ASP-Unit 4

Topic: Web Server Controls

Read Web server control = Same answer of **ASP.Net Server Controls** in Unit 1

(Hun tak asi sirf html controls padhe, dekhde ee agle type de controls:-)

Standard Controls below here:-

(textbox, checkbox, radiobutton, button, imagebutton, image**)-read from Unit 3**

****Label Control in ASP.NET:****

1. **Definition:**

- A Label control in ASP.NET is like a placeholder for text on your web page. It's used to display static text that doesn't change based on user interactions. You can use it to show instructions, titles, or any other non-interactive text.

2. **Syntax (How to Write It):**

- To use a Label control, you use the `<asp:Label>` tag in your ASP.NET page. Here's a basic example:

```html
<asp:Label runat="server" ID="MyLabel" Text="Welcome to ASP.NET"></asp:Label>
```

3. **Working (What It Does):**

- The Label control doesn't do much by itself; it's a way to display text on your web page. You can set its `Text` property in the markup or dynamically in your server-side code (C# or VB.NET). When the page is loaded, the text specified in the Label control is rendered on the webpage.

4. **Detailed Working Example:**

- Let's say you have a Label control in your ASP.NET page:

```html
```

<asp:Label runat="server" ID="MyLabel" Text="Welcome to ASP.NET"></asp:Label>

- In your ASP.NET code (C#), you can dynamically change the text of the label:

```csharp
protected void Page_Load(object sender, EventArgs e)
{
    // Your custom logic here, for example, change the text of the label dynamically
    MyLabel.Text = "Hello, Greetings from ASP.NET!";
}
```

- Now, when the page is loaded, the Label control will display the updated text.

****ImageMap Control in ASP.NET:****

1. **Definition:**

- An ImageMap control in ASP.NET is like a map overlaid on an image, allowing you to define clickable regions. It turns different areas of an image into interactive zones, and you can associate each zone with a specific action or link.

2. **Syntax (How to Write It):**

- To use an ImageMap control, you use the `<asp:ImageMap>` tag in your ASP.NET page. Here's a basic example:

```html
<asp:ImageMap runat="server" ID="MyImageMap" ImageUrl="image.jpg">
    <asp:RectangleHotSpot Left="0" Top="0" Right="50" Bottom="50" NavigateUrl="Page1.aspx" />
    <asp:RectangleHotSpot Left="50" Top="0" Right="100" Bottom="50" NavigateUrl="Page2.aspx" />
```

</asp:ImageMap>

In this example, `RectangleHotSpot` defines clickable rectangles on the image, and each rectangle has a `NavigateUrl` property, specifying the page to navigate to when clicked.

3. **Working (What It Does):**

- The ImageMap control allows you to define hotspots (clickable areas) on an image. Each hotspot is associated with a specific action, such as navigating to another page or triggering a server-side event. When a user clicks on a defined hotspot, the associated action is executed.

4. **Detailed Working Example:**

- Let's say you have an ImageMap control in your ASP.NET page:

```html
<asp:ImageMap runat="server" ID="MyImageMap" ImageUrl="worldmap.jpg">
    <asp:RectangleHotSpot Left="0" Top="0" Right="50" Bottom="50" NavigateUrl="NorthAmerica.aspx" />
    <asp:RectangleHotSpot Left="50" Top="0" Right="100" Bottom="50" NavigateUrl="Europe.aspx" />
</asp:ImageMap>
```

- When a user clicks on the North America region, they are navigated to the "NorthAmerica.aspx" page. Clicking on the Europe region navigates them to the "Europe.aspx" page.

****Panel Control in ASP.NET:****

1. **Definition:**

- A Panel control in ASP.NET is like a container that helps organize and group other controls on a web page. It provides a way to manage the layout and appearance of related elements.

2. **Syntax (How to Write It):**

- To use a Panel control, you use the `<asp:Panel>` tag in your ASP.NET page. Here's a basic example:

```html
<asp:Panel runat="server" ID="MyPanel">
    <!-- Other controls or content go here -->
    <asp:Label runat="server" Text="This is inside the panel"></asp:Label>
</asp:Panel>
```

3. **Working (What It Does):**

- The Panel control acts as a container for other controls or content. It allows you to group related elements together. You can manipulate the visibility, style, or other properties of the entire panel, affecting all the controls within it.

