DotNetNuke Skinning Guide

Shaun Walker

Version 1.0.0
Last Updated: June 20, 2006
Category: Skinning

Information in this document, including URL and other Internet Web site references, is subject to change without notice. The entire risk of the use or the results of the use of this document remains with the user.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, places, or events is intended or should be inferred.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Perpetual Motion Interactive Systems, Inc. Perpetual Motion Interactive Systems may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Perpetual Motion, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

DotNetNuke® and the DotNetNuke logo are either registered trademarks or trademarks of Perpetual Motion Interactive Systems, Inc. in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Abstract

In order to clarify the intellectual property license granted with contributions of software from any person or entity (the "Contributor"), Perpetual Motion Interactive Systems Inc. must have a Contributor License Agreement on file that has been signed by the Contributor.

Contents

DotNetNuke Skinning Whitepaper

Introduction

" Traditionally, professional web design companies have relied on static HTML web sites for a substantial portion of their revenue. However, with the emergence of economical portal applications, the consumer has quickly become savvy to the benefits offered by dynamic web sites. In order to cope with this new demand, web designers need to become acquainted with the opportunities presented by these new technological advancements. The skinning architecture in DotNetNuke allows professional web designers to create stunning user interfaces which integrate seamlessly with the underlying portal application to produce a powerful, yet visually appealing customer web site."

The term "Skinning" refers to a software architecture which provides you with a manageable way to separate application logic and content from its presentation. This abstraction of "form" and "function" is sometimes referred to as a two-tiered presentation model. The advantages of skinning are that application logic and web page layout are separated and can be designed and modified independently. A software developer and a web page designer can collaborate without the risk of interfering with each others work.

One of the nice things about ASP - and one of the primary reasons for its popularity as a rapid web development (RAD) tool - is the fact that ASP script can be inserted into regular HTML markup to turn otherwise static HTML content into intelligent web pages. This feature makes it possible to quickly write ASP applications that build web pages dynamically from a database (or other external data source), and to create powerful web applications.

However, this ease of use comes with a price. Most ASP web sites contain a mangled spaghetti of intermingled HTML markup and ASP script, making them hard to decipher and even more difficult to maintain. This problem most commonly rears its ugly head when web designers need to alter the user interface. Since the presentation attributes are

embedded in the ASP script, changes typically require a joint effort between a web developer and a designer to ensure the integrity of the modification. This ultimately results in more effort, more time, and higher cost.

Skinning involves the use of "templates" which separate presentation and layout attributes from application logic. An intelligent skinning engine is then used to assemble the artifacts into a final product. This two-tiered approach affords both developers and designers a fair degree of independence when it comes to maintaining a web site, and can substantially reduce the time and effort required in the post-release phases of a development project.

However, skinning also introduces some complexities to your web application. The process of merging multiple files into a single page can lead to serious performance degradation. Not to mention, developers must be well educated to keep the presentation details of layout and appearance abstracted from their script code. Both of these items can be mitigated through the use of technology and a solid skinning architecture.

Terminology

The term “Skinning” has many different interpretations depending on the audience. At the most basic level, skinning provides you with a static layout but allows you change colors and styles and possibly override images with your own custom graphics. At the other extreme, skinning can allow you with the capability to customize every aspect of the user interface except for the actual content. There are obviously many stages in between these two extremities and your business requirements should drive the optimal solution for your application.

Windows Clients

Skinning techniques have long been popular in Windows client applications. This would lead you to believe that it should be a trivial effort to incorporate the same skinning architecture into your web applications. However, this approach would prove to be a tragic mistake.

Due to their stateful nature Windows client applications do not have the same requirements as a web application. A skin can be applied to a Windows client application using a wide variety of parsing techniques with little or no performance impact. This is because the user interface in a Windows client application is persistent for the lifetime of

the container. Compare this to a web application where the entire user interface is assembled and transmitted to the web browser on every page request. It is quite obvious that an extremely high performance rendering engine is required in a web application to meet these demands.

Technology

The abstraction of user interface elements from a page can be accomplished using many different strategies. Each strategy invariably includes some degree of parsing to merge the presentation with the business logic. Therefore, defining where, when, and how this parsing will take place becomes critical to the entire solution.

