GoDiagram Web for ASP.NET Web Forms Introduction

GoDiagram™ Web for Microsoft® ASP.NET Web Forms ("Go") is a .NET class library containing a set of Web Forms controls for easily building interactive diagrams in ASP.NET-based web applications.

GoDiagram Web shares much of the design and implementation with GoDiagram Win, which you use to build Windows Forms applications. The User Guide provides details about Go, most of which apply to both products. You will need to read this before you can really make good use of Go.

GoDiagram Web applications can be viewed in many different kinds of browsers since they consist of images and some JavaScript. They can easily support AJAX-style reloading of images and data without doing ASP.NET postbacks – and even without the partial postbacks of Microsoft AJAX.

GoDiagram Web controls run on the ASP.NET server. They do not run on the browser's machine. Each click on the diagram results in a round trip to the server to update the <IMG> tag which represents the diagram. We do, however, offer two products that update in the client:

Silverlight and HTML5 Canvas

Note that we have two completely separate products **that we recommend over GoDiagram Web.**

- GoXam for Silverlight http://www.nwoods.com/components/silverlight-wpf/goxam-overview.htm
- GoJS for HTML5 Canvas http://www.nwoods.com/components/canvas/gojs-overview.htm

If your goal is to create a highly interactive graphical website application, we recommend you look at these products.

Installation kit

The installation kit is a Windows Installer file for ASP.NET. It serves as both an evaluation kit as well as the full binary product kit—the only difference is whether you have purchased and installed a full binary development license.

The installation kits are Windows Installer files

- GoWeb5003.msi for ASP.NET version 3.5 / VS2008
- GoWeb5004.msi for ASP.NET version 4.0 / VS2010 (includes ASP.NET 3.5 DLLs)
- GoWeb50045.msi for ASP.NET version 4.5 / VS2012 (includes ASP.NET 3.5 & .NET 4.0 DLLs)

Before you install Go, you should already have installed the .NET Framework SDK and ASP.NET.

An installation kit for .NET 1.1 and ASP.NET 1.1 is no longer available. For .NET 2.0, use version 4.2.

GoDiagram Win is available in a different installation kit.

GoDiagram for ASP.NET Web Forms Files

Go consists of five assemblies:

- **`Northwoods.GoWeb.dll`**, holding the **Northwoods.GoWeb** namespace
- **`Northwoods.GoWeb.Layout.dll`**, holding the **Northwoods.GoWeb.Layout** namespace
- **`Northwoods.GoWeb.Instruments.dll`**, holding the **Northwoods.GoWeb.Instruments** namespace
- **`Northwoods.GoWeb.Xml.dll`**, holding the **Northwoods.GoWeb.Xml** namespace
- **`Northwoods.GoWeb.Svg.dll`**, holding the **Northwoods.GoWeb.Svg** namespace
- **`Northwoods.GoWeb.Pdf.dll`**, holding the **Northwoods.GoWeb.Pdf** namespace

The five assemblies are in the **`lib`** subdirectory of the Go installation. They only depend on the Microsoft .NET System, Web, and Drawing assemblies. They do not include any unmanaged code and do not require any particular permissions beyond what any ASP.NET Web Forms controls would need.

Detailed documentation on the types in these libraries is provided in the **`GoWeb`** Visual Studio help file. Other documentation is in the **`docs`** subdirectory of the Go installation. You may find it instructive to see a listing of the differences between Windows Forms and ASP.NET Web Forms; this list is maintained in **`GoWinWebDiffs.doc`**.

It also places some example code in the **`Samples`** and **`SamplesVB`** subdirectories. You can open a subdirectory as a Web Site in Visual Studio, and compile and debug them individually. The start page for each project/website is always named `WebForm1.aspx`.

Initial Experiences

If you haven't already run the sample applications, just to get a feel for what Go can do, please try them. You can also try them at: http://www.nwoods.com/components/dotnet/webforms-samples.htm

Reading the source code for the applications will really help you understand how easily you can implement different kinds of features. Remember that these are sample applications. Sometimes functionality is implemented just for the sake of demonstration—no real application would want to have that combination of features, or so many different ways to achieve the same kind of functionality. Furthermore to simplify the samples, we use very simple buttons instead of menus or other fancier controls.

If you have certain features you know you want to implement, but are not sure how to do so, it might help to read the Frequently Asked Questions (FAQ) document, **`GoDiagramFAQ.htm`**, in the **`docs`** subdirectory. Another source of inspiration can be the GoDiagram forum at http://www.nwoods.com/forum.

It might also help to read the entire User Guide, because it discusses much of the programming model embodied in Go. If you don't have that much time, at least read the *Go Concepts* chapter in the User Guide.

Customizing Visual Studio

If you are using Visual Studio, you'll want to customize your Toolbox to include the three controls provided by the **`Northwoods.GoWeb.dll`** assembly.

1. Start up Visual Studio

2. View the Toolbox, if it isn't already visible.
3. Open up the tab that you want to hold the Go controls. You may want to create a new tab, or you may want to use an existing tab of WebForms controls.
4. Context click (right-mouse click) in the desired toolbox tab window. Choose the "Add/Remove Items" or "Choose Items…" context menu command. The Toolbox customization dialog will appear.
5. Select the ".NET Framework Components" tab.
6. Scroll down until you find the **GoView**, **GoPalette**, **GoOverview**, and **GoPrintView** controls, in the GoWeb assembly. If you do not see these controls, you may need to click the "Browse…" button to open the assembly in the **`lib`** subdirectory of the Go installation. Make sure all four controls have check marks by them.
7. Click OK for this dialog. The three controls should appear in your toolbox.

You can now drag any of the controls onto your Web Form that you are designing. The Properties window will let you specify many of the properties and events to customize the appearance and behavior of the selected view.

Server Requirements

Go requires the use of session state to be able to produce images representing the view when referenced by HTML `IMG` tags. The **GoView**, its **GoDocument** and its **GoObjects** are all serialized as part of the session state.

If a session cannot be maintained, **GoView** state will be lost. This will happen when a session times out—the **GoView.SessionStarted** event is raised when the state is requested again.

Browser Requirements

The HTML generated for **GoView** depends on JavaScript support on the client. If JavaScript scripting is disabled in the browser, the user should be able to see the view but will not be able to interact with it.

Currently **GoView** supports rendering for a number of different browsers with varying levels of functionality. We have tested Internet Explorer 8 and 9 and Firefox 14+, Apple Safari 5+ and Google Chrome 21+ running on various versions of Windows (Vista, Windows 7 and Windows 8). The functionality is most complete on Internet Explorer.

