User & Custom Controls

Introduction

- User controls allow you to reuse a portion of a page by placing it in a special .ascx file.
- A *user control* encapsulates existing ASP.NET controls into a single container control.
- Custom-derived controls allow you to build a new control by inheriting from an ASP.NET control class.
- A *server control* encapsulates the visual design, behavior, and logic for an element that the user interacts with.
- The controls are reusable across multiple pages.

User Controls

- Simplest form of ASP.NET control encapsulation.
- Grouping of existing server controls into a single-container control.
- Add User control to the website from Website menu of VS2K5.
- The file has an .ascx extension.
- Every ascx file starts with *Control* direcive.

User Control

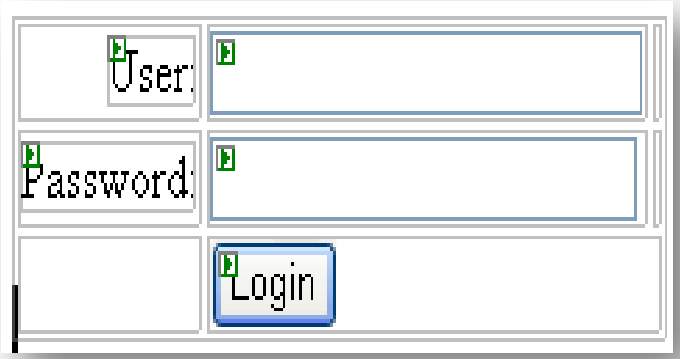
```
<%@ Control Language="C#" AutoEventWireup="true"
CodeFile="MyControl.ascx.cs" Inherits="MyControl" %>
```

- Every user control starts with above line.
- The attributes of control directive are similar to page directive.
- If code inline model is used, it'll have <script> tag with *runat* attribute.
- Form, html, body, head tags are provided by containing page. Hence no tags. We can add new as required.

User Control UI

```
%@ Control Language="C#" AutoEventWireup="true“
CodeFile="MyControl.ascx.cs" Inherits="MyControl" %>
<table>
   <tr> <td align="right"><asp:Label ID="Label1" runat="server" Text="User:“ />
      </td>
      <td><asp:TextBox ID="txtUser" runat="server"></asp:TextBox></td>
      <td></td>
   </tr>
   <tr> <td align="right">
<asp:Label ID="Label2" runat="server" Text="Password:">
            </asp:Label> </td>
      <td><asp:TextBox ID="txtPass" runat="server" TextMode="Password" />
      </td>
      <td> </td>
   </tr>
   <tr> <td> </td>
      <td colspan=2> <asp:Button ID="butLogin" runat="server" Text="Login" />
      </td>
   </tr> </table>
```

User Control UI & Event Handling



- One can add code to event handlers. Or
- One can publish events in User control which can be subscribed by control user.
- To add the controls to the user control, drag the ascx file on aspx page.

User Control Logic

```
private string url;

  public string RedirectUrl
  {
      get { return url; }
      set { url = value; }
  }
 protected void butLogin_Click(object sender, EventArgs e)
  {
      if ((txtUser.Text == "iconnect") && (txtPass.Text == "iconnect"))
      {
          if (url == null)
              throw new Exception("Redirect URL not specified.");
          Response.Redirect(url);
      }
  }
```

Put the code in code-behind class file.

User Control Code-Behind

- In Code-behind
  - One can program for
    - Fields
    - Properties
    - Methods
    - Events (Event Handlers)
  - User controls can expose events also.