A popular technique employed in many script languages is to use tokens or identifiers in the user interface files to represent dynamic functionality. When the page is processed, the identifiers are replaced with the appropriate application logic. Regardless of the method chosen to perform the replacement (ie. intelligent parsing, string functions, regular expressions, etc...), the key point to emphasize in web application skinning is the phrase “when the page is processed”. If the replacement is being done on each page request, the performance of the application is going to be affected. To mitigate this performance impact, the optimal solution needs to take advantage of pre-processing or compilation technologies.

ASP.NET provides a very powerful feature for abstracting web pages into individual components. User Controls are similar to classic ASP #include directives... except on steroids. User controls allow you to isolate user interface functions and reuse them in other ASP.NET pages. A user control is almost identical to a normal .aspx page, with two differences: the user control has the .ascx extension rather than .aspx, and it may not have <HTML>, <Body>, or <Form> tags. User controls can be broken into multiple files, separating the presentation from the application logic or “code-behind” file. This feature allows multiple presentation files to be created which can all reference the same “code-behind” file. And due to the fact that ASP.NET is a compiled language, pages comprised of multiple user controls are assembled and rendered very efficiently – certainly faster than non-compiled or interpreted code.

Definition

Another very important aspect to consider in any skinning solution is the identification of the skin author. Who will be creating skins? What is their comfort level in terms of

technology? What design tools do they use in their business activities? Requirements analysis in this area plays a large part in determining how skins should be defined.

Although ASP.NET user controls provide us with a powerful core technology, they are a relatively new concept and are proprietary to the Microsoft platform. Some web design tools (ie. FrontPage, DreamWeaver, etc...) are now providing embedded support for user controls; however, this concept is still foreign to many professional web designers. Not to mention the fact that user controls contain references to proprietary ASP.NET server controls which should not be exposed to web designers. The risk of a web designer mangling or removing critical server controls is not acceptable. Ultimately we would like web designers to be able to use their tool of choice for creating skins and keep the gory details of user controls separated.

As a result the skinning solution needs a method for a web designer to define a skin using the simplest base technology. HTML is the lowest common denominator when it comes to the World Wide Web and has the widest support in terms of design tools. The only problem with HTML is that it is completely static - yet a web application has requirements for dynamic elements as well.

Revisiting the token solution described above, it would seem useful to allow web designers to include placeholders into their HTML markup which represent dynamic functionality. This provides the cleanest abstraction of “form” and “function”. The skin file could be modified without affecting the application logic. The application logic could be re-factored without affecting the user interface. The only issue is the performance penalty imposed by this additional layer of separation.

DotNetNuke

In versions 1.0.0 to 1.0.10, DotNetNuke contained a very simplistic skinning implementation. It allowed you to modify the logo, colors, and styles to produce a minimally customized website. It did not take long to realize that this implementation did not meet the presentation needs of the general community and various projects were initiated to implement a more robust skinning solution.

Unfortunately, the requirements were slow to evolve as many different people had their own opinions in regards to the optimal solution. Most of these opinions took the form of technical implementations and soon there were multiple skinning options available to

the community – each with their own specific strengths and weaknesses. An incredible amount of pressure was placed on DotNetNuke to select one of these solutions for inclusion into the core. But the fact remained that none of them represented a full implementation, encompassing all of the business requirements we were trying to achieve. And we had to be very careful about releasing a half-baked solution as we understood the issues in creating an incomplete standard and then trying to support it.

DotNetNuke 2.0 represented a ground-up rewrite of the core portal framework. This initially only included the data access layer and business logic layer but it stood to reason that the presentation layer also needed to be overhauled to allow us realize our goals. As a result, DotNetNuke now includes a robust skinning architecture which allows the clean separation of "form" and "function". The rest of this document focuses on the technical details of the implementation.

Overview

One of the guiding principles in DotNetNuke is simplicity. This principle has different interpretations in different application areas but in terms of skinning, the goal was to expose a complex architecture in a manner which was simple to use and manage. In addition, a critical aspect of any web application is performance; therefore, a great deal of emphasis was placed on this criteria as well. The good news is that simplicity often compliments performance and in the case of DotNetNuke skinning, this definitely proved to be the case.

In terms of base technology, the benefits of ASP.NET user controls were a clear winner in terms of template management. The fact that user controls are compiled provides a definite advantage over parsed or interpretive methods in terms of performance. The abstraction of the presentation from business logic in the "code-behind" is also a key feature.