Your pages must be HTML 5 compliant and not require quirks mode. The DOCTYPE tag should be the one recommended for HTML 5:

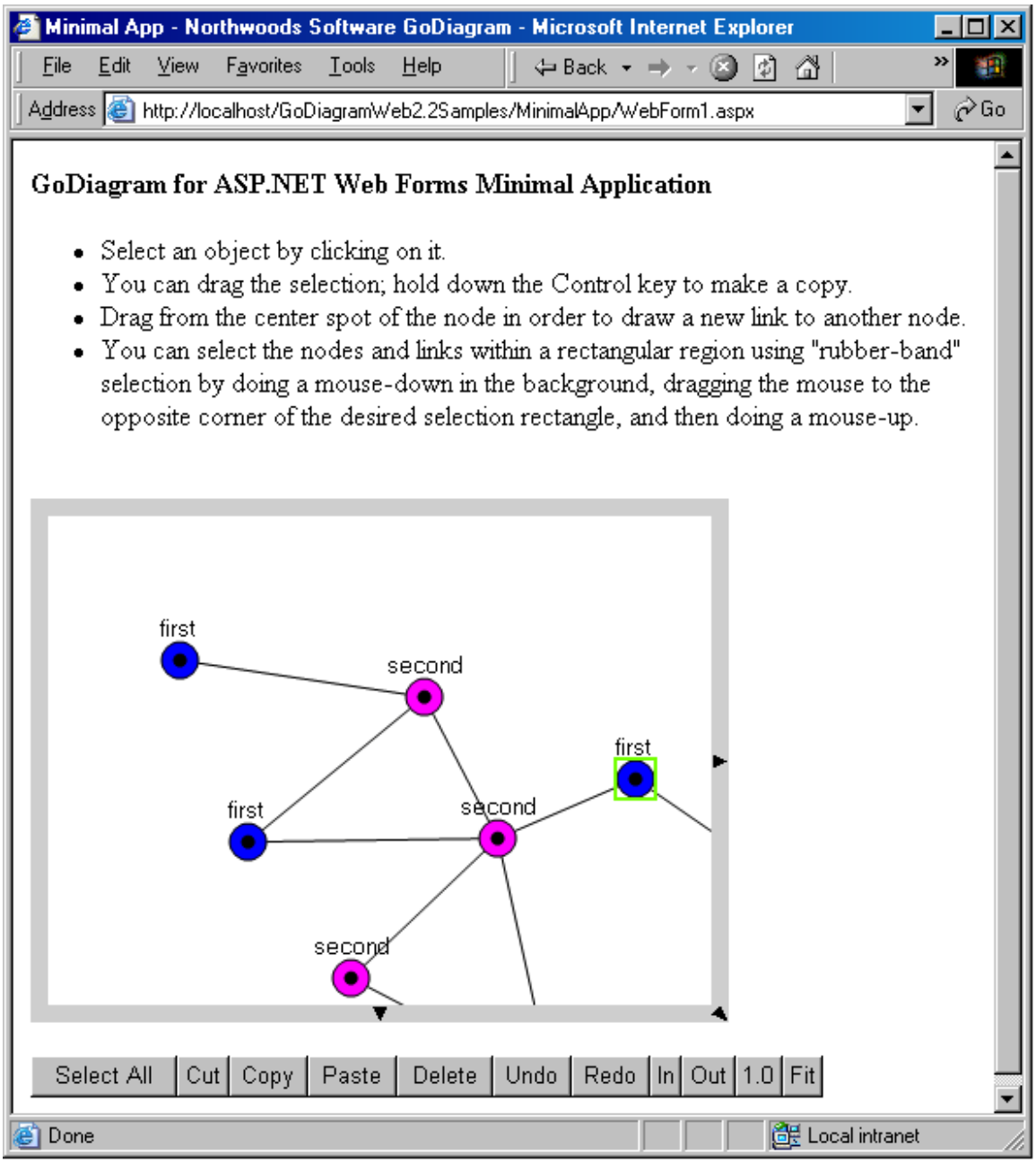
```
<!DOCTYPE HTML>
```

The examples are also dependent on cookies.

Appearance

A **GoView** will appear as an image in an HTML page. The image includes a margin where scrolling buttons are displayed if the view can be scrolled in that direction. You can set properties on a **GoView** to change the appearance of the scroll buttons and appearance and size of the scrolling margin.

Of course, you can set many other useful properties and handle many events on a **GoView**, in common with the Windows Forms version. Read the User Guide or API reference for more details.



Rendering

An HTML page cannot have images embedded in them—they must refer to the result of a separate request. Thus the **GoView** control in Go renders as HTML that includes an `IMG` tag.

For example, a simple commonplace ASPX use of a **GoView**:

```
<GoWeb:GoView id="MyView" runat="Server" Height="300" Width="400"
  NoPost="true" ImagePage="GoWebImage.axd" ScriptFile="GoWeb.js" CssFile="none">
</GoWeb:GoView>
```

might render into HTML such as:

```
<span style="background-color:White;margin:0;">
<img src="GoWebImage.axd?GoView=d337af23d9d945a79a8f87b6dcef6aa5"
  id="Img1" name="MyView" width="400" height="300"
  onmousedown="goMouseDown(event,'MyView')" onmouseup="goMouseUp(event,'MyView')"
  ondblclick="goDblClick(event,'MyView')" onmousemove="goMouseMove(event,'MyView')"
  oncontextmenu="return false"
onload="goInit('MyView','MyView','d337af23d9d945a79a8f87b6dcef6aa5',true,'GoWebImage.a
xd','',goview_MyView,false)"
  onerror="goLoadError('MyView')"
  tabindex="1" title="" alt="? loading... GoWebImage.axd"
  galleryimg="no" />
</span>
```

The `SRC` attribute is a reference to a GoDiagram-defined ASP.NET HTTP handler, **GoWebImageHandler**, an **IHttpHandler**, that must be declared in the ASP.NET configuration. You can do this by adding (or merging) the following lines into your application's `Web.config` file:

For IIS prior to version 7 (and for the Visual Studio development server):

In the <system.web> section of your web.config file:

```
<httpHandlers>
  <add verb="GET" path="GoWebImage.axd"
type="Northwoods.GoWeb.GoWebImageHandler"/>
</httpHandlers>
```

For IIS 7:

In the to the <configuration> section:

```
<system.webServer>
    <handlers>
      <add verb="GET" name="GoWebHandler" path="GoWebImage.axd"
         type="Northwoods.GoWeb.GoWebImageHandler"/>
    </handlers>
</system.webServer>
```

and change the pool to ASP.NET 4.0.