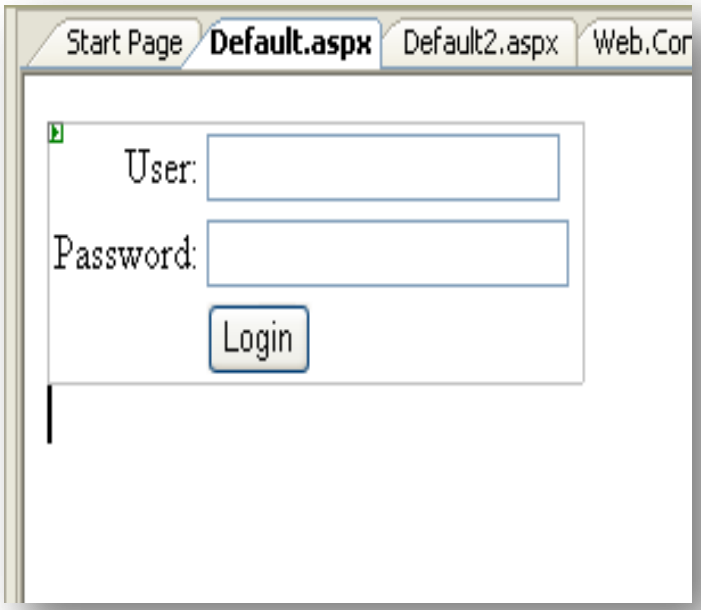
Using User Control

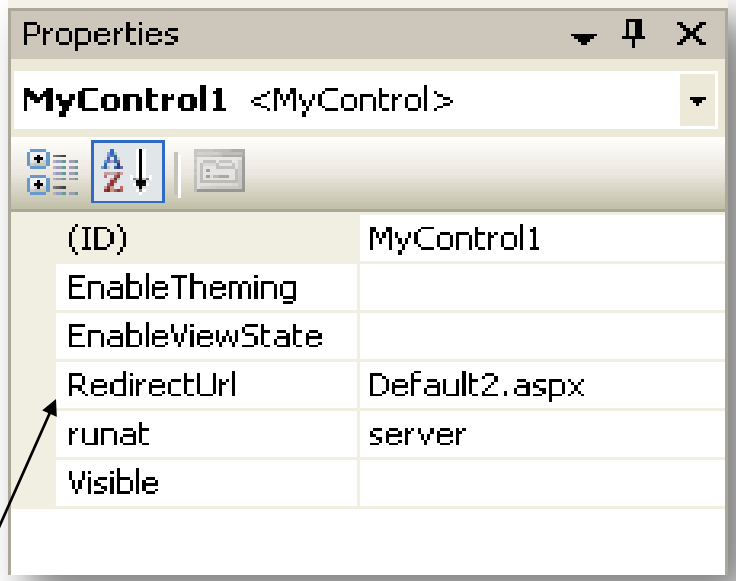
- To use control
  - Either drag ascx on aspx from solution explorer.
  - Or
    - Register user control with page.

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

```
<div>
    <uc1:MyControl ID="MyControl1" runat="server" RedirectUrl="Default2.aspx" />
     </div>
```

User Control on ASP.NET page





Property Browser show user control properties also.

Adding User Control Dynamically

- Use *LoadControl* method of *Page* class.
- Add the returned reference to container control's (like PlaceHolder or Panel) *Controls* collection.

```
MyControl ctrl = this.LoadControl("MyControl.ascx") as MyControl;
 ctrl.RedirectUrl = "Default2.aspx";
 Panel1.Controls.Add(ctrl);
```

MyControl must be registered with the page. If not, cannot typecast.

Custom Control

- All controls in ASP.NET are derived from two basic classes: System.Web.UI.Control

  or

- System.Web.UI.WebControls.WebControl

- Controls derived from *WebControl,* support for many of the basic styling elements such as Font, Height, and Width.

Creating Custom Control

- Create a *Web Control Library* project.
- It has a class derived from *WebControl*.
- By default it has *Text* property & *RenderContents* (overridden) method.
- RenderContents has parameter of *HtmlTextWriter* which writes HTML to render the control.

HtmlTextWriter Class

- Writes markup characters and text to an ASP.NET server control output stream

- Methods
  - RenderBeginTag
    - Writes the opening tag of a markup element to the output stream.
  - RenderEndTag
    - Writes the end tag of a markup element to the output stream.

