known customers, a user may click each customer in the list, causing the application to perform many data retrievals from the database.

For an example that illustrates each of the techniques listed above, see "How to: Design Data Maintenance Forms to Support Create, Read, Update, and Delete Operations" in Appendix B of this guide.

Implementing Separate Forms for the List and Entity Display

When implementing separate forms to display a list of business entities and individual business entity details, it is a good idea to use different controller classes for the list view and the single-entity view, effectively creating different user interface "processes." This simplifies reuse of the business entity details form for insert and edit operations.

Figure 4.5 illustrates a solution that uses separate entity list and entity details forms. The interaction with the server from the controller functions is illustrated as a call into a service agent to reduce clutter. For CRUD operations, calls typically go to the data access layer.

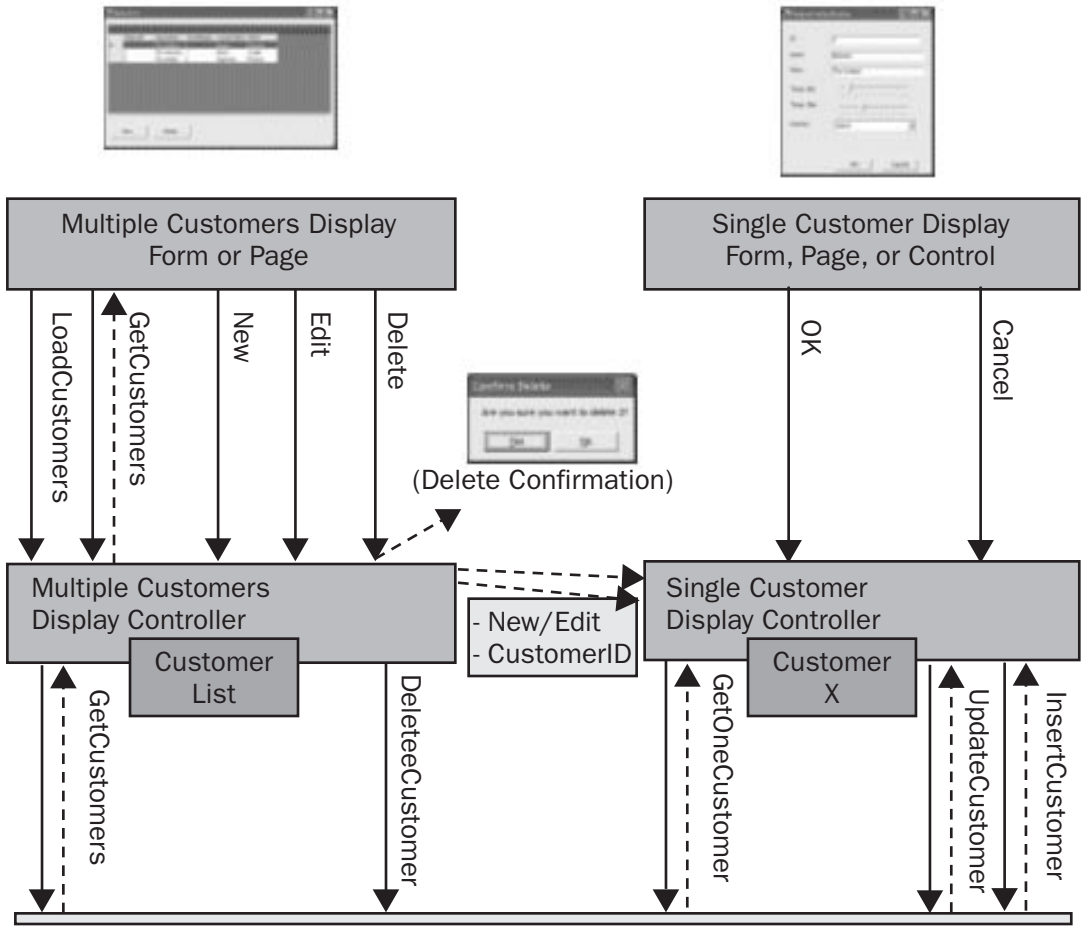


Figure 4.5

Using separate forms to display a list of business entities and an individual business entity

It is a good idea to use one schema, type, or data set for the data that appears in the list (the list might contain aggregated, computed, or formatted data), and another schema, type, or data set for the single entity.

Implementing List and Details in the Same Form

When you have to edit only one entity at a time, you might consider implementing the entity list and details user interface elements on the same form. Reusability is reduced in this scenario because the single-entity view is embedded in the same view as the list, and programming the state machine in the forms and controls can be more difficult because users can change selection in the list while editing the attributes of the active entity. For example, if you use a **System.Windows.Forms.DataGrid** control to display an entity list in a Windows Forms-based application, you must decide how to respond to user actions, such as moving to another row or re-selecting the current row. For each user action, you must make decisions such as whether to accept the data that currently appears or to cancel the update.

When using this approach, you are advised to encapsulate the single-entity view in a containing control; this allows you to enable or disable the control instead of opening a new form when an entity is to be edited.

You are also recommended to use different controller classes for the list view and the single-entity view, as described in the previous section. If you use the same controller class for the list view and the single-entity view, there will be increased coupling as shown in Figure 4.6. The dotted lines indicate that the list view might capture user actions that inform the single-entity view to accept or discard changes on the current entity.

If a user clicks on many rows while browsing through data, the properties that appear in the single-entity view must be updated in fast succession. To address this issue, it may be efficient to preload as much of this information as possible.

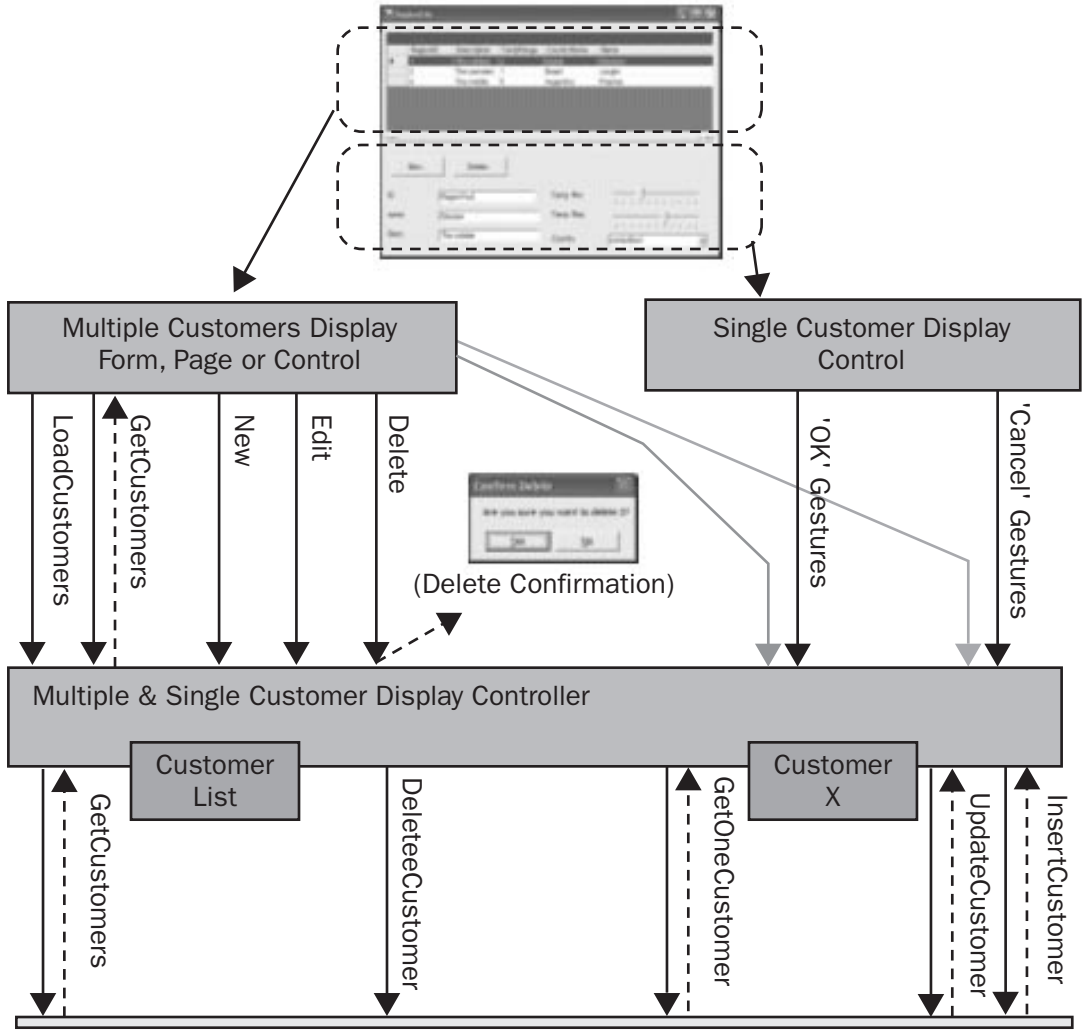


Figure 4.6
Using a single form to display a list of business entities and an individual business entity

Implementing Inline Editing in Grids or Specialized Controls

An inline editing approach is suitable in Windows Forms-based applications where you allow users to edit data in a grid directly. In this approach, a set of entities appears in the user interface and facilitates editing inline so that a separate view is not required. All data insertions and updates are performed on the display of the entity list itself.

The inline editing approach is best suited for simple entities that have relatively few fields and do not require specialized user interfaces to edit or create. It is assumed

that the user can act directly on the data in the grid, so special interactions are not required with the business or data layers to get the entity data. This approach is also appropriate when updates to multiple rows can be batched, for example by taking a modified data set and sending it back to the business and data layers for processing.

You can develop an inline editing user interface manually with any grid that supports editing. Follow these guidelines:

- Capture the appropriate events from the controls that receive user actions, and tie the events to the controller functions described earlier in this chapter.
- Use data binding with feature-rich grid controls that handle all the user actions relating to navigation, editing, and inserting new rows; this allows you to respond only to events to confirm row deletion. To update the data source, take the modified data set and use batched updates, or iterate through the data set and invoke appropriate single-entity methods on the server.

Figure 4.7 shows an inline editing user interface design.

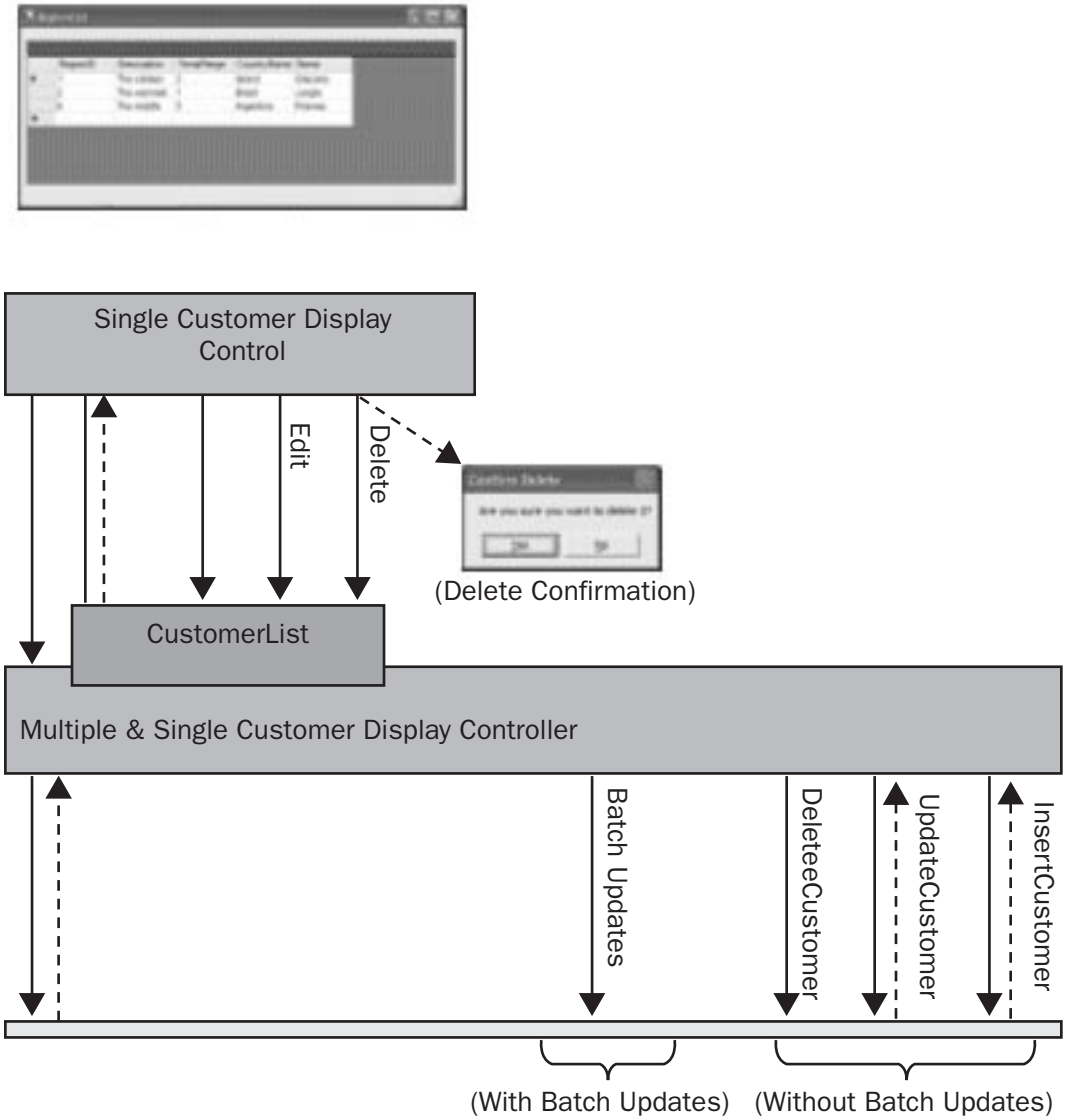


Figure 4.7
Using inline editing to display a modifiable list of entities in a grid control

Advanced Data Maintenance Implementation Techniques

If you have a recurrent structure in your code that depends only on the description of the entities and information such as how they are displayed and validated, you can use metadata-based techniques to automatically generate CRUD user interfaces. With this approach, the development team does not have to perform tedious, repetitive tasks for each different entity in your application.

You use metadata to create a simple meta-model for your application; the meta-model describes the structure of classes and their inter-relationships. For example, you can write an XML document containing appropriate elements to help with things such as automatic generation of CRUD-related components, pages, and forms. Figure 4.8 shows a fragment of an example of such a model.

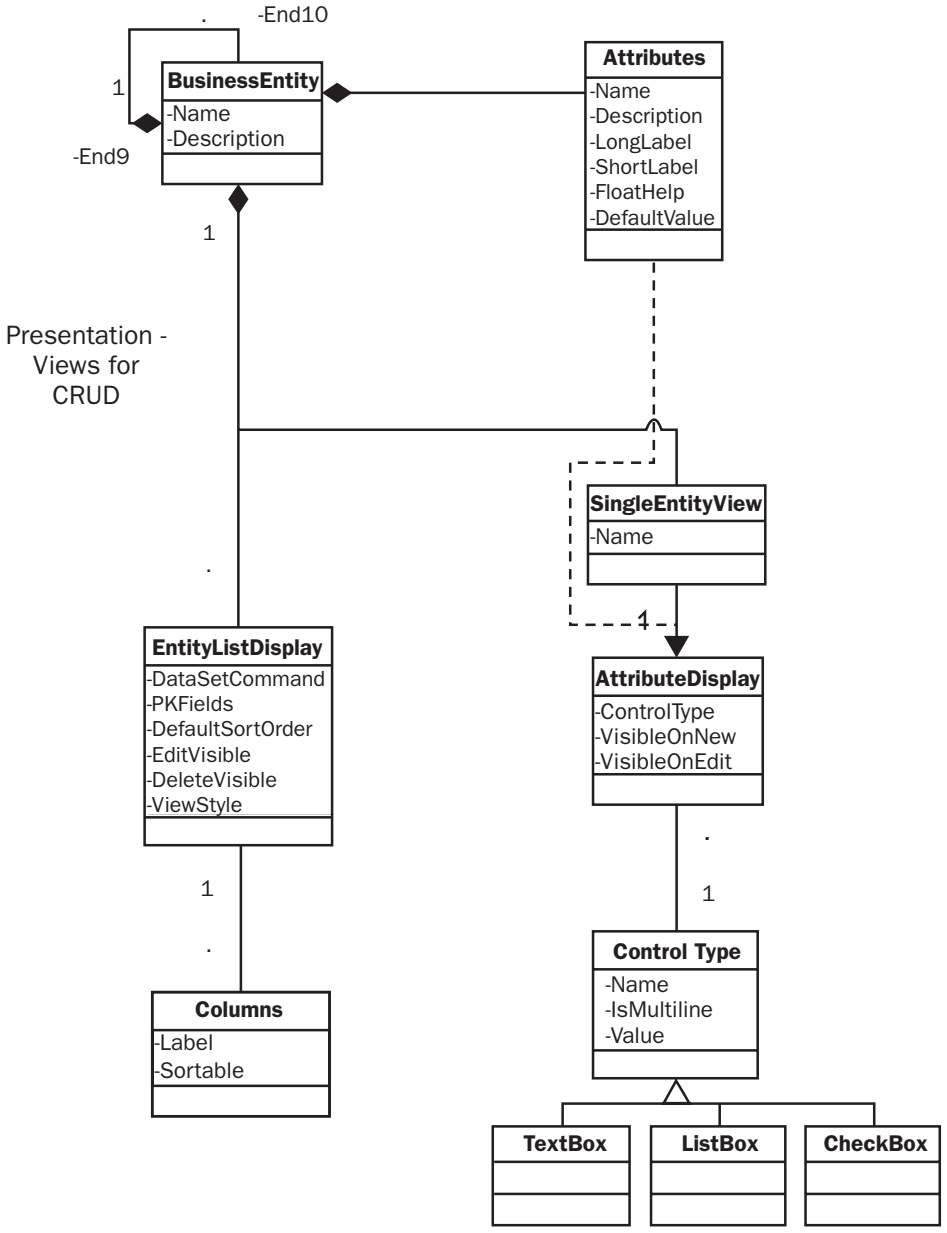


Figure 4.8
Fragment of a data maintenance meta-model

The first step in using a metadata-based approach is to model the metadata you require. Generally, you require information about the entities, how to display them, and how to allow data entry. The best way to determine what metadata you require is to analyze a small set of CRUD implementations that follow the same structure, and that express the variability points you believe will be required across all the entities.

Table 4.1 shows some common pieces of metadata that are required and some sources where that metadata can already be found, or sources that can be used to capture metadata.

Table 4.1: Common sources of metadata for data maintenance user interfaces

Metadata	RDBMS	DataSet Schema	Visual Studio .NET Designer	Manual Entry
Entities				
Attributes and data types	✓	✓		
Validation rules for attributes	✓	✓		
Validation rules for entities	✓	✓		✓
Default values for attributes	✓	✓		
Entity display				
Controls to use for attributes			✓	
Layout of controls			✓	
Friendly names for attributes and entities				✓
Data binding				
Binding of view, controller, and components				✓
Names of forms and controls			✓	✓
Names of assemblies that perform insert, read, update, and delete operations				✓
Names of controller classes				✓

There are two main ways to implement a metadata-based approach to building a data maintenance user interface:

- **Using metadata for code generation**–Use metadata at design time, in the tool, or during the build process to automatically generate all data maintenance user interface code. This approach is appropriate if you use code generation as a starting point for the development process but expect to change or customize the generated code after it is working, or if the effort of acquiring and interpreting the metadata at runtime is costly in terms of security, performance, and initial development effort.

  To use this approach, you have to know:

  - The implementation structure you are trying to repeat.
  - Whether it relies on reusable components (for example, a validation framework or a specialized **ListView**).
  - The templates for the code you have to generate.
  - The schema for the metadata required to generate the code.
  - The sources for the metadata.

- **Interpreting metadata at run time**–Using metadata at run time can enhance maintainability. Maintainability is improved because only the metadata must be changed to modify the application behavior; the source code is not affected. This means that modifications can be implemented by business users if they are provided with appropriate editing tools.

  However, this approach might require:

  - Extra effort in design and development of the code that acquires and interprets the metadata. For example, you have to load assemblies at run time, create controls on a form and display them at run time, and bind events.
  - Careful analysis of performance degradation. For example, performance might suffer if the metadata is interpreted many times across users.
  - Security reviews and testing. For example, the use of metadata at run time might open the door to elevation of privileges and component injection techniques.

  To use this approach, you have to know:

  - How to load controls, bind events, and call components at run time
  - Whether the framework that interprets the metadata has sufficient extension points to grow with the requirements
  - Whether the dynamic behavior relies on techniques that require full trust security privileges, for example by using reflection
  - The sources for the metadata that can be used at run time

You can use the Configuration Management Application Block to store the metadata in a secure way. You can download this block from MSDN (see *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/cmab.asp*).

Figure 4.9 shows a metadata-based approach to data maintenance user interface design.

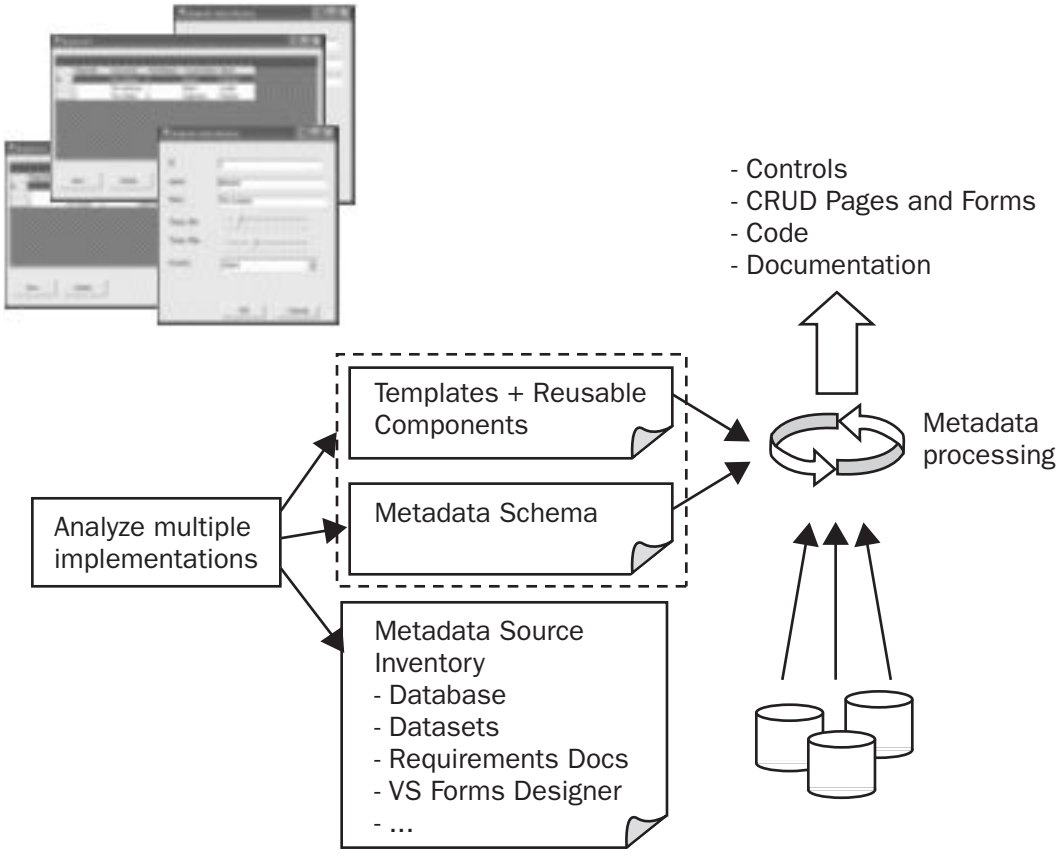


Figure 4.9
A meta-data based approach to data maintenance user interface design

Metadata techniques apply mostly to enterprise-level application developers who require CRUD forms for many entities, and to ISVs who want to build very flexible and rapidly customizable applications.

Validating Data in the Presentation Layer

Validation is an important issue in the presentation layer. There are two kinds of validation:

- **Continuous validation**–Continuous validation occurs each time a form or page is submitted. The validator controls provided by the .NET Framework provide continuous validation.
- **One-time validation**–One-time validation occurs only once. To perform one-time validation of a control, you must write some code.

This section describes why validation is important, and it provides guidance on how to perform validation in the presentation layer.

Why Validate?

It is a good idea to never trust user input. Validating the data entered in your application can produce the following benefits:

- **Integrity**–Validation enables you to prevent integrity errors in the data in your application.
- **Usability**–Validation permits you to format user-entered data (for example, a user enters a phone number as "1234567890" and you want it to display it as "123-456-7890").
- **Security**–Validation on incoming data helps prevent security threats such as cross-site scripting attacks and code-injection.
- **Responsiveness**–The built-in validator controls (**RegularExpressionValidator** and **RequiredFieldValidator**) enable you to give users immediate, client-side feedback.
- **Simplicity**–The built-in validator controls provide a simple way to replace validation in client-side ECMAScipt (JScript, JavaScript) code . These validator controls are much cleaner, easier to implement, and easier to debug and maintain than writing custom scripting code at the client.
- **Client-side and server-side validation**–The concept of "is this page valid" is nicely abstracted on both the client and server. The **System.Web.UI.Page** class has a **Validators** property that returns a collection of validator controls contained on the requested page, and an **IsValid** property that indicates whether page validation succeeded.

Data validation is important because it protects the integrity of your application data and improves the usability of the application.

Choosing a Validation Strategy

Any input coming from a user must be validated before being used. The .NET Framework provides a rich set of validator controls that you can use in ASP.NET Web applications, to handle this task.

There are five main types of validator controls:

- **ComparisonValidator**–Verifies that a user's input is of the correct type, or that the input matches a specific pre-defined value.
- **RequiredFieldValidator**–Verifies that the user has entered a value for a particular control.
- **RangeValidator**–Verifies that a user has entered a value within a permissible range. For example, you can test that the amount a user wants to withdraw from his or her checking account is between $0 and the total balance.
- **RegularExpressionValidator**–Verifies that user input matches a specific pattern. For example, you can test that a social security number matches the patter "nnn-nn-nnnn," where "n" is a number between 1 and 9.
- **CustomValidator**–If none of the built-in validator controls suit your validation requirements, you can write a custom validation function that performs server-side validation of user input.

By using the validation controls provided in the .NET Framework, you can prevent many of the problems associated with invalid data entry in ASP.NET Web applications.

Note: The validator controls are only available in ASP.NET Web applications. In Windows Forms-based applications, you must write your own code to perform these validation tasks.

Using Validation Controls

Consider the following issues when using validation controls:

- You might have to enable or disable some validator controls in response to user actions or if previous validator controls return as invalid. For example, you might have to validate the **Province** field only if the user selects **Canada** in the **Country** field. This must be done programmatically, and it can involve fairly complex logic.
- There are scenarios where the standard validator controls do not provide sufficient functionality and you must resort to the **CustomValidator**. Moreover, if you require client-side validation, you must write your own custom script code.
- There is no **RequiredFieldValidator** that works with a **CheckBoxList** control. To perform validation on this control, you must create your own validator class that inherits from **BaseValidator**.

Handing Validation Errors

There are several options for handling validation errors:

- Raise an exception. This might not be a useful action in the presentation layer. However, you might want to raise (or log) an exception if one of your validator controls detects some sort of attack.
- Display individual error messages, indicating the reason for the validation error and allowing the user to re-enter data accordingly.
- Use the **ValidationSummary** control to display a summary of all validation errors in a single location.

Whatever action you decide to perform when validation fails, you must make sure that the user is clearly notified about what is wrong and is given an opportunity to correct the data entry.

Summary

Accessing, presenting, modifying, and validating data are fundamental to most applications. You must make sure that you carefully plan how data will be accessed by the presentation layer, and in particular whether presentation layer components should have direct access to data access logic components. You must consider the impact of your data presentation approach on scalability, and you must carefully consider how you will implement data maintenance forms that allow users to view and modify data. Finally, it is a good idea to always implement at least a minimal level of data input validation to protect the integrity of your application's data and to improve usability.

5

Managing State in Web Applications

In This Chapter

This chapter describes how to manage state in the presentation layer in Web applications. The chapter includes the following sections:

- Understanding Presentation Layer State
- Planning State Management for Web Applications

State management is an important aspect of many applications. In the presentation layer, you frequently have to store information about the state of the application or keep track of user state information. Some examples of state include:

- The identity of the current user
- The contents of a shopping cart
- The current step in a navigation flow of the application
- A database connection string

When designing your application, you have to make decisions about how to handle state management, including what to store, where to store it, and how long information will be kept.

The Microsoft .NET Framework provides access to a rich set of state management mechanisms for both Windows-based and Web applications. This chapter describes state management mechanisms for Web applications.

Before looking at the individual mechanisms for state management, make sure that you understand the nature of application state to select the most appropriate mechanism.

Understanding Presentation Layer State

Three characteristics of state determine the state management mechanism most appropriate to your requirements. These three characteristics are:

- State lifetime
- State scope
- State type

The next sections describe each of these characteristics.

Determining State Lifetime

The lifetime of state refers to the period when that state is valid. Table 5.1 lists the common lifetime periods you use in your presentation layer.

Table 5.1: Common Lifetimes for Application State

State lifetime	Description	Example
Permanent	Data that is always valid	Information stored in a database
Process	Data that is valid only within the scope of a particular process	The process a user must go through to add an item to a shopping cart in an online store
Session	Data that is valid only within the scope of a single session for a single user	The contents of a user's shopping cart
Message	Data that is valid only within the scope of a single message or data exchange	The checkout request made by a user
Time span	Data that is valid only until a particular date and time	A coupon code that expires in two weeks

During the design process you must identify the most appropriate lifetime for your user interface state.

Determining State Scope

The scope of state defines the accessibility of an application's state. Scope is additionally divided as follows:

- Physical scope
- Logical scope

The following sections describe the various kinds of physical and logical scope available.

Determining Physical Scope

Physical scope defines the physical locations that state can be accessed from. Table 5.2 lists common physical scopes for application state.

Table 5.2: Common Physical Scopes for Application State

State Physical Scope	Description	Example
Organization	State is accessible to all applications in an organization	Organizational information stored in Microsoft Active Directory® directory service
Farm	State is accessible to all computers in an application farm	Information stored in a shared database associated with the farm
Computer	State is accessible to all applications running on a computer	Information stored on the file system or a local database (with no restricting permissions)
Process	State is accessible to multiple application domains running in the same process	Identity of the process
Application Domain	State is accessible only to code running in a single application domain	Objects stored in the application domain's **CurrentDomain** property

When designing your presentation layer, consider the affect of the various options for physical state storage locations, and select the most appropriate one for your application.

Determining Logical Scope

Logical scope defines the logical locations where state can be accessed. Table 5.3 lists common logical scopes for application state.

Table 5.3: Common Logical Scopes for Application State

State Logical Scope	Description	Example
Application	State is accessible only in a certain application	Application-specific information stored in a database
Business Process	State is accessible only to elements of a single business process	A purchase order being built by a business process
Role	State is accessible only to a particular role or group of users	Payroll information
User	State is accessible only to a particular user	The contents of a specific user's shopping cart
View	State is only relevant to a particular form, page, or control	The contents of a data grid

Selecting the correct logical scope for your presentation layer state is important because it has a significant affect on security and performance.