To make the process of creating skins as simple and flexible as possible to web designers we decided to use pure HTML as the basis of the skin definition. This allows designers to use their tool of choice for creating and maintaining skins. We defined some placeholders to separate the skin objects from the static markup which designers can then include in their skin. Placeholders are simply [TOKEN] text which uniquely identify a skin object. To mitigate the performance impact of replacing these placeholders at runtime, we created a simple skin upload mechanism which does the substitution of placeholders with the skin objects to produce a user control file which can then be

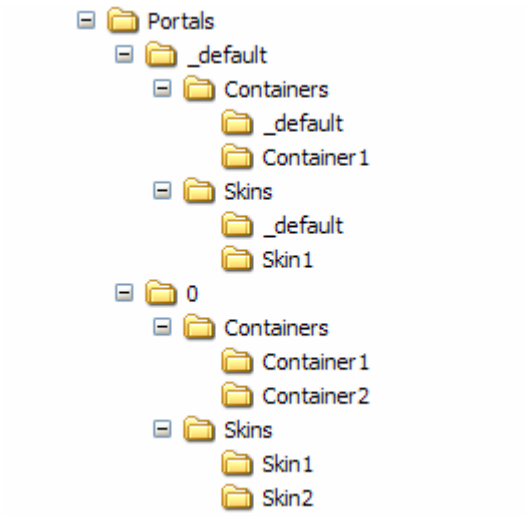
rendered by the engine. This pre-processing occurs only once when the skin is uploaded. This technique provides us with the performance benefit of user control skinning yet also includes the abstraction necessary for web designers to work independently from web developers.

In terms of layout, DotNetNuke uses a free-form skinning approach which allows the skin designer to organize the page in any layout they desire. For injection of content modules into the page, the designer can create an unlimited number of content panes which can be associated to the placement of modules in the database. This free-form approach offers the ultimate in design flexibility but also imposes some complications in terms of seamless plug-and-play skinning. For true plug-and-play, the layout, name, and quantity of content panes would have to consistent in each skin.

The Achilles Heel of any skinning solution is a component which attempts to define its own static layout or presentation. In DotNetNuke, content modules represent the biggest challenge to the skinning architecture. Depending on the complexity of the content module and the number of functions it attempts to expose in a single control, it is highly likely that it at least includes some embedded layout characteristics. This severely limits its usage in a skin depending on its footprint (ie. it may only fit inside of larger skin panes). Even worse is when a content module embeds static font, color, images, or styles as it severely reduces the effectiveness of the skinning solution.

File Organization

Skins can be applied at a host, portal, or tab level. Skins can also be applied at the module level; however, in this case we refer to them as Containers. All skin files are organized under the Portals folder. A special _default subfolder is used to designate the Host directory tree; whereas, each portal has its own directory tree named according to its ID in the database. This structure centralizes the “write-access” requirements to a single folder structure in your website. It also ties the physical organization of files with their logical usage within the application which means there is no external database required to manage the relationships.



Skins and Containers can contain an unlimited number of subfolders – each subfolder representing a package of files necessary to render a skin. Subfolders are stored according to skin name. Using a name may increase the chance of naming collisions between skins but it also allows you to manage your files directly on the file system without worrying about reconciling the changes to an external data source.

Page Processing

DotNetNuke uses a single ASPX page (Default.aspx) for rendering all controls and content. The benefit of this approach is two-fold. First it centralizes all of the logic for management of the user interface in a single page. Second it reduces the number of entry points into the application which results in a far more secure solution. The Default.aspx has very limited logic – it includes code for managing the page <HEAD> elements and includes a placeholder for injecting the skin.

When a user first enters the DotNetNuke application, it examines the URL and request header it receives from the user's browser, to determine which skin it should use. This process requires a database request to the Skins table where all of the assignments for skins and containers are stored. The assignments are stored in a hierarchical manner so that child assignments are able to override parent assignments (ie. a skin applied at the tab level should override a skin applied at the portal level). The performance implication of making a database request to retrieve the current skin assignment for the page and modules is mitigated by efficient data object caching within the core application.

Once the skin is identified, the associated user control is loaded dynamically and injected into the page placeholder. Every skin user control must reference a common skin.vb code-behind file stored in the \admin\Skins folder. This file performs all of the skin processing for managing security and injecting content.