Be sure that any configured HTTP handler for `*.axd` does not take precedence over this one; you can use a different name and file extension if it is more convenient. In IIS7, you will need to either update the integrated `<system.webServer> <handlers>` configuration section, or you will need to run your web app in Classic ASP.NET integration mode by moving it to the Classic ASP.NET Application Pool.

You can also control the format of the image that is generated by specifying the **GoView.ImageFormat** property. This defaults to **ImageFormat.Png**, which should work with nearly all browsers, but you may wish to use **ImageFormat.Gif**, which provides sharp images but with limited colors, or **ImageFormat.Jpeg**, that provides fuzzier images but should be supported for display by all browsers.

GoWebImageHandler produces an image by first getting a bitmap of a view, as identified by its document-wide unique ID. The ID is used as a key in the GoView.GetSessionViewsTable hashtable to get the saved GoView that should be displayed. The ID string is produced by the GoView.MakeSessionViewID method. The GoView.GetBitmap method paints any scrolling margin and draws any scroll buttons, and then calls PaintView to render all of the objects that are visible in the view.

Rendering also produces definitions of JavaScript functions such as **goMouseDown** as the event handler for the `onMouseDown` event. This code is generated by the **GoView.RenderScript** method.

Client-side Event Handling

A very thin layer of JavaScript code runs on the browser in order to handle mouse events that can be summarized and then passed back to the **GoView** via a postback. The appropriate `onMouseDown`, `onMouseUp`, `onDblClick`, and `onMouseMove` event handlers are automatically defined as attributes on the `IMG` tag if **GoView.Enabled** is true.

A web-server request only occurs upon a mouse up or a mouse double-click event. Just passing the mouse over a view or performing a mouse-down-and-drag (before the mouse up) is completely handled by the JavaScript running on the browser.

The predefined JavaScript can also handle key press events. However, the `IMG` tag cannot handle `onKeyDown`. Instead, if you want keystroke commands to be passed onto your **GoView**, you will need to add the event handler declaration on the HTML `BODY`:

```
<body onkeydown="goKeyDown(event, 'MyView')">
```

In doing this, you are limiting the user to passing the key press events to just one view. Often there can be conflicts and confusion with the keys that are handled by the browser, so handling keystrokes is not common.

GoView implements the **IPostBackEventHandler** interface. **GoView** supports a number of predefined postback event handler query commands. This facility is convenient when you want to define some buttons that perform standard operations on a view. For example, this is the definition of an HTML button that performs a copy on the current selection of the view named “MyView”:

```
<button id="Button5" onclick="goAction('copy','MyView')" type="button">
Copy</button>
```

The `goAction` JavaScript function is defined, along with other similar functions, as follows:

```
function goAction(act, id) {
  goPost(id, 'act=' + act);
}
```

The `goPost` JavaScript function is defined to either do a normal postback, by calling the ASP.NET-defined `__doPostBack` function, or to just execute commands on that one view and reload the resulting image. The latter choice is taken when the **GoView.NoPost** property is true, resulting in an AJAX-style web application. The advantage of avoiding a full postback is that the whole page does not need to be regenerated. However, if any other part of the page needs to be updated, for example by changing any

other controls or text, a standard postback must be done, or you need to implement all such updating with dynamic HTML in client-side JavaScript. One disadvantage of the "NoPost" method is that additional initialization may be required for the view, as described in the section below titled "Session State for NoPost GoViews".

The default value for **GoView.NoPost** is true, which causes mouse events in the "view" to reload the image without regenerating and reloading the whole page. You can also control whether a post or just an image reload occurs dynamically in JavaScript code running in the browser. Just set the **goNoPost** property on the IMG element that was rendered by the **GoView**. The initial value of **goNoPost** is provided by **GoView.NoPost**.

When scripting is disabled on the client browser, there is not much the user will be able to do with the view.

Server-side Event Handling

These are the kinds of query strings **GoView** handles by default on a postback or image reload, in the **GoView.RaisePostBackEvent** method:

- *Actions*. These take no other arguments.
  - `act=cut` (call **GoView.EditCut()**)
  - `act=copy` (call **GoView.EditCopy()**)
  - `act=paste` (call **GoView.EditPaste()**)
  - `act=delete` (call **GoView.EditDelete()**)
  - `act=selectall` (call **GoView.SelectAll()**)
  - `act=undo` (call **GoView.Undo()**)
  - `act=redo` (call **GoView.Redo()**)
- *Scrolling/panning*. The scrolling can be by page or by line, by amounts in both X and Y directions. This calls either **GoView.ScrollPage** or **GoView.ScrollLine**. Examples:
  - `scroll=page&dx=1&dy=-1` (towards the top-right)
  - `scroll=line&dx=-1&dy=0` (towards the left)

  You can also set the position of the view in the document to an absolute position, rather than modifying the **GoView.DocPosition** by incrementing/decrementing X and/or Y.
  - `position&x=100&y=200` (set **GoView.DocPosition** to 100,200)
  - `position&x=100` (set **GoView.DocPosition.X** to 100 but leave **Y** unchanged)
- *Zooming/rescaling*. There are three named commands, plus the ability to set the **GoView.DocScale** to an absolute value. Examples:
  - `zoom=in`
  - `zoom=out`
  - `zoom=fit`
  - `zoom=1.23`
- *Resizing*. The size of the view can be changed by amounts in both X and Y directions. The resize parameter specifies the step, which is multiplied with the dx and dy values to give the number of units to change size. The minimum size is 30. The units are always pixels. Examples:
  - `resize=10&dx=1&dy=-1` (wider and shorter)
  - `resize=10&dx=-1&dy=0` (narrower; height stays the same)

  You can also specify an absolute size in pixels:
  - `size&width=500&height=400` (set **GoView.Width** to 500 pixels and set **GoView.Height** to 400 pixels)