Determining State Type

The purpose, content, and quantity of state data are important factors when determining the state management mechanism to implement. Table 5.4 lists the characteristics of state that have the most effect on your choice of state management mechanism.

Table 5.4: Common Types of Application State

State Type	Description	Example
Secret / Sensitive	Information that is intended for a specific audience	Bank account transactions; private key used for encryption
Insensitive	Information that is not restricted in who, what, or both, that can see it	Public key used for encryption; hit count for a Web site
Object	Information that is stored in an object such as a **DataSet** or a **Hashtable**	The working set for a user performing an administration task offline
Scalar	Information that is stored in a single value	The number of times a user has executed an application
Large	State that takes up a large amount of storage space	The daily order report for a large e-commerce Web site
Small	State that takes up a small amount of storage space	The user ID of the user logged in to the application
Durable	Information that must be stored in a permanent medium to survive, for example, computer reboots	A submitted customer order
Transient	Information that is not required to be stored in a permanent medium	The current page that a user is on in a Web application

You must be familiar with the types of state your presentation layer must manage. Most applications use a combination of the state types listed in Table 5.4.

Planning State Management for Web Applications

Applications based on Web Forms are inherently stateless, but it is frequently necessary for an application to keep track of state information. ASP.NET simplifies state management by providing access to a variety of client-side and server-side state management mechanisms:

- **Client-side state management mechanisms**–Cookies, hidden form fields, query strings, and the **ViewState** property.
- **Server-side state management mechanisms**–Session state, application state, and a database.

Table 5.5 lists these mechanisms and provides an indication of the state lifetime, scope, and type that the particular mechanism is most appropriate for.

Table 5.5: State Management Mechanisms for Web Applications

Mechanism	Lifetime	Physical Scope	Logical Scope	Type
Session object: In-process	Session	Process	Application, User	Any
Session object: State server	Session	Organization	Application, User	Any
Session object: SQL Server	Session	Organization	Application, User	Any
Cookies	Permanent Session Time span	Web farm	Application, User	Small, non-essential, insensitive
Hidden form fields	Message	Web farm	Application, User, View	Small, non-essential, insensitive
Query strings (URL fields)	Message	Web farm	Application	Small, non-essential, insensitive
ViewState	Message	Web farm	View	Small, non-essential, insensitive
Application object	Process	Process	Application	Any
Database	Application-controlled	Datacenter	Application	Any

The following sections describe each of the state storage mechanisms listed in Table 5.5. Each section provides:

- An overview of the mechanism
- Guidance on when to use the mechanism, and points to consider before deciding to use the mechanism
- Sample code and configuration settings, to show how to use the mechanism

When planning your Web application, use the following guidance to identify and implement the most appropriate state storage mechanism for your presentation layer.

Note: If you want to encapsulate the choice of state storage mechanism in an ASP.NET Web application, you can use the Microsoft User Interface Process Application Block to manage state storage. You can configure the block to use the **Session** object or SQL Server to store state information.

For more information about using this block to manage state storage, see the "To create a controller class" procedure in Chapter 2 of this guide.

Storing State in the Session Object

By default, whenever a new client connects to an ASP.NET application, ASP.NET creates a cookie on the client containing a unique session identifier (ID). With each subsequent request, the client passes the cookie to the application. This allows the application to maintain context across a series of stateless Web requests. If the client or application is configured to disallow cookies, ASP.NET encodes the session ID as part of the URL the client uses when making requests to the application.

The **Session** object provides a state store that is unique for each session. Internally, ASP.NET manages **Session** objects for each active client session and makes the appropriate **Session** object available in your application through the **Session** property of the **System.Web.UI.Page** class, where all Web Forms pages inherit from.

The benefits of using the **Session** object include:

- The ASP.NET runtime manages the mapping of the session ID to session data.
- State data is easily accessible through the page-level **Session** property.
- You can configure a timeout after which ASP.NET terminates an inactive **Session** and disposes of the state for that session.
- State is not transmitted to clients, so this mechanism is efficient for network bandwidth purposes and more secure than mechanisms such as cookies and hidden form fields.
- Session management events can be raised and used by your application.

- Data placed in session-state variables can survive Internet Information Services (IIS) restarts and worker-process restarts without losing session data, because the data is stored in another process space.
- Session state can be used in both multi-computer and multi-process configurations, thereby improving scalability.
- Session state works with browsers that do not support HTTP cookies, although session state is most commonly used with cookies to provide user identification facilities to a Web application.

In addition to these benefits, you can configure the **Session** object to use one of the following three backing stores:

- In process
- State server
- SQL Server

Each backing store is described in the following sections.

Note: It is also possible to disable session state storage. To do this, add a **<sessionState mode="Off"/>** element to the **<system.web>** section of the Web.config file for your ASP.NET Web application.

Storing State in the In-Process Session Object

Using in-process session state in ASP.NET is closely analogous to using the classic ASP session state and is the default mechanism for managing state in an ASP.NET application. Session state is stored and managed in-process (using the Aspnet_wp.exe process); when that process recycles, the session state that was stored in it is lost.

The main benefit of storing state in-process is performance: the reading and updating of session state occurs much faster when that state information is stored in memory in the same process as the ASP.NET application, because there is no additional overhead for cross-process, network, or database communications.

Use the in-process **Session** object when:

- You have to store small or medium amounts of state information.
- You require high performance data access.
- You do not require durability; in-process **Session** object data does not survive computer reboot, IIS reset, or application domain unloading.
- Your application is hosted on a single server.

Before deciding to use the in-process **Session** object, consider the following points:

- In-process session state is the fastest of the three available backing stores for the **Session** object.

- The state information cannot be shared between multiple computers. If your application is running in a Web farm, you have to make sure that individual users are pinned to a specific server, or their state information will be lost as they bounce between the various servers in the Web farm.
- As the amount of state information stored in-process increases, so does the memory required to store that information. As more and more information is added to the in-process session state, the performance of the whole application may be degraded because it takes longer to access and modify the state information.

If in-process session state is appropriate for your application, you can configure it in the Web.config file associated with your ASP.NET Web site.

Note: The settings in Web.config take precedence over any machine-wide settings defined in the machine.config file. If you want to define machine-wide state management settings, edit machine.config instead of Web.config file.

- If you are using version 1.1 of the .NET Framework, machine.config is located in the %windir%\Microsoft.NET\Framework\v1.1.4322\Config folder.
- If you are using version 1.0 of the .NET Framework, machine.config is located in the %windir%\Microsoft.NET\Framework\v1.0.3705\Config folder.

► **To configure the in-process Session object**

1. In the application's Web.config file (or in the computer-wide machine.config file if you prefer), add the **<sessionState>** element if it is not already there.
2. In the **<sessionState>** element, set the **mode** attribute to **"InProc"**.

   This causes the **Session** object to be stored in the Aspnet_wp.exe process for the ASP.NET application.
3. If you want to specify a timeout period after which state for an idle process will be discarded, set the **timeout** attribute in the **<sessionState>** element. The default timeout period is 20 minutes.

The following example illustrates these points.

```
<configuration>
  <system.web>
    <sessionState mode="InProc" timeout="10"/>
  </system.web>
</configuration>
```

This example specifies in-process session management, with a timeout period of 10 minutes.

Storing State in the State Server-based Session Object

Another option for storing session state in an ASP.NET application is out-of-process, by using a state server process. When you use this option, session state is stored in the Aspnet_state.exe process; this process runs as a service and may be located on a separate state server. Using a state server allows applications running in a Web farm to access shared state information.

State information stored in a state server is not lost when a particular application recycles. However, the data is still transient: if the state server recycles, the state information is lost.

Use the state server-based **Session** object when:

- You have to share state information between multiple applications or multiple computers.
- You do not require durability. Data does not survive reboots of the state server.

Before deciding to use the state server-based **Session** object, you must consider the following points:

- Using out-of-process session state is slower than in-process session state.
- The state service does not require authentication.
- Network traffic between the application and state service is not encrypted. For security reasons, it is best to not run the state service on a server that is directly accessible through the Internet.
- Communications between applications and the state server use the well-understood (and plain-text) HTTP protocol. It is a good idea to encrypt traffic between the application and state server using Internet Protocol Security (IPSec). For detailed information about IPSec, see Chapter 4 of "Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/secnetlpMSDN.asp*).
- Information stored in a state server must be serialized; for more information, see "Serializing State" later in this chapter.
- The IP address or network name of the state server must be specified in Web.config.

If state server-based session state is appropriate for your application, you can configure it in the Web.config file associated with your ASP.NET Web site (or in machine.config)..

► **To configure the state server-based Session object**

1. Make sure that the state service (Aspnet_state.exe) is running on the state server computer.
2. The default port used by the state service is the "well-known" port number, 42424, making this an easy target for attack. To change the port number, set the value in the HKLM\SYSTEM\CurrentControlSet\Services\aspnet_state\Parameters\"Port" registry key.
3. In the Web.config file for your Web application (or in the computer-wide machine.config file if you prefer), add the **<sessionState>** element if it is not already there.
4. In the **<sessionState>** element, set the **mode** attribute to **"StateServer"**. Also set the **stateConnectionString** attribute to the IP address or network name of the state server, and indicate the port number the state service is running on.

   Optionally, also set the **stateNetworkTimeout** attribute to the number of seconds that the ASP.NET Web application will wait for the state server to respond to network requests. The default value is 10 seconds, but there are many factors that might warrant a higher timeout value, such as if the state server or the Web server are overloaded.

The following example illustrates these points.

```
<configuration>
  <system.web>
    <sessionState mode="StateServer"
                  stateConnectionString="tcpip=myStateServer:42424"
                  stateNetworkTimeout="20"/>
  </system.web>
</configuration>
```

This example specifies state server-based session management. The state service is running on a computer named myStateServer and is accessed through port 42424. A timeout period of 20 seconds has been specified.

Storing State in the SQL Server-based Session Object

Another option for storing session state for ASP.NET applications is to use SQL Server. Using SQL Server to store session state provides several benefits:

- State can survive SQL Server restarts, if it uses a database other than the default database, **TEMPDB**.
- You can share state between multiple instances of your Web application in a Web farm.
- You can take advantage of clustering; this persists the state if SQL Server stops unexpectedly.

The following sections describe when and how to store state in the SQL Server-based **Session** object.

Use the SQL Server-based **Session** object when:

- You have to share state information between multiple applications or multiple computers.
- You require fine-grained security control over who is permitted to access the data.
- Your state information must be durable.

Before deciding to use the SQL Server-based **Session** object, consider the following points:

- Using SQL Server for session state also allows user tasks in your application to span multiple sessions and devices.
- Using SQL Server to store session state is slower than using the in-process and state server-based **Session** object options.
- Session state stored in SQL Server must be serialized; for more information, see "Serializing State" later in this chapter.
- If the connection string used to connect to the SQL state server uses integrated security, your ASP.NET application must use impersonation. For information about how to use impersonation in ASP.NET applications, see Chapter 8 of *Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication* on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/secnetlpMSDN.asp*).
- SQL Server can use clustering and failover to make sure state data is available.
- Using SQL Server allows you to perform external backup of long-lived state data.

If SQL Server-based session state is appropriate for your application, you must configure the application to connect to the appropriate database server.

► To configure the SQL Server-based Session object

1. Configure SQL Server to handle **Session** object state. To do this, run the **InstallSqlState.sql** script on the computer that is running SQL Server. By default, the script is located in the \systemroot\Microsoft.NET\Framework*version* folder.

   This script creates the necessary databases, tables, and stored procedures to support session state storage.
2. In the Web.config file for your Web application (or in the computer-wide machine.config file if you prefer), add the **<sessionState>** element if it is not already there.
3. In the **<sessionState>** element, set the **mode** attribute to **"SQLServer"**. Also set the **sqlConnectionString**, to configure the database connection.

The following example illustrates these points.

```
<configuration>
  <system.web>
    <sessionState mode="SQLServer"
                  sqlConnectionString="datasource=x; user id=y; password=z"/>
  </system.web>
</configuration>
```

Note: If you are using version 1.1.4322 of the .NET Framework, you may choose to run the **InstallPersistSqlState.sql** script instead of running **InstallSqlState.sql**. The **InstallPersistSqlState.sql** script adds state management tables to the **ASPState** database instead of to the **TempDB** database, so that session state survives a restart of the state server.

If you use the **InstallSqlState.sql** script, the SQL state server uses **TempDB** to store state information. For details about how to configure persistent state in versions of the .NET Framework earlier than 1.1.4322, see article 311209, "HOW TO: Configure ASP.NET for Persistent SQL Server Session State Management," in the Microsoft Knowledge Base (*http://support.microsoft.com/*).

Using the Session Object

Whichever underlying storage mechanism you choose to use for the **Session** object, it does not affect how you use the object in your ASP.NET Web application code.

The **System.Web.UI.Page** class has a **Session** property that returns a **System.Web.SessionState.HttpSessionState** object; through this object you can manipulate the contents of the **Session** object associated with the current user session. State items are held as **Object** references in **HttpSessionState**.

The **HttpSessionState** class provides properties and methods to manipulate the collection's contents, enabling you to add and remove collection elements, get and set specific data elements, enumerate the collection, and clear the collection.

The following code sample shows how to set **Session** object state, by using the C# indexer of the **HttpSessionState** class.

```
Session["username"] = "myUserName";
Session["lastvisit"] = DateTime.Now;
```

The following code sample shows how to get **Session** object state. Notice the requirement to cast the retrieved data from **Object** into the required data type.

```
string userName = Session["username"].ToString();
DateTime lastVisit = (DateTime)Session["lastvisit"];
```

The following section describes how to use cookies as an alternative to the **Session** object for storing state information.

Storing State in Cookies

A cookie is a small amount of application data stored on the client—either on disk or in memory. The client passes the cookie as part of each request to the application, meaning that the application has access to the data held in the cookie. The application can use the cookie data to make runtime decisions, and it can modify or add to the cookie data before sending the cookie back to the client as part of the application's response.

By default, ASP.NET uses cookies to hold a client's session identifier; this allows ASP.NET to retrieve the correct **Session** object (discussed earlier). You can also use cookies to store your own state information.

Use cookies when:

- You have to store small amounts of state information.
- The state does not include secret or sensitive information.
- The state does not provide access to or drive secured parts of your application.
- Your application runs on a server farm and you have no means of providing a centralized state store.
- The state must survive across page requests or even across application invocations.
- Your application will still function if the state is not available.
- You have a relationship with the users and can be confident that they will enable cookies for your application.

Before deciding to use cookies, consider the following points:

- Cookies can survive across page requests, sessions, application invocations, and even computer reboots.
- Users can configure their browser to disallow cookies; in this case, your application must provide an alternate mechanism for storing the state information.
- The user's browser determines how much information a cookie can contain. Individual cookies are limited to a maximum size of 4 KB, and a specific domain may set a maximum of 20 cookies on a user's computer.
- Information stored in cookies is not durable; the cookie may expire based on information provided by the application or the user. The user can also delete the cookie.
- Because cookies are stored on the client, the information they contain is subject to modification.
- Cookies increase network traffic because the cookie is sent as part of each request and response.

- There is no state maintained on the server between client requests; this is particularly useful in a server farm because you do not have to maintain state centrally.
- Cookies allow you to set expiration times on the state information, after which the client browser discards the cookie.

Using Cookies

Cookies contain simple name/value string pairs. The easiest way to access cookie state is through the **Request.Cookies** and **Response.Cookies** properties of the current **System.Web.UI.Page** object. Both properties return a **System.Web.HttpCookieCollection** object containing the set of **System.Web.HttpCookie** objects that were received from, or will be sent to, the client.

The following code sample shows how to set state information in a cookie.

```
Response.Cookies["userinfo"]["username"] = "myUserName";
Response.Cookies["userinfo"]["lastvisit"] = DateTime.Now.ToString();
```

The following code sample shows how to get state information from a cookie. Notice the requirement to cast the retrieved data from **String** into the required data type.

```
string userName = Request.Cookies["userinfo"]["username"];
DateTime lastVisit = DateTime.Parse(Request.Cookies["userinfo"]["lastvisit"]);
```

Cookies can be a useful way to store state on the client. However, if you cannot guarantee that cookies will be enabled on every client, consider an alternative approach such as hidden form fields or query strings.

Storing State in Hidden Form Fields

Many Web applications store session state information in hidden form fields on the form that the user is currently working on. A hidden form field is not rendered visibly in the browser, but you can set its **value** property to hold a single piece of page-specific information. When the user re-submits the form to the server, the value of the hidden form field is sent in the HTTP **Form** collection along with the values of other controls on the form.

Use hidden form fields when:

- You have to store small amounts of state information.
- The state does not include secret or sensitive information.
- The state does not provide access to or drive secured parts of your application.
- Your application runs on a server farm and you have no means of providing a centralized state store.

- Cookies are disabled.
- The state does not have to survive across page requests.
- Your application will still function if the state is not available.

Before deciding to use hidden form fields, consider the following points:

- You can encrypt the information stored in hidden form fields to make it harder to attack. However, it is a good idea to use a back-end database to store any sensitive information.
- You can store only a limited amount of information in a hidden form field; the maximum size is limited by the control you choose for the hidden field.
- Information stored in hidden form fields is transmitted to the user and increases the size of the download. This can negatively impact performance.
- Pages must be submitted by way of an HTTP POST. If pages are submitted as an HTTP GET, your Web application will not be able to retrieve the information in the hidden form fields.
- It is a good idea to never trust data that comes from the user, especially if that data is of a sensitive nature. Some early shopping cart applications stored item price information for their carts in hidden form fields. This information was stored in plain text, and attackers soon realized they could modify the price in the hidden form fields to obtain a discount. Sensitive information should never be stored in hidden form fields, especially in plain text.

The following section describes how to use hidden form fields in an ASP.NET Web page.

► To use hidden form fields

1. Add a hidden field to a form on your ASP.NET Web page.

   One of the ways to do this is to use the **HtmlInputHidden** control; this control is located on the **HTML** tab on the Toolbox. When you add this control to your form, an **<INPUT>** tag is generated as shown in the following example.

   ```
   <INPUT type="hidden">
   ```

   Note: Another way to add a hidden form field is to use a standard Web Forms control such as a text box, and set its **Visible** attribute to **false**. If you adopt this approach, you do not have to perform Steps 2 and 3 in this procedure.

2. Add a **runat="server"** attribute to the **<INPUT>** tag, to enable you to access the hidden control in server-side postbacks. Also add an **id** attribute, to assign a programmatic identifier for the hidden control.

   ```
   <INPUT type="hidden" runat="server" id="MyHiddenControl">
   ```

3. In the code-behind file for your ASP.NET Web page, declare an **HtmlInputHidden** instance variable to correspond to the hidden control. The name of the instance variable must be exactly the same as the **id** attribute of the hidden control.

```
protected System.Web.UI.HtmlControls.HtmlInputHidden
MyHiddenControl;
```

4. At an appropriate juncture in your ASP.NET Web page, assign a value to the hidden control to store some information in the control.

   The hidden control, and its value, will be returned as part of the response to the client's browser.

```
MyHiddenControl.Value = "myUserName";
```

5. On a subsequent postback, retrieve the value of the hidden control and use the information as required.

```
string userName = MyHiddenControl.Value;
```

Hidden form fields provide a simple mechanism for storing state. A closely related mechanism is to use query strings.

Storing State in Query Strings (URL fields)

The HTTP specification allows for client browsers to pass data as part of the request's query string. This feature is typically used in HTTP GET requests to pass arguments to an application, but you can also use it to pass state information back and forth between the client and server. For example, in the URL *http://www.asp.net/somepage.aspx?userid=12&location=Boston*, the query string contains two attribute-value pairs: one named *userid* and the other named *location*.

Use query strings when:

- You have to store small amounts of state information.
- The state does not include secret or sensitive information.
- The state does not provide access to or drive secured parts of your application.
- Your application runs on a server farm and you have no means of providing a centralized state store.
- Cookies are disabled.
- The state must survive across page requests or even across application invocations.
- Your application will still function if the state is not available.

Before deciding to use the query strings to manage state, consider the following points:

- Some browsers or proxies have URL length restrictions. The current IETF RFC about URL length limits them to 4096 characters.
- If cookies are disabled, ASP.NET embeds the session ID in the URL of the page, which means the number of characters available for query string state is reduced.
- Users can easily change the contents of the query string. Make sure that no pages assume authorization, and write code to redirect users to a logon page if they have not already logged on.
- If the query string values contain characters such as spaces and punctuation, encode the query string values by using **HttpUtility.UrlEncode**.
- Use **HttpUtility.UrlDecode** when reading information from the query string to decode any encoded query string values and to protect against invalid characters in the query string.
- To make sure people do not modify a query string, it is a good idea to create a keyed hash code for the data.
- If you want to store sensitive state in the query string, you must use encryption. However, this raises the problem of key management, so it is a good idea to avoid the query string in favor of a more secure state mechanism.

The following procedure describes how to use query strings in an ASP.NET Web page.

► To use query strings

1. Write code in your ASP.NET Web page to programmatically assign a URL to a hyperlink control, form control, image control, or a similar control.

   The URL can contain a query string that relays information to the server-side postback when the user navigates the URL link. For example, the following statement sets the **NavigateUrl** property on a hyperlink control; the URL contains the user's ID in the query string.

   ```
   HyperLinkSubmitOrder.NavigateUrl = "orderVerify.aspx?userid=" +
                                      HttpUtility.UrlEncode("123 456 <789>");
   ```

2. On a subsequent postback, retrieve the required information from the query string as shown in the following example.

   ```
   string userID =
      HttpUtility.UrlDecode(Request.QueryString["userid"].ToString());
   ```

Query strings and hidden form fields are generic techniques that are used in many Web technologies. The following section describes an ASP.NET-specific mechanism, whereby state is stored in the **ViewState** property of an ASP.NET Web page.

Storing State in ViewState

Each ASP.NET page has a **ViewState** property that is used by the .NET Framework to persist control state between page requests. Every time the application responds to a client request, ASP.NET serializes the content of the page's view state and includes it in the HTML response as a hidden field. When the client submits the next request to the application, the request includes the serialized view state. ASP.NET parses the view state and uses it to set the initial state of the page and the contained controls. You can also use view state to store custom state information for your application

Use view state when:

- You have to keep track of page-specific information between postbacks.
- You have to store small amounts of state information.
- The state does not include secret or sensitive information.

Before deciding to use view state, consider the following points:

- State stored in the **ViewState** property is available only to the current page; view state is not persisted across pages.
- The values in view state are sent to the client, increasing the amount of network traffic.
- Storing large sets of information in the view state property can slow the rendering of the page and the parsing of requests, because ASP.NET has to manipulate the view state values.
- When rendered to HTML, the view state values are compressed, encoded and hashed. This improves efficiency by reducing the amount of data sent, and it helps make sure people cannot submit modified view state. However, because the view state is not encrypted, merely encoded, people can easily decompress and decode the view state to obtain the contained data.
- When using ASP.NET mobile controls to deliver applications to mobile devices, view state is not sent to the client device, but instead it is stored in the **Session** object automatically by the ASP.NET runtime.

The following procedure describes how to use view state in an ASP.NET Web page.

► To use ViewState

1. Write code in your ASP.NET Web page, to assign a value to a **ViewState** property for your Web page.

   If a **ViewState** setting with the specified name does not already exist, it is created automatically. The following example creates a **ViewState** property named "username," and sets its value to "myUserName."

   ```
   ViewState["username"] = "myUserName";
   ```

2. On a subsequent postback, retrieve the required information from the view state and cast it to the appropriate type.

```
string userName = ViewState["username"].ToString();
```

Using view state in this way makes it easy to persist small amounts of state between page postbacks on pages where controls have server-side event handlers.

Storing State in the Application Object

The **System.Web.UI.Page** class, that all ASP.NET pages inherit from, provides an **Application** property through which you can share state between all users and all sessions of an application.

Use the **Application** object when:

- You have to manage application-wide state that is not specific to any particular user or session.
- The state does not have to survive the life of the process the application is running in.
- The state is sensitive and cannot be passed across the network.
- You have substantial amounts of read-only state that you can load into the **Application** object at the start of application execution. An alternative solution might be to cache the data; for more information, see "Caching State" later in this chapter.

Before deciding to use the **Application** object, consider the following points:

- The **Application** object is a dictionary-based, in-memory storage mechanism, meaning that it is fast, but storing large amounts of data affects the performance of your server.
- Because the state is stored on the server, the **Application** object offers higher levels of security than the client-based state mechanisms such as cookies, hidden form fields, and query strings; however, because all application code can access the **Application** object, it does not provide as secure a solution as the **Session** object.
- You must synchronize access to **Application** state data, because other threads may be accessing the state at the same time. This means the **Application** object is best for read-only state.
- Unlike the **Session** object, you cannot specify a timeout value after which the **Application** object state expires. Data in the **Application** object is retained until you remove it or until the application terminates.

The following section describes how to store state in the **Application** object.

Using the Application Object

The **System.Web.UI.Page** class has an **Application** property that returns a **System.Web.HttpApplicationState** object; through this object, you can manipulate the contents of the **Application** object. State items are held as **Object** references in **HttpApplicationState**.

The **HttpApplicationState** class provides properties and methods to manipulate the collection's contents, enabling you to add and remove collection elements, get and set specific data elements, enumerate the collection, and clear the collection.

The following code sample shows how to access **Application** object state, by using the C# indexer of the **HttpApplicationState** class.

```
Application.Lock();
if (Application["hitCount"] == null)
{
  Application["hitCount"] = 1;
}
else
{
  Application["hitCount"] = 1 + (int)Application["hitCount"];
}
Application.UnLock();
```

Note the use of the **Lock** and **UnLock** methods to serialize access to the **Application** object state. Note also that the data is cast to the appropriate data type (in this case an integer) when retrieved.

Serializing State

When you store state out-of-process, such as in a state server or in SQL Server, the data must be serialized. Depending on the type of data that you store, this might lead to a major performance hit.

If you store basic types such as integers, strings, and GUIDs, the .NET Framework uses optimized methods to serialize the data. However, if you want to store non-trivial types such as a **Hashtable**, the framework uses the **BinaryFormatter** by default to perform serialization. This method of serialization is slower than that used for basic types.

► **To enable custom types to be serialized**

1. Annotate your custom type with the **[Serializable]** attribute.

   For example, the following code defines a serializable class named CustomerDetails.

   ```
   [Serializable]
   public class CustomerDetails
   ```

```
{
  // Plus other members ...
}
```

2. Declare fields, properties, and methods in the class as usual.

   Note that **BinaryFormatter** serializes all the fields of a class—even fields marked as private. In this respect, binary serialization differs from the **XmlSerializer** class; the **XmlSerializer** class serializes only public fields.

3. If you want selective fields to be excluded from binary serialization, annotate the fields with the **[NonSerialized]** attribute.

   The following example defines three fields; the mName and mAddress fields will be serialized, but the mLastUpdate field will not be serialized.

```
[Serializable]
public class CustomerDetails
{
  private string mName;
  private string mAddress;
  [NonSerialized] private DateTime mLastUpdate;
  // Plus other members ...
}
```

If your custom type is particularly large, or if it contains information that can be summarized into a more concise format, consider defining a custom serialization format for your type.

► **To define a custom serialization format**

1. Implement the **ISerializable** interface in your custom type.
2. Provide a **GetObject** method to define how you want the object's information to be serialized.
3. Provide a deserialization constructor to enable a copy of the object to be reconstituted during deserialization.

For more information about how to perform custom serialization, see "Custom Serialization" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcustomserialization.asp*).

Caching State

Caching is a technique used to improve the performance and scalability of applications. Caching is most useful in server-based applications, in particular Web applications, but it also applies to many smart client situations. The purpose of caching is to keep state that is expensive to create or retrieve in a more easily accessible form or location. For example, when a user has an active session on your

Web application, if you keep a copy of their profile in memory instead of always reading from a remote SQL Server, your application performance improves.

You must consider your caching requirements when you design the state management aspects of your application. Caching can be difficult to implement in an existing application, and the capabilities of the caching mechanism must be appropriate to the lifetime, scope, and type of state you have to manage.

A complete discussion of caching is beyond the scope of this guide. For comprehensive coverage, see the "Caching Architecture Guide for .NET Framework Applications" on MSDN (*http://msdn.microsoft.com/architecture/application/default.aspx?pull=/library/en-us/dnbda/html/CachingArch.asp*).

Summary

State management is a crucial consideration in Web applications. You must choose the appropriate storage locations and mechanisms to manage state in your presentation layer to attain the optimal levels of security, scalability, manageability, and extensibility.

6

Multithreading and Asynchronous Programming in Web Applications

In This Chapter

This chapter describes how to use two closely related mechanisms to enable you to design scaleable and responsive presentation layers for ASP.NET Web applications. The two mechanisms are:

- Multithreading
- Asynchronous programming

Performance and responsiveness are important factors in the success of your application. Users quickly tire of using even the most functional application if it is unresponsive or regularly appears to freeze when the user initiates an action. Even though it may be a back-end process or external service causing these problems, it is the user interface where the problems become evident.

Multithreading and asynchronous programming techniques enable you to overcome these difficulties. The Microsoft .NET Framework class library makes these mechanisms easily accessible, but they are still inherently complex, and you must design your application with a full understanding of the benefits and consequences that these mechanisms bring. In particular, you must keep in mind the following points as you decide whether to use one of these threading techniques in your application:

- More threads does not necessarily mean a faster application. In fact, the use of too many threads has an adverse effect on the performance of your application. For more information, see "Using the Thread Pool" later in this chapter.
- Each time you create a thread, the system consumes memory to hold context information for the thread. Therefore, the number of threads that you can create is limited by the amount of memory available.

- Implementation of threading techniques without sufficient design is likely to lead to overly complex code that is difficult to scale and extend.
- You must be aware of what could happen when you destroy threads in your application, and make sure you handle these possible outcomes accordingly.
- Threading-related bugs are generally intermittent and difficult to isolate, debug, and resolve.

The following sections describe multithreading and asynchronous programming from the perspective of presentation layer design in ASP.NET Web applications. For information about how to use these mechanisms in Windows Forms-based applications, see "Multithreading and Asynchronous Programming in Windows Forms-Based Applications" in the appendix of this guide.

Multithreading

There are many situations where using additional threads to execute tasks allows you to provide your users with better performance and higher responsiveness in your application, including:

- When there is background processing to perform, such as waiting for authorization from a credit-card company in an online retailing Web application
- When you have a one-way operation, such as invoking a Web service to pass data entered by the user to a back-end system
- When you have discrete work units that can be processed independently, such as calling several SQL stored procedures simultaneously to gather information that you require to build a Web response page

Used appropriately, additional threads allow you to avoid your user interface from becoming unresponsive during long-running and computationally intensive tasks. Depending on the nature of your application, the use of additional threads can enable the user to continue with other tasks while an existing operation continues in the background. For example, an online retailing application can display a "Credit Card Authorization In Progress" page in the client's Web browser while a background thread at the Web server performs the authorization task. When the authorization task is complete, the background thread can return an appropriate "Success" or "Failure" page to the client. For an example of how to implement this scenario, see "How to: Execute a Long-Running Task in a Web Application" in Appendix B of this guide.