4. **Detailed Working Example:**

- Let's say you have a Panel control in your ASP.NET page:

```html
<asp:Panel runat="server" ID="MyPanel" BorderStyle="Solid" BorderWidth="1">
    <asp:Label runat="server" Text="This is inside the panel"></asp:Label>
    <asp:Button runat="server" Text="Click Me" OnClick="btn_Click"></asp:Button>
</asp:Panel>
```

In this example, the Panel contains a Label and a Button. The panel has a solid border, and all the controls inside it are part of the panel.

In the code-behind (C#), you might have an event handler for the button click:

```csharp
protected void btn_Click(object sender, EventArgs e)
```

```
{
    // Your custom logic here, for example, change the label text
    MyLabel.Text = "Button clicked!";
}
```

- Now, when the button inside the panel is clicked, the label inside the panel is updated.

****Hyperlink Control in ASP.NET:****

1. **Definition:**

- A Hyperlink control in ASP.NET is like a clickable link that directs users to another page or resource on the web. It allows you to create navigation elements in your web applications.

2. **Syntax (How to Write It):**

- To use a Hyperlink control, you use the `<asp:HyperLink>` tag in your ASP.NET page. Here's a basic example:

```html
<asp:HyperLink runat="server" ID="MyHyperlink" NavigateUrl="https://www.example.com" Text="Visit Example.com"></asp:HyperLink>
```

3. **Working (What It Does):**

- The Hyperlink control creates a clickable link on your webpage. When a user clicks on the link, it can navigate to another page or resource specified by the `NavigateUrl` property. The link text is specified by the `Text` property.

4. **Detailed Working Example:**

- Let's say you have a Hyperlink control in your ASP.NET page:

```html

<asp:HyperLink runat="server" ID="MyHyperlink" NavigateUrl="Page2.aspx" Text="Go to Page 2"></asp:HyperLink>

In this example, clicking on the link will navigate the user to "Page2.aspx."

You can also dynamically change the hyperlink's properties in your code-behind (C#):

```csharp

protected void Page_Load(object sender, EventArgs e)

{

    // Your custom logic here, for example, change the hyperlink URL

    MyHyperlink.NavigateUrl = "UpdatedPage.aspx";

}
```

- Now, when the page is loaded, the hyperlink's destination is dynamically updated.

List Controls in ASP.NET:

List controls in ASP.NET are a group of server-side controls that enable the presentation and manipulation of lists or collections of items. These controls provide a convenient way to work with data in a structured manner, and they include various types such as dropdown lists, radio button lists, list boxes, checkbox lists, and bulleted lists.

1. **DropdownList Control:**

- **Definition:**

  - A DropDownList control in ASP.NET is like a dropdown menu that allows users to select one option from a list. It's often used when you have a long list of options, and you want users to choose one.
```

- **Syntax:**

```html
<asp:DropDownList runat="server" ID="MyDropDownList">
    <asp:ListItem Text="Option 1" Value="1"></asp:ListItem>
    <asp:ListItem Text="Option 2" Value="2"></asp:ListItem>
    <asp:ListItem Text="Option 3" Value="3"></asp:ListItem>
</asp:DropDownList>
```

2. **RadioButtonList Control:**

- **Definition:**

  - A RadioButtonList control in ASP.NET is like a group of radio buttons where users can choose one option. It's useful when you have a set of mutually exclusive choices.

- **Syntax:**

```html
<asp:RadioButtonList runat="server" ID="MyRadioButtonList">
    <asp:ListItem Text="Option A" Value="A"></asp:ListItem>
    <asp:ListItem Text="Option B" Value="B"></asp:ListItem>
    <asp:ListItem Text="Option C" Value="C"></asp:ListItem>
</asp:RadioButtonList>
```

3. **ListBox Control:**

- **Definition:**

  - A ListBox control in ASP.NET is like a list where users can select one or more items. It's suitable when you want users to make multiple selections from a list.

- **Syntax:**

  ```html
  <asp:ListBox runat="server" ID="MyListBox" SelectionMode="Multiple">
      <asp:ListItem Text="Item 1" Value="1"></asp:ListItem>
      <asp:ListItem Text="Item 2" Value="2"></asp:ListItem>
      <asp:ListItem Text="Item 3" Value="3"></asp:ListItem>
  </asp:ListBox>
  ```

4. **CheckboxList Control:**

  - **Definition:**

    - A CheckboxList control in ASP.NET is like a group of checkboxes where users can select multiple options. It's suitable for scenarios where users can make multiple independent selections.