Skin Package

A skin or container package is comprised of multiple files which constitute a complete skin:

- *.htm, *.html files – abstract skin definition files which will be processed by the skin uploader to create an *.ascx file
- *.ascx files – skin definition user controls which are precompiled in the format required by the skin engine.
- *.css files – styles sheets related to skins
- *.gif, *.jpg, *.jpeg, *.png – supporting graphics files
- *.* - any other resource files required for your skin (please note that the allowable file extensions are secured by the Host File Upload Extensions setting)

A skin package can contain multiple skin files. This allows you to create skins which leverage the same graphics but vary slightly based on layout. Obviously the more skin files you have in a package, the more maintenance will be required when you wish to make a general change to the presentation in the future.

Most “packaging” schemes employ the concept of a manifest file to identify the various files included in the package and define additional package attributes. Although this concept has benefits in terms of the metadata capability, it also represents another artifact which needs to be created and maintained for the life of the package. Maintaining our product focus, we felt the simplicity of zipping a group of files into a package far exceeded the benefit provided by introducing a manifest requirement.

Skin Definition

Skin definitions can be created using two different methods, HTML or ASCX (user controls). If you are a web designer with limited or no exposure to ASP.NET then the HTML option is best. On the other hand, if you are proficient in ASP.NET and plan on creating your skin in a tool such as VS.NET then ASCX is best. Basically the only difference between the two methods is the file extension of the skin definition file and the use of tokens versus actual user control tags (defined in Appendix A).

At a minimum there will likely be two skin files for each package – one which defines the layout of the public portal and one which defines the layout of the private admin area. The public portal has the ability to host multiple content controls in varied pane layouts whereas the private admin area can only host admin controls in a single pane per page.

Skin Creation

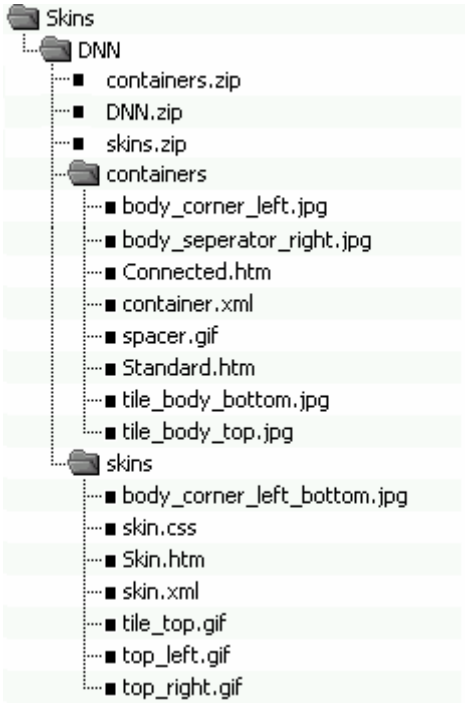
There is no particular order to this process, but the order below seems to work the best.

✧ 1. Set up your skin development environment

To simplify the development of skin files as well as expedite the packaging process later, it is recommended that you use the following organizational folder structure:

\Skins
\SkinName (this is the custom name for the skin package you are developing)
... (this is where you create the skin package ZIP files for deployment)
\containers (this is a static name to identify the container files for the skin package)
... (this will contain all resource files related to your containers)
\skins (this is a static name to identify the skin files for the skin package
... (this will contain all resource files related to your skins)

eg.



- Design your skin(s)

The free-form nature of skinning provides almost unlimited creative freedom with your design. Designers typically create the initial prototypes as full graphical images (possibly in PhotoShop or some other professional design tool). One thing to be aware of when creating skins is that you need to include all user interface elements in your design. This includes static elements such as graphics and text – but it should also include active elements such as Login links, a navigation/menu system, etc... Since this is a portal system which injects dynamic content at runtime, you will need to leave room in your design for this content.

Once the design is frozen, the designer will move onto the next step of "cutting-up" the image into an HTML document. This is where some of the more technical issues arise in terms of how to most effectively partition the various graphical elements into an HTML representation. HTML layout is much more "boxy" than free-flow graphics and the translation from a graphical image to an HTML document is critical to the presentation of the final product. Since HTML also has some of its own dynamic properties, you will need to tackle these behavioral issues as well. For

example, you will need to decide whether you want portions of the skin to scale according to the client browser screen resolution or remain as a fixed width.

- Choose an HTML or ASCX editor.