Note: Do not display visual indications of how long it will take for a long-running task to complete. Inaccurate time estimations confuse and annoy users. If you do not know the scope of an operation, distract the user by displaying some other kind of activity indictor, such as an animated GIF image, promotional advertisement, or similar page.

Unfortunately, there is a run-time overhead associated with creating and destroying threads. In a large application that creates new threads frequently, this overhead can affect the overall application performance. Additionally, having too many threads running at the same time can drastically decrease the performance of a whole system as Windows tries to give each thread an opportunity to execute.

Using the Thread Pool

A common solution to the cost of excessive thread creation is to create a reusable pool of threads. When an application requires a new thread, instead of creating one, the application takes one from the thread pool. As the thread completes its task, instead of terminating, the thread returns to the pool until the next time the application requires another thread.

Thread pools are a common requirement in the development of scaleable, high-performance applications. Because optimized thread pools are notoriously difficult to implement correctly, the .NET Framework provides a standard implementation in the **System.Threading.ThreadPool** class. The thread pool is created the first time you create an instance of the **System.Threading.ThreadPool** class.

The runtime creates a single thread pool for each run-time process (multiple application domains can run in the same runtime process.) By default, this pool contains a maximum of 25 worker threads and 25 asynchronous I/O threads per processor (these sizes are set by the application hosting the common language runtime).

Because the maximum number of threads in the pool is constrained, all the threads may be busy at some point. To overcome this problem, the thread pool provides a queue for tasks awaiting execution. As a thread finishes a task and returns to the pool, the pool takes the next work item from the queue and assigns it to the thread for execution.

Benefits of Using the Thread Pool

The runtime-managed thread pool is the easiest and most reliable approach to implement multithreaded applications. The thread pool offers the following benefits:

- You do not have to worry about thread creation, scheduling, management, and termination.
- Because the thread pool size is constrained by the runtime, the chance of too many threads being created and causing performance problems is avoided.
- The thread pool code is well tested and is less likely to contain bugs than a new custom thread pool implementation.
- You have to write less code, because the thread start and stop routines are managed internally by the .NET Framework.

The following procedure describes how to use the thread pool to perform a background task in a separate thread.

► **To use the thread pool to perform a background task**

1. Write a method that has the same signature as the **WaitCallback** delegate. This delegate is located in the **System.Threading** namespace, and is defined as follows.

```
[Serializable]
public delegate void WaitCallback(object state);
```

2. Create a **WaitCallback** delegate instance, specifying your method as the callback.
3. Pass the delegate instance into the **ThreadPool.QueueUserWorkItem** method to add your task to the thread pool queue. The thread pool allocates a thread for your method from the thread pool and calls your method on that thread.

In the following code, the AuthorizePayment method is executed in a thread allocated from the thread pool.

```
using System.Threading;

public class CreditCardAuthorizationManager
{
  private void AuthorizePayment(object o)
  {
    // Do work here ...
  }

  public void BeginAuthorizePayment(int amount)
  {
    ThreadPool.QueueUserWorkItem(new WaitCallback(AuthorizePayment));
  }
}
```

For a more detailed discussion of the thread pool, see "Programming the Thread Pool in the .NET Framework" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/progthrepool.asp*).

Limitations of Using the Thread Pool

Unfortunately, the thread pool suffers limitations resulting from its shared nature that may prevent its use in some situations. In particular, these limitations are:

- The .NET Framework also uses the thread pool for asynchronous processing, placing additional demands on the limited number of threads available.
- Even though application domains provide robust application isolation boundaries, code in one application domain can affect code in other application domains in the same process if it consumes all the threads in the thread pool.

- When you submit a work item to the thread pool, you do not know when a thread becomes available to process it. If the application makes particularly heavy use of the thread pool, it may be some time before the work item executes.
- You have no control over the state and priority of a thread pool thread.
- The thread pool is unsuitable for processing simultaneous sequential operations, such as two different execution pipelines where each pipeline must proceed from step to step in a deterministic fashion.
- The thread pool is unsuitable when you need a stable identity associated with the thread, for example if you want to use a dedicated thread that you can discover by name, suspend, or abort.

In situations where use of the thread pool is inappropriate, you can create new threads manually. Manual thread creation is significantly more complex than using the thread pool, and it requires you to have a deeper understanding of the thread lifecycle and thread management. A discussion of manual thread creation and management is beyond the scope of this guide. For more information, see "Threading" in the ".NET Framework Developer's Guide" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconthreading.asp*).

Synchronizing Threads

If you use multiple threads in your applications, you must address the issue of thread synchronization. Consider the situation where you have one thread iterating over the contents of a hash table and another thread that tries to add or delete hash table items. The thread that is performing the iteration is having the hash table changed without its knowledge; this causes the iteration to fail.

The ideal solution to this problem is to avoid shared data. In some situations, you can structure your application so that threads do not share data with other threads. This is generally possible only when you use threads to execute simple one-way tasks that do not have to interact or share results with the main application. The thread pool described earlier in this chapter is particularly suited to this model of execution.

Synchronizing Threads by Using a Monitor

It is not always feasible to isolate all the data a thread requires. To get thread synchronization, you can use a **Monitor** object to serialize access to shared resources by multiple threads. In the hash table example cited earlier, the iterating thread would obtain a lock on the **Hashtable** object using the **Monitor.Enter** method, signaling to other threads that it requires exclusive access to the **Hashtable**. Any other thread that tries to obtain a lock on the **Hashtable** waits until the first thread releases the lock using the **Monitor.Exit** method.

The use of **Monitor** objects is common, and both Visual C# and Visual Basic .NET include language level support for obtaining and releasing locks:

- In C#, the **lock** statement provides the mechanism through which you obtain the lock on an object as shown in the following example.

```
lock (myHashtable)
{
  // Exclusive access to myHashtable here...
}
```

- In Visual Basic .NET, the **SyncLock** and **End SyncLock** statements provide the mechanism through which you obtain the lock on an object as shown in the following example.

```
SyncLock (myHashtable)
  ' Exclusive access to myHashtable here...
End SyncLock
```

When entering the **lock** (or **SyncLock**) block, the static (**Shared** in Visual Basic .NET) **System.Monitor.Enter** method is called on the specified expression. This method blocks until the thread of execution has an exclusive lock on the object returned by the expression.

The **lock** (or **SyncLock**) block is implicitly contained by a **try** statement whose **finally** block calls the static (or **Shared**) **System.Monitor.Exit** method on the expression. This ensures the lock is freed even when an exception is thrown. As a result, it is invalid to branch into a **lock** (or **SyncLock**) block from outside of the block.

For more information about the **Monitor** class, see "Monitor Class" in the ".NET Framework Class Library" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemthreadingmonitorclasstopic.asp*).

Using Alternative Thread Synchronization Mechanisms

The .NET Framework provides several other mechanisms that enable you to synchronize the execution of threads. These mechanisms are all exposed through classes in the **System.Threading** namespace. The mechanisms relevant to the presentation layer are listed in Table 6.1.

Table 6.1: Thread Synchronization Mechanisms

Mechanism	Description	Links for More Information
ReaderWriterLock	Defines a lock that implements single-writer/multiple-reader semantics; this allows many readers, but only a single writer, to access a synchronized object. Used where classes do much more reading than writing.	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemthreadingreaderwriterlockclasstopic.asp*
AutoResetEvent	Notifies one or more waiting threads that an event has occurred. When the **AutoResetEvent** transitions from a non-signaled to signaled state, it allows only a single waiting thread to resume execution before reverting to the non-signaled state.	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemthreadingautoreseteventclasstopic.asp*
ManualResetEvent	Notifies one or more waiting threads that an event has occurred. When the **ManualResetEvent** transitions from a non-signaled to signaled state, all waiting threads are allowed to resume execution.	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemthreadingmanualreseteventclasstopic.asp*
Mutex	A **Mutex** can have a name; this allows threads in other processes to synchronize on the **Mutex**; only one thread can own the **Mutex** at any particular time providing a machine-wide synchronization mechanism. Another thread can obtain the **Mutex** when the owner releases it. Principally used to make sure only a single application instance can be run at the same time.	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemthreadingmutexclasstopic.asp*

With such a rich selection of synchronization mechanisms available to you, you must plan your thread synchronization design carefully and consider the following points:

- It is a good idea for threads to hold locks for the shortest time possible. If threads hold locks for long periods of time, the resulting thread contention can become a major bottleneck, negating the benefits of using multiple threads in the first place.
- Be careful about introducing deadlocks caused by threads waiting for locks held by other threads. For example, if one thread holds a lock on object A and waits for a lock on object B, while another thread holds a lock on object B, but waits to lock object A, both threads end up waiting forever.
- If for some reason an object is never unlocked, all threads waiting for the lock end up waiting forever. The **lock** (C#) and **SyncLock** (Visual Basic .NET) statements make sure that a lock is always released even if an exception occurs. If you use **Monitor.Enter** manually, you must make sure that your code calls **Monitor.Exit**.

Using multiple threads can significantly enhance the performance of your presentation layer components, but you must make sure you pay close attention to thread synchronization issues to prevent locking problems.

Troubleshooting

The difficulties in identifying and resolving problems in multi-threaded applications occur because the CPU's scheduling of threads is non-deterministic; you cannot reproduce the exact same code execution sequence across multiple test runs. This means that a problem may occur one time you run the application, but it may not occur another time you run it. To make things worse, the steps you typically take to debug an application—such as using breakpoints, stepping through code, and logging—change the threading behavior of a multithreaded program and frequently mask thread-related problems. To resolve thread-related problems, you typically have to set up long-running test cycles that log sufficient debug information to allow you to understand the problem when it occurs.

Note: For more in-depth information about debugging, see "Production Debugging for .NET Framework Applications" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/DBGrm.asp*).

Using Asynchronous Operations

Some operations take a long time to complete. These operations generally fall into two categories:

- I/O bound operations such as calling SQL Server, calling a Web service, or calling a remote object using .NET Framework remoting
- CPU-bound operations such as sorting collections, performing complex mathematical calculations, or converting large amounts of data

The use of additional threads to execute long running tasks is a common way to maintain responsiveness in your application while the operation executes. Because threads are used so frequently to overcome the problem of long running processes, the .NET Framework provides a standardized mechanism for the invocation of asynchronous operations that saves you from working directly with threads.

Typically, when you invoke a method, your application blocks until the method is complete; this is known as synchronous invocation. When you invoke a method asynchronously, control returns immediately to your application; your application continues to execute while the asynchronous operation executes independently. Your application either monitors the asynchronous operation or receives notification by way of a callback when the operation is complete; this is when your application can obtain and process the results.

The fact that your application does not block while the asynchronous operation executes means the application can perform other processing. The approach you use to invoke the asynchronous operation (discussed in the next section) determines how much scope you have for processing other tasks while waiting for the operation to complete.

Using the .NET Framework Asynchronous Execution Pattern

The .NET Framework allows you to execute any method asynchronously using the asynchronous execution pattern. This pattern involves the use of a delegate and three methods named **Invoke**, **BeginInvoke**, and **EndInvoke**.

The following example declares a delegate named **AuthorizeDelegate**. The delegate specifies the signature for methods that perform credit card authorization.

```
public delegate int AuthorizeDelegate(string creditcardNumber,
                                       DateTime expiryDate,
                                       double amount);
```

When you compile this code, the compiler generates **Invoke**, **BeginInvoke**, and **EndInvoke** methods for the delegate. Figure 6.1 on the next page shows how these methods appear in the IL Disassembler.

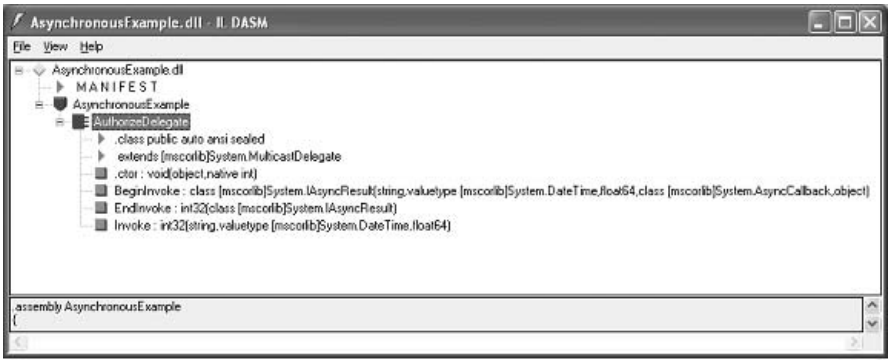


Figure 6.1
MSIL signatures for the Invoke, BeginInvoke, and EndInvoke methods in a delegate

The equivalent C# signatures for these methods are as follows.

```
// Signature of compiler-generated BeginInvoke method
public IAsyncResult BeginInvoke(string creditcardNumber,
                                DateTime expiryDate,
                                double amount,
                                AsyncCallback callback,
                                object asyncState);

// Signature of compiler-generated EndInvoke method
public int EndInvoke(IAsyncResult ar);

// Signature of compiler-generated Invoke method
public int Invoke(string creditcardNumber,
                  DateTime expiryDate,
                  double amount);
```

The following sections describe the **BeginInvoke**, **EndInvoke**, and **Invoke** methods, and clarify their role in the asynchronous execution pattern. For full details on how to use the asynchronous execution pattern, see "Including Asynchronous Calls" in the ".NET Framework Developer's Guide" on MSDN (*http://msdn.microsoft.com /library/default.asp?url=/library/en-us/cpguide/html/cpconasynchronousprogramming.asp*).

Performing Synchronous Execution with the Invoke Method

The **Invoke** method synchronously executes the method referenced by the delegate instance. If you call a method by using **Invoke**, your code blocks until the method returns.

Using **Invoke** is similar to calling the referenced method directly, but there is one significant difference. The delegate simulates synchronous execution by calling **BeginInvoke** and **EndInvoke** internally. Therefore your method is executed in the context of a different thread to the calling code, even though the method appears to execute synchronously. For more information, see the description of **BeginInvoke** in the next section.

Initiating Asynchronous Operations with the BeginInvoke Method

The **BeginInvoke** method initiates the asynchronous execution of the method referenced by the delegate instance. Control returns to the calling code immediately, and the method referenced by the delegate executes independently in the context of a thread from the runtime's thread pool.

The "Multithreading" section earlier in this chapter describes the thread pool in detail; however, it is worth highlighting the consequences of using a separate thread, and in particular one drawn from the thread pool:

- The runtime manages the thread pool. You have no control over the scheduling of the thread, nor can you change the thread's priority.
- The runtime's thread pool contains 25 threads per processor. If you invoke asynchronous operations too liberally, you can easily exhaust the pool causing the runtime to queue excess asynchronous operations until a thread becomes available.
- The asynchronous method runs in the context of a different thread to the calling code. This causes problems when asynchronous operations try to update Windows Forms components.

The signature of the **BeginInvoke** method includes the same arguments as those specified by the delegate signature. It also includes two additional arguments to support asynchronous completion:

- **callback** argument–Specifies an **AsyncCallback** delegate instance. If you specify a non-null value for this argument, the runtime calls the specified callback method when the asynchronous method completes. If this argument is a null reference, you must monitor the asynchronous operation to determine when it is complete. For more information, see "Managing Asynchronous Completion with the EndInvoke Method" later in this chapter.
- **asyncState** argument–Takes a reference to any **object**. The asynchronous method does not use this object, but it is available to your code when the method completes; this allows you to associate useful state information with an asynchronous operation. For example, this object allows you to map results against initiated operations in situations where you initiate many asynchronous operations that use a common callback method to perform completion.

The **IAsyncResult** object returned by **BeginInvoke** provides a reference to the asynchronous operation. You can use the **IAsyncResult** object for the following purposes:

- Monitor the status of an asynchronous operation
- Block execution of the current thread until an asynchronous operation completes
- Obtain the results of an asynchronous operation using the **EndInvoke** method

The following procedure shows how to invoke a method asynchronously by using the **BeginInvoke** method.

► **To invoke a method asynchronously by using BeginInvoke**

1. Declare a delegate with a signature to match the method you want to execute.
2. Create a delegate instance containing a reference to the method you want to execute.
3. Execute the method asynchronously by calling the **BeginInvoke** method on the delegate instance you just created.

The following code fragment demonstrates the implementation of these steps. The example also shows how to register a callback method; this method is called automatically when the asynchronous method completes. For more information about defining callback methods and other possible techniques for dealing with asynchronous method completion, see "Managing Asynchronous Completion with the EndInvoke Method" later in this chapter.

```
public class CreditCardAuthorizationManager
{
  // Delegate, defines signature of method(s) you want to execute asynchronously
  public delegate int AuthorizeDelegate(string creditcardNumber,
                                        DateTime expiryDate,
                                        double amount);

  // Method to initiate the asynchronous operation
  public void StartAuthorize()
  {
      AuthorizeDelegate ad = new AuthorizeDelegate(AuthorizePayment);
      IAsyncResult ar = ad.BeginInvoke(creditcardNumber,
                                       expiryDate,
                                       amount,
                                       new AsyncCallback(AuthorizationComplete),
                                       null);
  }

  // Method to perform a time-consuming operation (this method executes
  // asynchronously on a thread from the thread pool)
  private int AuthorizePayment(string creditcardNumber,
                               DateTime expiryDate,
                               double amount)
  {
    int authorizationCode = 0;

    // Open connection to Credit Card Authorization Service ...
    // Authorize Credit Card (assigning the result to authorizationCode) ...
    // Close connection to Credit Card Authorization Service ...
    return authorizationCode;
  }

  // Method to handle completion of the asynchronous operation
  public void AuthorizationComplete(IAsyncResult ar)
  {
    // See "Managing Asynchronous Completion with the EndInvoke Method"
```

```
    // later in this chapter.
  }
}
```

The following section describes all the possible ways to manage asynchronous method completion.

Managing Asynchronous Completion with the EndInvoke Method

In most situations, you will want to obtain the return value of an asynchronous operation that you initiated. To obtain the result, you must know when the operation is complete. The asynchronous execution pattern provides the following mechanisms to determine whether an asynchronous operation is complete:

- **Blocking**–This is rarely used because it provides few advantages over synchronous execution. One use for blocking is to perform impersonation on a different thread. It is never used for parallelism.
- **Polling**–It is generally a good idea to not use this because it is inefficient; use waiting or callbacks instead.
- **Waiting**–This is typically used for displaying a progress or activity indicator during asynchronous operations.
- **Callbacks**– These provide the most flexibility; this allows you to execute other functionality while an asynchronous operation executes.

The process involved in obtaining the results of an asynchronous operation varies depending on the method of asynchronous completion you use. However, eventually you must call the **EndInvoke** method of the delegate. The **EndInvoke** method takes an **IAsyncResult** object that identifies the asynchronous operation to obtain the result from. The **EndInvoke** method returns the data that you would receive if you called the original method synchronously.

The following sections explore each approach to asynchronous method completion in more detail.

Using the Blocking Approach

To use blocking, call **EndInvoke** on the delegate instance and pass the **IAsyncResult** object representing an incomplete asynchronous operation. The calling thread blocks until the asynchronous operation completes. If the operation is already complete, **EndInvoke** returns immediately.

The following code sample shows how to invoke a method asynchronously, and then block until the method has completed.

```
AuthorizeDelegate ad = new AuthorizeDelegate(AuthorizePayment);
IAsyncResult ar = ad.BeginInvoke(creditcardNumber,  // 1st param into async method
                                 expiryDate,        // 2nd param into async method
                                 amount,            // 3rd param into async method
                                 null,              // No callback
                                 null);             // No additional state

// Block until the asynchronous operation is complete
int authorizationCode = ad.EndInvoke(ar);
```

The use of blocking might seem a strange approach to asynchronous completion, offering the same functionality as a synchronous method call. However, occasionally blocking is a useful approach because you can decide when your thread enters the blocked state as opposed to synchronous execution; synchronous execution blocks immediately. Blocking can be useful if the user initiates an asynchronous operation after which there are a limited number of steps or operations they can perform before the application must have the result of the asynchronous operation.

Using the Polling Approach

To use polling, write a loop that repeatedly tests the completion state of an asynchronous operation using the **IsCompleted** property of the **IAsyncResult** object.

The following code sample shows how to invoke a method asynchronously, and then poll until the method completes.

```
AuthorizeDelegate ad = new AuthorizeDelegate(AuthorizePayment);
IAsyncResult ar = ad.BeginInvoke(creditcardNumber,  // 1st param into async method
                                 expiryDate,        // 2nd param into async method
                                 amount,            // 3rd param into async method
                                 null,              // No callback
                                 null);             // No additional state

// Poll until the asynchronous operation completes
while (!ar.IsCompleted)
{
  // Do some other work...
}

// Get the result of the asynchronous operation
int authorizationCode = ad.EndInvoke(ar);
```

Polling is a simple but inefficient approach that imposes major limitations on what you can do while the asynchronous operation completes. Because your code is in a loop, the user's workflow is heavily restricted, providing few benefits over synchronous method invocation. Polling is really only suitable for displaying a progress

indicator on smart client applications during short asynchronous operations. Generally, it is a good idea to avoid using polling and look instead to using waiting or callbacks.

Using the Waiting Approach

Waiting is similar to blocking, but you can also specify a timeout value after which the thread resumes execution if the asynchronous operation is still incomplete. Using waiting with timeouts in a loop provides functionality similar to polling, but it is more efficient because the runtime places the thread in a CPU efficient sleep instead of using a code level loop.

To use the waiting approach, you use the **AsyncWaitHandle** property of the **IAsyncResult** object. The **AsyncWaitHandle** property returns a **WaitHandle** object. Call the **WaitOne** method on this object to wait for a single asynchronous operation to complete.

The following code sample shows how to invoke a method asynchronously, and then wait for a maximum of 2 seconds for the method to complete.

```
AuthorizeDelegate ad = new AuthorizeDelegate(AuthorizePayment);
IAsyncResult ar = ad.BeginInvoke(creditcardNumber,  // 1st param into async method
                                 expiryDate,        // 2nd param into async method
                                 amount,            // 3rd param into async method
                                 null,              // No callback
                                 null);             // No additional state

// Wait up to 2 seconds for the asynchronous operation to complete
WaitHandle waitHandle = ar.AsyncWaitHandle;
waitHandle.WaitOne(2000, false);

// If the asynchronous operation completed, get its result
if (ar.IsCompleted)
{
  // Get the result of the asynchronous operation
  int authorizationCode = ad.EndInvoke(ar);
  ...
}
```

Despite the advantages, waiting imposes the same limitations as polling—the functionality available to the user is restricted because you are in a loop, even though it is an efficient one. Waiting is useful if you want to show a progress or activity indicator when executing long-running processes that must complete before the user can proceed.

Another advantage of waiting is that you can use the static methods of the **System.Threading.WaitHandle** class to wait on a set of asynchronous operations. You can wait either for the first one to complete (using the **WaitAny** method) or for them all to complete (using the **WaitAll** method). This is very useful if you initiate a number of asynchronous operations at the same time and have to coordinate the

execution of your application based on the completion of one or more of these operations.

Using Callbacks

When you specify an **AsyncCallback** delegate instance in the **BeginInvoke** method, you do not have to actively monitor the asynchronous operation for completion. Instead, when the operation completes, the runtime calls the method referenced by the **AsyncCallback** delegate and passes an **IAsyncResult** object identifying the completed operation. The runtime executes the callback method in the context of a thread from the runtime's thread pool.

The following code sample shows how to invoke a method asynchronously, and specify a callback method that will be called on completion.

```
AuthorizeDelegate ad = new AuthorizeDelegate(AuthorizePayment);
IAsyncResult ar = ad.BeginInvoke(creditcardNumber,
                                 expiryDate,
                                 amount,
                                 new AsyncCallback(AuthorizationComplete),
                                 null);
...

// Method to handle completion of the asynchronous operation
public void AuthorizationComplete(IAsyncResult ar)
{
    // Retrieve the delegate that corresponds to the asynchronous method
    AuthorizeDelegate ad = (AuthorizeDelegate)((AsyncResult)ar).AsyncDelegate;

    // Get the result of the asynchronous method
    int authorizationCode = ad.EndInvoke(ar);
  }
}
```

The great benefit of using callbacks is that your code is completely free to continue with other processes, and it does not constrain the workflow of the application user. However, because the callback method executes in the context of another thread, you face the same threading issues highlighted earlier in the discussion of the **BeginInvoke** method.

Using Built-In Asynchronous I/O Support

I/O is a situation where you frequently use asynchronous method calls. Because of this, many .NET Framework classes that provide access to I/O operations expose methods that implement the asynchronous execution pattern. This saves you from declaring and instantiating delegates to execute the I/O operations asynchronously. The following list identifies the most common scenarios where you would use

asynchronous I/O in your presentation layer and provides a link to a document where you can find implementation details:

- Consuming XML Web services:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconinvokingwebservicesasynchronously.asp
- Calling methods on remote objects using .NET Framework remoting:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconasynchronousremoting.asp
- File access:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconasynchronousfileio.asp
- Network communications:
  - *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconmakingasynchronousrequests.asp*
  - *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconusingnon-blockingclientsocket.asp*
  - *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconusingnon-blockingserversocket.asp*
- Microsoft message queue:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconAsynchronousProcessing.asp?frame=true

Using the built-in asynchronous capabilities of the .NET Framework makes the development of asynchronous solutions easier than it would be to explicitly create delegates to implement asynchronous operations.

Summary

Application performance and scalability can be greatly enhanced using multithreading and asynchronous operations. Wherever possible, try to use these techniques to increase the responsiveness of your presentation layer components.

7

Globalization and Localization

In This Chapter

This chapter describes how to address globalization and localization issues in Web applications. The chapter includes the following sections:

- Understanding Globalization and Localization Issues
- Using Cultures
- Formatting Data
- Creating Localized Resources

Applications might be used in multiple geographical locations and cultures; this can bring many challenges to software architects, particularly when designing the presentation layer. This chapter describes how to address these challenges so that your applications can support as broad a customer base as possible. This chapter also describes how to design and implement your applications so that they can be easily extended to support new cultures when necessary.

For a summary of best practices and recommendations discussed in this chapter, see "Best Practices for Developing World-Ready Applications" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconbestpracticesforglobalapplicationdesign.asp*).

Understanding Globalization and Localization Issues

Globalization is the term given to the process of making sure that an application does not contain any internal dependencies on a particular culture; for example, it is best for a globalized application to not have any hard-coded number formats that can differ across multiple countries, or to assume a particular sorting mechanism for

strings. A globalized application can correctly accept, process, and display a worldwide assortment of scripts, data formats, and languages.

Localization is the process of adapting an application, and in particular the user interface, to suit a specific culture. Localization typically involves tasks such as translating strings into different natural languages, resizing user interface elements to fit on the screen, and regenerating images for specific cultures.

To simplify localization efforts and minimize costs, deal with globalization and localization issues during the design phase of a project. Failure to correctly identify such requirements at design time can lead to expensive and inferior attempts at localization later in development.

There are several issues you must take into account:

- **Natural language issues**–Natural languages differ in display, alphabets, grammar, and syntactical rules. For more information about how this affects globalization and localization efforts, see "Language Issues" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconlanguageissues.asp*).
- **Data formatting issues**–Cultures around the world have different rules for formatting data such as numbers, dates, and times. These issues are described in the "Formatting Data" section later in this chapter. For additional information, see "Formatting Issues" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconformattingissues.asp*).
- **String-related issues**–To support different cultures, user interfaces that display text may have to be amenable to change. For information about the issues to consider, see "String-Related Issues" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconstring-relatedconsiderations.asp*).
- **User interface issues**–In addition to text, user interfaces also typically contain elements such as images, user interface controls, and message boxes. To support localization, you must design these user interface elements carefully. For information about the user interface issues to consider, see "User Interface Issues" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxcondesigninginternational-awareuserinterface.asp)*.

Globalization and localization are particularly important in Web applications because users can potentially access these applications from anywhere in the world. By providing culture-sensitive user interfaces and application logic, you increase the reach of the application and improve the user's experience when they use the application.

Effective globalization and localization helps reduce the effort you expend to develop world-ready Web applications and also simplifies maintenance and extensibility after the application has been deployed.

Additional Information

For a summary of general best practices for globalization and localization, see "Best Practices for Globalization and Localization" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconbestglobalizationpractices.asp*).

For advice on how to test your application for globalization and localization, see "Testing for Globalization and Localization" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxcontestingforglobalizationlocalization.asp*).

Using Cultures

The various rules for languages and countries, such as number formats, currency symbols, and sort orders, are aggregated into a number of standard *cultures*.

Each culture is identified by a culture ID as defined in RFC 1766; each culture ID is made up of a *neutral* culture code indicating the language of the culture and an optional *specific* culture code indicating the country represented by the culture. Neutral culture codes are written in lowercase, for example "en" represents English, while "fr" represents French. Specific culture codes are appended in uppercase, allowing specific language and country combinations such as "en-US" (English in the United States), "en-GB" (English in the United Kingdom), "fr-FR" (French in France) and "fr-CA" (French in Canada) to be represented.

The .NET Framework supports all cultures defined in RFC 1766 in addition to an *invariant* culture for culturally-insensitive data; an invariant culture uses an empty string as the culture ID and is defined as English language with no specific country. A culture is represented programmatically in the .NET Framework using the **System.Globalization.CultureInfo** class.

Identifying the Current Culture

On any particular computer running Windows, the system's current culture is determined by the system regional and language settings. By default, .NET Framework applications inherit the current culture from these settings and use it when performing tasks such as formatting numbers, sorting strings, and displaying currency.

The .NET Framework actually uses a combination of two cultures to handle different aspects of localization:

- **Current culture**–Determines how various data types are formatted, such as numbers and dates.

- **Current user interface culture**–Determines which localized resource file the resource manager loads. For more information about how to create localized resource files, see "Creating Localized Resources" later in this chapter.

These two aspects of localization are represented by the **CurrentCulture** and **CurrentUICulture** static properties of the **CultureInfo** class:

- **CultureInfo.CurrentCulture**–Returns the **Thread.CurrentCulture** property; this property indicates the culture of the currently executing thread.
- **CultureInfo.CurrentUICulture**–Returns the **Thread.CurrentUICulture** property; this property indicates the user interface culture of the currently executing thread.

The following code shows how to get the culture and user interface culture of the currently executing thread.

```
using System.Globalization;
using System.Threading;
…
string currentCulture   = CultureInfo.CurrentCulture.Name;
string currentUICulture = CultureInfo.CurrentUICulture.Name;
```

In addition to retrieving the current culture and user interface culture, it is also possible to change these settings to influence how information is presented to the user. The following section describes how to use an alternative culture.

Using an Alternative Culture

In most Windows-based applications (and on smart-device applications), it is reasonable to assume the system regional and language settings indicate the culture that an individual application uses. However, in some cases you might want to allow the user to select an alternative culture in your application, regardless of the underlying system settings.

For Web applications, the requirement to programmatically use an alternative culture is even more pronounced. The underlying system settings indicate the regional and language settings of the server hosting the application, not necessarily those of the user accessing it.