    - `size&width=500` (set **GoView.Width** to 500 pixels but leave **GoView.Height** unchanged)
- *Moving the Selection.* You can move the currently selected objects, by amounts in both X and Y directions. The movesel parameter specifies the step, which is multiplied with the dx and dy values to give the distance to move the objects. This calls **GoView.MoveSelection**. Example:
    - `movesel=1&dx=10&dy=0` (towards the right)
- *Copying the Selection.* You can copy and shift the currently selected objects, by amounts in both X and Y directions. The copysel parameter specifies the step, which is multiplied with the dx and dy values to give the distance to shift the copied objects. This calls **GoView.CopySelection**. Example:
    - `copysel=1&dx=10&dy=10` (towards the right and down)
- *Key press*. This includes the key code and any `ctrl`, `shift` or `alt`. Example:
    - `key=65&ctrl=1` (the user typed a Ctrl-A)
- *Mouse gesture*. This includes the mouse down and mouse up positions, along with any `ctrl`, `shift` or `alt` modifier, any `left`, `right`, or `middle` mouse button indication, and whether it is a double-click (the mouse down and mouse up positions must also be very close to each other). Examples:
    - `downx=342&downy=454&upx=374&upy=478&left=1`
    - `downx=342&downy=454&upx=342&upy=454&dblclick=1`
- *Requests*. This predefined case just calls **GoViewDataRenderer.HandleClientRequest**, so that you can easily implement your own actions. Example:
    - `request&color=fuschia`

The standard ASP.NET event handling still occurs, of course. For example, the ASP.NET button:

```
<asp:button id="Button10" OnClick="EditInsert" Runat="Server"
Text="Insert"></asp:button>
```

when clicked, will result in a call on the server to this method on your page, which might be defined as follows:

```
  void EditInsert(Object sender, EventArgs evt) {
    if (!MyPalette.Selection.IsEmpty) {
      MyView.StartTransaction();
      GoObject obj = MyPalette.Selection.Primary;
      MyView.Document.AddCopy(obj, MyView.DocExtentCenter);
      MyView.FinishTransaction("inserted from palette");
    }
  }
```

Furthermore, event handling on **GoView** can be used to perform useful work. For example, you can define **GoView.ObjectGotSelection** and **GoView.ObjectLostSelection** handlers to modify the appearance of the page:

```
  protected Northwoods.GoWeb.GoView MyView;   // Page variables
  protected Northwoods.GoWeb.GoPalette MyPalette;
  protected System.Web.UI.WebControls.Panel LabelPanel;
  protected System.Web.UI.WebControls.TextBox LabelTextBox;
  protected System.Web.UI.WebControls.Button SetLabelButton;

  private void InitializeComponent() {
```

```
    MyView.ObjectLostSelection +=
              new GoSelectionEventHandler(MyView_ObjectLostSelection);
    MyView.ObjectGotSelection +=
              new GoSelectionEventHandler(MyView_ObjectGotSelection);
    . . .
  }

  private void MyView_ObjectGotSelection(object sender,
                                         GoSelectionEventArgs e) {
    PersonNode pnode = MyView.Selection.Primary as PersonNode;
    if (pnode != null) {
      LabelPanel.Visible = true;
      LabelTextBox.Text = pnode.Text + ": " + pnode.ToolTipText;
    } else {
      LabelPanel.Visible = false;
    }
  }

  private void MyView_ObjectLostSelection(object sender,
                                          GoSelectionEventArgs e) {
    MyView_ObjectGotSelection(sender, e);
  }
```

Sessions and Initialization

This discussion assumes you are already very familiar with the life cycle for ASP.NET pages and web controls. Search for "Control Execution Lifecycle" in the ASP.NET 1.1 documentation, or for "ASP.NET Page Life Cycle" in the ASP.NET 2.0 documentation.

The state of a **GoView** is maintained between page requests by using session state rather than by using ASP.NET view state. A **Hashtable** mapping **GoView** identifiers to **GoView**s is kept in the **Session**. Maintaining the **GoView** through the session is also essential for the `IMG` tag reference to be able to find the view for generating an image to be streamed to the browser.

Using ASP.NET view state as many simple web controls do would be extremely inefficient because that would entail serializing into the HTML page all of the state of the **GoView**, its **GoSelection**, and its **GoDocument**, including the document's layers of **GoObject**s. That would consume considerable time and communication bandwidth. Serializing these objects in the **Session** will be much faster, although doing so does increase the server-side memory requirements for your web application.

View state is only used to remember the unique identifier for the view in the session state. This unique identifier is assigned when each view is loaded not during a postback. If you set **EnableViewState** to false, the unique identifier is just the **UniqueID** (but with modified syntax, as the value of **GoView.SafeID**).

Using session state does have the disadvantage that the state is lost when the session times out. The **GoView.SessionStarted** event is raised whenever the view is not found saved in the session state. You should always implement a **SessionStarted** event handler to initialize your **GoView** and in particular the view's **GoDocument**. For example, when your page is initialized, you will want to establish event handlers for all of your **GoView**s:

```
  private void InitializeComponent() {
    MyView.ObjectLostSelection +=
```

```
        new GoSelectionEventHandler(MyView_ObjectLostSelection);
    MyView.SelectionMoved +=
        new System.EventHandler(MyView_SelectionMoved);
    MyView.BackgroundSingleClicked +=
        new GoInputEventHandler(MyView_BackgroundSingleClicked);
    MyView.ObjectGotSelection +=
        new GoSelectionEventHandler(MyView_ObjectGotSelection);
    LinkButton.Click += new EventHandler(LinkButton_Click);
    SetLabelButton.Click += new EventHandler(SetLabelButton_Click);
    MyPalette.SessionStarted += new EventHandler(InitializePalette);
    MyView.SessionStarted += new EventHandler(InitializeCanvas);
  }
```

Your **SessionStarted** event handler named `InitializePalette` could be implemented as follows:

```
  public void InitializePalette(Object sender, EventArgs evt) {
    GoComment c = new GoComment();
    c.Text = "Enter your comments here";
    MyPalette.Document.Add(c);
    GraphNode n = new GraphNode(GraphNodeKind.Manager);
    MyPalette.Document.Add(n);
    GraphNode m = new GraphNode(GraphNodeKind.Individuals);
    MyPalette.Document.Add(m);
    GraphNode v = new GraphNode(GraphNodeKind.Vacancy);
    MyPalette.Document.Add(v);
  }
```

The sequence of calls and events can best be described by examining the definition of **GoView.OnLoad**:

```
  protected override void OnLoad(EventArgs evt) {
    . . .
    GoView saved = FindSessionView();
    if (saved != null) {
      LoadView(saved);
    } else {
      CreateView();
      OnSessionStarted(EventArgs.Empty);
    }
    // make sure the view's state is available later for GoWebImageHandler
    StoreSessionView();
    base.OnLoad(evt);
    . . .
  }
```

First we call **FindSessionView** to find the saved **GoView** in the **Session** state. If we find it, we call **LoadView** so that this **Page**-created instance of **GoView** gets a chance to restore the desired state from the saved **GoView** in the **Session**. Note that **LoadView** should be overridden by each subclass of **GoView**, in the following manner, to copy the state defined by that class:

```
  protected override void LoadView(GoView saved) {
    base.LoadView(saved);
    AppView v = (AppView)saved;
    myField = v.myField;
  }
```

If no saved **GoView** is found in the **Session** for this view, **OnLoad** calls **CreateView** to allow the view a chance to perform any expensive initialization that it only wants to do when a view is not found in the session state. This normally includes creating a **GoDocument**, a **GoSelection**, the default **GoTool**s, and a **GoViewDataRenderer**. Then it raises a **SessionStarted** event, to permit application-specific initialization such as document initialization. You can do such initialization based on control values that were loaded from the page's view state.

