1

Creating Your Own Controls

ASP.NET comes with its own set of server-side controls — so why create your own? And why would you need three different kinds of controls: custom controls, Web Parts, and user controls?

By creating your own controls, you can build powerful, reusable visual components for your Web application's user interface, including components that allow your users to customize and personalize your application. This chapter introduces you to the two primary types of controls (custom controls and Web Parts) along with user controls. You'll also see how creating your own controls can simultaneously improve the quality of your Web applications, make you more productive, and improve your user interfaces.

The Three Kinds of Controls

Why three different kinds of controls? Part of the reason is historical: Custom controls and user controls were introduced in the first version of ASP.NET, while Web Parts are new to ASP.NET 2.0 and add functionality that wasn't available in user controls and custom controls. So, from one perspective, Web Parts are different from user controls and custom controls because they are "newer" — not a very important difference. As you'll see, Web Parts are really just an extension of custom controls, but that new functionality provides developers with the opportunity to deliver something new: the ability for users to customize Web pages. But even that description doesn't really help distinguish among the three types of controls: while Web Parts are a special class of control, you can use both user controls and custom controls as Web parts (although they won't have all the customization facilities of a full-fledged Web Part).

Web Parts, custom controls, and user controls all allow you to create reusable components that can be used, in turn, to create Web pages in ASP.NET. Web Parts, custom controls and user controls in ASP.NET 2.0 look very much alike when you are using them to build Web pages. All can be used in Design view, for instance — you can drag them onto a page, resize them, and set their properties in the Property window. The major difference is that you drag custom controls and Web Parts from the toolbox in Visual Studio .NET but you drag user controls from Solution Explorer. However,

because both user controls and custom controls can be used as Web Parts, you can drag customization components from both the Toolbox and Solution Explorer (full-fledged Web Parts appear in the Toolbox). Whether you are building Web Parts, custom controls, or user controls, you can add your own properties, methods, and events to them.

User Controls

While the focus of this book is on custom controls and Web Parts, user controls shouldn't be ignored. For developers, the major difference between user controls and custom controls is in ease of development—a powerful incentive to use user controls. User controls provide an easy way to create reusable controls: If you know how to create a Web page in ASP.NET, then you know how to create a user control. As an example, you can add controls to your user control the same way that you add controls to a Web page: by dragging and dropping the controls onto a design surface. Figure 1-1 shows a user control in Design view in Visual Studio .NET.

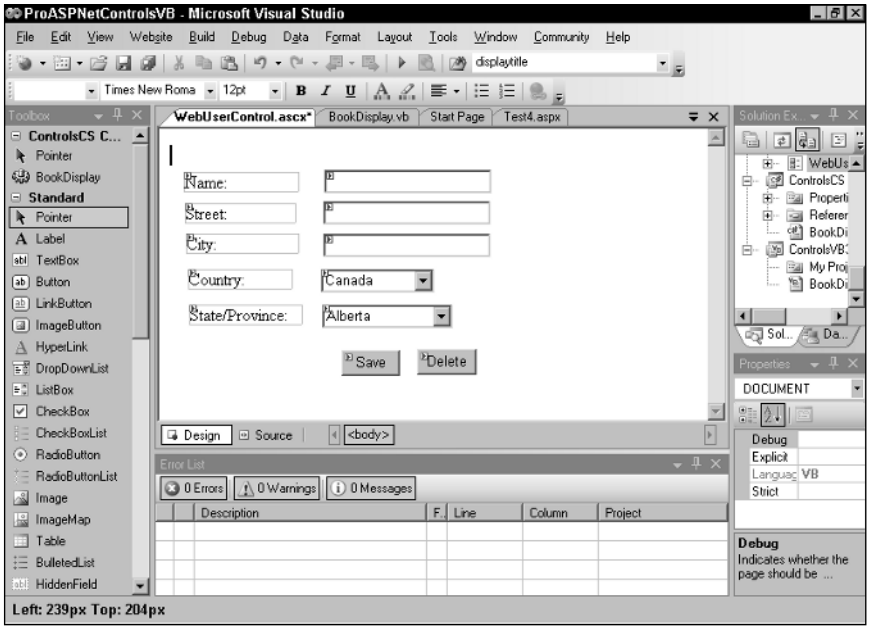


Figure 1-1

Of course, nothing comes for free: In order to gain this ease of development, user controls have several limitations. The first major limitation in the functionality of user controls is that they cannot inherit from other ASP.NET controls, while a custom control can inherit from other controls. The capability to inherit from other controls enables you, for instance, to create a custom control that inherits from the ASP.NET ListBox control and extends that control.

However, this limitation often just means thinking about the problem differently: If you want to create a user control that functions like a ListBox, you could just drop a list box on your user control and then add any new methods, properties, or events that you want (with a custom control, you would have to do

all of your development without the benefit of a drag-and-drop designer). As you'll see in this book, all the features that you can take advantage of in a custom control are available to you in a user control—it's just that the user control's drag-and-drop designer means that you don't need them.

The second major limitation of user controls is in their reusability: user controls can't be shared among projects or Web sites. A user control can be used only on Web pages in the project that the user control is part of. There is no way around this limitation.

Custom Controls

Custom controls are a more code-intensive way to create reusable components for Web applications. For instance, to add new controls to your custom controls, you must write code that will create the controls and add them to the Controls collection of your custom control—there is no drag-and-drop facility as in a user control. In return for taking longer to create, custom controls offer you more power.

Custom controls are more flexible than user controls. You can create a custom control that inherits from another server-side control and then extend that control. You could, for instance, create a custom control based on another custom control—even one as complex as the TreeView control—and then add new methods or properties to create your own, even more powerful control.

Custom controls are more reusable than user controls. You can share a custom control among projects. Typically, you'll create your custom control in a Web Custom Control library that is compiled separately from your Web application. As a result, you can add that library to any project in order to use your custom control in that project.

Web Parts

It's not really correct to compare Web Parts with user controls and custom controls. User controls and custom controls can be used as Web Parts, although they will lack all of the features of a full-fledged Web Part. But, if you want to take full advantage of the Web Part feature set, then you must build your control as a Web Part right from the beginning. As you'll see, Web Parts are an extension of custom controls—think of full-fledged Web Parts as custom controls with superpowers.

