Displays CodeBehind Reference

Add logic to displays with CodeBehind.

How-to Guides **Pillars** **Operator** **Displays** CodeBehind | How-To Guide | Reference

---

Displays CodeBehind (Reference) runs in each client and adds pre-defined event methods and custom functions to a display, in VB.NET, C#, or JavaScript (HTML5-only displays).

CodeBehind enables:

- Display lifecycle event handling
- Custom methods for user interactions
- Mouse and keyboard event processing
- Custom Properties for local variables
- Integration with Display namespace
- Automatic language conversion (VB.NET C#)

Access via **Displays** **Draw** **CodeBehind** tab.

Language Support
Pre-defined Methods
Custom Properties
Performance Considerations
Display Namespace Access
Multiple Popups Management
Code Editor Features
External References
Best Practices
Common Patterns
Troubleshooting

Language Support

Portable and WPF Displays

- **VB.NET** - Visual Basic syntax
- **C#** - C-Sharp syntax
- Auto-conversion between languages
- Note: C# is more restrictive on type conversions

HTML5-Only Displays

- **JavaScript** tab appears alongside VB.NET/C#
- Both execute side-by-side in web runtime
- Legacy approach (WebAssembly preferred)
- Similar method structure to .NET languages

Pre-defined Methods

Every display includes these event handlers:

Method	Triggered When	Purpose
DisplayOpening()	Display starts opening	Initialize variables, setup
DisplayIsOpen()	While display remains open	Continuous updates, monitoring
DisplayClosing()	Display starts closing	Cleanup, save state
DialogOnOK()	OK button pressed (dialogs only)	Validate input, return 1 to close, 0 to prevent

Example Structure

csharp

```
public void DisplayOpening()
{
    // Initialize display
}

public void DisplayIsOpen()
{
    // Called continuously while open
    // Use sparingly for performance
}

public void DisplayClosing()
{
    // Cleanup before close
}

public int DialogOnOK()
{
    // Dialogs only - validate and return
    if (ValidateInput())
        return 1; // Allow close
    else
        return 0; // Prevent close
}
```

Custom Properties

Local variables exclusive to the display instance:

Benefits

- Reduce communication points
- Store display-specific state
- Access from both display and CodeBehind
- Link to tags for extended functionality

Limitations

- Data doesn't persist between pages
- Specific to display instance
- Lost when display closes

Runtime Methods

csharp

```
// Set property value
SetCustomPropertyValue("PropertyName", value);

// Get property value
var value = GetCustomPropertyValue("PropertyName");

// Get all as string
string allProps = GetCustomPropertiesAsString();

// Clear all properties
ClearCustomProperties();
```

See Custom Properties for detailed documentation.

Performance Considerations

UI Blocking

- CodeBehind runs on client thread
- Display locks during execution
- User cannot interact while code runs

Solutions

1. **Use Async Functions**

csharp

```csharp
public async void DisplayOpening()
{
    await Task.Run(() => LongRunningOperation());
}
```

1. **Offload to Scripts**
   - Use Script Tasks for heavy processing
   - Run on server or background thread
   - Prevents UI freezing
2. **Optimize Code**
   - Minimize DisplayIsOpen() logic
   - Cache calculations
   - Avoid loops in event handlers

Display Namespace Access

CurrentDisplay Object

`CurrentDisplay` is available in C# and VB.NET CodeBehind. It refers to the display instance running the code, so each popup instance works on itself.

Member	Purpose
`GetThis()`	The display element, for size and layout, for example `GetThis().ActualWidth`.
`GetControl("uid")`	An element of the display, found by its Uid, not its Name. Cast the result to the control type, for example `as TTrendChart`.
`GetLabel("name")`	Reads a parameter the display was opened with. See Popup Parameters below.
`SetLabel("name", value)`	Writes a value to the tag a parameter names.
`SetPosition(left, top)`	Moves the display.
`Close()`	Closes this display instance.

```
// Size of this display
double width = this.CurrentDisplay.GetThis().ActualWidth;
double height = this.CurrentDisplay.GetThis().ActualHeight;

// Close this display instance
this.CurrentDisplay.Close();
```

Popup Parameters: GetLabel and SetLabel

A display opens with optional parameters: `name=value` pairs separated by semicolons. Its CodeBehind reads them with `GetLabel` and writes through them with `SetLabel`, so one display serves many tags.

```
// Opening the popup, in another display or in a button Action:
// @Display.MotorDetail.NewPopup("Motor=Tag.Pump1;Mode=2;Title='Pump 1'", 100, 100);

// In the CodeBehind of MotorDetail, called from a button:
public void ApplySettings()
{
    // Value of the tag passed as Motor (here, Tag.Pump1)
    object motorValue = this.CurrentDisplay.GetLabel("Motor");

    // Number and text parameters
    int mode = Convert.ToInt32(this.CurrentDisplay.GetLabel("Mode"));
    string title = Convert.ToString(this.CurrentDisplay.GetLabel("Title"));

    // Writes 0 to the tag passed as Motor (here, Tag.Pump1)
    this.CurrentDisplay.SetLabel("Motor", 0);
}
```