Note that **OnLoad** calls **StoreSessionView** to be sure that this **GoView** instance is saved in the session state. This is needed for the **GoWebImageHandler** HTTP handler to find the view in order to generate and stream an image to the browser, and so that the next call to **FindSessionView** in the session will be able to find the view and restore the state with a call to **LoadView**.

After the page and its controls have been restored to their earlier state in the session, input events are handled by the **RaisePostBackEvent** method. A **GoView** can then raise its own events, thereby invoking any event handlers that you may have registered. Remember that **GoView** event handlers are not serialized, so you must re-establish them in an **Init** or a **Load** event handler on the **Page** or the **GoView** itself, as shown above in the definition of **InitializeComponent** on the **Page**, or perhaps in your overrides of **GoView.LoadView** and **GoView.CreateView**.

Finally, just before the page is rendered to HTML, **GoView** raises an **Updated** event to make it easy for you to add code that might help save the current document state. Here is part of the definition of **GoView.Render**, and the definition of **OnUpdated**:

```
protected override void Render(HtmlTextWriter wrt) {
  OnUpdated(EventArgs.Empty);
  . . . produce HTML for the GoView
}

protected virtual void OnUpdated(EventArgs evt) {
  if (Updated != null)
    Updated(this, evt);  // call all handlers
}
```

You can easily have two views looking at the same document. Just be sure to set the **Document** property when the session is started.

```
MyView2.SessionStarted += new EventHandler(InitializeView2);
void InitializeView2(Object sender, EventArgs evt) {
  MyView2.Document = MyView.Document;
}
```

State that is not serialized will also need to be restored upon each load. This includes the **GoView.BackgroundImage** property and static properties such as **GoImage.DefaultResourceManager**. You can do this either in the **Page**'s **Load** event handler, the **GoView**'s **Load** event handler or in your override of **GoView.LoadView**.

Since session state must be used in order to produce an image in the <IMG> reference to the **GoWebImageHandler** HTTP handler, there can be quite a bit of memory stored in the session state. If you know that you no longer need any GoDiagram-related session state, you can discard it by performing:

```
  MyView.GetSessionViewsTable().Clear();
```

You might want to do this when you know that the user won't be returning to your page with the current state. For example, imagine a diagram having a bunch of hyperlink objects representing what in HTML would be `<a href=...>` tags. You can implement this with the following event handler:

```
private void MyView_ObjectSingleClicked(Object sender,
                                        GoObjectEventArgs evt) {
  GoTextNode n = evt.GoObject.ParentNode as GoTextNode;
  if (n != null) {
    String url = ... get URL for the node ...
    this.Response.Redirect(url, true);
    MyView.GetSessionViewsTable().Clear();
  }
}
```

The preceding code is modifying the **HttpResponse** object, only available when the whole **Page** is being generated.

Session State for NoPost GoViews

The standard scenario, described in the previous section, conforms to the normal **Page** and **WebControl** lifecycle in ASP.NET web applications. For an HTML request ASP.NET automatically re-creates the **Page** and its controls, initializes them individually with the **Init** event, restores their state, starts them with the **Load** event, and has them handle events from the request.

But if you are building an AJAX-style web application, you cannot afford to lose client-side state or communication time by having any postbacks happen unnecessarily. That is why there are "NoPost" views. (Our terminology precedes "AJAX", since GoDiagram Web has supported the **GoView.NoPost** property for years. Also, GoDiagram Web uses the JSON format for passing data rather than XML.)

When an HTML query goes directly to the **GoWebImageHandler** (`GoWebImage.axd`) HTTP handler, it must always specify the unique ID of the **GoView** that needs to be produced as an image. For normal pages, this is the only information needed—**GoWebImageHandler** just uses the unique ID to find the desired **GoView** in the session state and calls **GoView.GetBitmap** to produce the image.

However, in a **GoView.NoPost** scenario, the JavaScript-generated mouse or other action event request query string is passed directly to the **GoWebImageHandler** instead of through the page that contains the **GoView** web control. In both scenarios the **GoView** is deserialized from the session state. But in the **NoPost** case, the additional setup that is done by the **Page** and **WebControl**, i.e. the **Init** and **Load** events, does not happen because there is no **Page** that is reconstructed to contain the **GoView** control.

Clearly this can be much more efficient than the normal ASP.NET page request: no master **Page** is constructed on the server along with all its children, the deserialized **GoView** is modified directly and then reserialized, and state is maintained in the client browser. However, you will need to copy the **GoView** initialization code that is normally done in the **Init** or **Load** event handler for the **Page**, into an override of **GoView.OnNoPostLoad** so that it is performed even when the **GoView** deserialized instance is used directly. You only need to override this method if **GoView.NoPost** is true and you need to re-initialize any non-serialized state and event handlers.

Customizing Client-side Behavior

When a **GoView** is rendered, it produces some HTML. Primarily it generates an <IMG> element. But it also generates JavaScript data (JSON format) to describe parts of the image to implement client-side behavior.

In version 2.4 and earlier, it was possible to do some customization by defining the JavaScript functions that your application needed and modifying some of the standard code included in `GoWeb.js`. You could also override some methods on **GoView**, but the functionality was limited and awkward. The new **GoViewDataRenderer** class in version 2.5 improves your ability to customize the data that accompanies an image and makes it easier to specify certain predefined interactive behavior on the client.