Web Parts actually first appeared not in ASP.NET but in Windows SharePoint Services (in 2003, when SharePoint was re-architected to run on top of ASP.NET). SharePoint is Microsoft's Web-based tool for creating document-based solutions that can be customized by the user. As part of visiting a SharePoint site, users can build pages in SharePoint by adding Web Parts to a SharePoint page or modifying the Web Parts already on the page. With ASP.NET 2.0, a version of Web Parts was added to ASP.NET.

For most developers, the statement that "users can build pages in SharePoint" seems counterintuitive. The usual division of labor is to have developers build Web pages and users . . . well, users just use the pages. SharePoint, however, was designed to empower users, to let users build the pages they needed without having to call on the IT staff. In ASP.NET, Web Parts can be used to fulfill the same function: to let users build the pages they need from the inventory of Web Parts available to a page. Because of this ability, Web Part developers have a new and more interesting job to do. Web Part developers don't just build applications; they build components that enable users to build applications.

This description of how a Web Part is used on a SharePoint site omits an important step. After a Web Part is created it is added to one of several Web Part galleries available to the SharePoint site. Once a Web Part is available to a site, developers then add the Web Part to a page. Users can add Web Parts to a SharePoint page only if the Web Part is in one of the galleries for the site. Galleries aren't part of the ASP.NET 2.0 implementation of Web Parts.

While Web Parts have become part of the toolkit for all ASP.NET developers, currently Web Parts developed in ASP.NET 2.0 can't be used in SharePoint (and Web Parts built for SharePoint can't be used outside of SharePoint). However, Microsoft has committed to providing an upgrade to SharePoint that will allow ASP.NET 2.0 Web Parts to be used in SharePoint in the near future.

The Benefits of Reusable Controls

By creating your own controls, you can build a toolkit of controls to draw on when building a Web application. Think of these controls as reusable *visual* components. A control can range from something as simple as displaying a title to being a complete business application in itself.

Much of the talk about the benefits of creating objects and components seems to revolve around abstract features (encapsulation, polymorphism, and so on). For a developer working in the real world, creating components really provides three practical benefits in terms of reusability:

- **Productivity:** By creating reusable components, you don't have to re-invent the wheel when implementing similar functionality in different applications (or parts of the same application).
- **Standardization:** By using the same components to perform operations that are common to different pages, you are guaranteed that the functionality is implemented in a common way.
- **Simplification:** By dividing functionality between specialized components and other parts of the application (such as workflow management, business logic, data access), the complexity in any one part is reduced.

Web Parts, custom controls, and user controls provide all three of these benefits. Web Parts add features that custom controls and user controls do not. These features include:

- **Integration:** Web Parts on the same page can find one another and exchange information.
- **Property settings that can be changed by the user:** At run time, users can change property settings for a Web Part to further customize the application for themselves or others.
- **Page design:** Web Parts can be added or removed from the page, relocated to other parts of the page, or just minimized to "de-clutter" the page.

The benefits of reusability with object-oriented development are so well known that they form part of the conventional wisdom of modern application developers. But Web Parts also provide another benefit: customization. The benefits of customization are not as commonly known, however, so the next section describes why customization matters to you.

Beyond Reusability with Web Parts

Through customization, Web Parts give you the opportunity to gain a new and more challenging class of benefits: you can create Web Parts that end users can add to their pages in order to create their own solutions. Think of Web Parts as reusable visual *tools* (rather than just visual components): Web Parts are tools that users employ to meet their goals. When you create a user control or a custom control you design it to help you and other developers solve problems in creating applications. With Web Parts you create controls designed to let end users solve problems, often in situations that you may not have even thought of.

This opportunity is challenging because it's difficult to predict all the ways that users will find to employ a genuinely useful Web Part. If building a reusable visual tool isn't enough of a challenge, you can also offer users the capability to customize your Web Part to meet their needs. In addition to adding your Web Part to a page, users can also modify the way that your Web Part behaves.

Developing with Web Parts isn't about what you can do for your users. Web Parts are about what you can allow your users to do for themselves—how you can empower your users. You can give users the ability to add Web Parts to pages, remove Web Parts from pages, move Web Parts from one location to another on the page, customize the Web Parts on a page, and join Web Parts together so that they can pass information between themselves. Users can perform all of these activities through your site's user interface—other than a browser, no additional tools are required. So, in addition to building applications, you can provide the tools that allow users to build their own solutions.

Allowing Customization with Web Parts

Initially it may seem that incorporating Web Part *tools* into your application isn't all that different from incorporating Web Part *components* into your page. When you decide to use a control as a Web Part, it may appear that all you've done is delay when the control will be incorporated into a page or when the control's properties will be set. For instance, instead of adding your control to a page at design time, you've delayed adding the control to the point when the page is being used at run time. You may be thinking that all that's required is some additional planning to ensure that your page will work correctly no matter when controls are added. You may even be thinking that all you'll really need to do is add some more error handling to your code in order to deal with conditions that may not have been considered at development time. If you do, then you're missing the point of Web Parts.

Here's the point: Incorporating Web Parts into your application development makes it possible to create a new kind of Web application. SharePoint, where Web Parts first appeared, was designed to empower users to build solutions that met their needs. With Web Parts now part of the ASP.NET developer's toolkit, you (and every other ASP.NET developer) also have the ability to empower your users. Instead of just designing an application to perform some task, you can consider the entire range of activities that your users need to perform and build tools that support those activities in any combination. Instead of delivering a rigid system that implements your design, you can build a discrete set of tools that allows users to meet their own needs. As your users' needs change and grow over time, they can call on your tools to deal with those changes.

Piggy Banks and Lego Kits

What's the difference between building a standard application and building a customizable solution? Let's say that you have some spare change rattling around in a drawer (or, worse, in several drawers). The obvious solution is to go out and buy a piggy bank. The piggy bank is a great tool for collecting and holding coins — but that's all it can do. Most piggy banks don't even do a very good job of holding paper money, let alone all the other things that you might want to save.