  - **Syntax:**

    ```html
    <asp:CheckBoxList runat="server" ID="MyCheckBoxList">
        <asp:ListItem Text="Option X" Value="X"></asp:ListItem>
        <asp:ListItem Text="Option Y" Value="Y"></asp:ListItem>
        <asp:ListItem Text="Option Z" Value="Z"></asp:ListItem>
    </asp:CheckBoxList>
    ```

5. **BulletedList Control:**

  - **Definition:**

    - A BulletedList control in ASP.NET is like a list where items are presented in a bulleted format. It's often used for displaying a simple, visually appealing list of items.

- **Syntax:**

  ```html
  <asp:BulletedList runat="server" ID="MyBulletedList">
    <asp:ListItem Text="Item Alpha"></asp:ListItem>
    <asp:ListItem Text="Item Beta"></asp:ListItem>
    <asp:ListItem Text="Item Gamma"></asp:ListItem>
  </asp:BulletedList>

**Differentiate Simple List Controls and Template List Controls ?**

1. **Rendering and Customization:**
   - **Simple List Controls:** These controls, like DropDownList or BulletedList, have fixed appearances. You can't change how each item looks very much.
   - **Template List Controls:** With controls like Repeater or GridView, you have more power. You can design how each item appears in detail.
2. **Handling Data:**
   - **Simple List Controls:** These work well with basic data setups. They're good when you just need to show a list.
   - **Template List Controls:** They handle more complex data scenarios. If your data is intricate, these controls let you display it in a customized way.
3. **Customizing Item Presentation:**
   - **Simple List Controls:** You can modify some aspects of each item but can't change the overall structure much.
  ```

- **Template List Controls:** These controls allow a lot of customization. You can create templates for various parts, giving you control over how everything looks.

4. **Performance Considerations:**
    - **Simple List Controls:** Generally, these are faster, especially with fewer items.
    - **Template List Controls:** Using templates might slow things down, especially with many items or complex templates. You need to be careful about performance.
5. **Best Uses:**
    - **Simple List Controls:** Use them when you just need a basic list, like a simple dropdown or a bulleted list.
    - **Template List Controls:** Use them when you want a lot of control over how your data looks. For instance, if you're showing data in a table and need each row to look different.

Write a program in ASP.Net to demonstrate the use of checkbox in a form ?

ASP.NET Web Form (aspx file):

```html
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="CheckboxDemo.aspx.cs"
Inherits="YourNamespace.CheckboxDemo" %>


<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">

<head runat="server">

  <title>Checkbox Demo</title>
```

```
</head>
<body>
    <form id="form1" runat="server">
        <h2>Checkbox Demo</h2>

        <asp:CheckBox runat="server" ID="MyCheckbox" Text="Check me" />
        <br />
        <asp:Button runat="server" Text="Submit" OnClick="SubmitButton_Click" />
    </form>
</body>
</html>
```

Code-Behind File (aspx.cs file):

```csharp
using System;

namespace YourNamespace
{
    public partial class CheckboxDemo : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            // Code that runs when the page is loaded
        }
```

```
    protected void SubmitButton_Click(object sender, EventArgs e)
    {
        // Check if the checkbox is checked
        if (MyCheckbox.Checked)
        {
            Response.Write("Checkbox is checked!");
        }
        else
        {
            Response.Write("Checkbox is unchecked.");
        }
    }
  }
}
```

In this example:

- The web form contains a checkbox (`MyCheckbox`) and a submit button.

- When the submit button is clicked, the `SubmitButton_Click` event handler is triggered.

- In the event handler, it checks whether the checkbox is checked (`MyCheckbox.Checked`) and displays a message accordingly.

Validation Controls below here:

Define Validation. Discuss properties and methods of validation control.

Validation in ASP.NET refers to the process of ensuring that user input meets specific criteria or rules before it is processed by the server. This helps in maintaining data integrity, improving user experience, and preventing potential issues caused by invalid or malicious input.

ASP.NET provides a set of validation controls that simplify the implementation of client-side and server-side validation logic. These controls can be added to web forms, making it easier to define rules and constraints for user input.