Each **GoView** has a **DataRenderer** property that is an instance of a **GoViewDataRenderer**. You can set some of its properties to easily customize the client-side behavior. For example, if you want to execute some JavaScript code on the client when the user clicks a node, your **Page_Load** method could do something like:

```
// client-side behavior:
MyView.DataRenderer.LabeledNodeSingleClick = "EditLabel()";
MyView.DataRenderer.NoClick = "HideAll()";
```

Then you can define those JavaScript functions in your ASPX file:

```
    <script type="text/javascript">
<!--
function EditLabel() {
  goShowPanel('LabelPanel', 'LabelTextBox', goInfo.Text);
}
function HideAll() {
  goHide('LabelPanel');
}
// -->
    </script>
```

This makes use of some standard convenience functions that `GoWeb.js` provides (**goShowPanel** and **goHide**), some information passed from the server to the client (**goInfo**), and assumes that you have defined the corresponding web controls such as:

```
  <asp:Panel id="LabelPanel" style="display:none" runat="server">
   <BR>Label:
   <asp:TextBox id="LabelTextBox" runat="server"></asp:TextBox>
   <asp:Button id="SetLabelButton" runat="server" Text="Set Label"
               onclick="SetLabelButton_Click"></asp:Button>
  </asp:Panel>
```

Then the user will be able to click on a node, the LabelPanel will appear, and the user can modify the text. If they click the SetLabelButton, your **SetLabelButton_Click** event handler will be called on the server. You might have defined it to be something like:

```
protected void SetLabelButton_Click(object sender, System.EventArgs e) {
  IGoLabeledNode lnode = MyView.Selection.Primary as IGoLabeledNode;
  if (lnode != null) {
    MyView.StartTransaction();
    lnode.Label.Text = LabelTextBox.Text;
```

```
    MyView.FinishTransaction("set label");
  }
}
```

Until the user clicks the SetLabelButton, which in this example is defined to perform a postback, there is no communication with the web server.

You can easily specify what JavaScript to execute when the user clicks on an **IGoLabeledNode** such as a **GoNode** or a **GoLabeledLink** by setting some of the following **GoViewDataRenderer** properties:

- **LabeledNodeSingleClick**, executed when there is a single click on a **IGoLabeledNode** or on a **GoLabeledLink** containing a **GoText**
- **SingleClickDefault**, executed when there is a single click somewhere else in the view
- **LabeledNodeDoubleClick**
- **DoubleClickDefault**
- **LabeledNodeContextClick**
- **ContextClickDefault**
- **NoClick,** executed when none of the previous cases apply, or if the mouse moved too far between the mouse down point and the mouse up point (some kind of drag)

As another example, you might want to implement something similar to the hyperlink `MyView_ObjectSingleClicked` event handler shown above (which runs on the server), but open another page in JavaScript (without involving any server-side round-trips). You can do that by telling the GoView's DataRenderer that it should handle clicks on labeled nodes by calling a JavaScript function:

```
goView1.DataRenderer.LabeledNodeSingleClick = "NodeClicked()";
```

Then this JavaScript function could be defined in your ASPX page <SCRIPT> element, in a very simplistic fashion:

```
function NodeClicked() {
  window.open(goInfo.Text);
  return false;
}
```

This JavaScript opens a window displaying a URL given by the **Text** property for the node. Note again how the **goInfo** variable has been initialized by the Go code to refer to a JavaScript object holding properties corresponding to some of the properties of the node on the server. (More on this later.)

Normally all mouse events will be passed on to the server. (This may or may not cause a postback, depending on whether **GoView.NoPost** is false or true.) However, if you want to handle a click entirely on the client, you can just return **false** from any of these **…Click…** event-handling functions. The `NodeClicked` function above does this, so that clicking on a node will open a browser window without getting a new image from the server.

If you want to prevent all other mouse events from being passed on to the server, you initialize your **GoView.DataRenderer** as follows:

```
goView1.DataRenderer.NoClick = "false";
```

This need not just be the **false** expression – it can be any expression, including a function call, that returns false.

You can also implement your own client-side mouse-over behavior. Just define a JavaScript function in your ASPX page <SCRIPT> element, named **goOnMouseOver**, as follows:

```
function goOnMouseOver(e, id) {
  var info = goFindInfoEvent(e, id);
  var sdiv = document.getElementById('Status');
  if (sdiv) {
    if (info)
      sdiv.innerHTML = 'over: ' + info.Text;
    else
      sdiv.innerHTML = '';
  }
}
```

This example depends on an HTML element such as:

```
  <br><div id="Status"></div>
```

The **goOnMouseOver** function will be called for all GoViews that are in your HTML document. This is different than when defining click behavior, because you can easily specify different click behaviors for different objects (and when there is no object) and for different kinds of clicks.

The **goFindInfoEvent** function searches for a JavaScript object at the point of the mouse event. This JavaScript object will have various property/value pairs, provided by a **GoPartInfo** on the server, as described in the next section.

GoViewDataRenderer

Specifying client-side click behavior is only part of the purpose of the **GoViewDataRenderer** class. The primary purpose is actually to determine what data to send to the client. The **GoViewDataRenderer** class can generate information for cursors, tooltips, context menus, and general information about the parts of the diagram. It is the latter type of data that passes the text strings of visible **IGoLabeledNode**s to the browser.

On the server side your application needs to associate **GoPartInfo**s holding property values corresponding to objects visible in the view. These are reconstructed as JavaScript objects on the client side. In JavaScript code you can easily find such a particular "info" object given a point in the image by calling the **goFindInfoAt** or **goFindInfoEvent** function.

The **GoViewDataRenderer** iterates over all of the visible objects in the view. If **GoViewDataRenderer.PartInfos** is true (which it is by default), it calls **GoObject.GetPartInfo** on each top-level object and each immediate child of **GoSubGraph**s. This method is responsible for deciding whether any data should be associated with the **GoObject**'s area in the image, and if so, for specifying that data in a **GoPartInfo**.

The default implementation of **GoObject.GetPartInfo** just calls **GoViewDataRenderer.GetStandardPartInfo.** This checks whether the given **GoObject** is an **IGoLabeledNode** or a **GoLabeledLink** that holds a **GoText**

object. If it is, then it allocates a **GoPartInfo** by calling **CreatePartInfo** and adds property/value pairs. This is useful for the common case where you just want to provide a **GoPartInfo** to let the user see and perhaps modify the value of a text label locally on the client.

If you want to optimize the data generation to avoid producing this standard **GoPartInfo** when the **Label** is not **Editable**, because you only want to have the data on the client when the user might be able to modify the text, you can set **GoViewDataRenderer.PartInfosIfLabelNotEditable** to false.