So instead of buying a piggy bank, you could buy a Lego kit. With a Lego kit you can build your own piggy bank — perhaps even figure out how to build a bank that works well with all the different things that you want to save. You can also build a tower, a plane, a car, and anything else that you can think of.

In addition, different Lego kits have different building blocks. The greater the variety of Lego building blocks available to you, the greater the variety of things that you can build and the easier it is to build specific items (a car is considerably easier to build if your kit includes wheels and axles, for instance). Within any application domain, domain-specific tools are more useful than general-purpose tools.

With Web Parts, your job is to provide your user with a set of building blocks that your users can build solutions with. And, besides, who doesn't enjoy playing with Legos?

Undoubtedly, the user community for your application will contain a variety of users, many with specialized needs. It's entirely possible that every user has a unique set of needs. As a result, different users may assemble the Web Parts that you've built in different ways. Instead of building a single application that must meet the diverse needs of all of your users, you build the Web Parts that your users need, and let your users each build a set of unique applications that meet their needs. Instead of building a single application, you enable the creation of an infinite number of applications, each tailored to its user. This is the ultimate goal of user customization: Each user builds a custom application for himself. With customization, each user sees her own custom version of the underlying application, as shown in Figure 1-2.

This is X-customization: eXtreme customization. But you don't have to go that far in order for Web Parts to be useful to you. If you do allow users to customize your application, it's likely that you'll support only limited customization of restricted portions of your application. And, in all likelihood, rather than each user building a unique application, a single customization will be implemented by many users in the user community.

But you can still consider X-customization as the ultimate goal of Web development — empowering your users with the tools they need to meet their goals.

Customization doesn't have to be limited to your application's users. You probably already recognize that you have different kinds of users in your user community. As a result, you may be planning different parts of your site to serve different user types. As part of this design process, you can create a series of roles for your application, where each role represents a different segment of your user community. The next step is to create a set of controls that can be assembled in different ways to create the pages that make up your application. You can then go one step further and, after adding your controls to the Web page, use the controls as Web Parts and customize them for various types of users. The final step is to assign users to roles so that when a user logs on, she receives the pages designed for her role.

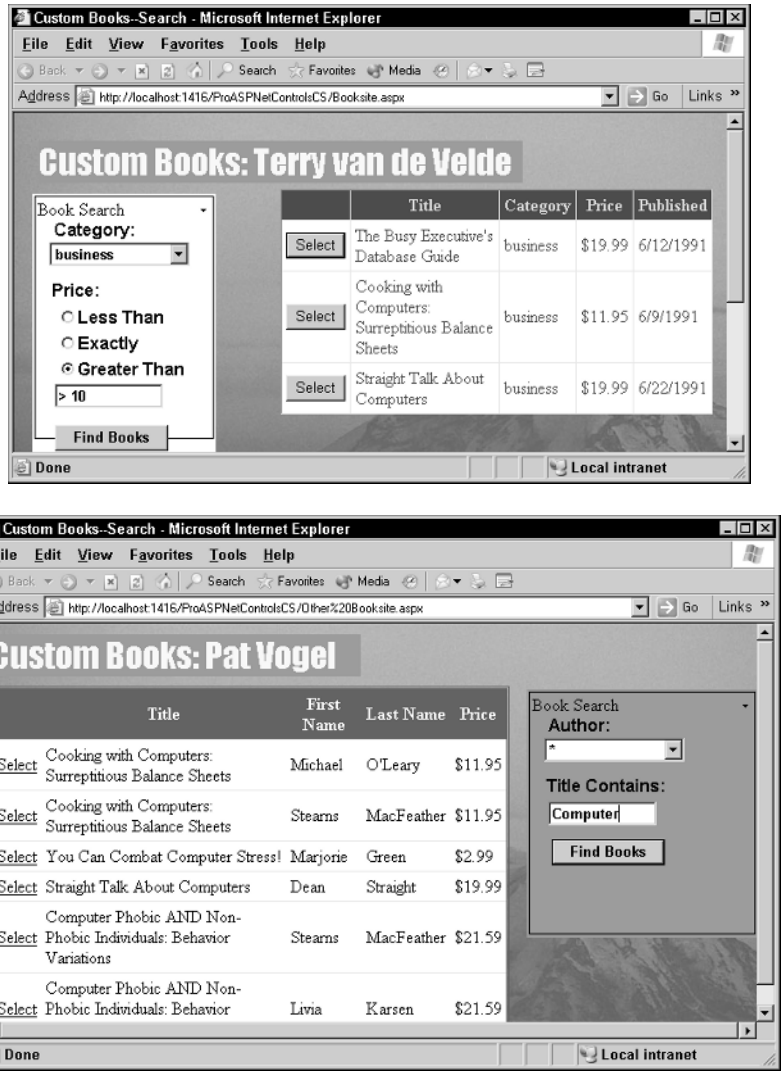


Figure 1-2

Implementing Reusability with Controls

One of the key ways to improve your productivity as a developer is through the re-use of components. If you've been building any kind of application, you've been using (or thinking about using) reusable objects. If you've been developing with ASP.NET, then you've been using (or thinking about using) custom controls and user controls as a means of creating reusable components.

When most developers think of "objects" they think of middleware components — items used to encapsulate business logic and data access that are called from the application code to perform some task. One example of these kinds of objects is the ADO.NET objects that are used in .NET to retrieve and update data (or the DAO and ADO objects in the COM world). There are two fundamental differences between that definition of objects and the ASP.NET-specific tools (custom controls, user controls, and Web Parts).

The first difference is that ASP.NET custom controls, user controls, and Web Parts can be used only with ASP.NET. The ADO.NET objects, for instance, can be accessed from any kind of code. It doesn't matter if your code is in a Windows Form, an ASP.NET page, or a middle-tier object, you can use the ADO.NET objects. However, limiting user controls, custom controls, and Web Parts to ASP.NET has its compensations: the ASP.NET tools, because they are tailored to ASP.NET, leverage the capabilities of the ASP.NET environment.