- **GetLabel** returns the value of the parameter: the value of a tag, a number, or a text. Numbers come back as `double`. To read a member of a tag passed as a parameter, add it after a dot, for example `GetLabel("Motor.Quality")`. Convert the result to the type you need, as in the example.
- **SetLabel** writes the value to the tag the parameter names. The parameter keeps pointing to the same tag. Its return value does not confirm the write: read the tag back when the result matters.
- **Parameter values:** a value starting with a digit, `+` or `-` is a number. A value in single quotes is a text, and it cannot contain `;` or =. Any other value is a tag or object name.
- **Names** are not case-sensitive, and the # prefix is optional in the open call and in GetLabel. When a name appears twice in the open call, the first value wins.
- A name the display was not opened with returns an empty text. A parameter whose value names a tag or object missing from the solution makes GetLabel and SetLabel throw an exception.
- In HTML5 displays, GetLabel returns the value the browser holds at that moment and does not wait for the server. Call it from a button or a later event, not while the display opens.
- GetLabel and SetLabel read the parameters of the display. Inside a symbol, use symbol parameters (see Symbols Control Reference).

The parameters come from `@Display.<Name>.NewPopup(...)`, `@Client.UI.NewPopup("Name", ...)` and `@Client.UI.OpenDisplay ("Name", ...)`. When a display contains symbols, a parameter with the same name as a symbol label also fills that label; see Displays Multiple Popups Example.

Opening Displays

Multiple methods available:

Method	Auto-Updates on Rename	Usage
`@Display.PageName.Open()`	Yes	Preferred for pages
`@Display.PageName.NewPopup()`	Yes	Create popup instance
`@Client.OpenDisplay("PageName")`	No	String-based navigation
`@Client.NewPopup("PageName")`	No	String-based popup

Mouse Wheel Example

csharp

```csharp
public void DisplayIsOpen()
{
    this.CurrentDisplay.GetThis().PreviewMouseWheel += PreviewMouseWheelChanged;
}

private void PreviewMouseWheelChanged(object sender, MouseWheelEventArgs e)
{
    e.Handled = true;

    if (e.Delta > 0)
        @Display.MainPage.ZoomLevel += 0.1;
    else if (e.Delta < 0)
        @Display.MainPage.ZoomLevel -= 0.1;
}
```

Multiple Popups Management

Key Considerations

- Each call to `NewPopup` opens a new, independent instance of the display. A display whose mode is Page does not open as a popup.
- `NewPopup` returns an empty string. Pass the data each instance needs as parameters, and read them in the popup with `GetLabel`.
- Each instance closes itself with `this.CurrentDisplay.Close()`.
- Keep the number of open instances in proportion to the memory of the client.

Common Pattern

```
// In the calling display: one popup instance per item
public void OpenDetail(string tagName)
{
    @Display.DetailPopup.NewPopup("Item=" + tagName);
}

// In the CodeBehind of DetailPopup, called from a button
public void Acknowledge()
{
    this.CurrentDisplay.SetLabel("Item", 0);
    this.CurrentDisplay.Close();
}
```

See Displays Multiple Popups Example for more scenarios.

Code Editor Features

- Syntax highlighting
- IntelliSense support
- Error checking
- Find/Replace
- Code folding
- Breakpoint debugging (development mode)

See Code Editor for complete feature documentation.

External References

For using custom DLLs and assemblies:

- Add references via Scripts References
- Available to all CodeBehind
- Shared with Script modules

See Scripts References for configuration.

Best Practices

1. **Keep Methods Light** - Avoid heavy processing in event handlers
2. **Use Async When Needed** - Prevent UI blocking
3. **Clean Up Resources** - Use DisplayClosing() for cleanup
4. **Cache References** - Store frequently used objects
5. **Handle Errors** - Try-catch in event handlers
6. **Document Code** - Comment complex logic
7. **Test Performance** - Monitor UI responsiveness

Common Patterns

Initialize Once

csharp

```
private bool initialized = false;

public void DisplayIsOpen()
{
    if (!initialized)
    {
        InitializeDisplay();
        initialized = true;
    }
}
```

Periodic Updates

csharp

```
private DateTime lastUpdate = DateTime.MinValue;

public void DisplayIsOpen()
{
    if (DateTime.Now - lastUpdate > TimeSpan.FromSeconds(1))
    {
        UpdateDisplay();
        lastUpdate = DateTime.Now;
    }
}
```

Troubleshooting

Display Freezing:

- Check for blocking code in events
- Use async methods
- Move heavy logic to Scripts

Methods Not Firing:

- Verify method signatures
- Check display is correct type (Page/Popup/Dialog)
- Review build errors

Variable Scope Issues:

- Use Custom Properties for persistence
- Class-level variables for display lifetime
- Static variables for cross-display data

In this section...
