

Google, can you hear me?

How to design URLs that are search engine friendly
by Michael C. Neel

If Pete Townsend wrote his rock opera *Tommy* today, we might have a very different story: Tommy is a young lad who loves pinball, runs a website called Pinball Wizard, and blogs all things pinball. Despite having the most pinball information on the Internet, his traffic is a trickle. After a deep soul-searching journey he finds out why: Google can't see him.

This may make for a bad soundtrack, but the lesson is important. As developers, it's easy to forget how our sites appear to search engines when we are knee deep in business logic implementation. No matter how well we design the interface or target content to the visitor, nothing will affect site traffic more than search engine rankings. This article will show you how to build a database-driven website with search engine friendly URLs using tools that already exist in the ASP.NET 2.0 Framework.

What Google Wants

Note: There are thick tomes written solely on cracking the DaGoogle Code; this article is not about fooling Google or tricks to raise rankings. The goal here is to correctly identify site content to a search engine.

When parsing a link, a search engine considers (among other things) the URL, the text of the anchor tag, the title tag of the document it links to, and the content of that document. The more parts that match, the better the chance the search engine will assign the keywords found in the link, title, and content to the document's index. Let's look at a poorly designed (yet all too common) link:

```
<a href="http://www.ddbkig.org/blog/post.aspx?id=34">World  
Finals</a>
```

The only part of this link with relevant content is the text of the anchor tag, "World Finals." "Blog," "post," and "id" are found in the URL, but these are keywords we do not want indexed. Let's rewrite a better URL:

```
<a  
href="http://www.ddbkid.org/Articles/Tournaments/2006/06/12/World-  
Finals.aspx">World Finals</a>
```

"World Finals" is now in both the anchor tag text and URL. We have added meta data to our URL: the category "Tournaments" and the date of the article, 12/6/2006. This meta data will match the title tag and/or content of the actual article.

When designing your site's URL structure consider what meta data is most useful and keep it simple. You do not want to dilute the information by providing too much, this could backfire and lower your rankings. URLs should also be

friendly to the eye, and imply a site structure. When done correctly, friendly URLs will give a visitor an impression of efficiency and stability.

Solution Overview

Now that we know what we want our URLs to look like, let's see how ASP.NET can help. The bulk of our solution will come from a custom SiteMapProvider, which will handle the conversion of database content to friendly URLs. A method updated in 2.0, HttpContext.RewritePath, will direct friendly URL requests to a template page. Finally, the PostBackUrl attribute will send posted form data to the correct URL.

In this example there are two template pages. First is the category page, which will be accessed when the URL is a directory of the category and ends in Default.aspx. The category page will list all articles in that category. The second page will display the article content. A master page will be used to display the site navigation on both pages and ensure proper page titles.

Custom SiteMapProvider

ASP.NET 2.0 uses a provider model to separate the data storage from the service using the data. This insulates the application from storage details and provides a flexible and extensible base for custom storage solutions. Using a custom SiteMapProvider we can control how the sitemap structure is built. In other frameworks this would be a chore, but Microsoft designed the ASP.NET 2.0 providers with extensibility as a primary goal. With the SiteMapProvider interface, no longer is there any reason to have “deep” web pages - pages not linked to by other pages.

A custom SiteMapProvider must inherit the SiteMapProvider abstract class, and at the minimum implement the following methods:

- FindSiteMapNode – Overloaded to take a URL or a HttpContext, however you are only required to implement the URL version. It returns a SiteMapNode or null if there was not a match.
- GetChildNodes – Passed a SiteMapNode and returns a SiteMapNodeCollection of child SiteMapNodes or null if there are no children.
- GetParentNode – Passed a SiteMapNode and returns the parent SiteMapNode or null if there is no parent node.
- GetRootNodeCode – Returns the root SiteMapNode of the sitemap. All SiteMapProviders are required to have only one root node.

In addition to these methods, we also implement Initialize, called when the provider is first loaded, to build our sitemap. Not part of SiteMapProvider, a few static methods are included to generate article and category URLs. The BuildNodes method does the actual work and is a public method so the SiteMap can be rebuilt when posts are added, changed or deleted.

Listing one shows the complete custom SiteMapProvider. The BuildNodes method stores the articles in a Dictionary keyed by the category. Categories are placed in a SiteMapNodeCollection. All of this occurs inside a lock to ensure two calls to BuildNodes do not overlap.

SiteMapNodes have a Key property that is described as “a string representing a lookup key for a site map node.” If the custom provider is used in combination with the providers that ship with .NET, Key is required to be unique across all providers. In this implementation the Key holds the database primary key for posts and an alpha-numeric key for categories. This prevents a situation where the post primary key could equal the category primary key (both are integer identity fields).