There are various ways to set the culture and user interface culture in an ASP.NET Web application, depending on your intended scope:

- **Set the culture and user interface culture in Web.config**–Use this approach if you want to set the default culture and user interface culture for all the pages in a Web application. The following fragment from a Web.config file illustrates this technique.

```
<configuration>
   <system.web>
      <globalization culture=»en-US» uiCulture=»de-DE»/>
   </system.web>
</configuration>
```

- **Set the culture and user interface culture in the @ Page directive**–Use this approach if you want to override the default culture and user interface culture for a specific page in a Web application. The following fragment from an .aspx file illustrates this technique.

```
<%@ Page Culture="en-GB" UICulture="Fr-FR" ... %>
```

- **Set the culture and user interface culture programmatically**–Use this approach if you want to select which culture and user interface culture to use at run time.

Note: You cannot change a thread's culture in semi-trusted code; changing the culture requires a **SecurityPermission** with the **SecurityPermissionFlag,ControlThread** set. Manipulating threads is dangerous because of the security state associated with threads. Therefore, this permission should be given only to trustworthy code, and then only as necessary.

The following code in an ASP.NET Web page retrieves the user's language preferences from the **Request.UserLanguages** property and uses the culture and user interface culture for the preferred language.

```
using System.Globalization;
using System.Threading;

// Set the culture to the browser's accept language
Thread.CurrentThread.CurrentCulture =
     CultureInfo.CreateSpecificCulture(Request.UserLanguages[0]);

// Set the user interface culture to the browser's accept language
Thread.CurrentThread.CurrentUICulture =
     new CultureInfo(Request.UserLanguages[0]);
```

Note: You cannot change the **CurrentCulture** or **CurrentUICulture** properties when programming with the .NET Compact Framework. If you have to support per-application localization on smart-client devices such as Pocket PCs, you must use a **CultureInfo** object to store the user-selected culture internally, and use it explicitly whenever loading resource files or formatting data.

Formatting Data

Different cultures format data in different ways. An example of this is the formatting of numbers in different currencies, but there are many other data type-specific issues that must be considered when using multiple cultures:

- Localizing string data
- Localizing numeric data
- Localizing date and time data

The following sections describe each of these localization issues.

Localizing String Data

Different cultures use different character sets to represent string values. Generally, a globalized application stores all strings internally as Unicode and converts them to the appropriate character set code-pages only if necessary to display them (for example by sending a code-page specific array of bytes to a browser). You can use the static **GetEncoding** method of the **System.Text.Encoding** class to retrieve an **Encoding** object for a specific code-page, and then use its **GetBytes** method to convert Unicode strings to a code-page specific byte array.

Another aspect of dealing with localized string data is that sort orders are culture-specific. For example, an application that sorts the names Ändré and Andrew produces different results depending on the culture used (in German cultures, Ändré is at the beginning; while in Swedish cultures, Andrew is first). The static **Sort** method of the **Array** class automatically sorts according to the **Thread.CurrentThread.CurrentCulture** setting.

You can also compare strings using the static **Compare** method of the **String** class; this allows a **CultureInfo** object to be passed so that the sort order from a culture other than the current culture can be used. Alternatively, the **CultureInfo** class has a **CompareInfo** object property; this provides a **Compare** method similar to that of the **String** class.

Localizing Numeric Data

You can display localized number formats using the **ToString** method of the base numeric types (such as **int**, **long**, and **double**) or any derived class that implements the **IFormattable** interface.

With the **ToString** method, you can pass a **format** parameter to indicate the kind of numeric formatting required (such as currency or decimal) to format the number according to the **Thread.CurrentThread.CurrentCulture** setting.

The following example formats a **double** value as a currency by passing the **"c"** format string to **ToString**; if the current culture is en-US, the result string is formatted as "$12,345,678,910.11."

```
double number = 12345678910.11;
string resultString = number.ToString("c");
```

You can also pass a **CultureInfo** object in the **provider** parameter of the **ToString** method to format the number for an alternative culture. The following example formats a **double** value as a currency using the "fr-FR" culture. The result string is formatted as "12 345 678 910,11 ."

```
double number = 12345678910.11;
string resultString = number.ToString("c", new CultureInfo("fr-FR"));
```

For a complete list of the numeric format strings, see "Standard Numeric Format Strings" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconstandardnumericformatstrings.asp*).

Localizing Date and Time Data

Date and time data presents three challenges in respect to localization:

- Formatting dates
- Using different calendars
- Handling time zones

The following sections describe how to address each of these issues.

Formatting Dates

To display dates in an appropriate localized format, you can use any of the **ToXxxxString** methods (such as **ToShortDateString** or **ToShortTimeString**) provided by the **DateTime** structure; these methods automatically format the date according to the **Thread.CurrentThread.CurrentCulture** setting.

Formating Dates and Times for a Particular Culture

The following example sets the current culture to "fr-FR," and then formats a **DateTime** object as a short and long date string, followed by a short and long time string. Sample results are shown in each case.

```
Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR");
DateTime dt = DateTime.Now;

string shortDateString = dt.ToShortDateString();  // "31/12/2003"
string longDateString  = dt.ToLongDateString();   // "mercredi 31 décembre 2003"

string shortTimeString = dt.ToShortTimeString();  // "09:32"
string longTimeString  = dt.ToLongTimeString();   // "09:32:57"
```

Customizing Date and Time Formatting for a Particular Culture

If you want more control over how the date or time is formatted, you can use the **DateTimeFormat** property of a **CultureInfo** object to obtain a **DateTimeFormatInfo** object for that culture. **DateTimeFormatInfo** has properties and methods that enable you to specify how to format items such as long dates, short dates, long times, and short times.

The following example modifies the long date pattern for the current culture, so that long dates appear in the format [year—monthName—day].

```
DateTimeFormatInfo fmt = Thread.CurrentThread.CurrentCulture.DateTimeFormat;
fmt.LongDatePattern = @"\[yyyy-MMMM-dd\]";

string longDateString = DateTime.Now.ToLongDateString();  // «[2003-décembre-31]»
```

Parsing Culture-Specific Date and Time Strings

To read a date from a string into a **DateTime** object, you can use the static **Parse** method of the **DateTime** structure. By default, the **Parse** method assumes that the date is formatted according to the **Thread.CurrentThread.CurrentCulture** setting, but you can also pass a **CultureInfo** object to read dates in formats for an alternative culture.

The following example parses a date string that has been formatted using the "fr-FR" culture.

```
DateTime dt = DateTime.Parse(aFrenchDateString, new CultureInfo("fr-FR"));
```

Using Different Calendars

Although most people are familiar with the Gregorian calendar, some cultures provide alternative calendars that your application might have to support. A globalized application typically displays and uses calendars based on the current culture.

The .NET Framework provides a **Calendar** class that you can use to perform calendar-related operations. Each culture has a default calendar (available from the **Calendar** property of the **CultureInfo** object) and a collection of optional calendars (available from the **OptionalCalendars** property of the **CultureInfo** object). The .NET Framework includes the following **Calendar** implementations: **GregorianCalendar**, **HebrewCalendar**, **HijriCalendar**, **JapaneseCalendar**, **JulianCalendar**, **KoreanCalendar**, **TaiwanCalendar**, and **ThaiBuddhistCalendar**.

Enumerating Calendars for a Culture

The following example retrieves the default calendar and any optional calendars for the current culture. Note that **GregorianCalendar** has a **CalendarType** property; this indicates the language version of the **GregorianCalendar**.

```
// Display details for the default calendar for the current culture
Calendar calendar = Thread.CurrentThread.CurrentCulture.Calendar;
Console.Write("Default calendar:  {0}", calendar);
if (calendar is GregorianCalendar)
  Console.Write(", subtype: {0}", ((GregorianCalendar)calendar).CalendarType);

// Display details for all the optional calendars for the current culture
Calendar[] calendars = Thread.CurrentThread.CurrentCulture.OptionalCalendars;
foreach (Calendar cal in calendars)
{
  Console.Write("\nOptional calendar: {0}", cal);

  if (cal is GregorianCalendar)
    Console.Write(", subtype: {0}", ((GregorianCalendar)cal).CalendarType);
}
```

The following sample output shows the default calendar and optional calendars for the "ja-JP" culture.

```
Default calendar:  System.Globalization.GregorianCalendar, subtype: Localized
Optional calendar: System.Globalization.JapaneseCalendar
Optional calendar: System.Globalization.GregorianCalendar, subtype: USEnglish
Optional calendar: System.Globalization.GregorianCalendar, subtype: Localized
```

Interpreting Dates and Times for a Culture

The following example shows how to interpret dates and times correctly for a particular calendar. The culture is set to "he-IL" (Hebrew in Israel), and the calendar is set to the **HebrewCalendar**. A **DateTime** object is then created using the "he-IL" culture, and the **HebrewCalendar** is used to retrieve the year according to the Hebrew calendar. The year is also obtained directly by using the **DateTime.Year** property; note that the **DateTime** structure always returns date information in the **GregorianCalendar**.

```
// Set the current culture to "he-IL"
CultureInfo he = new CultureInfo("he-IL");
Thread.CurrentThread.CurrentCulture = he;

// Use the Hebrew calendar for date-and-time formatting
he.DateTimeFormat.Calendar = new HebrewCalendar();

// Create a DateTime, using HebrewCalendar rules
DateTime dt = new DateTime(5763, 11, 4, he.DateTimeFormat.Calendar);

// Retrieve the year, using HebrewCalendar rules
Console.WriteLine("Hebrew year: {0}", he.DateTimeFormat.Calendar.GetYear(dt));

// Retrieve the year, using GregorianCalendar rules
Console.WriteLine("Gregorian year: {0}", dt.Year);
```

The preceding example produces the following output.

```
Hebrew year: 5763
Gregorian year: 2003
```

Displaying Calendar Controls for a Culture

When you add a **Calendar** control to an ASP.NET Web page, the calendar is displayed using the current culture. Figure 7.1 shows how a **Calendar** control appears in the "en-US" culture.

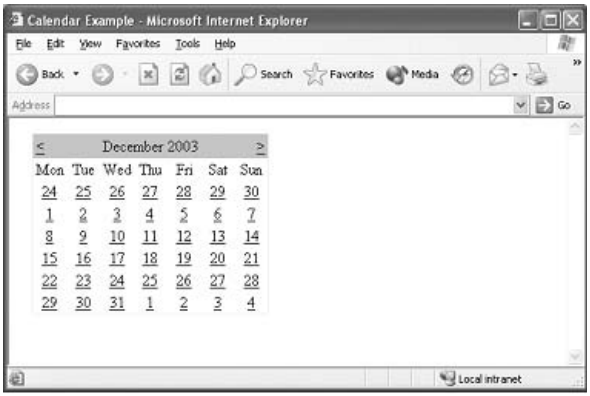


Figure 7.1
Calendar control in an ASP.NET Web form, using the "en-US" culture

To display the calendar control in a culture-specific fashion, add the following code to the **Page_Load** method (or another appropriate location) in your Web application.

```
// Set the culture to "Hebrew in Israel"
CultureInfo cultureInfo = new CultureInfo("he-IL");
Thread.CurrentThread.CurrentCulture = cultureInfo;

// Use the Hebrew calendar for formatting and interpreting dates and times
cultureInfo.DateTimeFormat.Calendar = new HebrewCalendar();
```

Figure 7.2 shows how the **Calendar** control appears in the "he-IL" culture.

The **Calendar** control in Figure 7.2 illustrates two aspects of localization. First, the month name and year number are displayed in Hebrew because the "he-IL" culture is being used. Second, the days of the month are displayed in Hebrew numerals because a **HebrewCalendar** object has been assigned to the **cultureInfo.DateTimeFormat.Calendar** property.

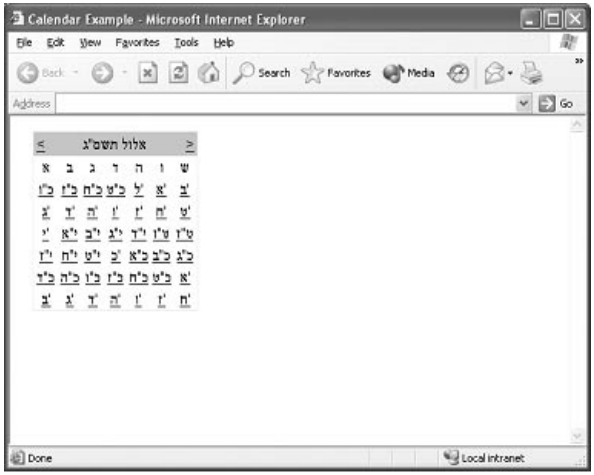


Figure 7.2
Calendar control in an ASP.NET Web form, using the "he-IL" culture

Handling Time Zones

Some applications might be used in multiple time zones. For example, a scheduling application might allow a user to schedule an event while in one time zone and automatically update the event details when the user moves to another time zone. To handle this kind of functionality, the .NET Framework supports *universal time*, a neutral time zone that can be used for the internal storage of date/time values.

You can use the **ToUniversalTime** and **ToLocalTime** methods of the **DateTime** structure to convert date/time values between the local time for the current system time zone and universal time. Additionally, the static **Parse** and **ParseExact** methods of the **DateTime** structure allow you to specify a styles parameter of **DateTimeStyles.AdjustToUniversal** to automatically read a date in a local time zone and adjust it to universal time. This makes it easier to accept input in the local **DateTime** format and to use universal time internally.

Persisting Dates and Times Using the Invariant Culture

When persisting a date as a string, it is a good idea to use the invariant culture to format it — this ensures that a consistent format is always used internally, regardless of the culture used for presenting dates to the user. For example, you can use the following code to persist a date/time value so that it can be adjusted for local time zones.

```
// Function to save DateTime value
private void SaveDateTime(DateTime localDateTime)
{
    // Convert the value to universal time
    DateTime udtDate = localDateTime.ToUniversalTime();

    // Format the date string using the invariant culture
    string dataToSave = udtDate.ToString(CultureInfo.InvariantCulture);
```

```
    // Save the data
    StreamWriter f = new StreamWriter(@"C:\SavedDate.txt");
    f.WriteLine(dataToSave);
    f.Close();
}

// Function to load a saved time
private DateTime LoadDateTime()
{
    // Load the data
    StreamReader f = new StreamReader(@"C:\SavedDate.txt");
    string loadedData = f.ReadToEnd();
    f.Close();

    // Create a DateTime object based on the string value formatted for
    // the invariant culture
    DateTime udtDate = DateTime.Parse(loadedData,CultureInfo.InvariantCulture);

    // Convert the value to local time and return it
    return udtDate.ToLocalTime();
}
```

With this approach, you can handle all internal date/time storage neutrally, only applying culture-specific formats and local time zones when required.

Creating Localized Resources

You can create localized collections of string and binary data that the .NET Framework common language runtime loads automatically depending on the **Thread.CurrentThread.CurrentUICulture** setting. This approach is typically used to create localized user interfaces, although it can also be used to manage any set of strings or binary data that require localization in your application.

To localize a user interface, avoid hard-coding localizable strings or images in your ASP.NET Web Forms or Windows Forms; store this data in localized resource files instead. Each application has a default set of resources; this set is used when no culture-specific resource file is available.

This section focuses on how to create custom resource files for ASP.NET Web applications and for other situations where you want to localize non-user interface data. For information about how to localize Windows Forms, see "How to Localize Windows Forms" in Appendix B in this guide.

Creating Custom Resource Files

If your application is not a Windows Forms-based application, or if you want to localize non-user interface data, you can create custom resource files. Custom resource files are compiled as satellite assemblies; you use the **System.Resources.ResourceManager** class to load the appropriate resources for the **Thread.CurrentThread.CurrentUICulture** setting.

Creating Resource Files

You can create resource files in three different ways:

- **Text (.txt) files**–Text files contain string resources in **name=value** pairs.

  You cannot directly embed a .txt file in an assembly; you must convert the .txt file into a .resources file using the Resource File Generator, Resgen.exe.

- **.resx files**–.resx files are XML files that contain references to strings and objects. When you view a .resx file, you can see the binary form of embedded objects (such as pictures) as long as the binary information is in the resource manifest.

  You cannot directly embed a .resx file in an assembly; you must convert the .resx file into a .resources file using the Resource File Generator, Resgen.exe.

- **.resources files**–.resources files contain references to strings and objects. You can create .resources files programmatically by using methods in the **System.Resources.ResourceWriter** class. You can also generate .resources files from .txt and .resx files by using the Resource File Generator, Resgen.exe. You can read .resources files programmatically by using methods in the **System.Resources.ResourceReader** class

For more information about these three types of resource files and how to use the Resgen.exe tool, see "Creating Resource Files" on MSDN (*http://msdn.microsoft.com /library/default.asp?url=/library/en-us/cpguide/html/cpconcreatingresourcefiles2.asp*).

Visual Studio .NET allows you to add XML .resx resource files to a project and automatically generates and embeds a binary version of the file when the project is compiled. You can use the resource editing window in Visual Studio .NET to add string and encoded binary resources to a resource file.

You must create a default resource file using the naming format **ResourceFileName.resx**; this file is used when no specific resource file can be found for the current user interface culture. For example, an assembly might contain a default resource file named **MyResources.resx**. Resource files for specifically supported cultures must be named using the format **ResourceFileName.Culture_ID.resx**. For example, a resource file for the "English in the UK" culture might be named **MyResources.en-GB.resx**.

Create the same list of resources in each resource file. It is a good idea for each resource file to use the same **name** property for each resource, with a localized **value**. For example, you might have a resource with the name *welcomeMessage*; this resource has the value "Welcome" in a **MyResources.en-GB.resx** file and the value "Bienvenue" in a **MyResources.fr-FR.resx** file.

When the solution is compiled, the default resource file is embedded in the application assembly and a satellite assembly for each culture-specific resource file is created and linked to the main assembly.

At run time, the common language runtime loads the most appropriate resources for the current user interface culture. If a specific culture resource file (that is, a resource file for a specific language and geographical location such as "en-GB") is available for the current user interface culture, it is loaded. If not, the runtime looks for an appropriate language culture (such as "en") and loads that. If no language culture resource file is available, the runtime uses the default resource file embedded in the main assembly. Note that this "closest match" approach allows you to make your localization language- and location-specific (for example, by having different resources for English speakers in the U.K., English speakers in the U.S., French speakers in Canada, and French speakers in France) or just language-specific (for example, by having different resources for English speakers and French speakers).

Retrieving Localized Resources

To retrieve the appropriate resource at run time, you can use the **System.Resources.ResourceManager** class. To create a **ResourceManager** object to load resources from satellite assemblies (or the main assembly if the default resource file is required), use the constructor that allows you to specify **baseName** and **assembly** parameters:

- The **baseName** is the root name of the resource files, including the namespace. For example, the **baseName** for the MyResources.resx and MyResources.en-GB.resources resource files in an assembly that uses the namespace "MyApplication" is "MyApplication.MyResources."
- The **assembly** parameter is used to specify the main assembly for the resources. This is generally the assembly that the code to create the **ResourceManager** object resides in, and you can specify the current assembly by passing **this.GetType().Assembly** to the constructor.

The following sample code shows how to create a **ResourceManager** object.

```
using System.Resources;
…
ResourceManager resMan =
  new ResourceManager("MyApplication.MyResources", this.GetType().Assembly);
```

You can retrieve resources by using the **GetString** and **GetObject** methods of the **ResourceManager** class and passing the name of the resource to be retrieved. For example, the appropriate *welcomeMessage* string resource for the current culture (as defined in the **CultureInfo.CurrentUICulture** object) can be retrieved using the following code.

```
string welcomeMessage = resMan.GetString("welcomeMessage");
```

The **GetString** method is overloaded to allow you to pass a **CultureInfo** object and therefore retrieve a resource for an alternative culture.

Additional Information

For more information about how to retrieve data from all kinds of resource files, see "Best Practices for Developing World-Ready Applications" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconbestpracticesforglobalapplicationdesign.asp*).

For additional information about how to use resources to localize the layout and presentation of an ASP.NET Web page, see "Enterprise Localization Toolkit" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspp/html/entloctoolkit.asp*). The Toolkit enables you to extend the .NET Framework foundation, and it includes code to manage resource strings through a database, localize complex data types, and build resource files automatically.

Summary

Globalization and localization are important factors in Web applications. By careful thought, you can design your user interface in a suitably generic manner so that it can be easily localized for different cultures. You can also take advantage of the extensive support provided by the .NET Framework class library to write code that is sensitive to the current culture.

Appendix A

Securing and Operating the Presentation Layer

In This Appendix

This appendix describes security and operational issues relating to the presentation layer. It includes the following sections:

- Securing the Presentation Layer
- Performing Operational Management

Security is a necessary consideration in most industrial-strength solutions to preserve the integrity and privacy of data as it passes over the network, and also to protect resources and business intelligence at the host.

Operational management is also a vital issue to make sure that applications are deployed correctly and run effectively.

Securing the Presentation Layer

You must approach application security from the earliest stages of your application design. For a good starting point, see "Designing for Securability" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxcondesigningforsecurability.asp*). This document describes how to assess and mitigate the security risks facing your application.

The presentation layer provides specific threats and risks to the security of your application. You must take care to secure it against these threats. Consider the following guidelines when thinking about presentation layer security:

- Rely on proven and tested security solutions based on industry-proven algorithms instead of creating custom solutions.
- Never trust external input; make sure your presentation layer validates all input before processing.
- Assume all external systems that your application accesses are insecure.
- Apply the principle of least privilege to the users of the system and hide functionality from those who do not have authorization to use a particular feature.
- If your application supports multiple user interfaces and client types, make sure you enforce a suitable security baseline across all interface and client types.
- Aim to stop unauthorized actions in the presentation layer before any process penetrates deeper into the application.

The following sections describe presentation-layer specific issues relating to the following aspects of application security:

- Achieving Secure Communications
- Performing Authentication
- Performing Authorization
- Using Code Access Security
- Implementing Security Across Tiers
- Auditing

Applying the guidance in these sections may help you to design and implement secure presentation layer solutions.

Achieving Secure Communications

Exchanging data over exposed networks, such as the Internet, introduces risks to the security of your communications. If your application processes sensitive data—such as credit card details or medical records—you must implement mechanisms to make sure of the secrecy and integrity of the data as it travels between application components.

Whether your presentation layer supports Web- or Windows-based clients, there are a number of widely used secure communications solutions available to you. These solutions are typically implemented by the underlying software or hardware infrastructure where your application runs. This means your application transparently gains the benefits of secure communications without the requirement to implement

secure communications mechanisms at the application level. The most common secure communications solutions are shown in Table A.1:

Table A.1: Secure Communications Options

Option	Use in Webs Forms	Use in Windows Forms
Secure Sockets Layer (SSL)	Use when securely communicating between the browser and Web server	Use when smart clients are calling Web services
Internet Protocol Security (IPSec)	Use between server tiers in a data center	Not used
Virtual Private Networking (VPN)	Not used	Use to access servers in the intranet/extranet

For detailed information about SSL/TLS and IPSec, see Chapter 4 of "Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/secnetlpMSDN.asp*).

For more information about Virtual Private Networking, see article 323441, "HOW TO: Install and Configure a Virtual Private Network Server in Windows Server 2003," in the Microsoft Knowledge Base (*http://support.microsoft.com/default.aspx?scid=kb;en-us;323441*).

Despite the availability and widely acknowledged success of these solutions, your application may demand different or additional levels of secure communications. Some examples include:

- Using message authentication codes to ensure data integrity
- Using digital signatures to ensure data integrity and support non-repudiation
- Using data encryption to provide end-to-end privacy of application data

You typically have to implement these features in your application, and the presentation layer is a common place to implement them. The .NET Framework provides cryptographic classes that support symmetric and asymmetric encryption, hashing, digital certificates, and key exchange. These cryptographic solutions are common to Windows and Web programming. Your selection of a cryptographic solution depends on the specific requirements of your application.

Be aware that such solutions bring with them their own problems. The primary problem with implementing cryptographic security solutions is key management and distribution. For more details about implementing secure communications solutions using the .NET Framework cryptographic classes, see "Cryptographic Services" in the ".NET Framework Developer's Guide" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcryptographicservices.asp*).

Performing Authentication

The purpose of authentication is to securely establish the identity of a person who wants to use your application. Authorization and auditing require authentication. The most common authentication technique on the Windows platform is the use of user names and passwords (using built-in Windows authentication or a mechanism such as Microsoft .NET Passport.) However, other mechanisms, such as smart cards, biometrics, and digital certificates, are gaining in popularity as they become more accessible and easier to implement. This is especially true in applications that require higher levels of security.

The techniques you use for authentication vary, depending primarily on whether you are creating a Windows-based or Web application. In both instances, there are operating system-provided mechanisms that are transparent to your application. Of course, you can implement a custom application-level authentication mechanism; however, implementing a secure and reliable authentication scheme is not trivial. Where possible, use the mechanism the operating system provides.

For .NET Framework applications, the result of authentication is an *identity* object and a *principal* object associated with each thread that the runtime can use to make authorization decisions.

- **Identity object**–An identity object implements the **IIdentity** interface and provides run-time information about the owner of the currently executing thread.

  The **IsAuthenticated** property in the **IIdentity** object indicates whether the user has been authenticated, and the **AuthenticatedType** property returns the authentication type as a string (for example, **"Basic"** or **"NTLM"**). There is also a **Name** property (this provides the name of the current user); if the user hasn't been authenticated, this is an empty string (""). You can use these details to decide whether the current user can access a particular resource or execute a particular piece of code.

  The .NET Framework defines a **GenericIdentity** class that you can use for most custom logon scenarios and a more specialized **WindowsIdentity** class that you can use if you want your application to use Windows authentication.

- **Principal object**–A principal object implements the **IPrincipal** interface and stores the roles associated with the current user. The **IPrincipal** object can then be used to authorize or reject access to particular resources.

  The **IPrincipal** interface has an **Identity** property; this returns an **IIdentity** object that identifies the current user. The **IPrincipal** interface also has an **IsInRole** method; this enables you to perform role-based checks to grant or deny access to a particular resource (or piece of code) depending on whether the current user is in a certain role. Principals allow fine-grained authorization checks at a very detailed programmatic level.

The .NET Framework defines **GenericPrincipal** and **WindowsPrincipal** classes to use with the **GenericIdentity** and **WindowsIdentity** classes described earlier.

IIS supports a variety of authentication mechanisms. Use the following guidelines to help you decide when to use these mechanisms:

- **Anonymous authentication**–Anonymous authentication is effectively synonymous with "no IIS authentication." Under Anonymous authentication, the IIS server creates a **guest account** to represent all anonymous users. By default, the anonymous account has the name **IUSR_COMPUTERNAME**, where **COMPUTERNAME** is the NetBIOS name of the computer at install time.

  Anonymous authentication is appropriate if you want to allow unfettered access to resources, and it offers the best performance because the authentication overheads are minimal. Another scenario where Anonymous authentication is appropriate is if you want to perform your own custom authentication.

  If you enable Anonymous authentication, IIS always attempts to authenticate the user with Anonymous authentication first, even if you enable additional authentication methods. You can change the account that is used for Anonymous authentication in IIS Manager. You can also change the security settings for the **IUSR_computername** account in Windows by using the Group Policy Manager snap-in of the Microsoft Management Console (MMC); when you change the **IUSR_computername** account, the changes affect every anonymous HTTP request that a Web server services.

- **Basic authentication**–Basic authentication is part of the HTTP specification; therefore it is supported by most browsers. Basic authentication requires the user to supply credentials—a user name and password—so that IIS can prove the user's identity. If a user's credentials are rejected, Internet Explorer displays an authentication dialog box to re-enter the user's credentials. Internet Explorer allows the user three connection attempts before failing the connection and reporting an error to the user.

  The user's credentials are submitted in unencrypted format; a network snooper can easily intercept the packets and steal these details. Therefore, use Basic authentication only in conjunction with SSL/TSL to ensure secure communication between the client and the IIS server. For more information, see "Achieving Secure Communications" earlier in this chapter.

- **Digest authentication**–Digest authentication was first introduced in IIS 5.0 as an enhancement to Basic authentication. The user's credentials are hashed before they are transmitted to the IIS server instead of being transmitted in clear text. To enable Digest authentication, the user and the IIS server must be part of the same domain, and the user must have a Windows user account stored in Active Directory on the domain controller. Additionally, the domain controller and the IIS server must be running Windows 2000 or later.

- **Integrated Windows authentication**–Integrated Windows authentication relies on an encrypted exchange of information between the client and the IIS server to confirm the identity of the user. Unlike Basic authentication, Integrated Windows authentication does not initially prompt for a user name and password; the current Windows user information on the client computer is used for Integrated Windows authentication.

  Integrated Windows authentication is either provided by **Kerberos** or NTLM (**Windows NT LAN Manager) Challenge/Response**, depending on the client and server configuration: Kerberos is used if the domain controller is running Windows 2000 or later and Active Directory Services is installed; NTLM authentication is used in all other cases.
- **Certificate authentication**–Certificate authentication relies on the user submitting a client certificate to the IIS server to prove the user's identity. The user obtains this client certificate from a Certification Authority (CA). At the IIS server, you must create a **client certificate map** to map client certificates to particular Windows user accounts; this enables IIS to test the client's certificate to verify that it corresponds with a recognized and authenticated user. When the user submits his or her certificate, it means IIS does not have to perform Basic, Digest, or Integrated Windows authentication to identify the user.

In addition to the security mechanisms provided by IIS, ASP.NET provides its own security mechanisms as follows:

- **Forms authentication**–Forms authentication provides an easily extensible mechanism that enables you to implement a custom authentication scheme.
- **.NET Passport authentication**–.NET Passport authentication provides access to the single logon capabilities of the Internet-based .NET Passport service.

Both Forms and .NET Passport authentication require significantly more effort to implement than IIS-based authentication; however, they are the only workable solutions if you have an Internet-facing Web application and do not want to establish Windows accounts for every user.

For complete information about designing and implementing an appropriate authentication and authorization mechanisms for your Web applications, see "Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication" on MSDN (*http://msdn.microsoft.com/library/en-us/dnnetsec/html/secnetlpMSDN.asp*).

Performing Authorization

Authorization is the process of determining whether a user has permission to access a particular resource or piece of functionality. To perform authorization, you must have an authentication mechanism in place to establish the identity of the user, and that mechanism must determine the identity of the user accurately and reliably.

In .NET Framework applications, authorization is generally based on two pieces of information associated with the active thread:

- The Windows access token
- The **IPrincipal** object

The Windows access token represents the capabilities of the active Windows account. In smart client applications, such as Windows Forms applications, this is generally the currently logged on user. In server-based applications, such as ASP.NET, this is generally a special service account configured to run the application; for example, the default account named for ASP.NET is named **ASPNET**. Windows uses the access token to determine whether the current thread can access resources and functionality secured at the operating system level.

The **IPrincipal** object is an application-level object that represents the identity and roles of the current user. Application code can use the **IPrincipal** object to make authorization decisions based on the roles of the active user. Frequently, the **IPrincipal** represents the same user as the Windows access token; however, applications can change the **IPrincipal** relatively easily to represent a different user. This is most frequently done in server applications to represent the user connected to the server instead of the account that the server service is running as.

Using Code Access Security

Code access security is a security feature that applies to all .NET Framework managed code to protect computer systems from malicious code and to provide a way to allow mobile code to run safely.

