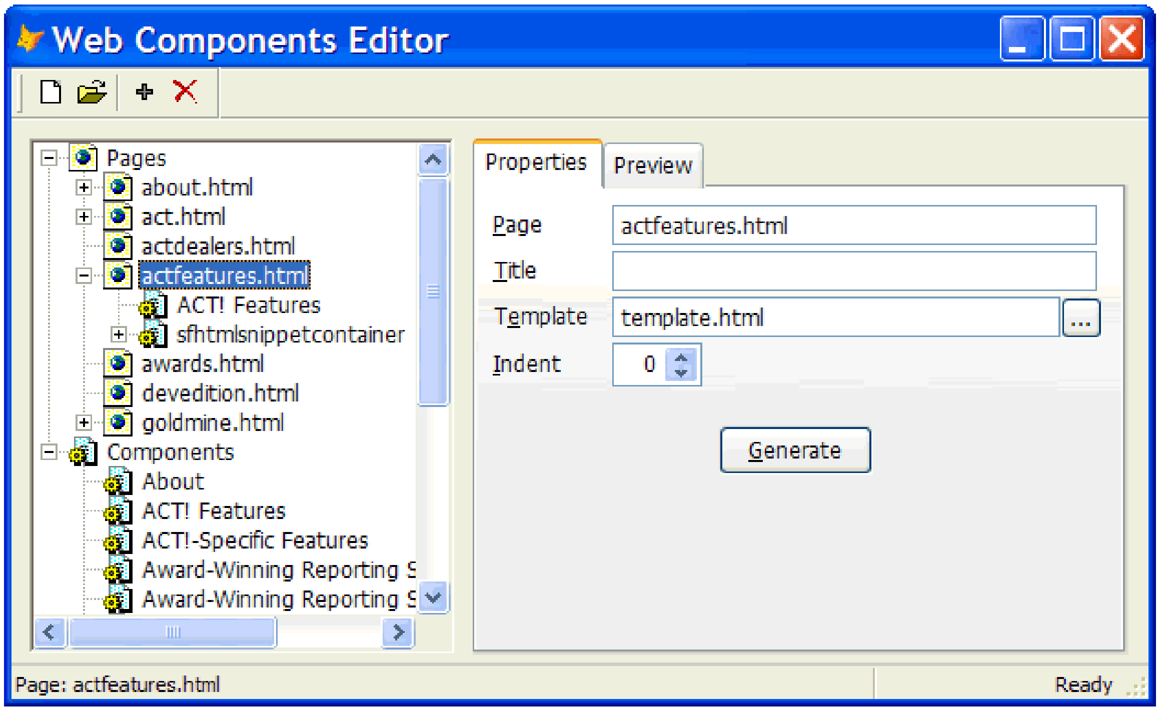
The Mother of All TreeViews

Doug Hennig

If you've ever had to work with a TreeView control, you'll appreciate Doug Hennig's article this month: it provides a class that encapsulates all of the quirky behavior of a TreeView so you don't have to worry about it. You need only implement a few methods to get a fully-functional control that provides the features your users need.

Last month, we looked at a set of classes that allow you to generate a Web page as a collection of reusable HTML components. However, as I quickly discovered, populating the tables that drive the classes isn't much fun in BROWSE windows. This month and next month, we'll look at a front-end UI for these tables to make that task much easier.

When I started thinking about what the editor tool should look like, I envisioned a form with a TreeView control at the left (since the HTML content can be hierarchical) and a pageframe with one page showing the properties of the selected node and another showing a preview of the HTML using an embedded Web browser control. Figure 1 shows the finished product, WebEditor.SCX.



Page: actfeatures.html
Title:
Template: template.html
Indent: 0

Page: actfeatures.html

Figure 1. WebEditor.SCX provides an easy-to-use interface for editing the Web content tables.

I then groaned because I've created similar forms before and they're a ton of work. The reason is because the TreeView is both a very rich control, so there are lots of properties, events, and methods (PEMs) to worry about, and a fairly ornery control. Here are just some of the gotchas about working with a TreeView:

- Loading nodes is fairly slow, so if there are a lot of them, it's better to load only the top-level or root nodes, and then load the child nodes on demand the first time a root node is expanded. This requires a bit of work: you have to add "dummy" child nodes or else the + won't appear, and when the node is expanded, you have to see if the first child is the "dummy" node, and if so, remove it and add the child nodes.