Warning: There is a known issue using SiteMapNode.Key with the TreeView and Menu controls. The TreeView and Menu controls incorrectly pass a Key to FindSiteMapNode method instead of a URL. As a work around the SiteMapProvider shown here compares both Key and Url properties in the FindSiteMapNode method. More on this issue is available at <http://weblogs.asp.net/dannychen/archive/2005/10/17/427714.aspx>

Note: If you plan to base your custom provider on the code found in listing one know its limitations. The code assumes it is the only SiteMapProvider for an application, and that it only has one instance. To address these limitations and learn more about custom SiteMapProviders see the MSDN article [How to: Implement ASP.NET Site-Map Providers](http://msdn2.microsoft.com/en-us/library/ms178432.aspx) at <http://msdn2.microsoft.com/en-us/library/ms178432.aspx>

HttpContext.RewritePath

Once the SiteMapProvider is in place, we need to direct these URLs to our template pages. If you are experienced with ASP you may suggest Server.Transfer. This method will work for loading the template page but will fail handling a postback done with PostBackUrl. There is an argument, *preserveForm*, that claims to keep form data when true, however when combining Server.Transfer with PostBackUrl I've found that not be the case. Luckily we have an alternative: HttpContext.RewritePath.

HttpContext.RewritePath is not new. Its primary use is described in the (short) documentation as, “[A]ssigns an internal rewrite path and allows for the URL that is requested to differ from the internal path to the resource. RewritePath is used in cookieless session state.” RewritePath can be thought of as an in-line Server.Transfer that doesn't restart the page request cycle. Prior to 2.0, RewritePath changed both the virtual path and the physical path. In 2.0 the parameter *rebaseClientPath* was added to allow changing the physical path only. Setting *rebaseClientPath* to false we can direct the framework to load our template page while leaving the virtual path unchanged. This is very helpful as

sitemap-aware controls use the virtual path. To call RewritePath at the start of the request place the code in the Global.asax method Application_BeginRequest, as shown in Figure One.

Displaying the Content

Figure Two lists the Master Page code. In Page_Load the Master Page walks the sitemap and sets a page title that matches the URL. A TreeView control displays site navigation and a SiteMapPath control generates a page header, again matching the URL.

The category template page (Figure Three) uses a Menu control to list all posts in the category. The article template (Listing Two) displays the correct article by using the value in SiteMap.CurrentNode.Key. You can see from the simplicity in the code there are advantages beyond URL structure for developing a custom SiteMapProvider.

Handling Form Postbacks

The article template (Listing Two) displays comments and a DetailsView for visitors to leave new comments. This is mostly standard fare, except the “Leave Comment” LinkButton sets PostBackUrl to SiteMap.CurrentNode.Url. Without this, the page would post back to the physical URL and we would lose the primary key in SiteMap.CurrentNode.Key – not to mention exposing a page we do not want a search engine or visitor to see.

The documentation states when PostBackUrl is used, controls of the posting page are placed in Page.PreviousPage and Page.IsCrossPagePostBack is set to true. In our case this is not correct. When a PostBackUrl matches the virtual path, it is treated as a normal post back. Controls properly load their viewstate, Page.PreviousPage is null, and Page.IsCrossPagePostBack is false. While this makes form processing easy and be the intended behavior, you should note it is undocumented and may change in a future version of .NET.

Trimming the TreeView

The example is complete but it not scalable. With a real site there may be hundreds of articles, exploding the TreeView. One solution is to display only categories on the TreeView, but what if you want to have the five most recent articles per category in the TreeView and still have all articles listed on the category page? You could implement a TreeNodeDataBound to hide excess nodes, but there is another solution: have the SiteMapProvider lie.

The SiteMapProvider methods GetParentNode, GetChildNodes can paint different pictures of the sitemap. When GetChildNodes is called with a category node, limit the return to the first five children nodes. If a “hidden” node is passed to GetParentNode, return the category node safe in knowing the framework will not uncover the ruse (by throwing a site-crashing exception). The Menu control on the category page should be replaced with a GridView or other paging control to

manage the growing list of articles. This control can use a separate data source to directly access the database of articles, then use the SiteMapProvider static methods MakeURL to generate the links. Other options include a custom method in the SiteMapProvider to retrieve all articles, or sub-classing the custom SiteMapProvider and using multiple providers.

An important note when using GridView for navigation: do not assume a search engine spider follows the JavaScript paging links generated by GridView. Google recommends including a sitemap page that lists all links in the site to address this issue.