If you are comfortable with HTML tables and related HTML attributes, and can handle simple CSS settings, you can build a skin. A WYSIWYG editor makes it particularly easy. You can use FrontPage, HotMetal, ColdFusion, DreamWeaver, VS.NET... whichever editor you prefer - it's just HTML, after all.

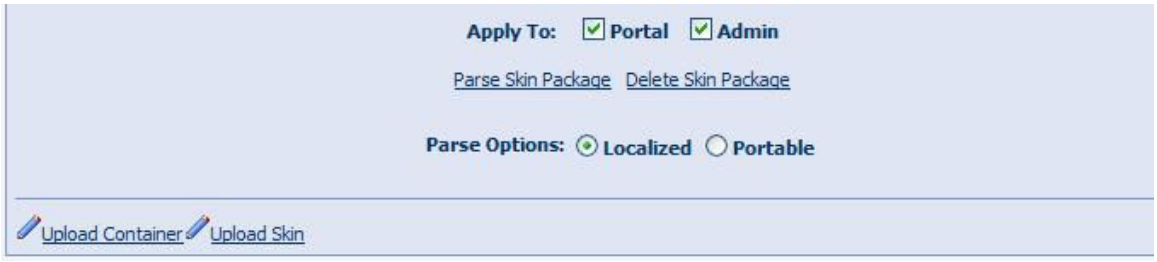
The HTML must be "well formed." That is, all HTML container tags must be closed. For instance, if you have a <TABLE> tag, it must be closed with a corresponding </TABLE> close tag. All tag attributes must have no spaces around the = signs, and attribute values must be in double quotes (ie. <TABLE WIDTH=”200”>... </TABLE>). Generally a professional HTML editor handles all this for you, but preferences can often be set for other renderings; therefore, it is best to double check.

- Manage graphics

Graphics are an integral part of any skin design. In order for the skinning engine to be able to locate your graphics files, you must specify the image references in a specific format. It is generally a good idea to place your graphics in the same folder as the skin file. However, this is not a requirement as the skinning engine will recognize any subfolders you specify.

In order for DotNetNuke to be able to locate your graphics files, The skinning engine uses two different methods to locate your graphics/media files: The “portable” method or the “Localized” method. By default, the Skin Uploader will try to make a “portable” skin file. This means that the Skin Uploader and parsing routines will add the <% = SkinPath %> tag the beginning or relative graphical references. This does require some extra resources at runtime, but the skin files are more portable and will transfer better to other server locations. For a more scalable setup, the skin parsing routine must add the explicit skin path to your relative URL references when it parses the files in your skin package. This explicit path specification results in the

best possible performance for loading skin graphics (since relative paths do not need to be determined at run-time for each request).



The Skin Uploader will manage the relative paths for the following HTML tags (contained in either HTML or ASCX files): IMG SRC, TD BACKGROUND, SCRIPT SRC. It will also manage the relative paths for the following style sheet (CSS) tag: BACKGROUND-IMAGE:URL(). In order to support the output of the widest variety of HTML editors, the order of the tag attribute specification is not important to the Skin Uploader. A previously uploaded skin package may be re-parsed at any time (sometimes useful during skin development) using the Parse Skin Package option at the bottom of the Skin Gallery.

DotNetNuke contains some graphics which are static and can not be customized as part of a skin package. These graphics represent global admin functions in the application and are stored in the /images folder of the site. In a fully flexible skinning architecture there should be absolutely no static graphics. That being said, there are always trade-offs between flexibility and performance, and in the case of DotNetNuke we chose the latter as the more critical criteria in this area. The use of static admin graphics has the following key benefits: 1) performance is preserved as the skin architecture can rely on the admin icons to be located in a centralized area – there is no file system query on an item-by-item basis to determine if an icon exists in a skin 2) skin package management is simplified since the skin only needs to contain the graphics which directly relate to the skin 3) disk space consumption is reduced as the admin icons do not need to be included in every skin package 4) DotNetNuke gets a consistent admin user interface experience – the admin icons will be the same for every skin which assists documentation and user training.

- Add skin objects and content panes

Skin objects are objects which will be rendered dynamically at runtime. If you are creating ASCX skins then you will need to specify the @Register and actual user control tag in your skin file (ie. <dnn:Login runat="server" id="dnnLogin" />). If you are creating HTML skins then you simply need to specify the token (ie. [LOGIN]). It is important to understand the functionality of each skin object so that you can design the skin accordingly (see Appendix B).