The second difference between the ASP.NET tools and what most developers think of as "objects" is where the ASP.NET tools are used. Most of the objects that developers create are designed to implement business logic and to reside in an application's middle tier — between the presentation layer (the user interface) and the data layer (the database). As part of creating a sales order system, for instance, a developer might create Customer, SalesOrder, and Invoice objects to handle all the activities involved with managing the application data. These objects would be called from code in the application's user interface and, in turn, update the application data in the database.

The ASP.NET tools, however, work only in the presentation layer, where they become an integral part of the user interface. In terms of the three benefits of using objects (productivity, standardization, simplification), the ASP.NET tools allow you to create a consistent user experience across many Web pages (and, with custom controls/Web Parts, across many Web sites). In addition, middle-tier objects can execute only on the Web server and are accessible only from application code that executes on your Web server. Web Parts, on the other hand, support writing code that will execute on both the server and in the browser, on the client. With the ASP.NET tools you can include client-side code that will be added to the HTML page that goes to the user and, as a result, executes in the Web browser.

Controls in Action

As an example of using controls to implement reusability in a user interface, consider a Web-based application that allows users to browse a catalog of books and order books. A user can log on to the application, search for a book by a list of criteria, read the details on a book, and place an order. Before they can access the site, users must register by entering information about themselves. In addition, when buying a book a customer must enter (or review and confirm) her billing, shipping, and contact information.

In this book application, users can list books in many different ways (for instance, wish lists, gift registries, recommended books, reminder lists). The application supports a variety of different customer types (such as individual consumers, companies, or libraries) and a variety of ways for customers to order books (individual orders, bulk orders, recurring orders, and so on). Customers can buy books using several different purchase mechanisms such as credit card, check, or purchase orders. Refer to Figure 1-2 to see typical pages from the bookstore Web site, listing books that match criteria entered by the user.

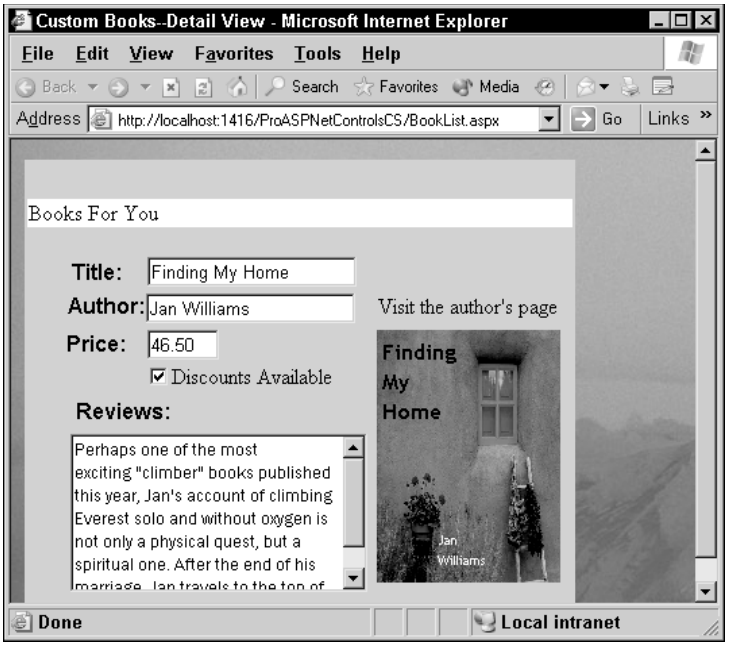
Obviously, many components in this application can be written once and then reused. For instance, a set of business rules that calculate a customer's creditworthiness or calculate the discounts available on an order shouldn't be duplicated in all the places where an order can be made—there's just too much danger that different versions of this code will get different answers. However, this code is implementing business rules and should be put in an object that is accessed by the parts of the application that need that processing.

There are a number of places in this application's user interface where a standardized, reusable control would be useful. Each page, for instance, should have the company logo and basic page information at the top. ASP.NET's master pages sound like a solution to this problem—but the page information isn't exactly the same for every customer (the customer name is included in the page's title bar, for instance). A Web Part would allow you to build a standard title bar for the top of the page that a user could modify to include their name as they want to be addressed and that would be automatically reloaded when that user returns to the site. For this Web Part, very little code may be required—just a set of properties that allow the Web Part's text and graphics to be updated may be sufficient. Figure 1-3 shows the title bars from two different pages implemented through the same Web Part.

Figure 1-3

While the application allows many different ways to list books, the way each book is displayed should be the same. By standardizing the way that book information is displayed, users can quickly figure out where to find particular information (such as price, genre, and so on). To meet this need, a custom control that can display a single book's information would be very useful. Because most of the book information is kept in a database, this Web Part could include the code to gather the information from the database either directly or by interacting with a middle-tier object. All that the application would have to do is pass a book's unique identifier to the custom control and the control would take care of displaying the book's data.

In fact, there's probably a need for two controls: a detailed book control that would display all the information on a book, and a summary book control that would display only basic information. Because the detailed information control would use a wide variety of controls and formatting to display all the information, it might be easiest to create it as a user control. Because the summary information control requires fewer controls, it might be easily created as a custom control. Figure 1-4 shows examples of these two Web Parts, with the detailed display above the summary display.