Yes, I think it's alright

The code required to generate friendly URLs is relatively small; a testament to design of the ASP.NET 2.0 Framework. Applying this to your own site will increase search engine visibility and build trust with site visitors. Besides articles, consider friendly URLs for product listings, department directories, and store locators.

There is more you can do to properly increase your search engine visibility. Many of the practices shown here are part of Google's [Webmaster Guidelines](http://www.google.com/webmasters/guidelines.html) at <http://www.google.com/webmasters/guidelines.html>

The code examples have been slimmed down to the minimum needed for demonstration. The full source code is available at www.aspnetPRO.com/download and includes both VB and C#.

Figure 1: Global.asax

```
<%@ Application Language="VB" %>
<script runat="server">
    Sub Application_BeginRequest(ByVal sender As [Object], _
                                ByVal e As EventArgs)
        Dim url As String
        url = Request.AppRelativeCurrentExecutionFilePath

        'Send the requests in the Articles folder
        'to the correct template page
        If url.StartsWith("~/Articles/") Then
            If url.EndsWith("/Default.aspx") Then
                HttpContext.Current.RewritePath( _
                    "~/Articles/Category.aspx", False)
            Else
                HttpContext.Current.RewritePath( _
                    "~/Articles/Article.aspx", False)
            End If
        End If

    End Sub
</script>
```

Figure 2: Masterpage.master

```
<%@ Master Language="VB" %>
<script runat="server">

    Protected Sub Page_Load(ByVal sender As Object, _
                            ByVal e As EventArgs)

        'Walk the sitemap backwards to build a page title
        Page.Title = SiteMap.CurrentNode.Title
        Dim smnCursor As _
            SiteMapNode = SiteMap.CurrentNode.ParentNode
        While Not (smnCursor Is Nothing)
            Page.Title = smnCursor.Title + ": " + Page.Title
            smnCursor = smnCursor.ParentNode
        End While

    End Sub

</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:SiteMapDataSource ID="SiteMapDataSource1"
            runat="server" StartingNodeUrl="~/Default.aspx" />
        <h1><asp:SiteMapPath ID="SiteMapPath1" runat="server"
            ParentLevelsDisplayed="1" PathSeparator=": " >
            <NodeTemplate><asp:Literal ID="Label1" runat="server"
                Text='<%# Eval("title") %>' /></NodeTemplate>
        </asp:SiteMapPath></h1>
        <asp:TreeView ID="TreeView1" runat="server"
            DataSourceID="SiteMapDataSource1"
            PopulateNodesFromClient="False" />
        <asp:contentplaceholder id="ContentPlaceHolder1"
            runat="server" />
    </form>
</body>
</html>
```

Figure 3: Category.aspx

```
<%@ Page Language="VB"
    MasterPageFile="~/MasterPage.master" %>
<asp:Content ID="Content1" Runat="Server"
    ContentPlaceHolderID="ContentPlaceHolder1">
    <asp:SiteMapDataSource ID="SiteMapDataSource1"
        runat="server" ShowStartingNode="False"
        StartFromCurrentNode="True" />
    <h2>Articles</h2>
    <asp:Menu ID="Menu1" runat="server"
        DataSourceID="SiteMapDataSource1">
    </asp:Menu>
```