- Removing the "dummy" node and adding child nodes can cause a lot of screen update. For VFP forms, we set LockScreen to .T. to prevent the screen from being updated until we're ready, but the

TreeView doesn't respect the setting of LockScreen and doesn't have its own equivalent property. Instead, you have to call a Windows API function to lock and unlock the TreeView.

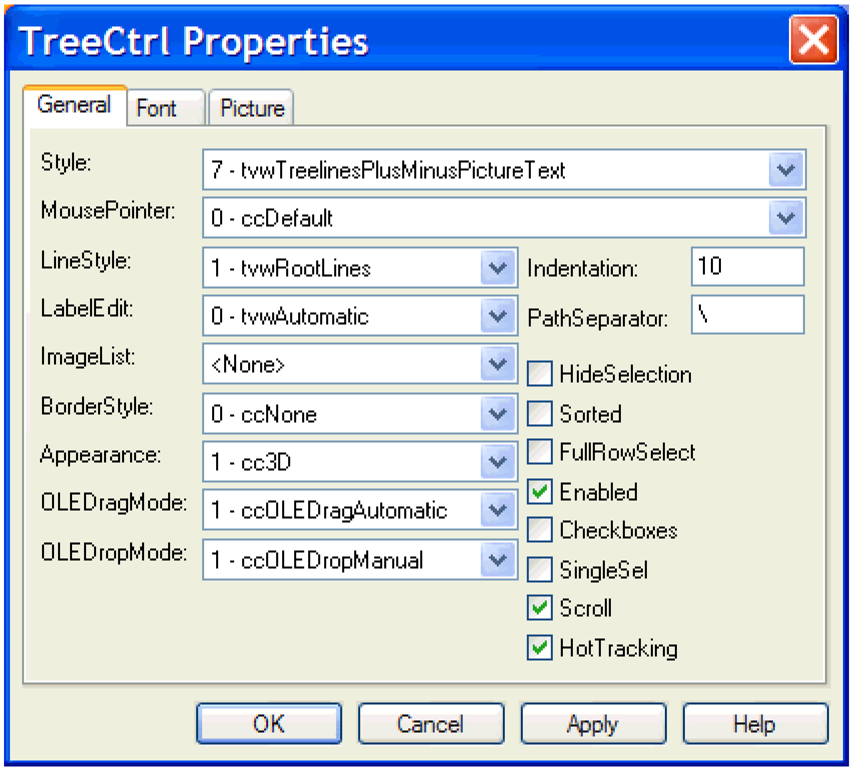
- The coordinate system for TreeViews is twips (1/1440th of an inch) rather than pixels, so you have to convert pixels to twips before calling TreeView methods that expect coordinate parameters. Again, this involves using Windows API functions.
- Right-clicking or dragging from a node doesn't make it the selected node, so code that displays a menu for the current node or supports node dragging must manually figure out which node the user clicked on.
- You may want the user to be able to double-click on a node to take some action, such as bringing up a dialog to edit the underlying record for the node. Unfortunately, double-clicking a node automatically causes it to expand or collapse if it has children.
- If you support drag and drop, the TreeView doesn't automatically scroll up or down when you drag into the top or bottom edges, nor does it automatically highlight the node the mouse is over to give visual feedback to the user. You have to handle these things yourself.

Having created forms before that had to deal with all of these quirks and coordinate TreeView node clicks with the rest of the form, I decided it was time once and for all to create a reusable component that would encapsulate all the behavior I wanted in one place. So, step one in creating the Web components editor was to create the reusable TreeView class.

SFTreeViewContainer, defined in SFTreeView.VCX, is a subclass of SFContainer (my base class Container located in SFCtrls.VCX). It contains both a TreeView and an ImageList control, the latter being the source of images for the TreeView. Most TreeView events call methods of the container for several reasons, including that good design dictates that events should call methods and it's easier to get at the PEMs of the container than to drill down into the TreeView in the Class Designer. I considered using the new VFP 8 BINDEVENTS() function so I didn't have to put any code at all into the TreeView events, but some TreeView events, such as BeforeLabelEdit, receive their parameters by reference, which BINDEVENTS() doesn't support.