In addition to skin objects, there is also a concept of Content Panes. Content panes are the containers for the content modules which are automatically injected at runtime. Content panes are simply HTML tags with some custom attributes specified which allow them to interact with the DotNetNuke skinning engine. Allowable content pane HTML tags include <TD>, <DIV>, <SPAN>, and <P>. At a bare minimum you must have at least one content pane and it must be appropriately named “ContentPane”. Content panes do not need to be contiguous – they can be located anywhere on the page. Content panes are collapsible as well – this means that if they contain no content at runtime, that they will be become invisible. If you are creating HTML skins then you can use the [CONTENTPANE] skin object to identify areas in your design where content will be injected.

Skin objects support the concept of named instances. For example, if you want to have two menu controls in your skin, you can specify [MENU:1] and [MENU:2] in your skin file. In most cases you will only be using a single instance of a skin object in your skin and in this case it is fine to use a singleton instance specification (ie. [MENU]). Named instances are important for Content Panes because in most cases you will have more than one content pane defined in your skin. In this case you would use [CONTENTPANE:1], [CONTENTPANE:2], etc.... (although you would still need to have one pane named [CONTENTPANE] since this is the default skin pane).

Skin objects also contain a feature known as attributes. Attributes allow you to customize the appearance of the skin object in your skin. Each skin object has its own set of supported attributes which are documented in Appendix B. If you are creating ASCX skins then you will need to specify the attribute directly in your skin file (ie. <dnn:Login runat=”server” id=”dnnLogin” Text=”Signin” />). If you are creating HTML skins then you must include your attributes specifications in a separate file – this preserves the presentation of the HTMLskin file for the designer.

A skin package can contain a global attributes specification named “skin.xml” (or “container.xml” for containers) which applies to all skin files in the package. In addition, you can also override the global skin attribute specification with a skin specific attribute specification by providing a “skinfilename.xml” file. The skin uploader will merge the skin attributes with the HTML presentation file to create an ASCX skin file. The following XML fragment represents the structure of the attributes file:

```
<Objects>
   <Object>
      <Token>[LOGIN]</Token>
      <Settings>
         <Setting>
            <Name>Text</Name>
            <Value>Signin</Value>
         </Setting>
      </Settings>
   </Object>
</Objects>
```

Please note there is a one to one correspondence of skin object declarations in your skin file (ie. [MENU]) with the attribute specification in the XML file. This is also true for named instances. For example if you want to include a vertical and horizontal menu in your skin, you can specify [MENU:1] and [MENU:2] named instances in your skin file and then create definitions for each with different attributes in your XML file.

When creating HTML skins and specifying multiple Content Panes, you will need to specify the “ID” attribute in the attributes file.

```
<Objects>
   <Object>
      <Token>[CONTENTPANE:1]</Token>
      <Settings>
         <Setting>
            <Name>ID</Name>
            <Value>LeftPane</Value>
         </Setting>
      </Settings>
   </Object>
</Objects>
```

- Create a style sheet

  DotNetNuke uses an external style sheet (or CSS) specification which takes full advantage of their cascading nature. Essentially this means that DotNetNuke has multiple external style sheet references on a page – each style sheet reference is specified in prioritized order so that hierarchical overriding can occur. The cascading order of style sheets is summarized below (with each item overriding the previous items):

  - Modules – styles for custom modules defined in PortalModuleControl.StyleSheet
  - Default – default host level styles – default.css
  - Skin – skin styles – skin.css or skinfilename.css
  - Container – container styles – container.css or containerfilename.css
  - Portal – custom styles defined by portal Administrator – portal.css

A skin package can contain a global style sheet named “skin.css” (or “container.css” for containers) which applies to all skin files in the package. In addition, you can also override the global skin style sheet with a skin specific style sheet by providing a “skinfilename.css” file. The default DotNetNuke style sheet (/Portals/_default/default.css) contains a number of default CSS "classes" (the entries that start with a period) that the portal relies on for a consistent user interface experience. You are free to add your own styles but at a bare minimum you should override the default styles to match your skin design.