</asp:Content>

Listing 1: Custom SiteMapProvider

```
Imports Microsoft.VisualBasic

Public Class PinballSiteMapProvider
    Inherits SiteMapProvider

    Private _smnRootAsSiteMapNode
    Private _smncCategoryAsSiteMapNodeCollection
    Private _dArticleAsHashtable
    Private lockobjAsNewObject()

    'Build the sitemap. Public so the application can
    'trigger a sitemap rebuild when an article is added
    Public Sub BuildNodes()
        SyncLock lockobj

            'Hard-coded site root
            _smnRoot = New SiteMapNode(Me, "PinballRoot", _
                "~/Default.aspx", "Pinball Wizard")
            _smncCategory = New SiteMapNodeCollection()
            _dArticle = New Hashtable

            'Select the articles from the database
            Dim taArticle As _
                New ContentTableAdapters.ArticleTableAdapter()
            Dim dtArticle As Content.ArticleDataTable = _
                taArticle.AllArticles()

            Dim row As Content.ArticleRow
            For Each row In dtArticle.Rows
                'Create a node for the article
                Dim smnNew As New SiteMapNode(Me, _
                    row.article_id.ToString(), _
                    MakeURL(row.title, row.category, row.post_date), _
                    row.title)

                If Not _dArticle.ContainsKey(row.code) Then
                    'Category does not exist, create a category node
                    Dim smnNewCat As New SiteMapNode(Me, row.code, _
                        MakeURL(row.category), row.category)
                    smnNew.ParentNode = smnNewCat
                    _smncCategory.Add(smnNewCat)
                    _dArticle(row.code) = New SiteMapNodeCollection()
                Else
                    smnNew.ParentNode = _
                        _dArticle(row.code)(0).ParentNode
                End If

                _dArticle(row.code).Add(smnNew)
            Next row
        End SyncLock
    End Sub
End Class
```

```
End Sub 'BuildNodes
```

```
'Internal method used to strip undesired characters from  
'a url. Used by MakeURL below.
```

```
Public Shared Function Sanitize(ByVal data As [String]) _  
    As [String]
```

```
    Dim allowed As [String] = _  
        "abcdefghijklmnopqrstuvwxyz0123456789."  
    Dim clean As [String] = ""
```

```
    Dim l As [Char]  
    For Each l In data.ToCharArray()  
        If allowed.Contains(l.ToString().ToLower()) Then  
            clean += l  
        Else  
            If l.Equals(" ") And (clean.Length > 0 And _  
                clean((clean.Length - 1)) <> "-") Then  
                clean += "-"  
            End If  
        End If  
    Next l  
    Return clean
```

```
End Function 'Sanitize
```

```
'Builds a URL for an article
```

```
Public Overloads Shared Function MakeURL( _  
    ByVal title As [String], _  
    ByVal category As [String], _  
    ByVal post_date As DateTime) As [String]  
    Dim url As [String] = ""  
    title = Sanitize(title)  
    category = Sanitize(category)  
  
    url = "~/Articles/" + category +  
        post_date.ToString("/yyy/MM/dd/") + title + ".aspx"  
  
    Return url  
End Function 'MakeURL
```

```
'Builds a URL for a category
```

```
Public Overloads Shared Function MakeURL( _  
    ByVal category As [String]) As [String]  
    Dim url As [String] = ""  
    category = Sanitize(category)  
  
    url = "~/Articles/" + category + "/Default.aspx"  
  
    Return url  
End Function 'MakeURL
```

```
'SiteMapProvider.Initialize, called during app start
```

```
Public Overrides Sub Initialize(ByVal name As String, _  
    ByVal attributes _  
    As System.Collections.Specialized.NameValueCollection)
```

```

    MyBase.Initialize(name, attributes)
    Me.BuildNodes()
End Sub 'Initialize

'SiteMapProvider.FindSiteMapNode, called to find a node
'by given URL. Note: TreeView and Menu pass a Key
'and not a URL, so we need to check both
Public Overrides Function FindSiteMapNode( _
    ByVal rawUrl As String) As SiteMapNode

    'Ensure our URL is relative
    Dim relUrl As [String] = rawUrl.Replace( _
        HttpRuntime.AppDomainAppVirtualPath, "~").ToLower()

    'Check the root
    If _smnRoot.Url.ToLower() = relUrl _
        Or _smnRoot.Key = rawUrl Then
        Return _smnRoot
    End If

    'Search the categories
    Dim smnCat As SiteMapNode
    For Each smnCat In _smncCategory
        If smnCat.Url.ToLower() = relUrl _
            Or smnCat.Key = rawUrl Then
            Return smnCat
        End If
    Next smnCat

    'Search the articles
    Dim smncArticle As SiteMapNodeCollection
    For Each smncArticle In _dArticle.Values
        Dim smnArticle As SiteMapNode
        For Each smnArticle In smncArticle
            If smnArticle.Url.ToLower() = relUrl _
                Or smnArticle.Key = rawUrl Then
                Return smnArticle
            End If
        Next smnArticle
    Next smncArticle

    'No match
    Return Nothing
End Function 'FindSiteMapNode

'SiteMapProvider.GetChildNodes, note we do not assume
'the node passed in may be a node we created, and
'use the Key instead. This is because a call to
'BuildNodes after app start will rebuild the lists
Public Overrides Function GetChildNodes( _
    ByVal node As SiteMapNode) As SiteMapNodeCollection

    'If this is the root, return the categories
    If _smnRoot.Key = node.Key Then
        Return _smncCategory
    End If