TreeView appearance

I set the properties of the TreeView control visually using the TreeCtrl Properties window as shown in Figure 2. If you want to use different settings in a subclass, such as using 0 for Appearance so the TreeView appears flat rather than 3-D, you'll need to change them in code; for some reason, changing the properties of the TreeView in a subclass doesn't seem to work. So, in the Init method of a subclass of SFTreeViewContainer, use DODEFAULT() followed by code that changes the properties of the TreeView as desired.



Style: 7 - tvwTreelinesPlusMinusPictureText
MousePointer: 0 - ccDefault
LineStyle: 1 - tvwRootLines
LabelEdit: 0 - tvwAutomatic
ImageList: <None>
BorderStyle: 0 - ccNone
Appearance: 1 - cc3D
OLEDragMode: 1 - ccOLEDragAutomatic
OLEDropMode: 1 - ccOLEDropManual
Indentation: 10
PathSeparator: \
☐ HideSelection
☐ Sorted
☐ FullRowSelect
✓ Enabled
☐ Checkboxes
☐ SingleSel
✓ Scroll
✓ HotTracking

Figure 2. The properties of the TreeView control are set visually.

The images used for the TreeView come from the ImageList control in the container. As with the TreeView, loading images into the ImageList visually in a subclass doesn't seem to work, so the Init method of SFTreeViewContainer calls the abstract LoadImages method. In a subclass, call the Add method of the ListImages collection of the ImageList object to add images. Note that you have to use the VFP LOADPICTURE() function to pass a reference to a loaded image object to the Add method.

Here's the code in the LoadImages method of the SFTreeViewContainer object in WebEditor.SCX. It sets the ImageHeight and ImageWidth properties of the ImageList to the values needed for the images being loaded, then loads PAGE.GIF as image 1 with a key of "Page" and COMPONENT.GIF as image 2 with a key of "Component".

```
with This.oImageList
  .ImageHeight = 16
  .ImageWidth  = 16
  .ListImages.Add(1, 'Page', ;
    loadpicture('PAGE.GIF'))
  .ListImages.Add(2, 'Component', ;
    loadpicture('COMPONENT.GIF'))
endwith
```

Loading the TreeView

There are four custom properties of SFTreeViewContainer associated with loading the TreeView with the proper nodes:

- lLoadTreeViewAtStartup: if this property is .T. (the default), the container's Init method calls the LoadTree method to load the TreeView. If it's .F., you'll have to manually call LoadTree when you want the TreeView loaded. This might be necessary if the Init of the form the container resides in must do some work before the TreeView is loaded.
- lAutoLoadChildren: if this property is .T., all nodes in the entire TreeView are loaded. This is fine if there aren't very many, but can cause a serious performance issue otherwise; it might take so long to load the TreeView that the user thinks the application has crashed. In that case, set

lAutoLoadChildren to .F. (the default setting). The container will only load the top-level nodes (or more, depending on other settings described later). Any loaded node that has child nodes will have a "Loading…" dummy node added under it rather than the real child nodes so the + will appear for the node, indicating it can be expanded. When a node is expanded for the first time, the TreeExpand method of the container (called from the Expand method of the TreeView) removes the dummy node and adds the real child nodes.

- nAutoExpand: this property indicates to what level the nodes of the TreeView are automatically expanded when the TreeView is loaded. The default is 0, meaning none of the nodes are expanded (although, as you'll see later, restoring the TreeView to its last viewed state will override this for some nodes). Setting it to 1 means that the top-level nodes should be expanded; this means that even if lAutoLoadChildren is .F., the container will load the immediate children (but not the children's children) of the top-level nodes. Setting it to 2 means the immediate children of the top-level nodes should be expanded (causing their immediate children to be loaded), and so forth.
- lUsePathAsKey: if this property is .F., you'll assign unique key values to nodes in the TreeView. If it's .T. (the default), you don't have to worry about keys; the container will use the path of the node (similar to the path of a file) as its key.

The LoadTree method does the main work of loading the TreeView. For space reasons, this code isn't shown here, but here's how it works. It starts by calling the LockTreeView method to lock the TreeView control (using a Windows API function) so updates aren't displayed as they're performed. Next, any existing nodes that are currently expanded are saved into the aExpandedNodes array property and the key for the currently selected node is saved in the cLastNode property; this allows you to call LoadTree a second time to refresh the contents (for example, if the TreeView displays records from a table that other users on a network may also be editing) and have the expanded state of each node and the currently selected node restored. LoadTree then loads the top-level nodes by calling the GetRootNodes method, passing it a Collection object. GetRootNodes is an abstract method that you must implement in a subclass to fill the collection with objects containing information about the top-level nodes; I'll discuss this further later. LoadTree then calls LoadNode for each object in the collection to load it into the TreeView. Any formerly expanded nodes are re-expanded and the former selected node is reselected (if these nodes still exist). Finally, LoadTree calls LockTreeView again to unlock the TreeView.