Properties of Validation Controls:

1. **ControlToValidate:**
    - Specifies the ID of the control to validate. The validation control checks the input of this control against the defined validation criteria.
2. **ErrorMessage:**
    - Sets the error message that is displayed when validation fails. This message helps users understand what went wrong with their input.
3. **InitialValue:**
    - Specifies the initial value against which the input is compared for validation. This is useful for controls like the **CompareValidator** that compare input to a fixed value.
4. **Display:**
    - Determines how the error message is displayed. Common values include "Static," "Dynamic," and "None," affecting whether the error message takes up space on the page when not visible.
5. **ForeColor and BorderColor:**
    - Control the color of the error message text and border, helping to make error messages more noticeable.

Methods of Validation Controls:

1. **Validate():**
    - This method is called to initiate the validation process. It checks the user input against the specified criteria and sets the IsValid property accordingly.
2. **Validate(string, string):**
    - An overloaded version of the **Validate()** method that accepts two string parameters. This allows you to pass custom values for validation and compare them to user input.
3. **IsValid:**
    - A boolean property that indicates whether the user input is valid according to the defined validation rules. If **IsValid** is true, the input is considered valid; otherwise, it's not.
4. **ValidateEmptyText():**
    - Used by certain validation controls to check whether the input is empty. This method is particularly relevant for controls like **RequiredFieldValidator**.

Common Validation Controls in ASP.NET:

1. **RequiredFieldValidator:**
    - Ensures that a required field is not left empty.
2. **RegularExpressionValidator:**
    - Validates input based on a specified regular expression pattern.
3. **RangeValidator:**
    - Checks whether the input falls within a specified numeric or date range.

4. **CompareValidator:**
    - Compares the input value with a constant value or another control's value.
5. **CustomValidator:**
    - Allows for custom validation logic by specifying client-side and server-side validation functions.

All types of validation controls:-

Sure, let's break down each of the mentioned validation controls in ASP.NET, including their definitions, syntax, and examples.

1. **RequiredFieldValidator:**

- **Definition:**

Ensures that a specific input field is not left empty. It's commonly used to mandate the completion of essential form fields, preventing users from submitting incomplete data.

- **Syntax:**

```html
<asp:RequiredFieldValidator runat="server" ControlToValidate="txtName" ErrorMessage="Name is required" />
```

- **Example:**

```html
<asp:TextBox runat="server" ID="txtName"></asp:TextBox>
<asp:RequiredFieldValidator runat="server" ControlToValidate="txtName" ErrorMessage="Name is required" />
```

2. **RangeValidator:**

- **Definition:**

Validates whether the input falls within a specified numeric or date range. This control is useful for enforcing constraints on values, such as age limits or acceptable date ranges.

- **Syntax:**

```html
<asp:RangeValidator runat="server" ControlToValidate="txtAge" Type="Integer" MinimumValue="18" MaximumValue="99" ErrorMessage="Age must be between 18 and 99" />
```

- **Example:**

```html
<asp:TextBox runat="server" ID="txtAge"></asp:TextBox>
<asp:RangeValidator runat="server" ControlToValidate="txtAge" Type="Integer" MinimumValue="18" MaximumValue="99" ErrorMessage="Age must be between 18 and 99" />
```

3. **CompareValidator:**

- **Definition:**

Compares the input value with a constant value or another control's value. It's employed to verify equality or inequality, commonly used in scenarios like password confirmation to ensure consistency.

- **Syntax:**

```html
<asp:CompareValidator runat="server" ControlToValidate="txtPassword" ControlToCompare="txtConfirmPassword" Operator="Equal" ErrorMessage="Passwords must match" />
```

- **Example:**

  ```html

  <asp:TextBox runat="server" ID="txtPassword"
TextMode="Password"></asp:TextBox>

  <asp:TextBox runat="server" ID="txtConfirmPassword"
TextMode="Password"></asp:TextBox>

  <asp:CompareValidator runat="server" ControlToValidate="txtPassword"
ControlToCompare="txtConfirmPassword" Operator="Equal"
ErrorMessage="Passwords must match" />

4. **RegularExpressionValidator:**

- **Definition:**

  Validates input based on a specified regular expression pattern. This control is valuable for enforcing specific formats, like email addresses or phone numbers, ensuring data conforms to defined patterns..

- **Syntax:**