```

```

End If

'If this is a category, return the articles
If _dArticle.ContainsKey(node.Key) Then
    Return _dArticle(node.Key)
End If

'If this is an article, there are no children
Return Nothing
End Function 'GetChildNodes

'SiteMapProvider.GetParentNode, as with GetChildNodes
'we use the Key to compare
Public Overrides Function GetParentNode( _
    ByVal node As SiteMapNode) As SiteMapNode

    'Root has no parent
    If _smnRoot.Key = node.Key Then
        Return Nothing
    End If

    'If this is a category, return the root
    If _dArticle.ContainsKey(node.Key) Then
        Return _smnRoot
    End If

    'If this is an article, return it's category node
    Dim smncArticle As SiteMapNodeCollection
    For Each smncArticle In _dArticle.Values
        Dim smnArticle As SiteMapNode
        For Each smnArticle In smncArticle
            If smnArticle.Key = node.Key Then
                Return smnArticle.ParentNode
            End If
        Next smnArticle
    Next smncArticle

    'No match
    Return Nothing
End Function 'GetParentNode

'SiteMapProvider.GetRootNodeCore, return the root
Protected Overrides Function GetRootNodeCore() _
    As SiteMapNode
    Return _smnRoot
End Function 'GetRootNodeCore

End Class 'PinballSiteMapProvider

```

Listing 2: Article.aspx

```

<%@ Page Language="VB"
    MasterPageFile="~/MasterPage.master" %>
<script runat="server">

```

```

Protected Sub Page_Load(ByVal sender As Object, _
                        ByVal e As EventArgs)
    'Select the article content from the database
    Dim taArticle _
        As New ContentTableAdapters.ArticleTableAdapter()
    Dim daArticle _
        As Content.ArticleDataTable = _
        taArticle.ArticleByID( _
            Convert.ToInt32(SiteMap.CurrentNode.Key))

    'Display the article content
    lBody.Text = daArticle(0).body
    lDate.Text = daArticle(0).post_date.ToString("D")

    'Set the article_id for the comment data source
    odsComments.SelectParameters( _
        "article_id").DefaultValue = SiteMap.CurrentNode.Key

```

```
End Sub
```

```

Protected Sub dvAddComment_ItemInserted( _
    ByVal sender As Object, _
    ByVal e As DetailsViewInsertedEventArgs)

```

```

    'After a comment is inserted, referesh the comment list
    rComments.DataBind()

```

```
End Sub
```

```

Protected Sub dvAddComment_ItemInserting( _
    ByVal sender As Object, _
    ByVal e As DetailsViewInsertEventArgs)

```

```

    'Before adding the comment, set the article_id
    e.Values("article_id") = SiteMap.CurrentNode.Key

```

```
End Sub
```

```
</script>
```

```

<asp:Content ID="Content1" Runat="Server"
    ContentPlaceHolderID="ContentPlaceholder1">
    <asp:ObjectDataSource ID="odsComments" runat="server"
        InsertMethod="AddComment"
        SelectMethod="CommentsByArticle"
        TypeName="ContentTableAdapters.CommentTableAdapter">
        <SelectParameters>
            <asp:Parameter Name="article_id" Type="Int32" />
        </SelectParameters>
        <InsertParameters>
            <asp:Parameter Name="article_id" Type="Int32" />
            <asp:Parameter Name="author" Type="String" />
            <asp:Parameter Name="comment" Type="String" />
        </InsertParameters>
    </asp:ObjectDataSource>
    <b><asp:Literal ID="lDate" runat="server" /></b>

```

```

<asp:Literal ID="lBody" runat="server" />
<asp:Repeater ID="rComments" runat="server"
  DataSourceID="odsComments">
  <ItemTemplate>
    <b><asp:Literal ID="lAuthor" runat="server"
      Text='<%# Eval("author") %>' /> said on
    <asp:Literal ID="lCommentDate" runat="server"
      Text='<%# Eval("comment_date") %>' />:</b>
    <p><asp:Literal ID="lComment" runat="server"
      Text='<%# Eval("comment") %>' /></p>
  </ItemTemplate>
</asp:Repeater>
<h2>Leave a comment...</h2>
<asp:DetailsView ID="dvAddComment" runat="server"
  AutoGenerateRows="False" DataKeyNames="comment_id"
  DataSourceID="odsComments" DefaultMode="Insert"
  OnItemInserted="dvAddComment_ItemInserted"
  OnItemInserting="dvAddComment_ItemInserting">
  <Fields>
    <asp:BoundField DataField="author"
      HeaderText="Name" />
    <asp:BoundField DataField="comment"
      HeaderText="Comment" />
    <asp:TemplateField ShowHeader="False">
      <InsertItemTemplate>
        <asp:LinkButton ID="lbInsert" runat="server"
          CommandName="Insert" Text="Leave Comment"
         PostBackUrl='<%# SiteMap.CurrentNode.Url %>' />
      </InsertItemTemplate>
    </asp:TemplateField>
  </Fields>
</asp:DetailsView>
</asp:Content>

```