You must fill in code in the GetRootNodes method of a subclass or instance of SFTreeViewContainer to fill the collection with objects containing information about the top-level nodes. To create such an object, call the CreateNodeObject method; it simply creates an Empty object and adds the following properties:

- Key: the key for the node in the TreeView. This is automatically set to SYS(2015) but you should change it to a unique value if the lUsePathAsKey property is .F.
- Text: the text to display for the node.
- Image: the name or number of an image in the ImageList control for this node.
- SelectedImage: the name or number of an image in the ImageList control to use when this node is selected. You can leave this blank to use the same value as Image.
- Sorted: .T. if the node should be sorted.
- HasChildren: .T. if this node has any children.

You can override CreateNodeObject if you need additional properties. In that case, you should also override GetNodeItemFromNode, which creates and fills the properties of a node item object from the current TreeView node.

GetRootNodes should create one object for each top-level node, fill in the properties, and add it to the passed-in collection. Here's the code in the SFTreeViewContainer object in WebEditor.SCX that creates two top-level nodes: one, "Pages", for pages defined in the PAGES table I discussed in last month's article, and the second, "Components", for the Web components defined in the CONTENT table.

ccPAGE_HEADER_KEY and ccCOMPONENT_HEADER_KEY are constants defined in WebEditor.H, the include file for this form.

```
lparameters toCollection
local loNodeItem
loNodeItem = This.CreateNodeObject()
go top in PAGES
with loNodeItem
  .Key         = ccPAGE_HEADER_KEY
  .Text        = 'Pages'
  .Image       = 'Page'
  .Sorted      = .T.
  .HasChildren = not eof('PAGES')
endwith
toCollection.Add(loNodeItem)

loNodeItem = This.CreateNodeObject()
go top in CONTENT
with loNodeItem
  .Key         = ccCOMPONENT_HEADER_KEY
  .Text        = 'Components'
  .Image       = 'Component'
  .Sorted      = .T.
  .HasChildren = not eof('CONTENT')
endwith
toCollection.Add(loNodeItem)
```

You must also implement the GetChildNodes method. Like GetRootNodes, this method fills a collection with node item objects. In this case, these objects represent the children of a node. GetChildNodes is passed three parameters: the type and ID for the node whose children are needed and the collection to fill.

Here's the code for GetChildNodes in the SFTreeViewContainer object in WebEditor.SCX. In the case of the "Pages" root node, the records in the PAGES table are loaded into the collection, with ccPAGE_KEY (another constant) and the ID as the value for Key. For the "Components" root node, the records in the CONTENT table are loaded, using ccCOMPONENT_KEY and the ID for Key. For a particular page node, the records in PAGECONTENT for the page that don't have a parent record are loaded; the reason records with PARENT filled in are ignored is because they will be loaded as children of their parent records.

```
lparameters tcType, ;
  tcKey, ;
  toCollection
local loNodeItem, ;
  lcWhere
do case

* If this is the "Pages" node, fill the collection with
* the pages defined so far.

  case tcType = ccPAGE_HEADER_KEY
    select PAGES
    scan
      loNodeItem = This.CreateNodeObject()
      with loNodeItem
        .Key         = ccPAGE_KEY + transform(ID)
        .Text        = trim(PAGE)
        .Image       = 'Page'
        .HasChildren = seek(PAGES.ID, 'PAGECONTENT', ;
          'PAGEID')
      endwith
      toCollection.Add(loNodeItem)
    endscan

* If this is the "Components" node, fill the collection
* with the components defined so far.
```

```
  case tcType = ccCOMPONENT_HEADER_KEY
    select CONTENT
    scan
      loNodeItem = This.CreateNodeObject()
      with loNodeItem
        .Key   = ccCOMPONENT_KEY + transform(ID)
        .Text  = trim(NAME)
        .Image = 'Component'
      endwith
      toCollection.Add(loNodeItem)
    endscan

* If this is a page node or a content node with children,
* load the content nodes.

  case tcType = ccPAGE_KEY or ;
    seek(tcKey, 'PAGECONTENT', 'PARENT')
    if tcType = ccPAGE_KEY
      lcWhere = 'PAGEID = tcKey and empty(PARENT)'
    else
      lcWhere = 'PARENT = tcKey'
    endif tcType = ccPAGE_KEY
    select PAGECONTENT.ID, ;
        PAGECONTENT.CLASS, ;
        CONTENT.NAME ;
      from PAGECONTENT ;
        left outer join CONTENT ;
          on PAGECONTENT.CONTENTID = CONTENT.ID ;
      into cursor _TEMP ;
      where &lcWhere ;
      order by PAGECONTENT.ORDER
    scan
      loNodeItem = This.CreateNodeObject()
      with loNodeItem
        .Key         = ccPAGE_COMPONENT_KEY + ;
          transform(ID)
        .Text        = trim(iif(empty(nvl(NAME, '')), ;
          CLASS, NAME))
        .Image       = 'Component'
        .HasChildren = indexseek(ID, .F., ;
          'PAGECONTENT', 'PARENT')
      endwith
      toCollection.Add(loNodeItem)
    endscan
    use

* We have a bad node type, so do nothing (we don't
* actually need the OTHERWISE statement, but this way,
* I won't get heck from Andy Kramek <gd&rfAK>).

  otherwise
endcase
```