You will have to consider the ramifications of code access security in the following scenarios:

- You are designing browser-hosted controls
- You are hosting third-party applications
- You are hosting assemblies from different vendors on a shared server
- You want to prevent certain native functions, such as file write APIs, to be available to certain assemblies

Code access security allows code to be trusted to varying degrees, depending on factors such as where the code comes from and its strong assembly name. Code access security enables you to specify the operations your code can perform and the operations your code cannot perform.

Code access security supports a permission support mechanism where code can explicitly request particular permissions and explicitly refuse others that it knows it never requires. Each permission represents the right for code to access a protected resource such as a file, directory, or registry entry, or the right for it to perform a protected operation such as calling into unmanaged code. Permissions can be

demanded by code and the run-time security policy determines which permissions to grant.

The .NET Framework allows administrators to assign a pre-defined set of permissions to an application. For example, applications running on a UNC share (running in the **Intranet** security zone) receive the **LocalIntranet** permission set. Applications running on the local computer (running in the **MyComputer** security zone) receive the **FullTrust** permission set.

ASP.NET Web applications can be configured by assigning them trust levels. Trust levels are configured using the **<trust>** element in the configuration file.

```
<trust level="Full | High | Medium | Low | Minimal" originUrl="url" />
```

Each level determines the application's permissions; an XML security policy file specifies the details of these permissions. Each level maps to a specific file. The default mappings for ASP.NET are:

- **Full**–This trust level has no associated configuration file. Full trust allows applications to use all resources (subject to operating system permissions); this is just like running without code access security (although code access security cannot be switched off for managed code).
- **High**–This trust level maps to **web_hightrust.config**. This trust level provides permissions that grant applications read/write access to the application directory (subject to operating system permissions) and allows the application to replace the authentication principal object.

  This trust level restricts applications from calling unmanaged code, calling serviced components, writing to the event log, accessing Microsoft Message Queuing queues, or accessing OLE database data sources.
- **Medium**–This trust level maps to **web_mediumtrust.config**. This trust level provides permissions that grant applications read/write access to the application directory (subject to operating system permissions) and allows the application to replace the authentication principal object.

  The restrictions listed for the **high** trust level also apply to the **medium** trust level. Additionally, file access is restricted to the current application directory, and registry access is not permitted.
- **Low**–This trust level maps to **web_lowtrust.config**. This trust level allows applications to read from the application directory and provides limited network connectivity. Applications can connect back to their host site, assuming the **originUrl** attribute of the **<trust>** element is configured appropriately.

  The restrictions listed for the **medium** trust level also apply to the **low** trust level. Additionally, the application is not able to connect to SQL Server data sources or call the **CodeAccessPermission.Assert** method.

- **Minimal**–This trust level maps to **web_minimaltrust.config**. In this trust level, only the execute permission is available.

You can override these mappings in the **<securityPolicy>** element of the configuration file, and you can customize and extend each level. You can also create your own levels that define arbitrary permission sets. The default **<securityPolicy>** mapping set is shown in the following example.

```
<securityPolicy>
   <trustLevel name="Full"    policyFile="internal" />
   <trustLevel name="High"    policyFile="web_hightrust.config" />
   <trustLevel name="Medium"  policyFile="web_mediumtrust.config" />
   <trustLevel name="Low"     policyFile="web_lowtrust.config" />
   <trustLevel name="Minimal" policyFile="web_minimaltrust.config" />
</securityPolicy>
```

ASP.NET configuration is hierarchical in nature, with configuration files optionally at the computer, application, and sub-application levels. Sub-level configuration files can be used to override settings made at a higher level or can be used to add additional configuration information. While this provides a high degree of flexibility, administrators may sometimes want to enforce the configuration settings and not allow them to be overridden by specific applications.

For example, an administrator of a hosted Web site may want to specify the code access security level and not allow it to be changed by individual applications. This can be achieved using the **<location>** element coupled with the **allowOverride** attribute. For example, an administrator of a hosted Web site may want to make sure that no applications are permitted to call into unmanaged code. The following configuration file fragment shows how an administrator can lock down the code access configuration settings for a whole site and restrict applications with the High trust level (this does not allow calls into unmanaged code).

```
<location path="somesitepath" allowOverride="false">
  <trust level="high" originUrl="http://somesite.com" />
</location>
```

The **path** attribute may refer to a site or a virtual directory, and it applies to the nominated directory and all sub-directories. In the preceding example, if you set **allowOverride** to "false," you can prevent any application in the site from overriding the configuration settings specified in the **<location>** element. Note that the ability to lock configuration settings applies to all settings, not just security settings such as trust levels.

Implementing Security Across Tiers

ASP.NET Web applications typically interact with business objects, .NET Framework remoting objects, or some other back-end application. The Web application typically undertakes the responsibility for authenticating and authorizing the user. There are two ways to pass the results of the authentication and authorization to downstream applications:

- **Trusted subsystem model**
- **Impersonation/delegation model**

The following sections describe these models and provide guidance on when to use each model.

Using the Trusted Subsystem Model

The ASP.NET Web application authenticates and authorizes the user at the first point of contact, and it creates a trusted identity to represent the user. By default, the ASP.NET Web application worker process (**aspnet_wp.exe**) runs using an account named **ASPNET**.

Whenever the ASP.NET application communicates with downstream applications, it does so using this security context. The downstream applications trust the ASP.NET application to correctly authenticate and authorize the original user.

There are two key benefits of the trusted subsystem model:

- **Simplicity**–Downstream applications have to authenticate only a single user account (that is, the **ASPNET** account of the ASP.NET application). Moreover, access control lists (ACLs) can be defined in terms of this single trusted identity instead of defining access rights for every authorized user or role.
- **Scalability**–The ASP.NET application always forwards the same security credentials to downstream applications, regardless of the identity of the user who actually contacted the ASP.NET application. This facilitates connection pooling. Connection pooling is an essential requirement for scalability and it allows multiple clients to reuse resources in an efficient manner, as long as the users have the same security context.

For more information, see "How to Use the Trusted Subsystem Model" in Appendix B in this guide.

Using the Impersonation/Delegation Model

In the impersonation/delegation model, the ASP.NET Web application authenticates and authorizes the user at the first point of contact as before. However, these original credentials are flowed to downstream applications instead of passing the same trusted identity for all users. This enables downstream applications to perform their own authentication and authorization tests using the real security credentials of the original user.

There are two key benefits of the impersonation/delegation model:

- **Flexibility**–Downstream applications can perform their own per-user and per-role security checks, using the user's real security context.
- **Auditing**–Downstream applications, and the resources that they access (such as databases), can keep an accurate audit trail of which users have accessed which resources.

Before you use the impersonation/delegation model, you must be aware of the following issues:

- **Scalability restrictions**–Connection pooling is not possible with impersonation/delegation, because each user's original security context is flowed downstream. Connection pooling works only if the same security context is used to access pooled resources.
- **Complexity**–You must define ACLs for particular users and roles instead of being able to express these ACLs in terms of a single trusted identity.
- **Technology difficulties**–Applications that perform impersonation require higher privileges than typical applications. Specifically, they require the operating system privilege.

There are two different ways to use impersonation/delegation model, depending on how you initially authenticate the user in the ASP.NET Web application:

- **Using Kerberos authentication**–Kerberos involves authenticating a user with a Windows NT Domain or Active Directory account. For more information about how to perform impersonation/delegation with Kerberos, see "How to: Use Impersonation/Delegation with Kerberos Authentication and Delegation" in Appendix B in this guide.
- **Using Basic authentication or Forms authentication**–Basic authentication is part of the HTTP 1.0 specification; it transmits the user's name and password to the Web server using Base64 encoding. Forms authentication uses HTTP client-side redirection to redirect unauthenticated users to an HTML login form. For more information about how to perform impersonation/delegation with Basic authentication or Forms authentication, see "How to Use Impersonation/Delegation with Basic or Forms Authentication" in Appendix B in this guide.

Use the appropriate impersonation/delegation model, depending on how you initially authenticate the user in your ASP.NET Web application.

Auditing

Auditing of presentation layer activities is generally limited to a small set of global events including:

- Users logging on
- Users logging off

- Sessions timing out
- Password changes
- Failed logon attempts
- Failed authorizations
- Failure and success of business processes

You might also decide to audit the business processes initiated by the user in the presentation layer. Having a high-level view of a user's actions can provide useful diagnostic information. However, it is best not to rely on presentation layer auditing to provide the sole audit trail of user activities. Typically, a presentation layer event triggers a series of events in the business and data access layers. To provide the granularity of information required for security auditing, make sure that each of the lower layers audits its own activities.

Aside from what to audit, the most important decision you must make is where to store audit logs. The most appropriate storage location depends on the architecture of your application.

Server-based applications that support thin Windows-based or Web clients generate all audit events at the server. In this scenario, it is easy to manage a handful of servers that write to their local Windows Event Logs. As the number of servers grows, it is better to switch to a centralized audit log. If all servers reside on a private network, the security risks are minimal, and the administrative and operation benefits of a global security log are significant.

Distributed applications that implement significant portions of their functionality in smart client applications might have to audit some presentation layer events on the clients. In these situations, writing to the local Windows Event Log becomes a problem. You must implement a mechanism to pull or push the events from the remote computers to a central store. Although there are tools available to automate this, the approach becomes unmanageable (or at least inefficient) as the number of clients grows. In this scenario, your application can implement a custom auditing service that your presentation layer logic can write audit records to. The service takes responsibility for delivering the audit logs to a central store. When implementing remote auditing mechanisms, consider the following guidelines:

- Sign the audit records with a digital signature or message authentication code before sending to ensure data integrity.
- Avoid including sensitive information in audit records; if unavoidable, it is a good idea to encrypt the records before sending them. For more information about using the .NET Framework cryptographic classes, see "Achieving Secure Communications" earlier in this chapter.
- Use a reliable transport and delivery mechanism to make sure audit records do not get lost.

- Make sure operating system security is in place to stop users from deleting audit records before delivery.
- Make sure users cannot change the location to where the audit service stores and delivers records.

Regardless of where you store your audit logs, audit records must be immutable and accessible only to authorized people.

Performing Operational Management

Operational management is concerned with the deployment and ongoing, day-to-day running of your application. For information about overall operational management issues relating to distributed applications, see "Application Architecture for .NET: Designing Applications and Services" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/distapp.asp*).

The following sections describe specific aspects of operational management to consider when designing your presentation layer:

- Managing Exceptions in the Presentation Layer
- Monitoring in the Presentation Layer
- Managing Metadata and Configuration Information
- Defining the Location of Services
- Deploying Applications

These sections provide guidance for planning the operational management of your application.

Managing Exceptions in the Presentation Layer

One important point that relates closely to the presentation layer is the issue of unhandled exceptions. As the name suggests, an unhandled exception is one that your application code does not explicitly handle. This might be because the exception represents an unrecoverable error, or perhaps the programmer did not anticipate and code for that type of exception. Whatever the case, these unhandled exceptions occur until they reach the outer boundary of your application, where they appear to the user as a confusing error. Not only is the presentation of a stack trace not useful for a typical user, but it might also provide attackers with useful information they can use to attack your system in the future. It is a good idea to never allow a user to see the raw details of an uncaught exception.

If you are developing an ASP.NET application, you can configure the application to display a specific page when an exception occurs. You can also use application- and page-level events to handle exceptions. For more information about these mechanisms, see the "Exception Management Application Block for .NET" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/emab-rm.asp*).

If you are developing Windows Forms-based applications, it is a good idea to implement a catch-all exception handler. For an example of how to do this, see "How to Define a Catch-All Exception Handler in Windows Forms Applications" in Appendix B in this guide.

Monitoring in the Presentation Layer

Implementing the appropriate levels of monitoring to your application allows you to know when things are running smoothly and when problems are encountered.

The types of monitoring to consider for your presentation layer include:

- Health monitoring, to determine:
  - Whether the application is running
  - If there are errors or other problems that might cause the application to not perform in an optimal manner
- Performance monitoring, to determine:
  - How long the application takes to process a user request
  - If there are any bottlenecks needing attention

It is also important that you communicate problems affecting application operation to users clearly, effectively, and in a timely manner. Some options for communicating these problems include Windows Management Instrumentation (WMI), writing to the Event Log, or publishing exceptions to isolated storage. As a minimum, display an informative message if your application is down, and include an estimate for when normal operation will resume. Depending on the nature of your application and user base, it may be appropriate to provide application health and status information directly to the users.

For information about how to write instrumentation code, see "Monitoring in .NET Distributed Application Design" on MSDN (*http://msdn.microsoft.com/library/en-us/dnbda/html/monitordotnet.asp*).

Managing Metadata and Configuration Information

Presentation layers frequently rely heavily on application configuration information and metadata. This is particularly true if your application:

- Provides an extensible framework that you load pluggable modules or add-ins into
- Dynamically renders user interfaces based on user profiles and run-time context
- Relies on external services that you have to provide location and authentication details for

The .NET Framework supports a range of storage mechanisms to hold application configuration information. Each mechanism has benefits and disadvantages that

make it more suitable in certain situations. The most common mechanisms for storing configuration information are listed in Table A.2.

Table A.2: Common Configuration Information Storage Mechanisms

Option	Notes
XML or INI Files	Using XML files to store configuration information provides an easy and standard way to store and read the information. Include built-in support for files such as Web.config and Machine.config through **IConfigurationSectionHandler**. Security is provided by Windows ACLs.
Databases (SQL Server, MSDE)	Configuration information is available to multiple computers and applications. Provide greatest flexibility in terms of the types of data stored and data security. Require additional infrastructure and ongoing management.
Active Directory	Within an organization, you may decide to store application metadata in Active Directory so that the metadata available for clients on the domain. Security is provided by Windows ACLs.
Constructor strings	If you are using Enterprise Services–based components, you can add configuration data to the constructor string for the components.
Windows Registry	Your application should store configuration information in the Windows registry only when absolutely necessary, such as when you must work with earlier versions of applications that use the registry. Storing configuration information in the Windows registry increases the deployment burden of your application. Configuration information is available to multiple applications on the same computer. Security is provided by Windows ACLs.

All these mechanisms provide some form of general-purpose administration tools, allowing you to manage your configuration information and configure its security. However, you frequently have to develop your own application-specific utilities to manage your applications configuration data effectively—especially if the configuration data is complex, or if you have to distribute administrative capabilities between different users. The development of usable, high-quality, administrative tools is important to the long-term success of your application and should not be sidelined as a trivial exercise to be done only if there is time.

Note: The Configuration Management Application Block available on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/cmab.asp*) simplifies the storage and retrieval of configuration information. The block allows you store application configuration information in XML files, the Windows registry, or SQL Server and is extensible so that you can add support for other data stores. Additionally, the block supports features such as encryption and message authentication codes to ensure the confidentiality and integrity of your application configuration data.

Defining the Location of Services

If your presentation layer code uses remote objects and XML Web services, make sure that you do not hard code the locations of these services into your code. Doing so reduces the maintainability of your code. If a service address changes, you have to update, rebuild, test, and distribute your application. It is a good idea for service location and authentication information to be stored securely with your other application configuration information. For information about application configuration, see the previous section of this chapter.

Deploying Applications

Deployment is rarely an application-specific decision in medium- to large-sized organizations that have standard application deployment policies and mechanisms owned by operations departments. None of the major application deployment mechanisms require you to implement specific features in your application; however, you can use the following guidelines to simplify application deployment:

- If security considerations allow, implement browser-based user interfaces where possible to avoid the requirement to distribute client software to a high number of computers.
- Minimize external software dependencies and resource dependencies to simplify installation procedures and reduce security requirements.

For comprehensive information about deploying .NET Framework applications, see "Deploying .NET Framework-based Applications" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/dalgroadmap.asp*).

Summary

This appendix has described how to secure the presentation layer in ASP.NET Web applications by applying built-in support provided by the IIS, ASP.NET, and the .NET Framework for authentication, authorization, and secure communications. It also described how to perform operational management tasks to enable applications to run smoothly and securely day-to-day.

Appendix B

How To Samples

In This Appendix:

- How to: Define a Formatter for Business Entity Objects
- How to: Perform Data Binding in ASP.NET Web Forms
- How to: Design Data Maintenance Forms to Support Create, Read, Update, and Delete Operations
- How to: Execute a Long-Running Task in a Web Application
- How to: Use the Trusted Subsystem Model
- How to: Use Impersonation/Delegation with Kerberos Authentication
- How to: Use Impersonation/Delegation with Basic or Forms Authentication
- How to: Localize Windows Forms
- How to: Define a Catch-All Exception Handler in Windows Forms-based Applications

---

Note: This code samples focus on the concepts in this appendix and as a result may not comply with Microsoft security recommendations. The code samples presented in this appendix have been written for Windows Server 2003 and the .NET Framework 1.1.

How To: Define a Formatter for Business Entity Objects

This example shows how to define a custom formatter class to control how business entity objects are displayed in the presentation layer.

There are three classes in this example:

- ReflectionFormattable–This is a base class that performs custom formatting for business entity objects. The class uses a formatting string to display selective properties of a business entity object in a particular format. For example, a formatting string such as "{LastName}, {Name}" displays the **LastName** and **Name** properties for a business entity object, separated by a comma and a space.
- CustomerEntity–This is a sample business entity class. **CustomerEntity** inherits from **ReflectionFormattable**, to make use of its formatting capabilities.
- CustomFormatting–This is a simple ASP.NET Web page that uses the formatting capabilities of the **ReflectionFormattable** class to display a **CustomerEntity** object in a particular format.

The following sections describe these classes.

Defining the ReflectionFormattable Class

The **ReflectionFormattable** class implements the **IFormattable** interface, and provides basic formatting capabilities that can be inherited by business entity objects.

ReflectionFormattable has a **ToString** method that receives a format string and extracts property names enclosed in braces {}. The method then uses reflection to obtain the value of each requested property on the current business entity object.

```
using System;
using System.Collections;
using System.Reflection;
using System.Text;
using System.Text.RegularExpressions;
using System.Runtime.InteropServices;

// Defines a custom formatting syntax that is used to create string using the
// properties for a given entity.
[ComVisible(false)]
public class ReflectionFormattable
    : IFormattable
{
  // The regular expresion used to parse the custom syntax
  const string format = @"\{(\w+)\}";

  // Object to use for synchronized operations in thread-safe code
  static object SyncRoot = new object();

  // Cached compiled regular expression
  static Regex regEx = null;
```

```
// Constructor safe for multi-threaded operations.
protected ReflectionFormattable()
{
  if (regEx == null)
  {
    lock (SyncRoot)
    {
      if (regEx == null)
      {
        regEx = new Regex(format, RegexOptions.Compiled);
      }
    }
  }
}

// ToString overload for the formattable object
public string ToString(string format, IFormatProvider formatProvider)
{
  if (format != null && format.Length != 0 && format.Trim().Length != 0)
  {
    string tempString = format;

    // Use the regular expression
    MatchCollection matchCollection;
    lock (typeof(Regex))
    {
      matchCollection = regEx.Matches( format );
    }

    // Use the matches to find the properties on the current instance
    foreach (Match m in matchCollection)
    {
      if (m.Groups.Count > 1)
      {
        foreach (Capture c in m.Groups[1].Captures)
        {
          if (c.Value != null &&
              c.Value.Length != 0 && c.Value.Trim().Length != 0)
          {
            tempString = tempString.Replace( "{" + c.Value + "}",
                                             GetPropertyValue(c.Value) );
          }
        }
      }
    }

    // Return the formatted string
    return tempString;
  }
  else
  {
    // Call base ToString method, because the format is null or an empty string
    return ToString();
  }
}
```

```
  // Use reflection to get the value of a property with a specified name
  protected string GetPropertyValue( string name )
  {
    PropertyInfo pi = this.GetType().GetProperty(name);
    if (pi != null)
    {
      object value = pi.GetValue(this, null);
      return value == null ? "" : value.ToString();
    }
    return "{" + name + "}";
  }
}
```

Defining the CustomerEntity Class

CustomerEntity is a sample business entity class for this sample. The class has three properties (and associated private fields), and inherits formatting capabilities from **ReflectionFormattable**.

```
using System;
using System.Runtime.InteropServices;

public class CustomerEntity
  : ReflectionFormattable
{
  // Entity ID
  public int ID
  {
    get { return _id;  }
    set { _id = value; }
  } int _id = 0;

  // Customer name
  public string Name
  {
    get { return _name;  }
    set { _name = value; }
  } string _name = "";

  // Customer last name
  public string LastName
  {
    get { return _lastName;  }
    set { _lastName = value; }
  } string _lastName = "";

  // Customer date created
  public DateTime DateCreated
  {
    get { return _dateCreated;  }
    set { _dateCreated = value; }
  } DateTime _dateCreated = DateTime.Now;
}
```

Defining the CustomFormatting Class

CustomFormatting is an ASP.NET Web page that asks the user to enter details for a **CustomerEntity** object. When the user clicks the **Set values** button, the customer details are displayed in the format {ID} – {LastName}, {FirstName}.

Figure B.1 shows how the page appears in the browser.

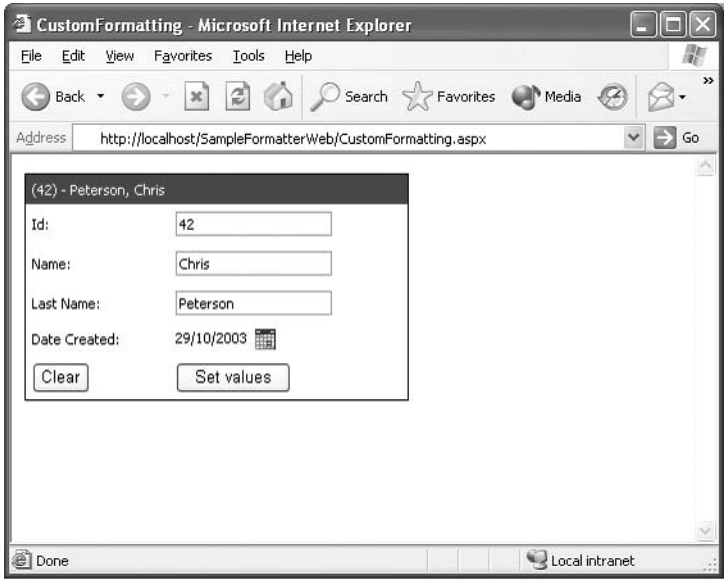


Figure B.1

Formatting business entity objects in an ASP.NET Web page

The relevant portions of code for the **CustomFormatting** class are shown in the following code sample. Notice that the **SetTitle** method calls **ToString** on the **CustomerEntity** object, passing the format string "({ID}) - {LastName}, {Name}" as the first parameter:

```
[ComVisible(false)]
public class CustomFormatting : System.Web.UI.Page
{
  private CustomerEntity customer = new CustomerEntity();

  protected System.Web.UI.WebControls.TextBox txtId;
  protected System.Web.UI.WebControls.TextBox txtName;
  protected System.Web.UI.WebControls.TextBox txtLastName;
  protected System.Web.UI.WebControls.Calendar clDateCreated;
  protected System.Web.UI.WebControls.ImageButton btnCalendar;

  protected System.Web.UI.WebControls.Button btnClear;
  protected System.Web.UI.WebControls.Button btnSetValues;

  protected System.Web.UI.WebControls.Label lblDateCreated;
  protected System.Web.UI.WebControls.Label lblTitle;
  protected System.Web.UI.WebControls.Label lblMessage;
```

```
  private void Page_Load(object sender, System.EventArgs e)
  {
    if (!IsPostBack)
    {
      clDateCreated.SelectedDate = DateTime.Now;
      lblDateCreated.Text = clDateCreated.SelectedDate.ToShortDateString();
    }

    try
    {
      // Populate the CustomerEntity object with data
      customer.ID = Int32.Parse(txtId.Text);
      customer.Name = txtName.Text;
      customer.LastName = txtLastName.Text;
      customer.DateCreated = clDateCreated.SelectedDate;

      // Set the title on the form
      SetTitle();
    }
    catch (Exception ex)
    {
      // Exception-handling code…
    }
    lblMessage.Visible = false;
  }

  private void btnSetValues_Click(object sender, System.EventArgs e)
  {
    try
    {
      // Populate the CustomerEntity object with data
      customer.ID = Int32.Parse(txtId.Text);
      customer.Name = txtName.Text;
      customer.LastName = txtLastName.Text;
      customer.DateCreated = clDateCreated.SelectedDate;

      // Set the title on the form
      SetTitle();
    }
    catch (Exception ex)
    {
      lblMessage.Text = "Incorrect data: " + ex.Message;
      lblMessage.Visible = true;
    }
  }

  private void SetTitle()
  {
    lblTitle.Text = (customer.ID == 0) ?
                        "New customer" :
                         customer.ToString("({ID}) - {LastName}, {Name}", null);
  }

  // Plus other code…
}
```

How To Perform Data Binding in ASP.NET Web Forms

Data binding in ASP.NET Web Forms is more straightforward than data binding in Windows Forms:

- In Windows Forms, objects are kept in memory. A notification mechanism is required between the objects and the controls they are bound to.
- In Web Forms, objects are bound to controls at the server. The data-bound output is returned to the client; there is no live data at the client. All user actions, such as editing and paging through the data, are achieved by postbacks and **ViewState** interactions at the server.

As a consequence of this simplified behavior in Web Forms, entity classes do not have to implement any special interfaces in order to support data binding. Likewise, you can easily create data-bound collections of entity objects by adding them to an **ArrayList** or any class that implements **IEnumerable**.

The following sections describe how to perform data binding in three particular scenarios:

- Data Binding an Entity Object to Simple Controls
- Data Binding a Collection of Entity Objects to a **DataList** Control
- Data Binding a Collection of Entity Objects to a **DataGrid** Control

Code samples are included where appropriate, to illustrate how to define entity classes and type-safe collections, and to show how to bind them to Web server controls.

Note: For simplicity in this example, the application does not validate the details entered by the user.

Data Binding an Entity Object to Simple Controls

This section describes how to data bind entity objects to simple controls on a Web Form, such as labels and text boxes.

This section includes the following topics:

- Defining an Entity Class for Data Binding
- Adding an Entity Object to a Web Form
- Specifying Data Binding Expressions for a Control
- Performing Data Binding at Run Time

A simple entity class and ASP.NET Web Form are presented to demonstrate the key points in this section.

Defining an Entity Class for Data Binding

If you want to data bind entity objects to Web server controls, it is a good idea for your entity class to inherit from **System.ComponentModel.Component**. This enables Visual Studio .NET to display entity objects in the Components Tray in the Designer window, to simplify design-time usage of the entity objects.

When you define a class that inherits from **System.ComponentModel.Component**, Solution Explorer displays the class with a "UI control" icon by default. If you double-click the icon, Visual Studio .NET displays the class in design view instead of code view. To suppress this unwanted behavior, annotate your class with the **[System.ComponentModel.DesignerCategory("Code")]** attribute. Solution Explorer then displays the class with a "code" icon, and double-clicking the icon opens the class in code view.

The following code sample shows an entity class named **CustomerEntity** to illustrate these points.

```
using System;
using System.ComponentModel;

[System.ComponentModel.DesignerCategory("Code")]
public class CustomerEntity : Component
{
  // Customer ID
  public int ID
  {
    get { return _id;  }
    set { _id = value; }
  } int _id = 0;

  // Customer first name
  public string FirstName
  {
    get { return _name;  }
    set { _name = value; }
  } string _name = "";

  // Customer last name
  public string LastName
  {
    get { return _lastName;  }
    set { _lastName = value; }
  } string _lastName = "";

  // Customer date created
  public DateTime DateCreated
  {
    get { return _dateCreated;  }
    set { _dateCreated = value; }
  } DateTime _dateCreated = DateTime.Now;
}
```

The **CustomerEntity** class inherits from **System.ComponentModel.Component** for design-time support in Visual Studio .NET. Also, the class is annotated with the **[System.ComponentModel.DesignerCategory("Code")]** attribute as described previously. **CustomerEntity** defines four public properties, to get and set private fields defined in the class.

Adding an Entity Object to a Web Form

When you define a class that inherits from **System.ComponentModel.Component**, you can add instances of the class directly to a Web Form in Visual Studio .NET. There are two ways to do this:

- Add the entity class to the Toolbox, and then drag an instance of the class onto the form.
- Create an entity object programmatically in the code-behind class for the Web Form. In this approach, you must declare a protected variable and initialize it in the **InitializeComponent** method. The following code sample illustrates this approach.

```
public class SimpleBinding : System.Web.UI.Page
{
  // Declare a protected instance variable, to refer to an entity object
  protected CustomerEntity customer;

  private void InitializeComponent()
  {
    // Create an entity object, and assign it to the instance variable
    this.customer = new CustomerEntity();

    // Plus other component-initialization code...
  }

  // Plus other members in the Web Form...
}
```

When you create the entity object using one of these two approaches, the entity object appears in the Component Tray in the Visual Studio .NET Designer.

Specifying Data Binding Expressions for a Control

This section describes how to specify data binding expressions for a control to bind a particular property on the control to a specific property on an entity object.

The sample form shown in Figure B.2 has four labels that can be bound to the **ID**, **FirstName**, **LastName**, and **DateCreated** properties of a **CustomerEntity** object. The form also has buttons to allow the user to navigate backward and forward through a collection of **CustomerEntity** objects:

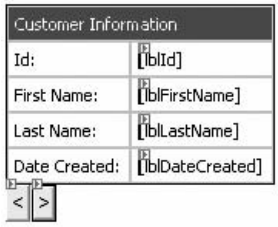


Figure B.2
Web Form containing simple controls, which can be data bound to an entity object

► **To specify a data binding expression for a control**

1. Select the control that you want to perform data binding on.
2. In the Properties window, locate the **(DataBindings)** property and click the **ellipsis** button (…).
3. In the **DataBindings** property editor, select one of the control's bindable properties. Then use either simple binding or complex binding to bind the property as appropriate.

When you bind a property on a control, Visual Studio .NET adds a binding expression to the .aspx file to describe the data binding. For example, the following code sample shows how an <asp:label> control can be data-bound to the **DateCreated** property on a customer component.

```
<asp:label
      id="lblDateCreated"
      runat="server"
      Text='<%# DataBinder.Eval(customer, "DateCreated", "{0:d}") %>'>
</asp:label>
```

The next section describes how and when the binding expressions are evaluated at run time.

Performing Data Binding at Run Time

To evaluate a binding expression at run time, call the **DataBind** method on the control. You can also call the **DataBind** method at the page level, which conveniently causes the call to be applied recursively to all controls on the page.

This section shows a sample implementation for the Web Form introduced earlier to illustrate how to perform data binding. The form creates some sample **CustomerEntity** objects in its static constructor and inserts them into an **ArrayList**.