Custom Books--Detail View - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:1416/ProASPNetControlsCS/BookList.aspx
Go Links
Books For You
Title: Finding My Home
Author: Jan Williams
Price: 46.50
Discounts Available
Reviews:
Perhaps one of the most exciting "climber" books published this year, Jan's account of climbing Everest solo and without oxygen is not only a physical quest, but a spiritual one. After the end of his
Visit the author's page
Finding My Home
Jan Williams
Done
Local intranet

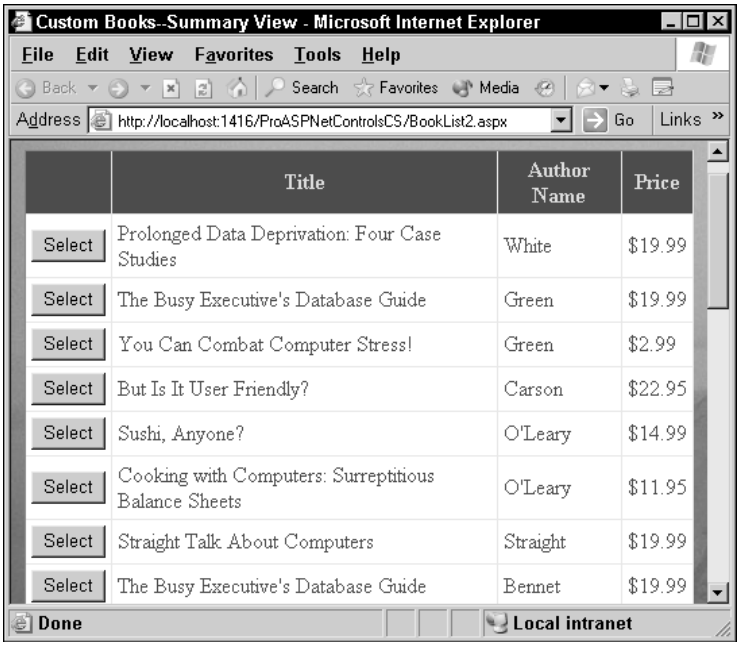

Custom Books--Summary View - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:1416/ProASPNetControlsCS/BookList2.aspx
Go Links
Title
Author Name
Price
Select Prolonged Data Deprivation: Four Case Studies White $19.99
Select The Busy Executive's Database Guide Green $19.99
Select You Can Combat Computer Stress! Green $2.99
Select But Is It User Friendly? Carson $22.95
Select Sushi, Anyone? O'Leary $14.99
Select Cooking with Computers: Surreptitious Balance Sheets O'Leary $11.95
Select Straight Talk About Computers Straight $19.99
Select The Busy Executive's Database Guide Bennet $19.99
Done
Local intranet

Figure 1-4

It would be convenient to have a single control that could be used to build all lists. This control could be implemented in one of two ways: it could either accept a set of criteria for building a list of books, or accept a list of book identifiers generated by some other part of the application. Either way, this control would use the previously created book summary control to display the individual books on the list in a standard way.

Throughout the application, customer information is gathered and displayed. Rather than make the user work with several different formats for customer information, why not create a customer information control? This control would work in two modes: data entry and data display. In the data display mode, the control would retrieve customer information from the database and display it. In the data entry mode, the control would accept customer information and do whatever validation is appropriate to the user interface (ensuring the customer phone number is correctly formatted, for instance). Once the data has passed validation, the control would take care of updating the database.

Here is a case where using a Web Part would be the best choice. If, for example, the Web page contains multiple controls, how do these controls commit their changes to the database? For instance, the Web page that lets customers order books might contain the control for entering customer information and a control for entering sales order header information. If these controls interact with the database (or middle-tier objects) individually, processing updates from the page could be very inefficient. The code on the Web page could coordinate updates from the controls, but then you lose some of the benefits of reusability that controls are supposed to provide. By taking advantage of the capability of Web Parts to find other Web Parts on the page and communicate with them, the code in the Web Parts could coordinate their updates. All the display-oriented Web Parts could look for an "update" Web Part on the page and send their updates to that Web Part for processing.

In either display or update mode, the customer control would have a set of properties that would expose the customer information to the application that's using the Web Part. Figure 1-5 shows the customer information Web Part in two different pages.

It's not hard to see that in addition to the benefits of standardization, there exist significant opportunities to improve the productivity of the person building or extending this application. When it's time to build a page that displays all the books for a specific customer, the developer who's building the page can start by adding the customer information control and the book listing control (which, in turn, uses the book summary control). From that point on, the developer just needs to add any specialized code required by the particular page.

Exploring the Different Kinds of Controls

Now that you've been introduced to the possibilities of Web Parts, custom controls, and user controls, you may be wondering how to decide which control to use in which situations. In this section, I offer guidelines on when you should use a user control or a custom control (remember that Web Parts are just a kind of custom control). Nothing is free, of course, so you also see what costs you incur by picking each type of control.