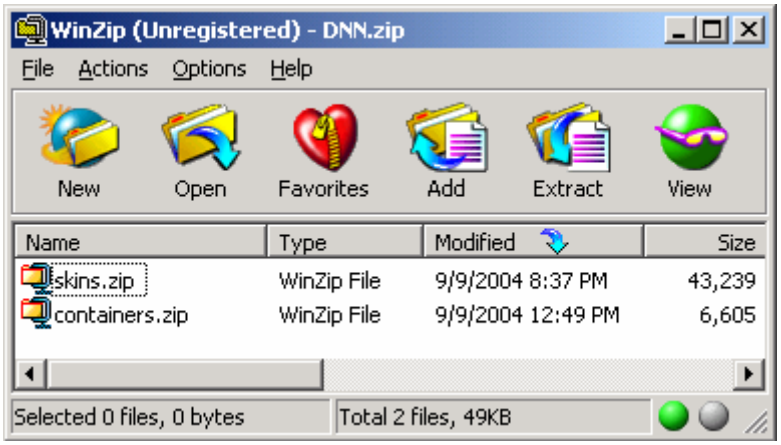
- Publish the skin

  In order for the skin to be viewable in the Skin Gallery, you need to create a high quality screen shot of your skin. For each skin or container source file you should also have a corresponding screen shot stored with a .JPG file extension (ie. if your skin file is named skin.html then your screen shot needs to be named skin.jpg).

- Package the skin

All of the files associated to a skin are packaged as a compressed *.zip file. If you use Windows XP, or have "Compressed Folders" installed in Windows ME, you can just right-click on the folder where you saved everything, choose "Send to >", and click "Compressed (zipped) folder." The operating system will ZIP it up for you, ready for upload. If you don't have one of these operating systems, use WinZIP or some other ZIP utility.

In many cases you will want to package a complementary set of skin files and container files in one distribution file. In order to do this you need to package your container files in a compressed *.zip file named “containers.zip”. Similarly, you must package your skin files in a compressed *.zip file name “skins.zip”. Then you need to package these 2 files into a single *.zip file which is named after your skin. This will allow people to install the full skin package (skins and containers) by uploading a single file through the Skin Uploader.



Optionally, you may also add an “About.htm” page to the skin package that will give info and credits about your entire Skin Package. A link to this file will be shown on the skin Gallery page when viewing your skin package.

Container Creation

As mentioned earlier, containers are skin definitions which can be applied to content modules. A container is defined in exactly the same manner as a skin except for the fact that there are a different set of skin objects used in containers.

The only extra restriction when creating containers is that an Actions control must be included in the container skin. The Actions control is a new feature in DotNetNuke which acts as the glue that ties the content module functionality to the portal framework. The Actions control is essentially a user interface control which exposes the content module functionality. General functions include the ability to edit module settings as well as the ability to manage the positioning of modules within skin content panes. Custom functions related to a specific module are also exposed, allowing you to edit content and navigate to other controls. There are a number of Actions controls included with DotNetNuke. The default actions control is the SolPartActions control which behaves as a popup menu when you hover over the edit icon in the upper left corner of the default container skin. Since this action control is best suited for higher level browsers, there is also a DropDownActions control which behaves like a simple dropdown combobox for downlevel browsers.

Although skins and containers are created, packaged, and deployed independently, it is very likely that you will create a skin and container combination which are intended to work together. Of course this can be accomplished by uploading both the skin and the container and then applying them to your portal user interface. To simply this operation and provide a higher degree of granularity, a concept known as Pane Level skinning is also available. Pane level skinning can only be configured at design-time when the skin designer constructs the skin. It involves the use of some custom attributes which can be included in the markup for the pane. The ContainerType, ContainerName, and ContainerSrc attributes can be used to identify a specific container to be used with all modules injected into the pane. In order for this to work correctly, the container must exist in the location specified otherwise the default container will be displayed.

```
<Objects>
   <Object>
      <Token>[CONTENTPANE:1]</Token>
      <Settings>
         <Setting>
            <Name>ID</Name>
            <Value>LeftPane</Value>
         </Setting>
         <Setting>
```

```
            <Name>ContainerType</Name>
            <Value>G</Value>
         </Setting>
         <Setting>
            <Name>ContainerName</Name>
            <Value>DNN</Value>
         </Setting>
         <Setting>
            <Name>ContainrSrc</Name>
            <Value>standard.ascx</Value>
         </Setting>
      </Settings>
   </Object>
</Objects>
```

In versions prior to 2.0, DotNetNuke contained a limited module container skin functionality which allowed the Administrator to wrap a content module in some HTML markup to provide a custom border or outline. This container concept had a variety of issues including the fact the supporting graphics were stored in the same portal upload folder, the associated HTML required parsing on every request, and the layout was completely static. Even with these limitations, the popularity of containers resulted in the creation and distribution of many container packages. And to help manage these containers, a third party enhancement was released which separated the container graphics into their own subfolders and provided a simple selection mechanism. The good news is that the conversion of these original containers to the new DotNetNuke skinning architecture is minimal (see Appendix C for more information).