The form also has an integer variable to indicate the index of the currently-bound **CustomerEntity** object:

```
public class SimpleBinding : System.Web.UI.Page
{
  // CustomerEntity component (as defined earlier), which is bound to controls
  protected CustomerEntity customer;

  // Collection of sample CustomerEntity objects
  private static ArrayList CustomerList = new ArrayList();

  // Index of currently-bound CustomerEntity object
  private int current = 0;

  // Static constructor, to create CustomerEntity objects and add to collection
  static SimpleBinding()
  {
    // Set customers on the list
    CustomerEntity tempCustomer = new CustomerEntity();
    tempCustomer.ID = 5;
    tempCustomer.FirstName = "Guy";
    tempCustomer.LastName = "Gilbert";
    tempCustomer.DateCreated = new DateTime( 1948, 3, 9 );
    CustomerList.Add(tempCustomer);

    tempCustomer = new CustomerEntity();
    tempCustomer.ID = 11;
    tempCustomer.FirstName = "Kendall";
    tempCustomer.LastName = "Keil";
    tempCustomer.DateCreated = new DateTime( 1974, 9, 4 );
    CustomerList.Add(tempCustomer);

    // Plus other sample entities...
  }

  // Plus other members...
}
```

The form's **OnPreRender** method persists the current index in the page **ViewState**, so that the current index is available in a postback.

```
public class SimpleBinding : System.Web.UI.Page
{
  protected override void OnPreRender(EventArgs e)
  {
    ViewState["Index"] = current;
    base.OnPreRender (e);
  }

  // Plus other members...
}
```

The form's **Page_Load** method retrieves the current index from the **ViewState**. The method sets the "customer" variable to refer to the current **CustomerEntity** object in the **ArrayList**, and then calls **DataBind** to bind all the controls on the form. This causes the data-bound labels on the form to display the details for the current customer. This causes the form to display the details for the newly selected **CustomerEntity** object.

```
public class SimpleBinding : System.Web.UI.Page
{
  private void Page_Load(object sender, System.EventArgs e)
  {
    if (IsPostBack)
    {
      current = (int)ViewState["Index"];
    }

    customer = CustomerList[current] as CustomerEntity;
    DataBind();
  }

  // Plus other members...
}
```

The form also has event handler methods for the **previous** button (<) and **next** button (>) on the form.

```
public class SimpleBinding : System.Web.UI.Page
{
  private void btnPrev_Click(object sender, System.EventArgs e)
  {
    if (--current <= 0)
    {
      current = 0;
    }
    customer = CustomerList[current] as CustomerEntity;
    DataBind();
  }

  private void btnNext_Click(object sender, System.EventArgs e)
  {
    if (++current >= CustomerList.Count)
    {
      current = CustomerList.Count - 1;
    }
    customer = CustomerList[current] as CustomerEntity;
    DataBind();
  }

  // Plus other members...
}
```

When the user clicks the **previous** or **next** buttons, the current index is decreased or increased as appropriate. The **DataBind** method causes the data binding expressions to be re-evaluated for each control on the form, to display the details for the newly selected **CustomerEntity** object.

Data Binding a Collection of Entity Objects to a DataList Control

This section describes how to bind a collection of entity objects to a **DataList** control. This allows the user to view all the entity objects at the same time, and to edit and delete entity objects in the collection.

This section includes the following topics:

- Defining a Typed Entity Collection Class
- Adding an Entity Collection Object to a Web Form
- Designing Item Templates for a **DataList** Control
- Specifying Data Binding Expressions for a **DataList** Control
- Performing Data Binding at Run Time

This section uses the **CustomerEntity** class from the previous section, but it uses a new ASP.NET Web Form to illustrate data binding techniques for a **DataList** control.

Defining a Typed Entity Collection Class

A **DataList** control can be bound to a collection object; the **DataList** displays each of the collection's items in a separate row.

You can either bind a **DataList** to a generic collection type such as **ArrayList**, or to a typed collection class that works with a particular type of entity object. The latter approach is recommended, because it enables Visual Studio .NET to expose the entity's properties when you define data binding expressions at design time.

The following code sample shows a typed collection class named **CustomerCollection** to hold a collection of **CustomerEntity** objects.

```
using System;
using System.ComponentModel;
using System.Collections;

[System.ComponentModel.DesignerCategory("Code")]
public class CustomerCollection : CollectionBase, IComponent
{
  // Default constructor
  public CustomerCollection()
  {}

  // Add a CustomerEntity object to the collection
  public int Add(CustomerEntity customer)
  {
    return base.InnerList.Add(customer);
  }
```

```
  // Remove a CustomerEntity object from the collection
  public void Remove(CustomerEntity customer)
  {
    base.InnerList.Remove(customer);
  }

  // Access a CustomerEntity object by its index in the collection
  public CustomerEntity this[int index]
  {
    get { return base.InnerList[index] as CustomerEntity; }
    set { base.InnerList[index] = value; }
  }

  // Implement the IComponent members
  public event System.EventHandler Disposed;
  ISite IComponent.Site
  {
    get { return _site;  }
    set { _site = value; }
  } ISite _site = null;

  // Implement the IDisposable members (IComponent inherits from IDisposable)
  public event System.EventHandler Disposed;

  void IDisposable.Dispose()
  {
    // We've nothing to dispose explicitly
    if (Disposed != null)
      Disposed(this, EventArgs.Empty);
  }
}
```

CustomerCollection inherits from **System.Collections.CollectionBase** and provides type-safe methods to add, remove, and access items in the collection. **CustomerCollection** also implements the **IComponent** interface, to allow **CustomerCollection** objects to be treated as components in the Visual Studio .NET Designer.

Adding an Entity Collection Object to a Web Form

There are two ways to add an entity collection object to a Web Form in Visual Studio .NET:

- Add the entity collection class to the Toolbox, and then drag an instance of the class onto the form.
- Create an entity collection object programmatically in the code-behind class for the Web Form. In this approach, you must declare a protected variable and initialize it in the **InitializeComponent** method. The following code sample illustrates this approach.

```
public class DataListBinding : System.Web.UI.Page
{
  // Declare an instance variable to refer to an entity collection object
  protected CustomerCollection customers;

  private void InitializeComponent()
  {
    this.customers = new CustomerCollection();

    // Plus other component-initialization code...
  }

  // Plus other members in the Web form...
}
```

When you create the entity collection object using one of these two approaches, the object appears in the Component Tray in the Visual Studio .NET Designer.

Designing Item Templates for a DataList Control

This section describes how to design item templates for a **DataList** control to govern how entity objects are displayed and edited in the **DataList** control.

► **To create and configure a DataList control in a form**

1. Drag a **DataList** control from the Toolbox and drop it onto your Web Form.
2. Right-click the **DataList** control. In the shortcut menu, point to **Edit Template** and then click **Item Templates**.
3. In the **ItemTemplate** section, add simple controls (such as **Label** controls) to display each of the properties of an entity object.
4. Add two buttons to the **ItemTemplate** section to enable the user to delete or edit the entity object. Set the buttons' **CommandName** properties to **Delete** and **Edit** respectively, so that the buttons raise **DeleteCommand** and **EditCommand** events on the **DataList** control.
5. In the **EditItemTemplate** section, add editable controls (such as **Textbox** controls) to edit each of the properties of an entity object.
6. Add two buttons to the **EditItemTemplate** section to enable the user to save or cancel edits on the entity object. Set the buttons' **CommandName** properties to **Update** and **Cancel** respectively, so that the buttons raise **UpdateCommand** and **CancelCommand** events on the **DataList** control.
7. Right-click the **DataList** control, and then click **End Template Editing** in the shortcut menu.

Figure B.3 shows sample Item Templates to display and edit **CustomerEntity** objects.

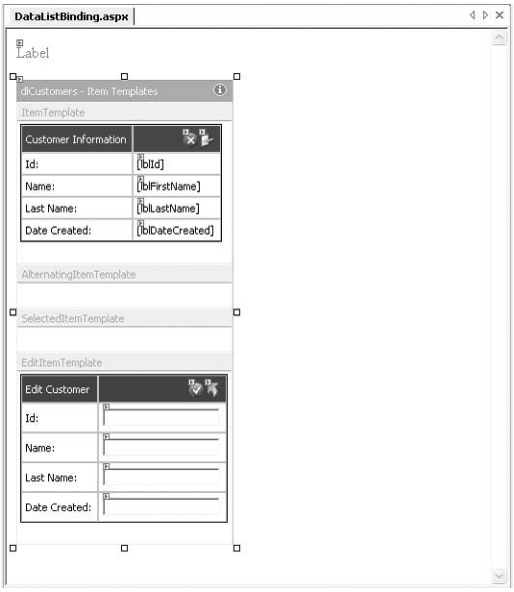


Figure B.3
Designing item templates for a DataList control

The **ItemTemplate** section in Figure B.3 has four **Label** controls, to display the **ID**, **FirstName**, **LastName**, and **DateCreated** properties for a **CustomerEntity** object. There are also two **ImageButton** controls to delete or edit the entity object.

The **EditItemTemplate** section in Figure B.3 has four **Textbox** controls, to edit the **ID**, **FirstName**, **LastName**, and **DateCreated** properties for a **CustomerEntity** object. There are also two **ImageButton** controls to save or cancel the edits.

Specifying Data Binding Expressions for a DataList Control

Before you can specify data binding expressions for a **DataList** control, you must bind the **DataList** control to a data source.

► To bind the DataList control to a data source

1. Select the **DataList** control that you want to perform data binding on.
2. In the Properties window, set the **DataSource** property to the entity collection object that you want to bind to (for example, the "customers" collection object created previously).

This causes Visual Studio .NET to add a **datasource** attribute to the **<asp:datalist>** tag in the .aspx file.

```
<asp:datalist id="dlCustomers" runat="server" datasource="<%# customers %>" >
</asp:datalist>
```

After you bind the **DataList** control to an entity collection object, you must define data binding expressions to specify how the **DataList** control displays and edits entity objects in each row.

► **To define data binding expressions**

1. Open the **ItemTemplate** for the **DataList** control.
2. Specify data binding expressions for each of the bindable controls in the **ItemTemplate**, as follows:
   a. Select a bindable control (such as the **lblDateCreated** control) in the template.
   b. Open the **DataBindings** property editor on the bindable control.
   c. Select one of the control's properties and bind it to a property on the entity object.
3. Repeat steps 1 and 2 for the **EditItemTemplate** for the **DataList** control.

The **DataBindings** property editor makes it easy for you to bind to specific properties on your entity class. The entity's properties are listed in the **Container.DataItem** node because you are using a typed collection class.

When you define data binding expressions for controls in a **DataList** template, Visual Studio .NET adds code such as the following to your .aspx file.

```
<asp:label
      id="lblDateCreated"
      runat="server"
      Text='<%# DataBinder.Eval(Container,
                                "DataItem.DateCreated",
                                "{0:dd-MM-yyyy}") %>'>
</asp:label>
```

The next section describes how and when the binding expressions are evaluated at run time.

Performing Data Binding at Run Time

This section describes how a Web Form can bind a **DataList** control to **CustomerEntity** objects held in a **CustomerCollection** object. The code-behind class for the Web Form has a static constructor to create the **CustomerEntity** objects and insert them into a static **CustomerCollection** object.

```
public class DataListBinding : System.Web.UI.Page
{
  // CustomerCollection component (as defined earlier), bound to a DataList
  protected CustomerCollection customers;

  // Static collection of CustomerEntity objects
  private static CustomerCollection CustomersList = new CustomerCollection();
```

```
  // Create CustomerEntity objects and add to static collection
  static DataListBinding()
  {
    CustomerEntity tempCustomer = new CustomerEntity();
    tempCustomer.ID = 5;
    tempCustomer.FirstName = "Guy";
    tempCustomer.LastName = "Gilbert";
    tempCustomer.DateCreated = new DateTime( 1948, 3, 9 );
    CustomersList.Add(tempCustomer);

    tempCustomer = new CustomerEntity();
    tempCustomer.ID = 11;
    tempCustomer.FirstName = "Kendall";
    tempCustomer.LastName = "Keil";
    tempCustomer.DateCreated = new DateTime( 1974, 9, 4 );
    CustomersList.Add(tempCustomer);

   // Plus other sample entities...
  }

  // Plus other members...
}
```

The form's **Page_Load** method performs data binding the first time the page is displayed. Note that the **Page_Load** method does not perform data binding on subsequent postbacks; this task is performed by the other event handler methods, described later.

```
public class DataListBinding : System.Web.UI.Page
{
  private void Page_Load(object sender, System.EventArgs e)
  {
    if (!IsPostBack)
    {
      // Set the "customers" instance variable to refer to the
      // static CustomerCollection object, "CustomersList"
      customers = CustomersList;

      // Re-evaluate all the data binding expressions on this page
      DataBind();
    }

    lblMessage.Visible = false;
  }

  // Plus other members...
}
```

The form defines event handler methods for the **DeleteCommand**, **EditCommand**, **UpdateCommand**, and **CancelCommand** events on the **DataList** control as follows:

- **DeleteCommand event handler method**–This method removes the currently selected entity from the collection, and rebinds the collection to the **DataList** control.

```
private void dlCustomers_DeleteCommand(
                        object source,
                         System.Web.UI.WebControls.DataListCommandEventArgs e)
{
  // Remove the item, and rebind
  customers = CustomersList;
  customers.RemoveAt(e.Item.ItemIndex);
  DataBind();
}
```

- **EditCommand event handler method**–This method sets the **EditItemIndex** property on the **DataList** control, to tell the **DataList** control which entity in the collection is to be edited.

```
private void dlCustomers_EditCommand(
                        object source,
                         System.Web.UI.WebControls.DataListCommandEventArgs e)
{
  // Set the edit index
  dlCustomers.EditItemIndex = e.Item.ItemIndex;

  // Rebind
  customers = CustomersList;
  DataBind();
}
```

- **UpdateCommand event handler method**–This method updates the currently selected entity object with the new values entered by the user while in edit mode.

```
private void dlCustomers_UpdateCommand(
                        object source,
                         System.Web.UI.WebControls.DataListCommandEventArgs e)
{
  // Find the TextBox controls for the current item on the DataList
  TextBox id    = e.Item.FindControl("txtId") as TextBox;
  TextBox fname = e.Item.FindControl("txtFirstName") as TextBox;
  TextBox lname = e.Item.FindControl("txtLastName") as TextBox;
  TextBox date  = e.Item.FindControl("txtDateCreated") as TextBox;

  // Get the appropriate entity and update values
  CustomerEntity customer = CustomersList[e.Item.ItemIndex];
```

```
    // Set the new values
    if (id != null && fname != null && lname != null && date != null)
    {
      try
      {
        customer.ID = Int32.Parse(id.Text);
        customer.FirstName = fname.Text;
        customer.LastName = lname.Text;
        customer.DateCreated = DateTime.Parse(date.Text);
      }
      catch (Exception ex)
      {
        lblMessage.Text = "Incorrect data: " + ex.Message;
        lblMessage.Visible = true;
      }

      // Switch back to view mode
      dlCustomers.EditItemIndex = -1;

      // Rebind
      customers = CustomersList;
      DataBind();
    }
    else
    {
      throw new Exception( "Invalid page structure" );
    }
  }
```

- **CancelCommand event handler method**–This method sets the **EditItemIndex** property on the **DataList** control to **-1**, to tell the **DataList** control that no entity is to be edited. This causes the **DataList** to redraw the entity in view mode instead of edit mode.

```
private void dlCustomers_CancelCommand(
                        object source,
                         System.Web.UI.WebControls.DataListCommandEventArgs e)
{
  // Reset the edit index
  dlCustomers.EditItemIndex = -1;

  // Rebind
  customers = CustomersList;
  DataBind();
}
```

These event handler methods enable the user to view, edit, and delete **CustomerEntity** objects by using the **DataList** control on the Web Form.

Data Binding a Collection of Entity Objects to a DataGrid Control

This section describes how to bind a collection of entity objects to a **DataGrid** control. This allows the user to view all the entity objects in a grid on the Web page, and to edit and delete entity objects.

This section includes the following topics:

- Performing Simple Data Binding to a **DataGrid** Control
- Performing Custom Data Binding to a **DataGrid** Control

This section uses the **CustomerEntity** and **CustomerCollection** classes introduced in the earlier sections of this chapter.

Performing Simple Data Binding to a DataGrid Control

This section describes the default data binding support provided by the **DataGrid** control.

► **To perform simple data binding to a DataGrid control**

1. Write code in your form to create the data that you want to be displayed in the **DataGrid** control. For example, the following code creates an **ArrayList** object and populates it with sample **CustomerEntity** objects.

```
public class GridBinding : System.Web.UI.Page
{
  // Customer list to be bound
  private ArrayList customerList = new ArrayList();

  // Populate the customer list
  private void SetupSources()
  {
    CustomerEntity tempCustomer = new CustomerEntity();
    tempCustomer.ID = 11;
    tempCustomer.FirstName = "Kendall";
    tempCustomer.LastName = "Keil";
    tempCustomer.DateCreated = new DateTime(2003, 10, 10);
    customerList.Add(tempCustomer);

    tempCustomer = new CustomerEntity();
    tempCustomer.ID = 22;
    tempCustomer.FirstName = "Jennifer";
    tempCustomer.LastName = "Riegle";
    tempCustomer.DateCreated = new DateTime(2003, 10, 10);
    customerList.Add(tempCustomer);

    // Plus other sample entities...
  }
}
```

2. Drag a **DataGrid** control from the Toolbox and drop it onto your Web Form.
3. Write code in the **Page_Load** method to bind the **DataGrid** control to your data.

```
public class GridBinding : System.Web.UI.Page
{
  // DataGrid control
  protected System.Web.UI.WebControls.DataGrid dgMain;

  private void Page_Load(object sender, System.EventArgs e)
  {
    // Populate the customer list
    SetupSources();

    // Bind the DataGrid to the customer list
    dgMain.DataSource = customerList;
    dgMain.DataBind();
  }
}
```

Figure B.4 shows how the **DataGrid** control appears when it is rendered in the browser.

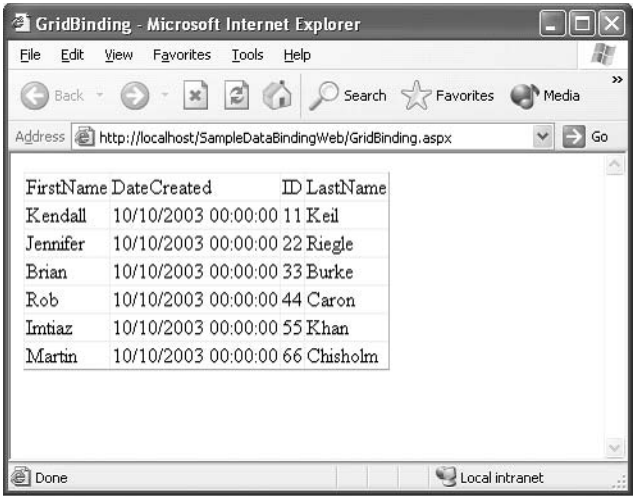


Figure B.4
Simple data binding for a DataGrid control

The **DataGrid** control has an **AutogenerateColumns** property, which is **true** by default. This causes the **DataGrid** control to perform default formatting for the columns when the control is rendered.

Performing Custom Data Binding to a DataGrid Control

This section describes how to perform custom formatting for a **DataGrid** control to improve its visual appearance when it is rendered.

► **To perform custom data binding to a DataGrid control**

1. In the Designer, select the **DataGrid** control.
2. Right-click the **DataGrid** control, and then click **Property Builder** on the shortcut menu.
3. In the **Property Builder** dialog box, customize the appearance of the **DataGrid** control as required. For example:
   - Click the **General** tab in the navigation pane, and then specify the data source, headers and footers, and sorting capabilities of the **DataGrid** control.
   - Click the **Columns** tab in the navigation pane, and then choose the properties of the entity object that you want to display in the **DataGrid** control. You can change the formatting string for these properties to specify how values are formatted in the columns. Alternatively, you can click the **Convert this column into a Template Column** link to use the templating feature described for the **DataList** control earlier in this chapter.
   - Click the **Paging** tab in the navigation pane to allow paging in the **DataGrid** control. You can also define the number of rows per page, and customize how the user moves between pages.
   - Click the **Format** tab in the navigation pane to specify fonts, colors, and other formatting characteristics for the columns, headers, and footers on the **DataGrid** control.
   - Click the **Borders** tab in the navigation pane to specify cell margins and borders on the **DataGrid** control.

The **Property Builder** dialog box also allows you to add **Delete**, **Edit**, **Update**, and **Cancel** buttons to the **DataGrid** control to enable the user to edit data directly in the **DataGrid** control.

► **To add Edit, Update, Cancel, and Delete buttons to the DataGrid control**

1. In the **Property Builder** dialog box, click the **Columns** tab in the navigation pane.
2. In the **Available columns** list, expand **Button Columns**, select **Edit, Update, Cancel**, and click the **>** button. This adds the **Edit, Update, Cancel** button column to the **Selected columns** list.
3. In the **Available columns** list, expand **Button Columns**, click **Delete**, and then click the **>** button. This adds the **Delete** button column to the **Selected columns** list.

You must then define event handler methods for the **DeleteCommand**, **EditCommand**, **UpdateCommand**, and **CancelCommand** events on the **DataGrid**

control as follows. (These methods are similar to the corresponding **DataList** methods shown earlier in this chapter.)

```
private void grdCustomers_DeleteCommand(
                      object source,
                       System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
  customers = CustomersList;
  customers.RemoveAt(e.Item.ItemIndex);
  DataBind();
}

private void grdCustomers_EditCommand(
                      object source,
                       System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
  customers = CustomersList;
  dgCustomers.EditItemIndex = e.Item.ItemIndex;
  DataBind();
}

private void grdCustomers_UpdateCommand(
                      object source,
                       System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
  // Find the updated controls
  TextBox id    = (TextBox) e.Item.Cells[0].Controls[0];
  TextBox fname = (TextBox) e.Item.Cells[1].Controls[0];
  TextBox lname = (TextBox) e.Item.Cells[2].Controls[0];
  TextBox date  = (TextBox) e.Item.Cells[3].Controls[0];

  // Get the appropriate entity and update values
  CustomerEntity customer = CustomersList[e.Item.ItemIndex];

  // Set the new values
  try
  {
    customer.ID = Int32.Parse(id.Text);
    customer.FirstName = fname.Text;
    customer.LastName = lname.Text;
    customer.DateCreated = DateTime.Parse(date.Text);
  }
  catch (Exception ex)
  {
    lblMessage.Text = "Incorrect data: " + ex.Message;
    lblMessage.Visible = true;
  }

  // Switch back to view mode
  dgCustomers.EditItemIndex = -1;
```

```
  // Rebind
  customers = CustomersList;
  DataBind();
}

private void grdCustomers_CancelCommand(
                      object source,
                       System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
  dgCustomers.EditItemIndex = -1;
  customers = CustomersList;
  DataBind();
}
```

The **DataGrid** control now allows the user to edit the properties in an entity object, as shown in Figure B.5.

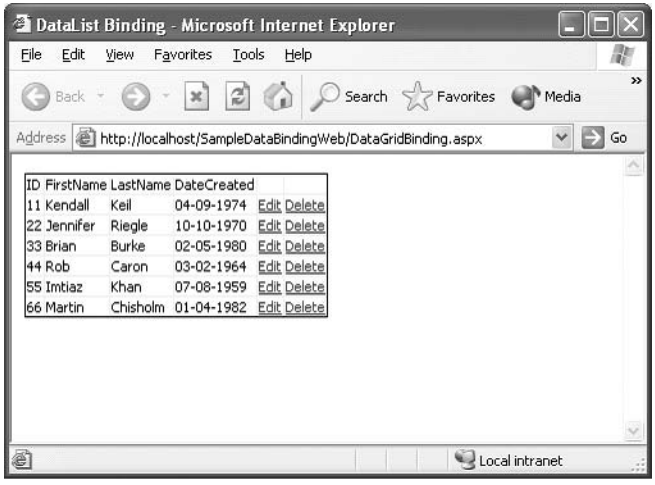


Figure B.5
Custom data binding for a DataGrid control

When the user clicks the **Edit** link on a row in the **DataGrid** control, text boxes appear in each column to enable the user to edit the values; the user can click **Update** or **Cancel** to save or cancel these changes to the underlying entity object. Alternatively, the user can click **Delete** to delete an entity object.

How To: Design Data Maintenance Forms to Support Create, Read, Update, and Delete Operations

This section describes how to design and implement Web Forms to support Create, Read, Update, and Delete ("CRUD") operations on data in a data store.

This section includes the following topics:

- Defining Business Entities
- Defining Data Access Logic Components
- Defining Business Components
- Designing CRUD Web Forms

The examples in this section are based on three business entities: **Customer**, **Order**, and **OrderItem**. Figure B.6 shows the relationships between these business entities.

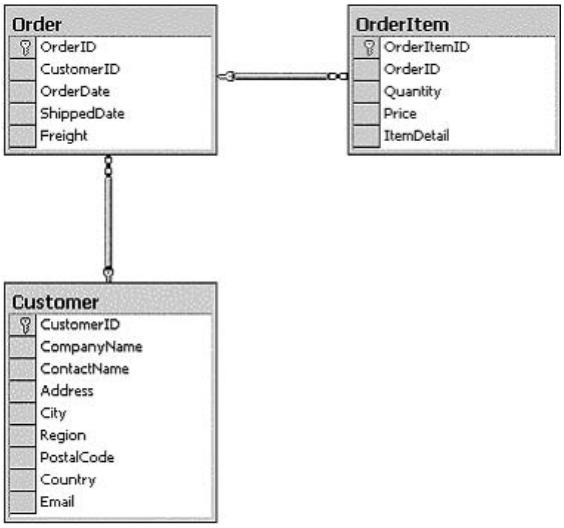


Figure B.6
Relationships between business entities

As shown in Figure B.6, each customer can have many orders, and each order can have many order items.

Defining Business Entities

There are several ways to represent business entities as they are passed internally between the components and layers in a distributed application, including:

- Data set
- Typed data set
- Data reader

- XML
- Custom "business entity" objects

This example uses typed data sets because they are straightforward to create and maintain. Additionally, typed data sets can be easily bound to **DataGrids** and other controls on a Web Form.

You can either define a single data set that encompasses all the business entities or create a separate typed data set for each business entity. The latter approach is preferred because it is easier to add new typed data sets if the business model grows in the future.

► **To define a typed data set for a business entity**

1. Open a Visual Studio .NET project.
2. In Solution Explorer, right-click the project. On the shortcut menu, point to **Add**, and then click **Add New Item**.
3. In the **Add New Item** dialog box, select the **Data Set** template. Enter a name for the new data set, such as CustomerDS.xsd, and then click **OK**.
4. Open Server Explorer in Visual Studio .NET, and then locate the database table that you want to create a data set for.
5. Drag the database table onto the Designer surface. Visual Studio .NET creates an XML schema to represent the columns in the database table.
6. Right-click the Designer surface, and make sure there is a check mark next to the **Generate Dataset** menu item.
7. Save the new file. Visual Studio .NET creates a typed data set class (for example, CustomersDS.cs), based on the information in the XML schema.

To view the typed data set class, click the **Show all Files** icon in the toolbar in Solution Explorer and expand the XML schema file (for example, CustomerDS.xsd). This reveals the typed data set class file (for example, CustomerDS.cs). The class has properties and methods to provide named and type-safe access to the fields in the business entity.

Defining Data Access Logic Components

Data access logic components encapsulate access to the underlying data in the data store. Data access logic components typically have methods to perform the following tasks:

- Create a new entity in the data store
- Update an existing entity in the data store
- Delete an existing entity in the data store
- Get a single entity in the data store
- Get all entities in the data store

For simplicity and maintainability, it is a good idea to define a separate data access logic component for each business entity. You can define a common base class to hold any behavior or properties that are the same across the data access logic components. For example, the **BaseDalc** class shown in the following code sample defines common features for the **CustomerDalc**, **OrderDalc**, and **OrderItemDalc** subclasses. **BaseDalc** retrieves the **"ConnectionString"** setting from the application configuration file (Web.config), and stores this setting in an instance variable named **connectionStr**. This setting contains the connection string that all data access logic components use when they want to access the database.

```
using System;
using System.Configuration;
using System.Reflection;
using System.Resources;
using System.Runtime.InteropServices;

namespace CRUDSample.DataAccessComponent
{
  // Base class for all Data Access Logic Components
  [ComVisible(false)]
  public class BaseDalc
  {
    // Database connection string
    protected string connectionStr;

    // Retrieve the resource file corresponding to this assembly
    protected static ResourceManager ResMgr =
         new ResourceManager(typeof(BaseDalc).Namespace + ".DalcMessages",
                             Assembly.GetExecutingAssembly());

    // Default constructor
    public BaseDalc()
    {
      connectionStr = ConfigurationSettings.AppSettings["ConnectionString"];
      if (connectionStr == null)
      {
        throw new ConfigurationException(
          ResMgr.GetString("ConfigurationException.ConnectionStringNotFound"));
      }
    }

    // Get the resource file corresponding to this assembly
    protected ResourceManager ResourceMgr
    {
      get { return ResMgr; }
    }
  }
}
```

To implement the data access logic component for a particular business entity, follow these guidelines:

- It is a good idea to write stored procedures to perform CRUD operations for a particular business entity in the database. Stored procedures give you an opportunity to centralize data access logic in the database and to achieve reuse.
- Devise a consistent naming convention for your CRUD stored procedures. For example, the CRUD stored procedures for the "customer" business entity might be named **Customer_Insert**, **Customer_Update**, **Customer_Delete**, **Customer_SelectByID**, and **Customer_SelectAll**.
- Evaluate using the Data Access Application Block for optimized data access code that helps you call stored procedures and issue SQL text commands against a SQL server database. The Data Access Application Block is available on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daab-rm.asp*).

The following code shows a sample implementation for the **CustomerDalc** data access logic component. **CustomerDalc** inherits from **BaseDalc**, and provides CRUD methods relating to the "customer" business entity; these methods make use of the Data Access Application Block for efficiency.

```
using System;
using System.Data;
using System.Data.SqlClient;
using Microsoft.ApplicationBlocks.Data;
using System.Runtime.InteropServices;

using CRUDSample.BusinessEntity;

namespace CRUDSample.DataAccessComponent
{
  [ComVisible(false)]
  public class CustomerDalc : BaseDalc
  {
    // Default constructor
    public CustomerDalc() : base()
    {}

    // Create a new customer record in the data store
    public int Create(string companyName, string contactName,
                      string address, string city,  string region,
                      string postalCode, string country, string email)
    {
      try
      {
        // Execute the "insert" stored procedure
        int customerId = (int)SqlHelper.ExecuteScalar(this.connectionStr,
                                                      "Customer_Insert",
                                                       companyName, contactName,
                                                       address, city, region,
                                                       postalCode, country, email);
```

```
    // Return the customer id
    return customerId;
  }
  catch (SqlException e)
  {
    throw new TechnicalException(
        this.ResourceMgr.GetString("TechnicalException.CantCreateCustomer",
        System.Globalization.CultureInfo.CurrentUICulture), e);
  }
}

// Update an existing customer record in the data store
public void Update(int customerId, string companyName, string contactName,
                   string address, string city, string region,
                   string postalCode, string country, string email)
{
  try
  {
    // Execute the "update" stored procedure
    SqlHelper.ExecuteNonQuery(this.connectionStr,
                              "Customer_Update", customerId,
                               companyName, contactName,
                               address, city, region,
                               postalCode, country, email);
  }
  catch (SqlException e)
  {
    throw new TechnicalException(
        this.ResourceMgr.GetString("TechnicalException.CantUpdateCustomer",
        System.Globalization.CultureInfo.CurrentUICulture), e);
  }
}

// Delete an existing customer record in the data store
public void Delete(int customerId)
{
  try
  {
    // Execute the "delete" stored procedure
    SqlHelper.ExecuteNonQuery(this.connectionStr,
                              "Customer_Delete",
                               customerId);
  }
  catch (SqlException e)
  {
    throw new TechnicalException(
        this.ResourceMgr.GetString("TechnicalException.CantDeleteCustomer",
        System.Globalization.CultureInfo.CurrentUICulture), e);
  }
}
```

```
// Return the customer with the specified ID
public CustomerDs.Customer GetById(int customerId)
{
  try
  {
    CustomerDs customer = new CustomerDs();

    // Execute the "select single entity" stored procedure
    using (SqlDataReader reader = SqlHelper.ExecuteReader(
                                                  this.connectionStr,
                                                  "Customer_SelectByID",
                                                  customerId))
    {
      // Populate the dataset with reader rows
      SQLHelperExtension.Fill(reader,
                              customer,
                                customer.Customers.TableName,
                                0, 1);
    }

    // Return the first row
    return customer.Customers[0];
  }
  catch (SqlException e)
  {
    throw new TechnicalException(
        this.ResourceMgr.GetString("TechnicalException.CantGetCustomer",
        System.Globalization.CultureInfo.CurrentUICulture), e);
  }
}

// Return all customers, as a typed data set object
public CustomerDs GetAll()
{
  try
  {
    CustomerDs customer = new CustomerDs();

    // Execute the "select all entities" stored procedure
    using (SqlDataReader reader = SqlHelper.ExecuteReader(
                                                  this.connectionStr,
                                                  "Customer_SelectAll"))
    {
      SQLHelperExtension.Fill(reader,
                              customer,
                                customer.Customers.TableName,
                                0, 0);
    }

    // Return the customer data set
    return customer;
  }
```

```
      catch (SqlException e)
      {
        throw new TechnicalException(
            this.ResourceMgr.GetString("TechnicalException.CantGetAllCustomers",
            System.Globalization.CultureInfo.CurrentUICulture), e);
      }
    }
  }
}
```

The **OrderDalc** and **OrderItemDalc** data access logic components follow a similar pattern to the **CustomerDalc** data access logic component; **OrderDalc** provides CRUD methods for "order" business entities, and **OrderItemDalc** provides CRUD methods for "order item" business entities.