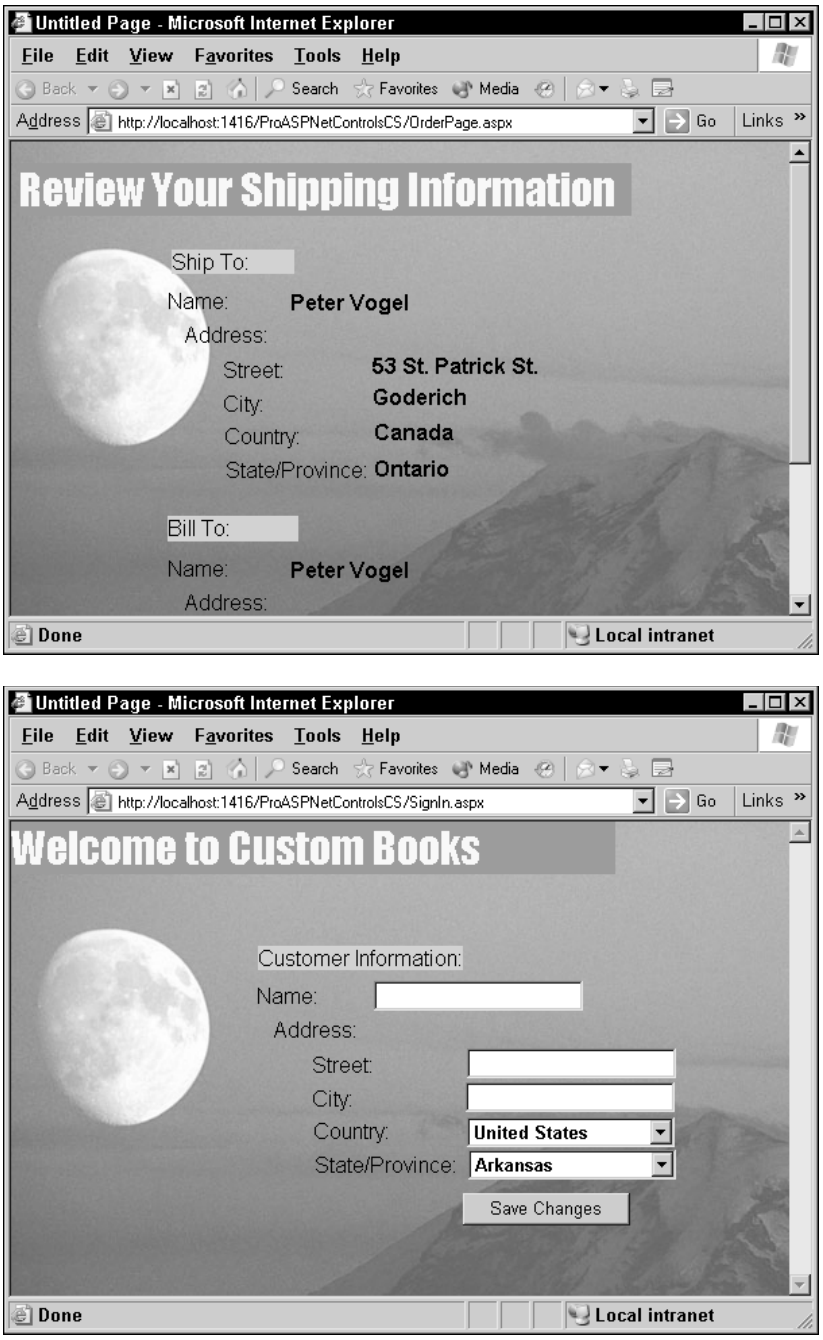

Untitled Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:1416/ProASPNetControlsCS/OrderPage.aspx
Go Links
Review Your Shipping Information
Ship To:
Name: Peter Vogel
Address:
Street: 53 St. Patrick St.
City: Goderich
Country: Canada
State/Province: Ontario
Bill To:
Name: Peter Vogel
Address:
Done
Local intranet
Untitled Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:1416/ProASPNetControlsCS/SignIn.aspx
Go Links
Welcome to Custom Books
Customer Information:
Name:
Address:
Street:
City:
Country: United States
State/Province: Arkansas
Save Changes
Done
Local intranet

Figure 1-5

When to Use a User Control

User controls should be your first choice for creating Web Parts where the control will be used on a single Web site only and isn't extending some other control. There are many benefits to using user controls:

- The drag-and-drop design interface supported by user controls is the simplest and fastest way to create your controls' user interface.
- You can put any amount of server-side code behind a Web user control, so you can have your Web Part perform any server-side processing that you want.
- You can give your Web Part custom properties, methods, and events, increasing the flexibility of the Web Part.
- You can include client-side code and HTML in your Web Part along with ASP.NET WebForm controls and server-side code just by switching to HTML view. This also lets you take advantage of Visual Studio .NET's support for writing client-side code (including IntelliSense).
- In addition to being more powerful than pure HTML, ASP.NET WebForm controls generate their HTML at run time and can configure the HTML they generate to match the device that is requesting them.

Because a user control can be used in a single project only, if you want to use a user control on two Web sites, you have to copy the control from the project that it was created in and paste it into the other project that it will be used in. This means that you will eventually end up with multiple copies of the same user control. When that happens you lose two of the benefits of components:

- **Standardization:** Inevitably, the multiple copies of the user control will start to diverge. Each version of the user control will be tweaked to work with the project that it's being used in. As a result, the functionality embedded in the user control will start to work differently in the different versions.
- **Productivity:** The first loss in productivity occurs because multiple developers continue to develop the different versions of the user control. The second loss occurs because developers will, eventually, have to take time to reconcile the different versions of the user control. For instance, when a bug is discovered, or a change is required because of changes in the organization, or the opportunity for an enhancement is recognized, developers will have to chase down all the versions of the control to make the change.

When to Use a Custom Control

Custom controls can do anything that a user control can do — it just takes longer to build them. (This makes custom controls an excellent choice if you are paid by the hour.) When building a custom control you may find that you have to write the server-side code that generates the client-side portion of your Web Part, including any HTML. This means that you give up the way that ASP.NET's standard controls automatically adapt their HTML to the client that requested them.

You can still use ASP.NET WebForm controls in your Web custom control, but you must write code to add those WebForm controls to your user interface rather than using drag-and-drop. The same is true of any client-side code that you add to your Web Part: The client-side code must be generated from your server-side code, rather than typed into the HTML view of a Web Page. Without the benefit of the Visual

Studio .NET designer, it's going to take much longer to lay out your user interface. And, without the benefit of Visual Studio .NET's IntelliSense support for writing client-side code, it's going to take longer to write bug-free client-side code, also.

You should use a custom control only if there is some compelling reason for not using a user control. In the days of the ASP.NET 1.0 and 1.1, there were several benefits to using custom controls compared to user controls — custom controls had a design-time interface and could be shared among projects, for instance. However, with ASP.NET 2.0, some of those differences have gone away. With ASP.NET 2.0, there are only three reasons that might cause you to consider using a custom control:

- You should use a custom control whenever the control will be used in more than one project.
- You should use a custom control when your HTML can't be generated by some combination of existing ASP.NET controls.
- You should use a custom control when you want to extend an existing ASP.NET control (rather than write all the functionality yourself). While you can extend an existing ASP.NET control by wrapping it inside a user control, it's often easier to extend an existing control by inheriting from the existing control and modifying it in some way.

Web custom controls do offer another benefit: they expose all the code involved in creating a control — a good thing in a book about creating your own controls.

In the book site example, a user control would be the best choice for the title bar because the title bar is limited to a single Web site and is built from existing Web server controls. The properties exposed by the title bar Web Part would allow the code behind a page to insert the text that the page needs. The listing control described in the case study would be best implemented through a custom control that extends the ASP.NET DataList control.

Web Parts in Action: Customization

But when should you use a Web Part? First, remember that any control can be used as a Web Part. When used as a Web Part, user controls and custom controls can be customized in several ways:

- The control can be added or removed from the page.
- The control's appearance can be modified.
- The control can be moved to a different location on the page.