Predefined **GoPartInfo** property names include **ID**, **Text**, **SingleClick**, **DoubleClick**, and **ContextClick**. These are also the names of properties of the **GoPartInfo** class which just call **GoPartInfo.GetProperty** and **SetProperty** with the corresponding predefined name. You can easily add your own property/value pairs by overriding **GoObject.GetPartInfo**, so that each of your interesting object classes can pass information to the browser. For example, consider this override on a **GoLabeledLink**:

```
public override GoPartInfo GetPartInfo(GoView view, IGoPartInfoRenderer renderer) {
  GoPartInfo info = renderer.CreatePartInfo();
  if (this.MidLabel is GoText) {
    info.Text = ((GoText)this.MidLabel).Text;
  }
  info["Curviness"] = this.Curviness;
  info.SingleClick = "ShowLink()";
  return info;
}
```

This definition of **GetPartInfo** passes along a "Text" property value and a "Curviness" property value. It also specifies the JavaScript function to call when the user clicks on that **GoLabeledLink**.

As always, being able to override methods on either **GoViewDataRenderer** or **GoObject** gives you the flexibility to put your code where it makes the most sense for your application.

The **GoViewDataRenderer.Render** method collects all of these **GoPartInfo**s that are associated with regions in the image and then generates some JavaScript that initializes a JavaScript object that corresponds to the view and is associated with the <IMG> DOM element. A **GoPartInfo** is rendered by calling **ToString()**, which generates a JavaScript Object Notation string. **GoPartInfo** can handle property values that are strings, booleans, integers, floats, and Arrays of those types. [By the way, **GoPartInfo** has some static methods that you may find useful in producing quoted JavaScript strings.]

GoViewDataRenderer.Render also generates JavaScript to initialize data structures for cursors, tooltips, and context menus. In addition to cursors, tooltips, and context menus associated with **GoObject**s, it also specfies default properties for the whole view: **ToolTipDefault**, **MenuDefault**, **LabeledNodeSingleClick, LabeledNodeDoubleClick, LabeledNodeContextClick, SingleClickDefault**, **DoubleClickDefault**, **ContextClickDefault**, and **NoClick**. The "…Click…" properties all provide JavaScript to be executed for the corresponding event; the **NoClick** property specifies the JavaScript to execute when there is a mouse-up-and-down that is not a click, such as a mouse-drag, or when there is a click for which no "…Click" or "…ClickDefault" JavaScript is defined.

On the client you can retrieve the JavaScript object that holds all the data for a **GoView** by calling the **goFindView** function. You can retrieve the JavaScript object corresponding to a **GoPartInfo** by calling the **goFindInfoAt** function, passing it X and Y view coordinates. Of course **goFindInfoAt** may very well return

null, if there wasn't any **GoObject** at that point, or if no **GoPartInfo** was produced for the **GoObject** at that point.

You may want to execute some JavaScript as soon as the data for a view has been downloaded. Define a **goOnLoad** JavaScript function; it will be called with an argument that is the id of the view.

```
function goOnLoad(id, reload) {  // called after data has been downloaded
  var v = goFindView(id);
  if (v != null) {
    // use DHTML to update your page
  }
}
```

Handling Requests from the Client

Finally, **GoViewDataRenderer** and the **goRequest** JavaScript function make it a little easier to implement custom query handlers. You have always been able to override **GoView.RaisePostBackEvent** to parse the argument string and decide what actions to take. (Remember that this method is called both for real postbacks as well as when the view is reloaded in a NoPost, AJAX-like situation.) By overriding **GoViewDataRenderer.HandleClientRequest,** you will get not only the original query string, but also a **Hashtable** holding the parsed name/value pairs from that string. Also, you may find that implementing a class inheriting from **GoViewDataRenderer** is simpler than subclassing **GoView.**

For example, let's say you want a button that changes the color of the currently selected objects.

```
<button onclick="changeSelectionColor('red')" type="button">Make Red</button>
```

You could implement a JavaScript function as follows:

```
function changeSelectionColor(colorname) {
  goRequest('MyView', 'color=' + colorname);
}
```

This makes use of the **goRequest** JavaScript function that is defined in `GoWeb.js`. It calls the **GoViewDataRenderer.HandleClientRequest** method, which you might implement as:

```
public override void HandleClientRequest(String evtargs, Hashtable parameters) {
  String color = (String)parameters["color"];
  if (color != null) {
    Color c = FromString(color);
    this.View.StartTransaction();
    foreach (GoObject obj in this.View.Selection) {
      GoShape shape = obj as GoShape;
      if (shape != null) {
        shape.BrushColor = color;
      }
    }
    this.View.FinishTransaction("changed colors");
  }
}
```

Context Menus

For version 2.5 we have implemented context menu classes that are subsets of the same-named Windows Forms classes. The GoDiagram Web versions, like their Windows Forms counterparts, are not **Control**s, but are just objects that hold descriptive information.

Here's an example that defines a context menu for a particular kind of node. In the node class you would override **GoObject.GetContextMenu** to return a **GoContextMenu**:

```
public override GoContextMenu GetContextMenu(GoView view) {
  GoContextMenu cm = new GoContextMenu(view);
  if (view.CanInsertObjects()) {
    cm.MenuItems.Add(new MenuItem("Add Port",
          new EventHandler(this.AddPort_Command)));
  }
  return cm;
}

private void AddPort_Command(Object sender, EventArgs e) {
  this.Document.StartTransaction();
  . . . create and Add a port . . .
  this.Document.FinishTransaction("Add Port");
}
```

The **GoViewDataRenderer** will see that there is a **GoContextMenu** for your node and will send the description of the context menu and its menu items to the browser. This information is used to construct a DHTML context menu. A click on the menu item will invoke the **AddPort_Command** shown above, on the server. The definition of the standard context menus uses DHTML and styles to describe the appearance and behavior of the whole context menu, its items, and any separators.

There are a few declarations you need to provide in order for context menus to be rendered by the server and constructed on the client. You need to set **GoViewDataRenderer.ContextMenus** to true and make sure **GoView.CssFile** is not `"none"`. And if you don't want the **GoView** to render the standard CSS definitions each time, you need to make sure **GoView.CssFile** is set to `"GoWeb.css"` and make sure that file is in your web site. You are of course free to use your own CSS definitions.

You can specify a default context menu, to be used for a context click that is not over an object where a context menu is defined, by setting **GoViewDataRenderer.DefaultContextMenu**, or by overriding **GoViewDataRenderer.GetDefaultContextMenu** if you want to generate it conditionally or dynamically based on the state of the view or document. The event handlers can also be defined on your subclass of **GoViewDataRenderer**, instead of on a **Page**, which is convenient for NoPost views when the **Page** might not exist.