HtmlTextWriter Class

- Methods
  - AddAttribute
    - Adds the specified markup attribute and value to the opening tag of the element that the *HtmlTextWriter* object creates with a subsequent call to the *RenderBeginTag* method.
  - AddStyleAttribute
    - Adds a markup style attribute to the opening tag of the element that the **HtmlTextWriter** object creates.

Adding Properties

- One can add properties to control to make it programmable & customizable.
- Public properties double as attributes in tags.
- Define property in control class.
- Server control's lifetime= one HTTP request.
- Changes in property values cannot be noted unless we persist property value in view-state. e.g. Contents of textbox, when changed, can be noted if existing view-state text & whatever user sends back on postback are not matching. One cannot fire events if property values are not persisted.

Adding Property

```
public class MyControl:WebControl
{
        public string Name
        {
                get
                {
                        if(ViewState[“name”]!=null)
                        return ViewState[“name”].ToString();
                        return “”;
                }
                set
                {
                        ViewState[“name”]=value;
                }
        }
        …
}
```

Custom Control Important Points

- If a tag output by a control includes a *Name* attribute, the value of that attribute should be taken from inherited *UniqueID* property.

- If a tag output by a control includes an *Id* attribute, the value assigned from inherited *ClientID* poperty.

UniqueID & ClientID

- They are never null.
- Even if the control lacks a Id attribute, *UniqueID* & *ClientID* give each control instance a unique value.
- When control is put in replicator-type control (e.g.GridView), each control has unique identity.

Handling Postbacks

- The interface *IPostBackDataHandler* must be implemented by control class to handle postback.
- It also allows to update control properties with new values.
- Two methods
  - LoadPostData
    - *Receives postback data & compares old & new values. If equals, returns false. Else returns true.*
  - RaisePostDataChangedEvent
    - If *LoadPostData* returns true, it fires an event to notify subscribers.

LoadPostData Method

- Receives two parameters: *postDataKey* as string & *postCollection* as NameValueCollection.
- *postCollection* holds all data that accompanied the postback.
- *postCollection* can be indexed, *postDataKey* holds the index of data that corresponds to our control.

```
public bool LoadPostData(string postDataKey,
                         NameValueCollection postCollection)
    {
        string oldname=Name;
        Name = postCollection[postDataKey];
        return (oldname != Name);
    }
```

Events

- Events are notification to normal happenings.
- Events can be shared also.
- Events are public or protected.
- Control class publishes events if required.

e.g. Button is clicked. Click is an event.

Publishing Events & RaisePostDataChangedEvent Methods

```
public class MyControl:WebControl,IPostBackDataHandler
{
        public event EventHandler NameChanged;
                ...
        public void RaisePostDataChangedEvent()
            {
                if (NameChanged != null)
                    NameChanged(this, new EventArgs());
            }
...
}
```

Tests whether event-handler is present for the event or not.

Generating Postbacks (AutoPostBack)

- Create a property *AutoPostBack*.
- While Rendering output check for *AutoPostBack* value. If true, using *GetPostBackEventReference* method, we'll set javascript to client-event, which in turn post backs the page to server.

```
if(AutoPostBack)
  writer.AddAttribute("onchange","javascript:"
            +Page.GetPostBackEventReference(this));
```

Consider TextBox textchanged event.

IPostBackEventHandler Interface

- *IPostBackEventHandler* enables controls that generate postbacks to be notified when they cause postbacks to occur.

- e.g. *LinkButton* control implements the interface. Server-side processing includes *Click* and *Command* events of link button, if *LinkButton* has caused the postback.

- *RaisePostBackEvent* is the only method defined in the interface.

RaisePostBackEvent Method

- Only parameter is *eventArgument* as string.
  - It is a second parameter of *GetPostBackEventReference*.
- Provides ability to pass application-specific data to controls that generate postbacks & its *RaisePostBackEvent* action depends on that data.