  ```html

  <asp:RegularExpressionValidator runat="server" ControlToValidate="txtEmail"
ValidationExpression="\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*"
ErrorMessage="Invalid email format" />

- **Example:**

  ```html

  <asp:TextBox runat="server" ID="txtEmail"></asp:TextBox>

  <asp:RegularExpressionValidator runat="server" ControlToValidate="txtEmail"
ValidationExpression="\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*"
ErrorMessage="Invalid email format" />
  ```

5. **CustomValidator:**

- **Definition:**

Allows the implementation of custom validation logic using both client-side and server-side validation functions. Developers can define and enforce specialized validation rules beyond the capabilities of built-in validators.

- **Syntax:**

```html
<asp:CustomValidator runat="server" ControlToValidate="txtCustom" OnServerValidate="CustomValidation" ErrorMessage="Custom validation failed" />
```

- **Example:**

```html
<asp:TextBox runat="server" ID="txtCustom"></asp:TextBox>
<asp:CustomValidator runat="server" ControlToValidate="txtCustom" OnServerValidate="CustomValidation" ErrorMessage="Custom validation failed" />
```

```csharp
protected void CustomValidation(object source, ServerValidateEventArgs args)
{
    // Custom validation logic
    args.IsValid = (args.Value.Length >= 6);
}
```

6. **ValidationSummary:**

- **Definition:**

Displays a summary of all validation errors on the page. This control consolidates and presents error messages in a centralized location, making it convenient for users to identify and address multiple validation issues at once.

- **Syntax:**

```html
<asp:ValidationSummary runat="server" DisplayMode="BulletList" />
```

- **Example:**

```html
<asp:ValidationSummary runat="server" DisplayMode="BulletList" />
```

Custom Web Controls:

User Controls:

1. **Definition:**
    - User Controls in ASP.NET are encapsulated, reusable components that group together related markup and code. They are like mini web pages with a specific functionality or purpose.
2. **Syntax:**
    - A User Control is created by adding an ASCX file to the project. It typically includes both HTML markup and server-side code.

```
<%@ Control Language="C#" AutoEventWireup="true"
CodeBehind="MyUserControl.ascx.cs"
Inherits="YourNamespace.MyUserControl" %>

<div>
```

```
    <!-- Your markup and code here -->
</div>
```

3. **Example:**
    - An example User Control might be a navigation bar that you reuse across multiple pages.
4. **Purpose:**
    - User Controls are great for encapsulating and reusing pieces of UI or functionality. They enhance maintainability and make it easy to update common elements across different pages.

How User controls are loaded dynamically ?

1. **Create the User Control:**

- First, create your User Control (ASCX file) with the desired UI and functionality.

2. **Instantiate the User Control in Code-Behind:**

- In your ASP.NET page's code-behind file (like .aspx.cs), create an instance of the User Control using the `LoadControl` method.

```csharp
MyUserControlType myControl = (MyUserControlType)LoadControl("MyUserControl.ascx");
```

3. **Set Properties (Optional):**

- If your User Control has properties that need to be set dynamically, you can do so after instantiation.

```csharp
myControl.PropertyName = "Value";
```

4. **Add to Page Controls Collection:**

   - Add the instantiated User Control to the Controls collection of a container on your page (e.g., a Panel or a Placeholder).

   ```csharp
   myContainer.Controls.Add(myControl);
   ```

5. **Ensure Unique IDs (Optional):**

   - If your User Control relies on unique IDs (for example, if it contains controls with runat="server"), make sure to assign unique IDs dynamically.

   ```csharp
   myControl.ID = "DynamicID";
   ```

6. **Handle Events (Optional):**

   - If your User Control raises events, you can handle them in the code-behind of your page after adding the control.

   ```csharp
   myControl.SomeEvent += MyEventHandlerMethod;
   ```

7. **Ensure Postback Handling (Optional):**

   - If your User Control triggers postbacks, ensure that you handle those postbacks in the Page_Load or other appropriate events of your page.