GoMenuItem has an additional constructor overload that lets you specify some JavaScript to run when that menu item is clicked. This is useful for implementing context menu behavior that does not invoke code on the server, but code running in the browser. Consider a node that overrides **GetContextMenu**:

```
public override GoContextMenu GetContextMenu(GoView view) {
  if (view is GoOverview) return null;
  GoContextMenu cm = new GoContextMenu(view);
  cm.MenuItems.Add(new GoMenuItem("Rename", "EditLabel()"));
```

```
    return cm;
  }
```

The call to **EditLabel** is actually a call to a JavaScript function that the application defines. In your JavaScript code you can use the global variable **goInfo** to get the JavaScript object corresponding to the **GoPartInfo** corresponding to the object in that view.

Printing

Support for printing in web applications has always been problematic. Although we cannot provide good printing support without having a DLL or Java Applet running on the client, we do provide a **WebControl** that you might want to consider. The **GoPrintView** web control, when initialized to refer to a **GoView**, renders as a number of large images that the user can print using the browser's printing commands. This provides a solution that will work on any browser.

The idea is that you define a separate ASPX page that consists of a **GoPrintView** control and whatever additional descriptive information that you want, such as some text and/or a legend. The **GoPrintView** will look at the **View**'s document and render as enough <IMG> elements to cover the needed area. Hopefully the images will be large enough to be moderately efficient in making use of the printable area of the page, and not too large to cause any image to be clipped when printing. You can control the image size by setting the **GoPrintView.ImageWidth** and **ImageHeight** properties.

You can also set the **PageLimit** and **ViewScale** properties to control how many images there are.

A typical usage would be to have a button in your ASPX page that called some JavaScript:

```
<button onClick="OpenPrintPage('MyView')" type="button">Print...</button>

function OpenPrintPage(id) {
  var img = goFindImg(id);
  if (img != null && img.goID != null) {
    open('PrintPage.aspx?GoView=' + img.goID, '');
  }
}
```

Note the use of the JavaScript function **goFindImg**, provided by `GoWeb.js`. This function takes an "id" and finds the <IMG> element with that "id". The "goID" property provides the unique string identifying the particular view in the session. This is what is needed to be able to find the right **GoView** on the server in session state, so we pass it on to the PrintPage.aspx page as a query parameter.

You would define your `PrintPage.aspx` file to set the **GoPrintView.ViewID** when then **Page** is loaded:

```
  protected void Page_Load(object sender, System.EventArgs e) {
    // always get the intended GoView
    MyPrinter.ViewID = this.Request.Params["GoView"];
  }
```

Of course you can use other methods for communicating between the two pages, but this is one simple, effective way. You might even be able to reuse your `PrintPage.aspx` file if you have other pages that show diagrams that the user might want to print.

Static Images

The normal use of a **GoView** is interactive. However, you may be interested in generating images to be saved on disk and served as image files.

You can do this at any time by constructing a **GoView**, initializing the **GoDocument**, calling **GoView.GetBitmap** or **GetBitmapFromCollection**, and then saving the **Bitmap** as a file in the desired **ImageFormat**. However, you will need to manage the naming, lifetime, and potential security risks of the files that you write to disk.

ASP.NET AJAX

Microsoft has introduced extensions to ASP.NET 2.0 that support AJAX-style interaction. GoDiagram views do work within **UpdatePanel**s, as well as outside of them. Of course we still suggest that you set **GoView.NoPost** to true, since that will permit the view's handling of events and updating of the image without any partial rendering on the server or any HTML DOM changes on the client.

Common Problems

If no image is displayed for your **GoView** or other view class, it means there was an error. The most common cause of this error is forgetting to implement the image-generation mechanism. There are two choices, controlled by the value of **GoView.ImagePage**:

- The default value, "GoWebImage.axd", is more efficient, but it requires you to include the following lines in your `Web.config` file:

```
<httpHandlers>
  <add verb="GET" path="GoWebImage.axd" type="Northwoods.GoWeb.GoWebImageHandler"/>
</httpHandlers>
```

- A value of "GoWebImage.aspx" requires you to include the `GoWebImage.aspx` file in your project. This file consists entirely of one ASP.NET Page directive:

```
<%@ Page Inherits="Northwoods.GoWeb.GoWebImage" %>
```

  This file does not need any code-behind class.

The other most common reason for not seeing a view image is that cookies were not enabled by the browser.

A less common reason for no image in the browser is the inability to use session state. If you are not using the InProc implementation of session state, the **GoView**, its **GoDocument**, and its **GoObject**s may be serialized and deserialized. If you have extended any of these classes, be sure that everything is serializable. Note that event handlers are not normally serializable, so any registered event handlers for **GoView** events may be lost. It may be easiest to override the corresponding desired **GoView.On…** methods instead.

Another problem is that an image is visible, but the user cannot interact with it (assuming that **GoView.Enabled** is true, of course). Usually this is caused by JavaScript code being disabled on the browser.

However, another possible reason for a non-interactive image is that you have specified a value for **GoView.ScriptFile** that is an invalid or inaccessible JavaScript file. The default value is "`GoWeb.js`".

Normally one should just copy the `GoWeb.js` file into your web site, but if you forget, or if the reference isn't valid within the web site, or if the contents of the file have errors in them, there will be run-time errors on the browser. Note also that the tilde character ("~") is not supported as part of the file path – the file reference should be a relative path.

If context menus are not appearing, be sure that the value of **GoView.CssFile** is not "`none`", and that if is not the empty string, that that CSS file actually exists on your web site and is accessible, with valid definitions corresponding to the ones given in the standard `GoWeb.css` file.

If you have multiple **GoView**s on the same page, for example by having both a **GoView** and a **GoPalette** in the same form, be sure that the values for **GoView.ImagePage**, **GoView.ScriptFile**, and **GoView.CssFile** are respectively the same for all the controls, to avoid any potential inconsistencies or conflicts.

For maximal rendering performance, be sure to specify references for **GoView.ScriptFile** and (if using client-side context menus) **GoView.CssFile**, typically to **"GoWeb.js"** and **"GoWeb.css"**, respectively.

For optimum image-generating performance, be sure to specify **"GoWebImage.axd"** as the value for **GoView.ImagePage**, and define the HttpHandler in your `web.config` file. Be sure that any configured HTTP handler for `*.axd` does not take precedence over this one; you can use a different name and file extension if it is more convenient. In IIS7, you will need to either update the integrated `<system.webServer> <handlers>` configuration section, or you will need to run your web app in Classic ASP.NET integration mode by moving it to the Classic ASP.NET Application Pool.

For optimum data-rendering performance, set to **false** all the properties of **GoWebDataRenderer** for features that you don't need to use. For example, if you don't use any client-side context menus, specify "none" for the **GoView.CssFile** and set:

```
goView1.DataRenderer.ContextMenus = false;
```