Defining Business Components

The general purpose of business components is to encapsulate business rules in a distributed application. Business components shield other parts of the application, such as the presentation layer, from direct interactions with data access logic components.

In CRUD applications, there are usually few business rules that need to be encapsulated by the business components. Therefore, the methods in a business component typically act as simple wrappers for the underlying methods in the data access logic component.

The following code shows a sample implementation for the **CustomerBc** business component. Each method in **CustomerBc** calls the corresponding method in the **CustomerDalc** data access logic component.

```
using System;
using System.Runtime.InteropServices;
using CRUDSample.BusinessEntity;
using CRUDSample.DataAccessComponent;

namespace CRUDSample.BusinessComponent
{
  [ComVisible(false)]
  public class CustomerBc : BaseBc
  {
    // Default constructor
    public CustomerBc() : base()
    {}

    // Create a new customer
    public int Create(string companyName, string contactName,
                      string address, string city,  string region,
                      string postalCode, string country, string email)
    {
```

```
    // Create the DALC component
    CustomerDalc customerDalc = new CustomerDalc();

    // Create a new customer using the DALC component
    int customerId = customerDalc.Create(companyName, contactName,
                                         address, city, region,
                                         postalCode, country, email);
    // Return the customer id
    return customerId;
  }

  // Update an existing customer
  public void Update(int customerId, string companyName, string contactName,
                     string address, string city, string region,
                     string postalCode, string country, string email)
  {
    // Create the DALC component
    CustomerDalc customerDalc = new CustomerDalc();

    // Update an existing customer using the DALC component
    customerDalc.Update(customerId, companyName, contactName, address,
                        city, region, postalCode, country, email);
  }

  // Delete an existing customer
  public void Delete(int customerId)
  {
    // Create the DALC component
    CustomerDalc customerDalc = new CustomerDalc();

    // Delete an existing customer using the DALC component
    customerDalc.Delete(customerId);
  }

  // Return the customer with the specified ID
  public CustomerDs.Customer GetById(int customerId)
  {
    // Create the DALC component
    CustomerDalc customerDalc = new CustomerDalc();

    // Get the specified customer using the DALC component
    return customerDalc.GetById(customerId);
  }

  // Return all customers, as a typed dataset object
  public CustomerDs GetAll()
  {
    // Create the DALC component
    CustomerDalc customerDalc = new CustomerDalc();

    // Get all customers using the DALC component
    return customerDalc.GetAll();
  }
 }
}
```

The **OrderBc** and **OrderItemBc** business components follow a similar pattern to the **CustomerBc** business component.

Designing CRUD Web Forms

This section describes how to design CRUD Web Forms to enable users to create, read, update, and delete customer records, order records, and order item records in the data store. There are six Web Forms in this sample scenario:

- **CustomerList.aspx**–Displays a list of customer records in a **DataGrid** control and allows the user to delete a customer record.
- **EditCustomer.aspx**–Enables the user to create a new customer record and edit an existing customer record.
- **OrderList.aspx**–Displays a list of a customer's order records in a **DataGrid** control and allows the user to delete an order record.
- **EditOrder.aspx**–Enables the user to create a new order record and edit an existing order record.
- **OrderItemList.aspx**–Displays a list of order item records in a **DataGrid** control and allows the user to delete an order item record.
- **EditOrderItem.aspx**–Enables the user to create a new order item record and edit an existing order item record.

The following sections describe each of these Web pages, focusing on the CRUD operations performed by each Web page.

Defining the CustomerList.aspx Web Page

The CustomerList.aspx Web page displays customer records in a **DataGrid** control, as shown in Figure B.7.

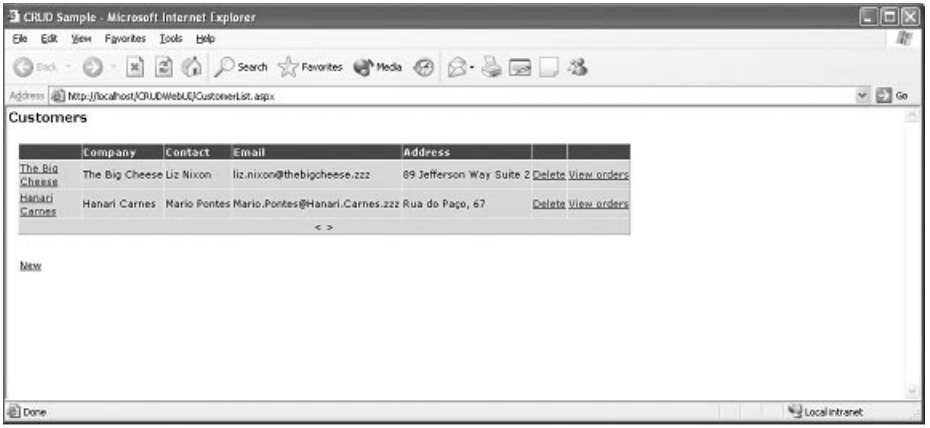


Figure B.7
Customer list Web page

Notice the following features in CustomerList.aspx:

- The Web page contains a **DataGrid** control that displays a page of customers, one customer per row. The arrows at the bottom of the **DataGrid** enable the user to page backward and forward through customer records in the data store; this is an important feature in CRUD forms, where there might be many records to display.
- The **DataGrid** shows a subset of information for each customer, instead of displaying all the fields. It is a good idea to display a subset of information in CRUD forms, because there may be too many fields to display neatly in a single **DataGrid**.
- The first column in the **DataGrid** contains a hyperlink to a details Web page (EditCustomer.aspx), where the user can display and edit full details for an individual customer.
- Each row in the **DataGrid** has a "Delete" hyperlink, to delete an existing customer in the data store. Each row also has a "View orders" hyperlink, which redirects the user to another Web page (OrderList.aspx) to display the orders for this customer.
- The "New" hyperlink at the bottom of CustomerList.aspx redirects the user to the details Web page (EditCustomer.aspx), where the user can enter details for a new customer.

► **To implement these features in the CustomerList.aspx Web page**

1. In Visual Studio .NET, drag a **DataGrid** control from the Toolbox onto the Web page.
2. Right-click the **DataGrid** control and select **Property Builder** from the shortcut menu.
3. In the **Property Builder** dialog box, click the **Columns** tab in the navigation pane.
4. Design the first (hyperlink) column as follows:
   - In the **Available columns** list, select **HyperLink Column**.
   - Click the **>** button to add the **HyperLink Column** to the **Selected columns** list.
   - In the **Text field** text box, enter "CompanyName" so that the hyperlink text displays the company name.
   - In the **URL field** text box, enter "CustomerID" so that the **CustomerID** property is used to generate the URL of the hyperlink.
   - In the **URL format string** text box, enter "EditCustomer.aspx?CustomerId={0}" so that the hyperlink redirects the user to EditCustomer.aspx to edit the current customer.
5. Design the "Company" column as follows:
   - In the **Available columns** list, select **Bound Column**.
   - Click the **>** button to add the **Bound Column** to the **Selected columns** list.

- In the **Header text** text box, enter "Company" as the header text for this column.
- In the **Data Field** text box, enter "CompanyName" so that the column is bound to the **CompanyName** property on the customer record.

6. Repeat step 5 for the "Contact," "Email," and "Address" columns. Bind the columns to the **ContactName**, **Email**, and **Address** properties on the customer record.
7. Design the "Delete" column as follows:
   - In the **Available columns** list, select **Template Column**.
   - Click the **>** button to add the **Template Column** to the **Selected columns** list.

   Note: After you finish using the Property Builder, you must add a "Delete" link button control into this template column to perform the "delete" command. To add a "Delete" link button control, edit the template for the **DataGrid** control; in the **ItemTemplate** section for this column, drag a **LinkButton** control from the Toolbox, and set its **CommandName** property to **Delete**.

8. Design the "View orders" column as follows:
   - In the **Available columns** list, select **HyperLink Column**.
   - Click the **>** button to add the **HyperLink Column** to the **Selected columns** list.
   - In the **Text** text box, enter "View orders" as the fixed text for the hyperlink.
   - In the **URL field** text box, enter "CustomerID" so that the **CustomerID** property is used to generate the URL of the hyperlink.
   - In the **URL format string** text box, enter "OrderList.aspx?CustomerId={0}" so that the hyperlink redirects the user to OrderList.aspx to display the orders for the current customer.
9. Specify paging behavior for the **DataGrid** control, as follows:
   - Click the **Paging** tab in the navigation pane of the **Property Builder** dialog box.
   - Place a check mark in the **Allow paging** check box.
   - Enter a suitable number in the **Page size** text box (for example, 10).
   - Place a check mark in the **Show navigation buttons** check box, and configure the page navigation details as you require.

The CustomerList.aspx Web page also has a "New" hyperlink to enable the user to create new customer records in the data store.

► To add this hyperlink to the Web page

1. View the HTML for the Web page.
2. Add an **<a>** HTML element as follows, to define a hyperlink to the EditCustomer.aspx Web page.

```
<a href="EditCustomer.aspx">New</a>
```

Note that EditCustomer.aspx is the same Web page that is used to edit existing customer records, but the absence of a customer ID in the URL indicates that a new customer record is to be created.

The next task is to write code in the code-behind class for CustomerList.aspx.

► To implement the functionality of the Web page

1. Add code to the **Page_Load** method to bind the **DataGrid** control to the customer records in the data store. The following code uses a helper method named **LoadCustomers**, because the task of loading customers will be needed elsewhere in the Web page. The **LoadCustomers** method creates a **CustomerBc** business component and calls the **GetAll** method to return all the customer records as a typed data set. The **DataGrid** control is then bound to the Customers table in the typed data set.

```
using CRUDSample.BusinessComponent;
using CRUDSample.BusinessEntity;

namespace CRUDSample.WebUI
{
  [ComVisible(false)]
  public class CustomerList : System.Web.UI.Page
  {
    // DataGrid to display a list of customers
    protected System.Web.UI.WebControls.DataGrid dgCustomers;

    private void Page_Load(object sender, System.EventArgs e)
    {
      if (!Page.IsPostBack)
      {
        LoadCustomers();
      }
    }

    private void LoadCustomers()
    {
      // Create a "customer" business component
      CustomerBc customerBc = new CustomerBc();

      // Get all customers
      CustomerDs customer = customerBc.GetAll();
```

```
      // Bind the DataGrid control to the "Customers" table in the dataset
      dgCustomers.DataSource = customer.Customers;
      dgCustomers.DataBind();
    }
    // Plus other members...
  }
}
```

2. Define an event handler for the **ItemCreated** event on the **DataGrid**. The event handler method adds client-side script to the **Delete** button for the new item. The client-side script displays an "Are you sure?" confirmation message when the user tries to delete the item to give the user an opportunity to cancel the deletion.

```
namespace CRUDSample.WebUI
{
  [ComVisible(false)]
  public class CustomerList : System.Web.UI.Page
  {
    private void customersGrid_ItemCreated(object sender,
                                           DataGridItemEventArgs e)
    {
      if (e.Item.ItemType == ListItemType.Item ||
          e.Item.ItemType == ListItemType.AlternatingItem)
      {
        // Add an "Are you sure?" confirmation dialog box to the Delete button
        LinkButton deleteButton =
                          e.Item.FindControl("lnkDelete") as LinkButton;
        if (deleteButton != null)
          deleteButton.Attributes.Add(
                           "onclick",
                           "return window.confirm('Are you sure ?');");
      }
    }
    // Plus other members...
  }
}
```

3. Define an event handler to delete the selected customer record from the data store.

```
namespace CRUDSample.WebUI
{
  [ComVisible(false)]
  public class CustomerList : System.Web.UI.Page
  {
    // Handle "item" events from the DataGrid control
    private void customersGrid_ItemCommand(object source,
                                           DataGridCommandEventArgs e)
    {
      // Is it the "Delete" command?
      if (e.CommandName == "Delete")
      {
```

```
        // Get the CustomerID of the selected customer
        int customerId = (int)dgCustomers.DataKeys[e.Item.ItemIndex];

        // Delete the selected customer
        CustomerBc customerBc = new CustomerBc();
        customerBc.Delete (customerId);

        // Update the page index
        if (dgCustomers.Items.Count == 1 && dgCustomers.CurrentPageIndex > 0)
        {
          dgCustomers.CurrentPageIndex -= 1;
        }

        // Rebind
        LoadCustomers();
      }
    }
    // Plus other members...
  }
}
```

4. Define an event handler method for the **PageIndexChanged** event on the **DataGrid** control. Implement the event handler method as follows to support paging behavior in the **DataGrid** control.

```
namespace CRUDSample.WebUI
{
  [ComVisible(false)]
  public class CustomerList : System.Web.UI.Page
  {
    private void customersGrid_PageIndexChanged(
                                    object source,
                                    DataGridPageChangedEventArgs e)
    {
      // Adjust the current page index
      dgCustomers.CurrentPageIndex = e.NewPageIndex;

      // Rebind
      LoadCustomers();
    }
    // Plus other members...
  }
}
```

The preceding code samples show how to display a collection of "customer" business entities in a **DataGrid** control, and how to incorporate hyperlinks that enable the user to create new customers, edit existing customers, and delete existing customers in the data store.

Defining the EditCustomer.aspx Web Page

The EditCustomer.aspx Web page allows the user to edit details for an existing customer record and to enter details for a new customer record. EditCustomer.aspx is shown in Figure B.8, with some sample data entered by the user.

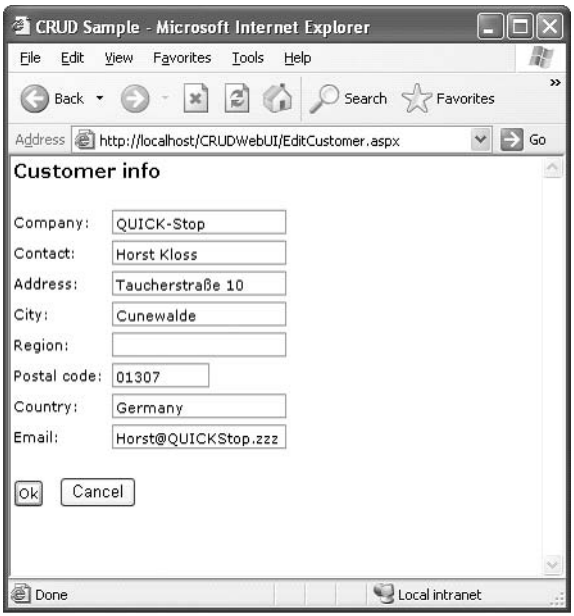


Figure B.8
Edit customer Web page

Notice the following features in EditCustomer.aspx:

- The Web page contains a series of text boxes, one for each field in the "customer" business entity.
- If the user invokes the Web page to edit an existing customer record, the text fields are populated with the customer's current details. However, if the user invokes the Web page to create a new customer record, the text fields are blank initially.
- When the user clicks the **Ok** button, the Web page either updates an existing record or creates a new record, depending on whether the user is editing an existing customer or creating a new customer. The user is then redirected back to the CustomerList.aspx Web page to view the customer list.
- When the user clicks the **Cancel** button, the user is immediately redirected back to CustomerList.aspx without any modifications to the data store.

To design this Web page, add **Label**, **TextBox**, and **Button** controls to the form as shown in Figure B.8. There are no special user interface issues that you must take into account for this Web page.

► To implement the functionality for the EditCustomer.aspx Web page

1. Add code to the **Page_Load** method to populate the text boxes on the form if the user is editing an existing customer record; in this scenario, the query string contains a **CustomerId** parameter to identify the current customer. The customer ID must be stored in the **ViewState**, so that it is available in subsequent postbacks.

```
using CRUDSample.BusinessComponent;
using CRUDSample.BusinessEntity;

namespace CRUDSample.WebUI
{
  [ComVisible(false)]
  public class EditCustomer : System.Web.UI.Page
  {
    private void Page_Load(object sender, System.EventArgs e)
    {
      if (!Page.IsPostBack)
      {
        // Is the user editing an existing customer record?
        if (Request.QueryString["CustomerId"] != null)
        {
          // Add customerID to ViewState, so that it is available in postbacks
          ViewState.Add("CustomerId",
                        int.Parse(Request.QueryString["CustomerId"]));

          // Create a "customer" business component
          CustomerBc customerBc = new CustomerBc();

          // Get the specified customer's details from the data store
          CustomerDs.Customer customer =
                        customerBc.GetById((int)ViewState["CustomerId"]);

          // Populate the form controls with customer details
          txtCompany.Text = customer.CompanyName;
          txtContact.Text = customer.ContactName;
          txtAddress.Text = customer.Address;
          txtCity.Text = customer.City;
          txtRegion.Text = customer.Region;
          txtPostal.Text = customer.PostalCode;
          txtCountry.Text = customer.Country;
          txtEmail.Text = customer.Email;
        }
      }
    }
    // Plus other members...
  }
}
```

2. Define an event handler method for the **OK** button-click event. If the **ViewState** contains a **CustomerId** value, update the current customer record in the data store; otherwise, create a new customer record in the data store. Finally, redirect the user back to the CustomerList.aspx Web page.

```
namespace CRUDSample.WebUI
{
  [ComVisible(false)]
  public class EditCustomer : System.Web.UI.Page
  {
    private void btnOk_Click(object sender, System.EventArgs e)
    {
      if (!Page.IsValid)
        return;

      // Create a "customer" business component
      CustomerBc customerBc = new CustomerBc();

      // If the ViewState contains a CustomerID value, we are in "edit mode"
      if (ViewState["CustomerId"] != null)
      {
        // Update the specified customer record in the data store
        customerBc.Update( (int)ViewState["CustomerId"],
                           txtCompany.Text, txtContact.Text,
                           txtAddress.Text, txtCity.Text, txtRegion.Text,
                           txtPostal.Text, txtCountry.Text, txtEmail.Text );
      }
      else
      {
        // Create a new customer record in the data store
        customerBc.Create (txtCompany.Text, txtContact.Text,
                          txtAddress.Text, txtCity.Text, txtRegion.Text,
                          txtPostal.Text, txtCountry.Text, txtEmail.Text );
      }

      // Redirect the user back the "customer list" Web page
      Response.Redirect("CustomerList.aspx");
    }
    // Plus other members...
  }
}
```

3. Define an event handler method for the **Cancel** button-click event to redirect the user back to the CustomerList.aspx Web page.

```
namespace CRUDSample.WebUI
{
  [ComVisible(false)]
  public class EditCustomer : System.Web.UI.Page
  {
    private void btnCancel_Click(object sender, System.EventArgs e)
    {
      Response.Redirect("CustomerList.aspx");
    }
```

```
        // Plus other members...
    }
}
```

The preceding code samples show how to edit an existing customer record in the data store, and how to create a new customer record in the data store.

Defining the OrderList.aspx Web Page

The OrderList.aspx Web page displays orders for a particular customer. This Web page is displayed when the user clicks the "View orders" hyperlink on the CustomerList.aspx Web page; the customer's ID is appended to the query string for OrderList.aspx to identify the current customer.

OrderList.aspx displays the order records in a **DataGrid** control, as shown in Figure B.9.

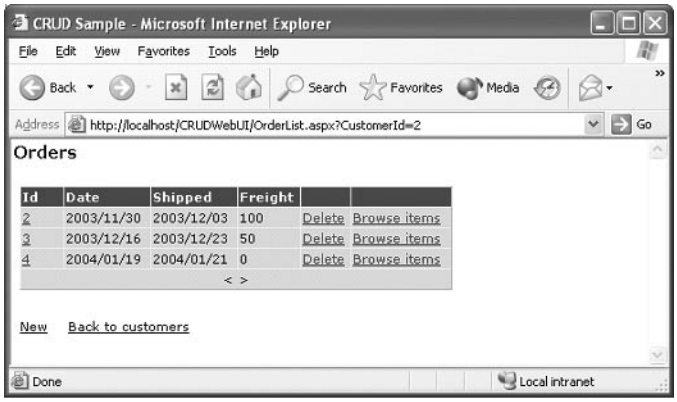


Figure B.9
Order list Web page

There are many similarities between the OrderList.aspx Web page and the CustomerList.aspx Web page introduced earlier:

- OrderList.aspx contains a **DataGrid** control that displays a page of orders, one order per row. The **DataGrid** supports paging, in case there are many orders to display.
- The first column in the **DataGrid** contains a hyperlink to a details Web page (EditOrder.aspx), where the user can display and edit full details for a particular order.
- Each row in the **DataGrid** has a "Delete" hyperlink to delete an existing order in the data store. Each row also has a "Browse items" hyperlink, which redirects the user to another Web page (OrderItemList.aspx) to display the order items in a particular order.
- The "New" hyperlink at the bottom of OrderList.aspx redirects the user to the details Web page (EditOrder.aspx), where the user can enter details for a new order.
- The "Back to customers" hyperlink at the bottom of OrderList.aspx redirects the user back to the CustomerList.aspx Web page, to redisplay the list of customers.

To design the visual interface for the OrderList.aspx Web page, follow the general guidelines presented earlier in this chapter for the CustomerList.aspx Web page. Note the following points:

- The "Id" column is a hyperlink column in the **DataGrid** property builder. The **URL field** is **OrderID**, and the **URL format string** is EditOrder.aspx?OrderId={0}.
- The "Date," "Shipped," and "Freight" columns are bound columns in the **DataGrid** property builder. These columns are bound to the **OrderDate**, **ShippedDate**, and **Freight** properties on the order record.
- The "Delete" column is a template column in the **DataGrid** property builder. This template column contains a **LinkButton** control, whose **CommandName** property is **DeleteCommand**.
- The "Browse items" column is a hyperlink column in the **DataGrid** property builder. The **URL field** is **OrderID**, and the **URL format string** is OrderItemList.aspx?OrderId={0}.
- The "New" hyperlink beneath the **DataGrid** control is a **HyperLink** control. The **NavigateUrl** property is EditOrder.aspx?CustomerId={0}, to create a new order for the current customer.
- The "Back to customers" hyperlink beneath the **DataGrid** control is a **HyperLink** control. The **NavigateUrl** property is CustomerList.aspx, to redirect the user back to the "customer list" page.

► To implement the functionality for the OrderList.aspx Web page

1. Add code to the **Page_Load** method, to bind the **DataGrid** control to the order records for the current customer. Retrieve the customer's ID from the query string, and store it in the **ViewState** so that it is available in subsequent postbacks. In the following code sample, the task of loading orders and binding them to the **DataGrid** control is performed in a helper method named **LoadOrders**.

```
using CRUDSample.BusinessEntity;
using CRUDSample.BusinessComponent;

namespace CRUDSample.WebUI
{
  [ComVisible(false)]
  public class OrderList : System.Web.UI.Page
  {
    private void Page_Load(object sender, System.EventArgs e)
    {
      if (!Page.IsPostBack)
      {
        // Add customerID to the ViewState, so it is available in postbacks
        ViewState.Add("CustomerId",
                      int.Parse(Request.QueryString["CustomerId"]));
```

```
        // Rebind the DataGrid control
        LoadOrders();

        // Append the customerID to the "New" hyperlink,
        // to identify the current customer in the hyperlink
        lnkNew.NavigateUrl =
          String.Format(System.Globalization.CultureInfo.CurrentUICulture,
                        lnkNew.NavigateUrl,
                        ViewState["CustomerId"]);
      }
    }

    private void LoadOrders()
    {
      // Create an "order" business component
      OrderBc orderBc = new OrderBc();

      // Get all orders for the current customer
      int customerId = (int)ViewState["CustomerId"];
      OrderDs order = orderBc.GetByCustomer(customerId);

      // Bind the DataGrid control to the data set
      dgOrders.DataSource = order.Orders;
      dgOrders.DataBind();
    }
    // Plus other members...
  }
}
```

2. Define an event handler for the "item created" event on the **DataGrid**. The event handler method adds client-side script to the **Delete** button for the new item. The client-side script displays an "Are you sure?" confirmation message when the user tries to delete the item to give the user an opportunity to cancel the deletion.

```
namespace CRUDSample.WebUI
{
  [ComVisible(false)]
  public class OrderList : System.Web.UI.Page
  {
    private void ordersGrid_ItemCreated(object sender,
                                        DataGridItemEventArgs e)
    {
      if (e.Item.ItemType == ListItemType.Item ||
          e.Item.ItemType == ListItemType.AlternatingItem)
      {
        // Add an "Are you sure?" confirmation dialog box to the Delete button
        LinkButton deleteButton =
                          e.Item.FindControl("lnkDelete") as LinkButton;

        if (deleteButton != null)
          deleteButton.Attributes.Add(
                           "onclick",
                           "return window.confirm('Are you sure ?');");
      }
    }
```

```
    // Plus other members...
  }
}
```

3. Define an event handler method for to delete the selected order record from the data store.

```
namespace CRUDSample.WebUI
{
  [ComVisible(false)]
  public class OrderList : System.Web.UI.Page
  {
    // Handle "item" events from the DataGrid control
    private void ordersGrid_ItemCommand(object source,
                                        DataGridCommandEventArgs e)
    {
      // Is it the "Delete" command?
      if (e.CommandName == "Delete")
      {
        // Get the OrderID of the selected order
        int orderId = (int)dgOrders.DataKeys[e.Item.ItemIndex];

        // Delete the selected order
        OrderBc orderBc = new OrderBc();
        orderBc.Delete(orderId);

        // Update the page index
        if (dgOrders.Items.Count == 1 && dgOrders.CurrentPageIndex > 0)
        {
          dgOrders.CurrentPageIndex -= 1;
        }

        // Rebind
        LoadOrders();
      }
    }
    // Plus other members...
  }
}
```

4. Define an event handler method for the **PageIndexChanged** event on the **DataGrid** control. Implement the event handler method as follows, to support paging behavior in the **DataGrid** control.

```
namespace CRUDSample.WebUI
{
  [ComVisible(false)]
  public class OrderList : System.Web.UI.Page
  {
    private void ordersGrid_PageIndexChanged(object source,
                                             DataGridPageChangedEventArgs e)
    {
```

```
            // Adjust the current page index
            dgOrders.CurrentPageIndex = e.NewPageIndex;

            // Rebind
            LoadOrders();
        }
        // Plus other members...
    }
}
```

The preceding code samples show how to display a collection of "order" business entities in a **DataGrid** control, and how to incorporate hyperlinks that enable the user to create new orders, edit existing orders, and delete existing orders.

Defining the EditOrder.aspx Web Page

The EditOrder.aspx Web page allows the user to edit details for an existing order record, and to enter details for a new order record. EditOrder.aspx is shown in Figure B.10 with some sample data entered by the user.

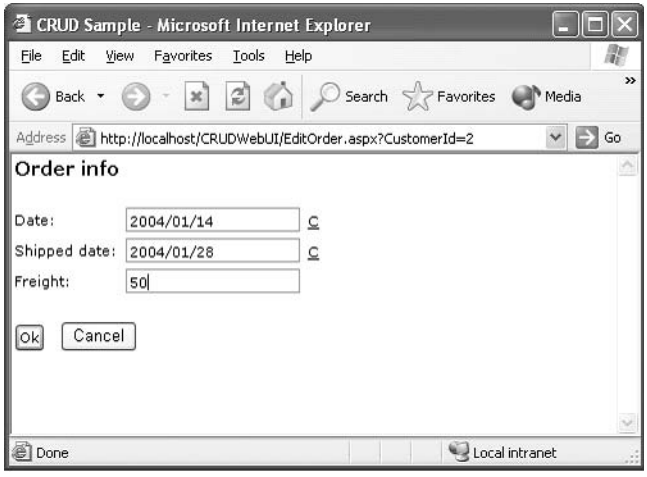


Figure B.10
Edit-order Web page

The EditOrder.aspx Web page is visually and functionally equivalent to the EditCustomer.aspx Web page, except that it works with "order" records instead of "customer" records. For a description of the mechanisms and techniques used to implement both of these Web pages, see "Defining the EditCustomer.aspx Web Page" earlier in this chapter.