   ```csharp
   protected void Page_Load(object sender, EventArgs e)
   {
   ```

```
        // Handle postbacks or other logic
    }
```

Here's a simple example illustrating these steps:

```csharp
protected void Page_Load(object sender, EventArgs e)
{
    // Step 2: Instantiate User Control
    MyUserControlType myControl =
(MyUserControlType)LoadControl("MyUserControl.ascx");

    // Step 3: Set Properties (Optional)
    myControl.PropertyName = "DynamicValue";

    // Step 4: Add to Page Controls Collection
    myContainer.Controls.Add(myControl);

    // Step 5: Ensure Unique IDs (Optional)
    myControl.ID = "DynamicID";

    // Step 6: Handle Events (Optional)
    myControl.SomeEvent += MyEventHandlerMethod;

    // Step 7: Ensure Postback Handling (Optional)
```

```
    // Handle postbacks or other logic
}
```

Illustrate how to create a basic form of user control for asp.net ?

1. **Create the ASCX File:**

   - In your Visual Studio project, right-click on the project in Solution Explorer.

   - Choose "Add" -> "Web Form..." and name it, for example, `MyUserControl.ascx`.

2. **Design the User Control:**

   - Open the newly created ASCX file.

   - Design your User Control by adding HTML elements and controls. For simplicity, let's create a control with a label and a button.

    ```html
    <%@ Control Language="C#" AutoEventWireup="true" CodeBehind="MyUserControl.ascx.cs" Inherits="YourNamespace.MyUserControl" %>

    <div>
        <asp:Label runat="server" ID="lblMessage" Text="Hello from User Control"></asp:Label>
        <asp:Button runat="server" ID="btnClickMe" Text="Click Me" OnClick="btnClickMe_Click" />
    </div>
```

3. **Code-Behind for the User Control:**

- Add the code-behind file by right-clicking on the ASCX file in Solution Explorer and choosing "View Code."

- Add the necessary using statements and implement any logic you need. In this example, we handle the button click event to change the label text.

```csharp
using System;
using System.Web.UI;

public partial class MyUserControl : UserControl
{
    protected void btnClickMe_Click(object sender, EventArgs e)
    {
        lblMessage.Text = "Button Clicked!";
    }
}
```

4. **Use the User Control in a Web Form:**

- Open a web form (ASPX file) where you want to use the User Control.

- Drag and drop the User Control onto the form, or add it programmatically. Ensure you register the control at the top of your ASPX file.

```html
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="MyWebForm.aspx.cs" Inherits="YourNamespace.MyWebForm" %>

<%@ Register Src="~/MyUserControl.ascx" TagName="MyUserControl" TagPrefix="uc" %>
```

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>My Web Form</title>
</head>
<body>
    <form id="form1" runat="server">
        <uc:MyUserControl runat="server" ID="myControl" />
    </form>
</body>
</html>
```

5. **Run Your Application:**

- Build and run your application. When you click the "Click Me" button in the User Control, the label text should change.

Custom Controls:

1. **Definition:**
    - Custom Controls in ASP.NET are more advanced, self-contained components that encapsulate both UI and behavior. They can be created by inheriting from existing controls or building from scratch.

2. **Syntax:**
    - Custom Controls are typically created by creating a class that inherits from **WebControl** or **Control**, and then overriding or adding necessary methods and properties.

      ```
      public class MyCustomControl : WebControl
      {
          // Custom control implementation
      }
      ```

3. **Example:**
    - A custom calendar control with specific styling and behavior could be an example of a custom control.
4. **Purpose:**
    - Custom Controls offer more control and flexibility than User Controls. They are suitable for scenarios where you need highly customized, reusable components with complex behavior.

How user control is different from custom controls ?

- **Reusability:**
    - **User Controls:** Primarily used for encapsulating UI elements.
    - **Custom Controls:** Used for creating reusable, self-contained components with custom behavior.
- **Development Approach:**
    - **User Controls:** Easier to create, suitable for simpler components.
    - **Custom Controls:** Require more advanced development skills, suitable for complex and highly customized components.
- **Flexibility:**

    - **User Controls:** Limited compared to Custom Controls.
    - **Custom Controls:** Highly flexible, allowing complete customization and extension.
- **Complexity:**
    - **User Controls:** Good for simple components with basic behavior.
    - **Custom Controls:** Ideal for complex components needing unique features.
- **Examples:**
    - **User Controls:** Navigation bars, login forms, simple date pickers.
    - **Custom Controls:** Specialized charts, complex grids, unique data displays.