Restoring the TreeView state

When you run a form that uses SFTreeViewContainer a second time, you may expect it to appear the same as it did last time: the node that was selected is still selected and all the nodes that were expanded are still expanded. This is handled by the SaveSelectedNode and RestoreSelectedNode methods, which are called from Destroy and Init, respectively. These methods are abstract in SFTreeViewContainer so it's up to you to implement them in a subclass or instance.

SaveSelectedNode should save the value of the Key property of the selected node and each expanded node. You can save them anywhere you wish: the Windows Registry, an INI file, a table, and so forth. Here's the SaveSelectedNode method in WebEditor.SCX; it saves the information to an INI file using WriteINI.PRG, which is included with this month's Subscriber Downloads. cWebSitePath is the location of the currently open Web site files.

```
local lcFile, ;
  lnNode, ;
  lnHandle, ;
  loNode, ;
  lcContent, ;
  laLines[1], ;
  lnLines, ;
  lnI, ;
  lcLine
with This.oTree
  if vartype(.SelectedItem) = 'O'
    lcFile = Thisform.cWebSitePath + 'settings.ini'
    lnNode = 0
    if not file(lcFile)
      lnHandle = fcreate(lcFile)
      fclose(lnHandle)
    endif not file(lcFile)
    WriteINI(lcFile, 'Nodes', 'SelectedNode', ;
      .SelectedItem.Key)
    for each loNode in .Nodes
      if loNode.Expanded
        lnNode = lnNode + 1
        WriteINI(lcFile, 'Nodes', 'Expanded' + ;
          transform(lnNode), loNode.Key)
      endif loNode.Expanded
    next loNode

* Remove extra lines from the INI file in case there
* were more expanded nodes in the INI file the last
* time it was written to.

    lcContent = ''
    lnLines   = alines(laLines, filetostr(lcFile))
    for lnI = 1 to lnLines
      lcLine = laLines[lnI]
      if lcLine <> 'Expanded' or ;
        val(strextract(lcLine,'Expanded','=')) <= lnNode
        lcContent = lcContent + lcLine + ccCRLF
      endif laLines[lnI] = 'Expanded' ...
    next lnI
    strtofile(lcContent, lcFile)
  endif vartype(.SelectedItem) = 'O'
endwith
```

Similarly, RestoreSelectedNode should restore these items. We won't look at the code in WebEditor.SCX here; it simply uses ReadINI.PRG to restore cLastNode and aExpandedNodes from the INI file.

Summary

That's all we have room to cover this month. Next month, we'll discuss handling node selection, drag and drop, and other behavior, then look at a form class that ties SFTreeViewContainer to a pageframe of properties for the selected node.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, author of the CursorAdapter and DataEnvironment builders that come with VFP 8, and co-author of "What's New in Visual FoxPro 8.0", "What's New in Visual FoxPro 7.0", and "The Hacker's Guide to Visual FoxPro 7.0", from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP). Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com