The code-behind class for EditOrder.aspx is shown in the following code sample. This code follows a similar pattern to the code-behind class for EditCustomer.aspx.

```
using CRUDSample.BusinessComponent;
using CRUDSample.BusinessEntity;
```

```
namespace CRUDSample.WebUI
{
  [ComVisible(false)]
  public class EditOrder : System.Web.UI.Page
  {
    private void Page_Load(object sender, System.EventArgs e)
    {
      if (!Page.IsPostBack)
      {
        // Is the user editing an existing order record?
        if (Request.QueryString["OrderId"] != null)
        {
          // Add orderID to the ViewState, so that it is available in postbacks
          ViewState.Add("OrderId", int.Parse(Request.QueryString["OrderId"]));

          // Create an "order" business component
          OrderBc orderBc = new OrderBc();

          // Get the specified order's details from the data store
          OrderDs.Order order = orderBc.GetById((int)ViewState["OrderId"]);

          // Populate the form controls with order details
          txtDate.Text    = order.OrderDate.ToString("yyyy/MM/dd");
          txtShipped.Text = order.ShippedDate.ToString("yyyy/MM/dd");
          txtFreight.Text = order.Freight.ToString();

          // Also add the customerID to the ViewState
          ViewState.Add("CustomerId", order.CustomerID);
        }
        else
        {
          // This is a new order, so get the customer's ID from the Query String
          // and add it to the ViewState
          ViewState.Add("CustomerId",
                        int.Parse(Request.QueryString["CustomerId"]));
        }
      }
    }

    private void btnOk_Click(object sender, System.EventArgs e)
    {
      // Create an "order" business component
      OrderBc orderBc = new OrderBc();

      // If the ViewState contains an OrderID value, we are in "edit" mode
      if (ViewState["OrderId"] != null)
      {
        int orderId = (int)ViewState["OrderId"];

        // Update the specified order record in the data store
        orderBc.Update( orderId,
                       (int)ViewState["CustomerId"],
                        DateTime.Parse(txtDate.Text),
                        DateTime.Parse(txtShipped.Text),
                        decimal.Parse(txtFreight.Text) );
      }
```

```
      else
      {
        // Create a new order record in the data store
        orderBc.Create( (int)ViewState["CustomerId"],
                        DateTime.Parse(txtDate.Text),
                        DateTime.Parse(txtShipped.Text),
                        decimal.Parse(txtFreight.Text) );
      }

      // Redirect the user back to the "order list" Web page for this customer
      Response.Redirect("OrderList.aspx?CustomerId=" +
                        ViewState["CustomerId"].ToString());
    }

    private void btnCancel_Click(object sender, System.EventArgs e)
    {
      Response.Redirect("OrderList.aspx?CustomerId=" +
                        ViewState["CustomerId"].ToString());
    }
  }
}
```

The preceding code samples show how to edit an existing order record in the data store, and how to create a new order record in the data store.

Defining the OrderItemList.aspx Web Page

The OrderItemList.aspx Web page displays the order items that comprise a particular order. This Web page is displayed when the user clicks the "Browse items" hyperlink on the OrderList.aspx Web page; the order ID is appended to the query string for OrderItemList.aspx to identify the order.

OrderItemList.aspx displays the order item records in a **DataGrid** control, as shown in Figure B.11.

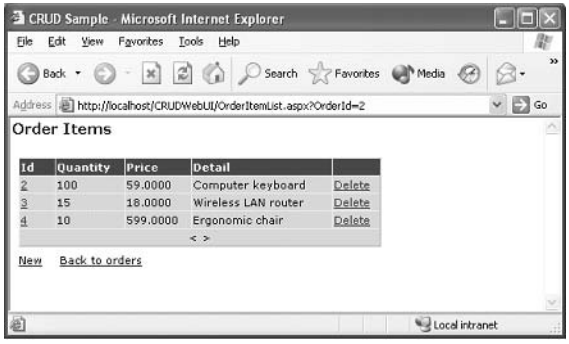


Figure B.11
Order-item list Web page

The OrderItemList.aspx Web page is visually and functionally equivalent to the OrderList.aspx Web page, except that it works with "order item" records instead of "order" records. For a description of the mechanisms and techniques used to implement both of these Web pages, see "Defining the OrderList.aspx Web Page" earlier in this chapter.

Defining the EditOrderItem.aspx Web Page

The EditOrderItem.aspx Web page allows the user to edit details for an existing order item record and to enter details for a new order item record. EditOrderItem.aspx is shown in Figure B.12, with some sample data entered by the user.

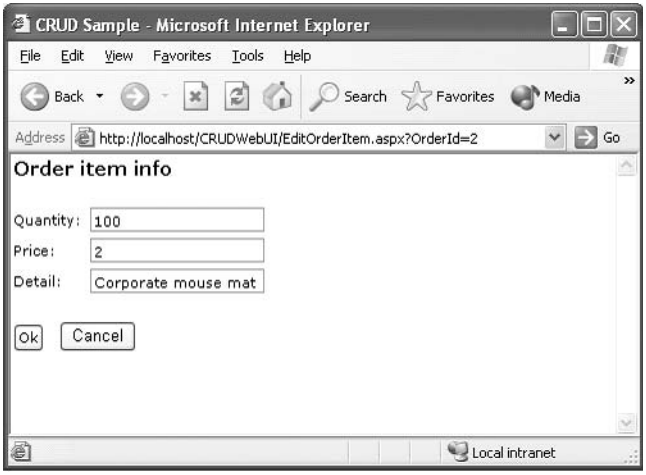


Figure B.12
Edit order-item Web page

The EditOrderItem.aspx Web page is visually and functionally equivalent to the EditOrder.aspx Web page, except that it works with "order item" records instead of "order" records. For a description of the mechanisms and techniques used to implement both of these Web pages, see "Defining the EditOrder.aspx Web Page" earlier in this chapter.

How To: Execute a Long-Running Task in a Web Application

This example shows how to create threads to perform long-running tasks in an ASP.NET Web application.

In this example, the user enters credit card details in a "payment" Web page (Payment.aspx). When the user submits the details, the payment page creates a worker thread to authorize the credit card details as a background task. In the meantime, the user is redirected to a "result" Web page (Result.aspx). The result page continually refreshes itself until the worker thread has completed the credit card authorization task.

There are four classes in this example:

- Payment–This is the code-behind class for the payment Web page.
- CCAuthorizationService–This is a service agent class that sends credit card transactions to a credit card authority.
- ThreadResults–This is a helper class that stores the credit card authorization results from the worker threads.
- Result–This is the code-behind class for the result Web page.

The following sections describe these classes.

Note: For simplicity in this example, the application does not validate the details entered by the user.

Defining the Payment Class

The **Payment** class is the code-behind class for the Payment.aspx Web page. This Web page asks the user to enter payment details, as shown in Figure B.13.

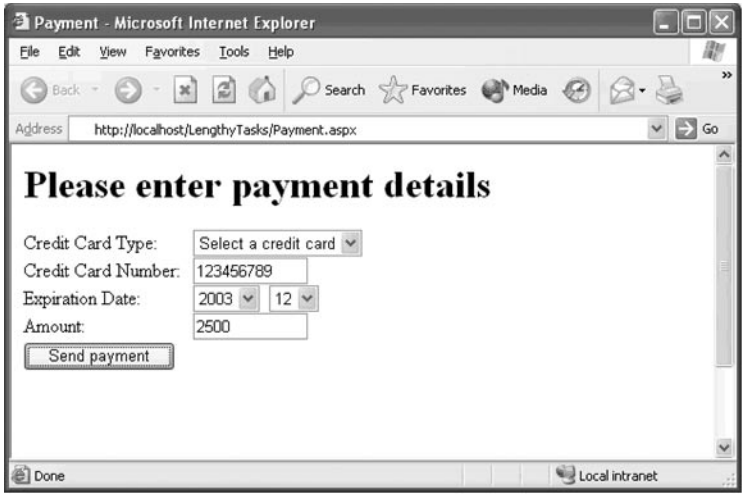


Figure B.13
Credit card payment page

The **Payment** class has three important members:

- **RequestId field**–This is a globally unique identifier (GUID) that identifies each credit card authorization task. Every time the user submits credit card details to the payment Web page, a new GUID will be generated.
- **Page_Load method**–This method is called when the payment Web page is first displayed, and on each subsequent postback. This method performs the following tasks:
  - When the payment Web page is first displayed, **Page_Load** initializes the controls on the page.
  - On subsequent postbacks, **Page_Load** creates a new request ID, and then calls the AuthorizePayment method in a worker thread. In the meantime, the user is redirected to the result Web page, Result.aspx. The request ID is appended to the query string for Result.aspx to identify the worker thread that the result Web page is waiting for.
- **AuthorizePayment method**–This method synchronously calls the **Authorize** method in the **CCAuthorizationService** class to authorize the credit card details. When the authorization is complete, the **Authorize** method returns an authorization ID. The authorization ID is added to the **ThreadResults** collection to indicate to the result Web page that the authorization is complete.

The following is the code for the **Payment** class.

```
using System;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Threading;

public class Payment : System.Web.UI.Page
{
  // GUID, holds a unique ID for this authorization request
  protected Guid RequestId;

  private void Page_Load(object sender, System.EventArgs e)
  {
    if (Page.IsPostBack)
    {
      // This is a postback, so authorize the credit card details...

      // Create a new request id
      RequestId = Guid.NewGuid();

      // Create and start a worker thread, to process the credit card details
      ThreadStart ts = new ThreadStart(AuthorizePayment);
      Thread workerThread = new Thread(ts);
      workerThread.Start();

      // Redirect to the "result page" (append the requestId to the query string)
       Response.Redirect("Result.aspx?RequestId=" + RequestId.ToString());
    }
```

```
    else
    {
      // This is not a postback, so initialize the Web UI controls
      int currentYear = DateTime.Now.Year;

      // Populate the expiration date dropdown lists
      for (int index = currentYear; index < currentYear + 6; index++)
        cmbExpDateYear.Items.Add(new ListItem(index.ToString(),
                                              index.ToString()));

      for (int index = 1; index < 13; index++)
        cmbExpDateMonth.Items.Add(new ListItem(index.ToString(),
                                               index.ToString()));

      for (int index = 1; index < 32; index++)
        cmbExpDateDay.Items.Add(new ListItem(index.ToString(),
                                             index.ToString()));
    }
  }

  // Send an authorization request to the credit card authority.
  // This method is executed by the worker thread.
  private void AuthorizePayment()
  {
    // Send the request
    int authorizationId = CCAuthorizationService.Authorize(
                                    cmbCCType.Items[cmbCCType.SelectedIndex].Value,
                                    txtCCNumber.Text,
                                    DateTime.MaxValue,
                                    Double.Parse(txtAmount.Text));

    // Add the authorization id to the result collection
    ThreadResults.Add(RequestId, authorizationId);
  }
  // Plus other members...
}
```

Defining the CCAuthorizationService Class

The **CCAuthorizationService** class interacts with a back-end credit card agency to authorize the user's credit card details. The exact mechanism for credit card authorization is irrelevant; the important point is that the authorization task is likely to take several seconds (or minutes) to complete.

The following code shows a mock-up implementation for the **CCAuthorizationService** class. The **Authorize** method waits for 5 seconds, and then returns a random authorization ID.

```
using System;
using System.Threading;
```

```
public sealed class CCAuthorizationService
{
  public static int Authorize(string ccType,
                              string ccNumber,
                              DateTime expDate,
                              double amount)
  {
    // Wait for the credit card authority...
    Thread.Sleep(new TimeSpan(0, 0, 0, 7, 0));

    // Return the authorization id
    return new Random(0).Next(100);
  }
}
```

Defining the ThreadResults Class

The **ThreadResults** class holds a collection of results from worker threads, so that the result page can access the result of each credit card authorization task when it is complete.

ThreadResults holds the results in a hash table. The keys in the hash table are the request IDs, and the values in the hash table are the corresponding authorization IDs. The **ThreadResults** class provides methods to insert, query, and remove items in the hash table.

The following is the code for the **ThreadResults** class.

```
using System;
using System.Collections;

public sealed class ThreadResults
{
  private static Hashtable results = new Hashtable();

  public static object Get(Guid itemId)
  {
    return results[itemId];
  }

  public static void Add(Guid itemId, object result)
  {
    results[itemId] = result;
  }

  public static void Remove(Guid itemId)
  {
    results.Remove(itemId);
  }

  public static bool Contains(Guid itemId)
  {
    return results.Contains(itemId);
  }
}
```

Defining the Result Class

The **Result** class is the code-behind class for the Result.aspx Web page. This Web page displays a "wait" message while the credit card authorization is taking place as shown in Figure B.14.

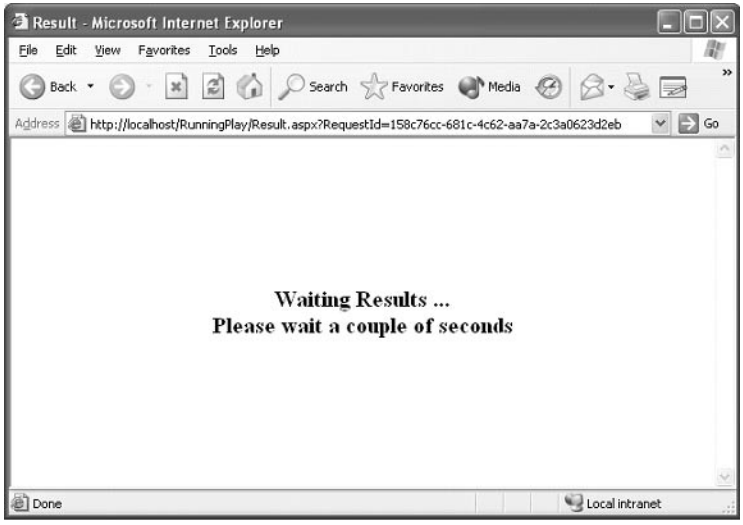


Figure B.14
The result page displays a "wait"' message while the long-running task is taking place

When the credit card authorization is complete, the Result.aspx Web page displays the authorization ID as shown in Figure B.15.

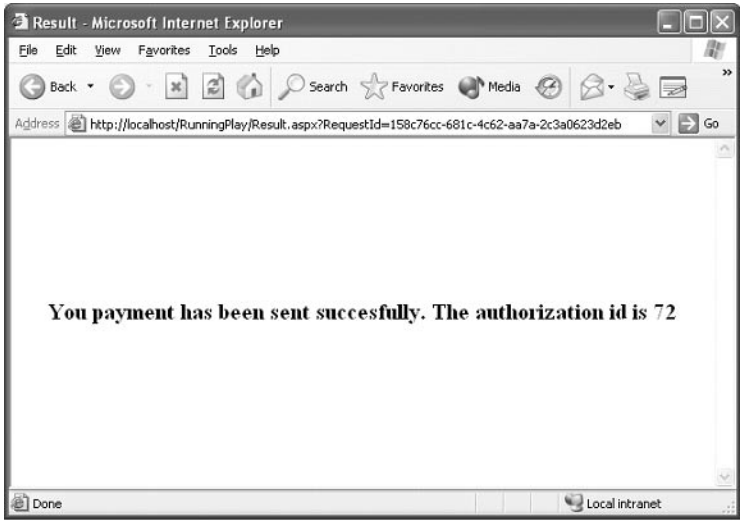


Figure B.15
The result page displays the authorization ID when the long-running task is complete

The **Page_Load** method in the **Result** class retrieves the **RequestId** parameter from the HTTP query string and tests whether the worker thread has completed authorizing this request:

- If the authorization request is not yet complete, a **Refresh** header is added to the HTTP response. This causes the Web page to reload itself automatically in 2 seconds.
- If the authorization request is complete, the authorization ID is retrieved from the **ThreadResults** class and is displayed on the Web page.

The following is the code for the **Result** class.

```
using System;

public class Result : System.Web.UI.Page
{
  protected System.Web.UI.WebControls.Label lblMessage;

  private void Page_Load(object sender, System.EventArgs e)
  {
    // Get the request id
    Guid requestId = new Guid(Page.Request.QueryString["RequestId"].ToString());

    // Check the thread result collection
    if(ThreadResults.Contains(requestId))
    {
      // The worker thread has finished

      // Get the authorization id from the thread result collection
      int authorizationId = (int)ThreadResults.Get(requestId);

      // Remove the result from the collection
      ThreadResults.Remove(requestId);

      // Show the result
      lblMessage.Text = "You payment has been sent succesfully. " +
                        "The authorization id is " + authorizationId.ToString();
    }
    else
    {
      // The worker thread has not yet finished authorizing the payment details.
      // Add a refresh header, to refresh the page in 2 seconds.
      Response.AddHeader("Refresh", "2");
    }
  }
  // Plus other members...
}
```

How To: Use the Trusted Subsystem Model

The trusted subsystem model enables middle-tier services to use a fixed identity to access downstream services and resources. The security context of the original caller does not flow through the service at the operating system level, although the application may choose to flow the original caller's identity at the application level. It may need to do so to support back-end auditing requirements or to support per-user data access and authorization.

You can define multiple trusted identities if necessary, based on the role membership of the caller. For example, you might have two groups of users, one who can perform read/write operations on a database, and the other who can perform read-only operations. In such a scenario, you can map each role to a different trusted identity in the underlying system.

The main advantages of using he trusted subsystem model are scalability, simple back-end ACL management, and protection of data from direct access by users.

To make use of the trusted subsystem model, you need to make the following configuration changes in IIS and ASP.NET on the Web server:

- **Configure IIS authentication**–Configure IIS so that it authenticates the original user. You can use Windows authentication (Basic, Digest, Integrated Windows, or Certificate authentication), Forms authentication, or Passport authentication.
- **Configure ASP.NET authentication**–Edit Web.config for the ASP.NET Web application so that it specifies which authentication mechanism to use for this application. Add an **<authentication>** element that specifies the authentication mode ("**Windows**," "**Forms**," or "**Passport**").

```
<authentication mode="Windows"/>  -or-
<authentication mode="Forms"/>    -or-
<authentication mode="Passport"/>
```

  You must also ensure impersonation is disabled. Impersonation is a mechanism whereby the user's original security context is passed on to downstream applications so that they can perform their own authentication and authorization tests on the user's real credentials. Impersonation is disabled by default in ASP.NET, unless you have an **<identity impersonate="true"/>** element. To explicitly disable impersonation, add the following element to Web.config.

```
<identity impersonate="false"/>
```

How To: Use Impersonation/Delegation with Kerberos Authentication

In the impersonation/delegation model, the ASP.NET Web application authenticates and authorizes the user at the first point of contact as before. These credentials are flowed to downstream applications, to enable the downstream applications to perform their own authentication and authorization tests using the real security credentials of the original user.

To enable impersonation/delegation with Kerberos authentication, you must make the following configuration changes in IIS and ASP.NET on the Web server:

- **Configure IIS authentication**–Configure IIS for the Web application's virtual root, so that it uses Integrated Windows authentication (that is, Kerberos authentication).
- **Configure ASP.NET authentication**–Edit Web.config for the ASP.NET Web application. You must set Windows authentication as the authentication mechanism and enable impersonation.

```
<authentication mode="Windows"/>
<identity impersonate="true"/>
```

Note: Impersonation/delegation with Kerberos authentication and delegation is feasible only if all computers are running Windows 2000 or later. Furthermore, each user account that needs to be impersonated must be stored in Active Directory and must not be configured as "Sensitive and cannot be delegated."

How To: Use Impersonation/Delegation with Basic or Forms Authentication

In the impersonation/delegation model, the ASP.NET Web application authenticates and authorizes the user at the first point of contact as before. These credentials are flowed to downstream applications, to enable the downstream applications to perform their own authentication and authorization tests using the real security credentials of the original user.

To enable impersonation/delegation with Basic authentication or Forms authentication, you must make the following configuration changes in IIS and ASP.NET on the Web server:

- **Configure IIS authentication**–Configure IIS for the Web application's virtual root so that it uses Basic authentication or Forms authentication.
- **Configure ASP.NET authentication**–Edit Web.config for the ASP.NET Web application so that it uses either Basic authentication or Forms authentication.

```
<authentication mode="Basic"/>   -or-
<authentication mode="Forms"/>
```

You must also enable impersonation.

```
<identity impersonate="true"/>
```

You must also write code in your ASP.NET Web application to obtain the user's name and password:

- If you use Basic authentication, get the user's name and password from HTTP server variables.

```
string username = Request.ServerVariables["AUTH_USER"];
string password = Request.ServerVariables["AUTH_PASSWORD"];
```

- If you use Forms authentication, get the user's name and password from fields in the HTML logon form. For example, if the logon form has text fields named **txtUsername** and **txtPassword**, you can get the user name and password as follows.

```
string username = txtUsername.Text;
string password = txtPassword.Text;
```

How To: Localize Windows Forms

In Windows Forms-based applications, you can use the Visual Studio .NET Forms Designer to create localized versions of your form.

► **To create localized versions of your form**

1. Design your form in the usual manner, laying out all the necessary controls. Any text labels you assign at this stage will be used by your application's default culture.
2. After you design the form, set the form's **Localizable** property to **True**.
3. For each alternative culture that your application will support, create a localized version of the form as follows:
   a. Set the form's **Language** property to the culture you want to work with.
   b. Modify each of the controls on your form to comply with the culture you are working with. For example, change labels on the form to display text in the appropriate natural language for the culture.

As you add cultures to your form, new .resx files are created for the form. To view the .resx files, make sure you have selected the **Show All Files** option in Solution Explorer. The .resx files appear beneath your form file; for example, if you add support for French (fr-FR) to a form named MyForm, Visual Studio. NET will create MyForm.fr-FR.resx.

When you run the application, the run time uses the most appropriate resource file to render the user interface based on the **Thread.CurrentThread.CurrentUICulture** setting. In general, you want to specify a culture so that every part of the application's user interface is appropriate to that culture; therefore, you must set the culture before the **InitializeComponent** method is called. The following code sample sets the current culture and current UI culture in the constructor of the **MyForm** class.

```
using System.Windows.Forms;
using System.Threading;
using System.Globalization;

public class MyForm : System.Windows.Forms.Form
{
  // Constructor
  public MyForm()
  {
    // Set the culture and UI culture before InitializeComponent is called
    Thread.CurrentThread.CurrentCulture   = new CultureInfo("fr-FR");
    Thread.CurrentThread.CurrentUICulture = new CultureInfo("fr-FR");

    // Initialize components, using the specified culture and UI culture
    InitializeComponent();
```

```
    // Perform any additional initialization, as appropriate
    // ...
  }
  // Plus other members...
}
```

For additional information about how localize Windows Forms, see "Let Your Apps Span the Globe with Windows Forms and Visual Studio .NET" on MSDN (*http://msdn.microsoft.com/msdnmag/issues/02/06/internat/toc.asp*).

How To: Define a Catch-All Exception Handler in Windows Forms-based Applications

In Windows Forms-based applications, you can define a catch-all exception handler method to catch all un-trapped thread exceptions. In the exception handler method, you can take appropriate recovery actions—usually to display a user-friendly error message.

► To define a catch-all exception handler

1. Implement an event handler method for the **ThreadException** event of the **System.Windows.Forms.Application** class.
2. Add the event handler method to the **Application.ThreadException** event. The best place to do this is in the **Main** method of your application.

The following code sample implements a catch-all event handler method that publishes exceptions using the Microsoft Exception Management Application Block.

```
using System.Windows.Forms;
using System.Threading;
using Microsoft.ApplicationBlocks.ExceptionManagement;

public class MyForm : System.Windows.Forms.Form
{
  // Catch-all event handler method
  protected static void CatchAllExceptions(object Sender,
                                           ThreadExceptionEventArgs e)
  {
    // Publish the exception
    ExceptionManager.Publish(ex);
  }

  // Main method in the application
  public static void Main()
  {
    // Create a new ThreadExceptionEventHandler delegate instance
    // and add it to the Application.ThreadException event
    Application.ThreadException +=
        new ThreadExceptionEventHandler(CatchAllExceptions);

    // Run the Windows Forms application
    Application.Run(new MyForm());
  }
}
```

For more information about the Microsoft Exception Management Application Block, see MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/emab-rm.asp*).

About Microsoft *patterns & practices*

Microsoft *patterns & practices* guides contain specific recommendations illustrating how to design, build, deploy, and operate architecturally sound solutions to challenging business and technical scenarios. They offer deep technical guidance based on real-world experience that goes far beyond white papers to help enterprise IT professionals, information workers, and developers quickly deliver sound solutions.

IT Professionals, information workers, and developers can choose from four types of *patterns & practices*:

- **Patterns**—Patterns are a consistent way of documenting solutions to commonly occurring problems. Patterns are available that address specific architecture, design, and implementation problems. Each pattern also has an associated GotDotNet Community.
- **Reference Architectures**—Reference Architectures are IT system-level architectures that address the business requirements, LifeCycle requirements, and technical constraints for commonly occurring scenarios. Reference Architectures focus on planning the architecture of IT systems.
- **Reference Building Blocks and IT Services**—References Building Blocks and IT Services are re-usable sub-system designs that address common technical challenges across a wide range of scenarios. Many include tested reference implementations to accelerate development. Reference Building Blocks and IT Services focus on the design and implementation of sub-systems.
- **Lifecycle Practices**—Lifecycle Practices provide guidance for tasks outside the scope of architecture and design such as deployment and operations in a production environment.

Patterns & practices guides are reviewed and approved by Microsoft engineering teams, consultants, Product Support Services, and by partners and customers. *Patterns & practices* guides are:

- **Proven**—They are based on field experience.
- **Authoritative**—They offer the best advice available.
- **Accurate**—They are technically validated and tested.
- **Actionable**—They provide the steps to success.
- **Relevant**—They address real-world problems based on customer scenarios.

Patterns & practices guides are designed to help IT professionals, information workers, and developers:

Reduce project cost

- Exploit the Microsoft engineering efforts to save time and money on your projects.
- Follow the Microsoft recommendations to lower your project risk and achieve predictable outcomes.

Increase confidence in solutions

- Build your solutions on proven Microsoft recommendations so you can have total confidence in your results.
- Rely on thoroughly tested and supported guidance, but production quality recommendations and code, not just samples.

Deliver strategic IT advantage

- Solve your problems today and take advantage of future Microsoft technologies with practical advice.

patterns & practices: Current Titles

October 2003

Title	Link to Online Version	Book
Patterns		
Enterprise Solution Patterns using Microsoft .NET	*http://msdn.microsoft.com/practices/type/Patterns/Enterprise/default.asp*	
Microsoft Data Patterns	*http://msdn.microsoft.com/practices/type/Patterns/Data/default.asp*	
Reference Architectures		
Application Architecture for .NET: Designing Applications and Services	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/distapp.asp*	
Enterprise Notification Reference Architecture for Exchange 2000 Server	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnentdevgen/html/enraelp.asp*	
Improving Web Application Security: Threats and Countermeasures	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/ThreatCounter.asp*	
Microsoft Accelerator for Six Sigma	*http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/mso/sixsigma/default.asp*	
Microsoft Active Directory Branch Office Guide: Volume 1: Planning	*http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/ad/windows2000/deploy/adguide/default.asp*	
Microsoft Active Directory Branch Office Series Volume 2: Deployment and Operations	*http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/ad/windows2000/deploy/adguide/default.asp*	
Microsoft Content Integration Pack for Content Management Server 2001 and SharePoint Portal Server 2001	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncip/html/cip.asp*	
Microsoft Exchange 2000 Server Hosting Series Volume 1: Planning	Online Version not available	
Microsoft Exchange 2000 Server Hosting Series Volume 2: Deployment	Online Version not available	

Title	Link to Online Version	Book
Microsoft Exchange 2000 Server Upgrade Series Volume 1: Planning	*http://www.microsoft.com/technet/treeview /default.asp?url=/technet/itsolutions/guide /default.asp*	
Microsoft Exchange 2000 Server Upgrade Series Volume 2: Deployment	http://www.microsoft.com/technet/treeview /default.asp?url=/technet/itsolutions/guide /default.asp	
Microsoft Solution for Intranets	*http://www.microsoft.com/technet/treeview /default.asp?url=/technet/itsolutions/mso /msi/Default.asp*	
Microsoft Solution for Securing Wireless LANs	*http://www.microsoft.com/downloads /details.aspx?FamilyId=CDB639B3-010B-47E7-B23 4-A27CDA291DAD&displaylang=en*	
Microsoft Systems Architecture— Enterprise Data Center	*http://www.microsoft.com/technet/treeview /default.asp?url=/technet/itsolutions/edc /Default.asp*	
Microsoft Systems Architecture— Internet Data Center	*http://www.microsoft.com/technet/treeview/ default.asp?url=/technet/itsolutions/idc/default.asp*	
The Enterprise Project Management Solution	*http://www.microsoft.com/technet/treeview /default.asp?url=/technet/itsolutions/mso/epm /default.asp*	
UNIX Application Migration Guide	*http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnucmg/html/ucmglp.asp*	
Reference Building Blocks and IT Services		
.NET Data Access Architecture Guide	*http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnbda/html/daag.asp*	
Application Updater Application Block	*http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnbda/html/updater.asp*	
Asynchronous Invocation Application Block	*http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnpag/html/paiblock.asp*	
Authentication in ASP.NET: .NET Security Guidance	*http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnbda/html/authaspdotnet.asp*	
Building Interoperable Web Services: WS-I Basic Profile 1.0	*http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnsvcinter/html/wsi-bp_msdn_ landingpage.asp*	
Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication	*http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnnetsec/html/secnetlpMSDN.asp*	

Title	Link to Online Version	Book
Caching Application Block	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/Cachingblock.asp*	
Caching Architecture Guide for .Net Framework Applications	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/CachingArch.asp?frame=true*	
Configuration Management Application Block	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/cmab.asp*	
Data Access Application Block for .NET	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daab-rm.asp*	
Designing Application-Managed Authorization	*http://msdn.microsoft.com/library/?url=/library/en-us/dnbda/html/damaz.asp*	
Designing Data Tier Components and Passing Data Through Tiers	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/BOAGag.asp*	
Exception Management Application Block for .NET	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/emab-rm.asp*	
Exception Management Architecture Guide	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/exceptdotnet.asp*	
Microsoft .NET/COM Migration and Interoperability	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/cominterop.asp*	
Microsoft Windows Server 2003 Security Guide	*http://www.microsoft.com/downloads/details.aspx?FamilyId=8A2643C1-0685-4D89-B655-521EA6C7B4DB&displaylang=en*	
Monitoring in .NET Distributed Application Design	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/monitordotnet.asp*	
New Application Installation using Systems Management Server	*http://www.microsoft.com/business/reducecosts/efficiency/manageability/application.mspx*	
Patch Management using Microsoft Systems Management Server - Operations Guide	*http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/msm/swdist/pmsms/pmsmsog.asp*	
Patch Management Using Microsoft Software Update Services - Operations Guide	*http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/msm/swdist/pmsus/pmsusog.asp*	
Service Aggregation Application Block	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/serviceagg.asp*	
Service Monitoring and Control using Microsoft Operations Manager	*http://www.microsoft.com/business/reducecosts/efficiency/manageability/monitoring.mspx*	

Title	Link to Online Version	Book
User Interface Process Application Block	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/uip.asp*	
Web Service Façade for Legacy Applications	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/wsfacadelegacyapp.asp*	
Lifecycle Practices		
Backup and Restore for Internet Data Center	*http://www.microsoft.com/technet/treeview/default.asp?url=/technet/ittasks/maintain/backuprest/Default.asp*	
Deploying .NET Applications: Lifecycle Guide	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/DALGRoadmap.asp*	
Microsoft Exchange 2000 Server Operations Guide	*http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/exchange/exchange2000/maintain/operate/opsguide/default.asp*	
Microsoft SQL Server 2000 High Availability Series: Volume 1: Planning	*http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/sql/deploy/confeat/sqlha/SQLHALP.asp*	
Microsoft SQL Server 2000 High Availability Series: Volume 2: Deployment	*http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/sql/deploy/confeat/sqlha/SQLHALP.asp*	
Microsoft SQL Server 2000 Operations Guide	*http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/sql/maintain/operate/opsguide/default.asp*	
Operating .NET-Based Applications	*http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/net/maintain/opnetapp/default.asp*	
Production Debugging for .NET-Connected Applications	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/DBGrm.asp*	
Security Operations for Microsoft Windows 2000 Server	*http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/prodtech/win2000/secwin2k/default.asp*	
Security Operations Guide for Exchange 2000 Server	*http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/prodtech/mailexch/opsguide/default.asp*	
Team Development with Visual Studio .NET and Visual SourceSafe	*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/tdlg_rm.asp*	

 This title is available as a Book