Users will also be able to set the control's property values interactively at run time. However, if you build a control as a Web Part from the ground up, you can give your control even more features — the capability to pass data between other controls, for instance.

When considering how you can use Web Parts, ask yourself if it's necessary that you build every page in your application. Can you improve your productivity by letting your users build parts of the application for you? This section offers a few examples based on the bookstore application. Your users will certainly come up with many suggestions for customizing your application by using Web Parts.

When you build an application, you are faced with decisions on what user interface will work best for your users. For instance, the application's designers may have decided not to display book detail information on the same page as a book list. The site's designers may have thought the amount of information on a page with both a list and detailed book information would be overwhelming — for *most* (but not all) users. So, while users can generate a list of books with summary information, when the user wants to look at detail information he has to click a button and go to a new page. To continue to work through the list, users must click a button to return to the list. The result is that users "ping-pong" back and forth between the list and detail pages. For many users, this is the optimal design.

But there may be users out there who resent having to click an item in a book list in order to go to a new page that displays the detail information on a book. They would prefer to have the list display the detailed book information, rather than just the summary information. Given access to the controls already discussed, using those controls as Web Parts would let users build this detailed list page by dropping the detail book Web Part onto a page with the listing control. Look again at Figure 1-4: The top portion shows the standard book listing using the detailed book information Web Part, while the bottom portion illustrates the same listing, but now using the summary book information Web Part.

With Web Parts, any user can build a page to meet her unique needs, the only limits being the toolkit of Web Parts that you provide. For instance, the application probably has a page that displays all the books purchased by the current customer. The user just navigates to the page to see the books she's bought. However, it's not hard to imagine that customers might want to list all the books purchased by their company or some other buying entity that they have permission to view. To get this list by using one of the application's built-in lists, the customer has to navigate to the page, enter the name of the organization into the search criteria, and then generate the list. You could expand your application to hold the customer's various affiliations and automatically display books purchased by affiliated organizations — but where does this stop?

Instead, you could allow your users to build a dedicated search page. The first step would be to enhance the customer information control so that the user can set the properties on the control, including the customer name. A user could then drag the customer information control to a page and customize the control by setting the customer name to some buying entity that they are allowed access to. With that done, your user could drag the listing Web Part onto the page and connect it to the customized customer information Web Part. The listing Web Part would now display the books purchased by the entity entered in the customer Web Part. The user could redisplay the list just by navigating to the page.

As this example indicates, you may want to create Web Parts whose sole purpose is to support user customizations. For instance, the application has several places where users can enter search criteria for listing books. However, it may not make sense to build a separate control for entering search criteria or listing books because there's no opportunity for reuse. It may be that every search page supports a different set of criteria and each list has a unique format for displaying the books found in the search. In addition, this version of the application uses only the summary book information and detail book information Web Parts.

Even though there's no opportunity for reuse, it may still make sense to create controls just to support customization. To create opportunities for customization, you could create sets of Web Parts for:

- **Entering search criteria:** One control might provide a limited number of search criteria (just author and title), another control might provide an extensive list of options familiar to the general audience (author, title, publisher), while another control might offer options only of interest to collectors (allowing the user to specify particular editions or publication dates, for instance).

- **Listing books:** One control might provide a single column with limited options for sorting the list, another control could support complex sorting options, another control might format the user into a rigid grid format with a single line for each book, and yet another control might allow the list to "snake" through multiple columns, allowing more books to be listed on a page.
- **Displaying book information:** Different controls might vary in the amount of information displayed about a book or which information stands out. One control might be formatted so that information about book size and weight stands out for retail buyers concerned about reducing shipping costs — information that the typical reader isn't interested in.

With those controls created, you could add a page to your application that consists of two areas:

- One area at the top of the page to place one of the search criteria controls
- An area below that to put one of the controls for listing books

Users could draw from your toolkit of Web Parts to build the search page that they want. Users would be able to put the search criteria Web Part that most meets their needs at the top of the page and put the listing Web Part they want below that. To work with the listing control, they could add the book information Web Part that supports them best. This is a solution aimed purely at giving users the opportunity to create the application they need.

Providing for Personalization

In order to implement customization, you also need *personalization*. Your users won't be happy if they have to recreate their application each time that they return to your site, or if, when they return, they get some other user's customization. Customization is of no use unless the application remembers what changes a user has made and associates those changes with the user that made them. Fortunately, ASP.NET 2.0 comes with a personalization framework. The ASP.NET 2.0 personalization framework allows you to implement an application that tracks users and the choices they make so that when the users return to your application, they find their own customizations waiting for them. Each user is connected to his customized application after he logs on with little effort on your part.

This description makes personalization sound like a "nice-to-have" feature. In fact, personalization is really just the extension of identity-based security, which is essential in building an application. When users access your Web site, they are automatically logged on to your Web site's server. The logging in process assigns the user an identity, even if it's only the default "anonymous user" assigned by IIS. If you've turned off the capability to log on as the anonymous user, then the user may be silently logged on using whatever credentials are currently available (for instance, the user ID and password that the user logged on to his workstation with). If no valid credentials are available, the user may be asked to enter a user ID and password. After the user has logged on to your site's server, your application may have an additional level of security that requires the user to enter a user ID and password into your application's login page.

All of this work, both in the infrastructure that supports your Web application and in the application code behind your login page, has just one purpose: to establish who the user is (the user ID) and to authenticate that claim (the password). Once a user is authenticated, she is then authorized to perform specific activities.

Whatever mechanism is used to authenticate the user, when the process is completed successfully, the user has been assigned an identity. From this point of view, security is just the base level of personalization; security assigns an identity that is authorized to perform some activities (and forbidden to perform others). Personalization extends this security feature up from the ASP.NET infrastructure and into the application domain. Personalization allows you to manage your application on the basis of who the user is.

> *The identity you are assigned when you log onto the Web server is used just within the application. When your code accesses other services (for example, reading or writing a database), those accesses are normally performed by an identity that represents ASP.NET. (On Windows 2003, this identity is called NETWORK SERVICE; on other versions of Windows the identity is called ASPNET.) In your application's Web.Config file you can turn on impersonation, which causes the ASP.NET application to adopt the identity used to log on to the server: the anonymous user if anonymous access is enabled, the user's identity if anonymous access is not enabled.*

Understanding the Personalization Framework

The good news is that the personalization framework will take care of itself—by and large you can just build on the personalization framework and count on it to work. However, there are some decisions that you will need to make as part of setting up a Web site (for example, selecting the correct provider for your site). In order to make those decisions you need to understand the components of the personalization framework.

The personalization framework has three main components:

- Login controls
- Membership providers
- Profile services and providers

The first components of the personalization framework that a user encounters are ASP.NET 2.0's new login and user controls. Rather than write all the code necessary to log in a user, you can simply drag and drop the new login controls to a Web page. These controls handle all the typical tasks associated with the log on process (including sending forgotten passwords to users). For personalization, these controls allow a user to be assigned an identity. Separate from this process, the site administrator has to register with the personalization datastore the identities that users can be assigned. Figure 1-6 illustrates the three elements of the personalization framework: login controls, membership providers, and profile services.

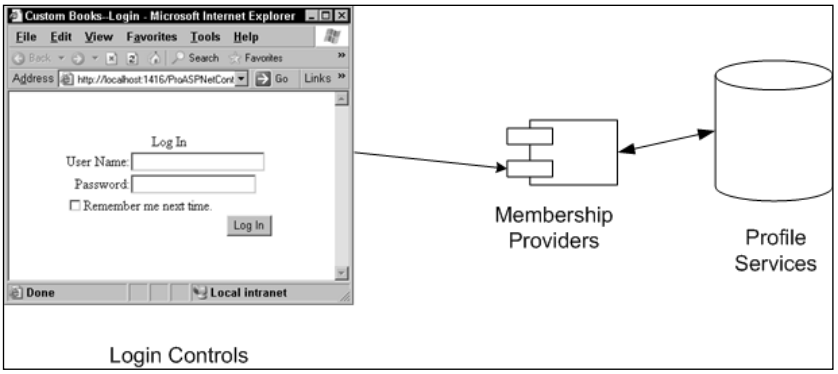


Figure 1-6

Within the area of user management, some other features of SharePoint have also migrated into ASP.NET 2.0. Within SharePoint, it's possible for authorized users to create new SharePoint sites, new users, new pages, and perform other administrative tasks.

In ASP.NET 2.0, as part of the personalization framework, it's possible to implement user management functions within the site itself (although you still can't create a whole new Web site from within an application). ASP.NET even comes with a set of pre-defined user administration pages that you can incorporate into your site.

Logging in is more than just assigning identities. For sites that have a small number of users, implementing personalization on an identity-by-identity basis is a viable choice. However, as the number of identities increases, the costs of maintaining personalized information increases. While costs increase with the number of identities, the benefits of personalization don't increase. In all likelihood, the customizations made for one identity will be applicable to other identities. In any user community, you can probably break your user community up into several groups. This can be handled by assigning individual users to roles and implementing customizations on a role-by-role (group) basis. As a result, you need a mechanism that not only assigns identities to users but also assigns users to roles.

The next component of the personalization framework, the membership provider, handles this. The membership provider is the glue that binds the site to the users that access the site and binds the site's functionality to the various user roles. Membership providers also handle all the tasks around storing user and role information. Two membership providers come with ASP.NET 2.0: one for storing information in Microsoft SQL Server and one for storing information in a Jet database. If you want, you can even build your own membership provider to connect to other data stores.

A provider is a component that extends or replaces some existing ASP.NET function. Any part of ASP.NET that is implemented through a provider model can be enhanced or replaced with a provider written by you (or some third party, such as IBM's Tivoli for transaction management). Once you have built a new provider, you plug it in to the list of available providers and select it when you build your application (which is why providers are said to be "pluggable"). The main requirement of a provider is that it has to reproduce the interface (methods, properties, events) for the provider that it extends or replaces. However, what happens behind that interface is up to the developer that creates the provider.

The final component of the personalization framework is the profile service. A profile is all the data associated with a specific identity. The profile service allows you to store and retrieve data for a particular identity or role. To access the profile service, you need a profile provider that handles all the data access for you. The profile service is very flexible: you can store any data from simple datatypes (for example, strings) right up to an object (provided that the object can be serialized). In addition, saving and restoring data from the profile is handled for you automatically.

> **The personalization framework allows you to store, as part of a profile, the Web Part customizations associated with some identity. The membership provider allows you to keep track of which identities are being used. The login controls assign identities to users.**

Summary

In this chapter you've learned about the two types of controls that you can use in ASP.NET:

- **User controls:** A combination of content, static HTML, ASP.NET tags, and code, built using the same tools that you use to create Web pages.
- **Custom controls/WebParts:** A code-only solution, very similar to ASP.NET custom controls. Unlike custom controls, you cannot inherit from other controls when building a Web Part. These controls are the focus of this book.

By the end of this book you'll have learned how to build the more powerful and flexible custom controls. I also show you how to use these controls as Web Parts and how to extend custom controls to let users customize your site. Along the way, you'll also see how easy it is to build user controls — and both how to add custom control features to user controls and use them as Web Parts.

While it's good to know about controls, what's important to you is what you do with those controls. You've seen how these ASP.NET tools support two different scenarios:

- **Reusability:** Controls support reusability in the user interface (or presentation) layer rather than in the business rules layer. Like other reusability tools, controls enable you to provide a standardized set of services and improve your own productivity.
- **Customization:** Web Parts allow you to support customization to a degree that simply wasn't possible in earlier versions of ASP and ASP.NET. With Web Parts, you can allow your users to create customized versions of existing pages, or to create pages that were not intended to be included in the application. While this empowers your users, it also opens up a whole new set of challenges for the developer, both in determining what Web Parts the user community will need and ensuring that those Web Parts will work in all the different ways that users want.

Finally, in order to support customization, you also need personalization: the ability to keep track of a user's choices, remember those changes from one session to another, and connect the user to his customized pages. Personalization is the extension of identity-based security into application development. Over the rest of this book, you'll see how to build custom controls, user controls, and Web Parts.