Skin Upload

Since skins are based on ASCX files which are essentially executable once they are added to an ASPX page, there is some risk that malicious script could be inserted into the skin files – putting your entire installation in danger. For this reason, the Host has the ability to grant Skin Upload Permission to either the Host or Portal. The option is available when you login as the Host User and select the Host Settings option from the Host tab. If the option is set to Host then only the Host User is able to upload skins to the site. If the option is set to Portal (default), then the Administrator of the portal is able to upload their own skins without Host intervention.

The upload of files has been centralized in DotNetNuke to the File Manager tab on the Admin or Host menu. To upload skins to a specific portal, you must browse to the portal's URL, login, and then use the File Manager option in the Admin tab. To upload skins which are available to all portals, the Host should use the File Manager option in the Host tab. The File Manager has an option to Upload New File(s). Selecting this option displays the File Upload interface which allows you to upload your skin and container packages. Select the appropriate option from the upload file type options prior to uploading a package (the application must be able to distinguish between the various ZIP file packages). Please note that depending on the Skin Upload Permission defined above, some of the options may not be available to you.

The Skin Upload will unzip the skin package; creating the necessary folder and decompress the files. It will convert any HTML files to their ASCX user control equivalent by replacing the placeholder tokens with the actual ASP.NET script. This replacement is done according to the skin objects defined in the ModuleControls database table. ASCX user control files and CSS style sheet files will also be parsed to include the relative path references for graphics files. Please note that if you are installing a skin package which contains both skins and containers (containers.zip and skins.zip) then you should choose the Upload Skin Package option.

XCOPY Deployment

In DotNetNuke 3.0, a new feature was added which allows you to deploy a skin or container at the host level without having to login to the application. This is useful in scenarios where you are a Hoster and wish to provision a client account with a new skin, or possibly you would like to upload your skin via FTP rather than through the user interface, or perhaps you are developing a new skin and wish to test it locally. In order to use this feature, must copy your skin or container package file (*.zip) to the appropriate application directory. The application will immediately recognize the file and execute the necessary installation routine.

Skin Administration

Skins can be applied to your application at a variety of levels. A generic skin selection control is used to expose the available skins in various areas of the portal user interface. Each portal has access to their own skins as well as skins uploaded by the host. Skins are assigned in a hierarchical manner where child skins will override parent skin assignments. For example, a skin applied at the tab level will always override a skin assigned at the portal level.

Host Level

Host level skins apply to all portals in your site. They can be assigned by logging in as the SuperUser and selecting the Host / Host Settings tab. You are able to assign a skin and/or container for both the public portal and private admin interfaces.

Portal Level

Portal level skins apply to a specific portal. They can be assigned by logging in as the Administrator for a portal and selecting the Admin / Site Settings tab. You are able to assign a skin and/or container for both the public portal and private admin interfaces.

Tab Level

Tab level skins apply to a specific tab in a portal. They can be assigned by logging in as the Administrator for a portal and selecting the Edit Tab Settings option from the tab admin control. You are able to assign a skin and/or container for a tab.

Pane Level

Pane level skins are actually module containers which apply to a specific pane on a portal tab. They must be configured by the skin designer when creating the skin and cannot be managed through the portal user interface. Module level skins assigned to a specific module will override the Pane level skins.

Module Level

Module level skins are referred to as containers and apply to a specific content module instance on a tab. The can be assigned by logging in as the Administrator for the portal and selecting the Edit Module Settings option on the module Actions menu.

Skin Gallery

DotNetNuke has a Skin Gallery which allows Administrators to view a thumbnail image of a skin prior to applying it to your site. In order for a thumbnail to be displayed, the skin designer must create a high quality screen shot of the skin, save it with a .JPG extension, and package it with their skin package. By adding an about.htm file with their skin an informational credits page link will also be available in the skin gallery.

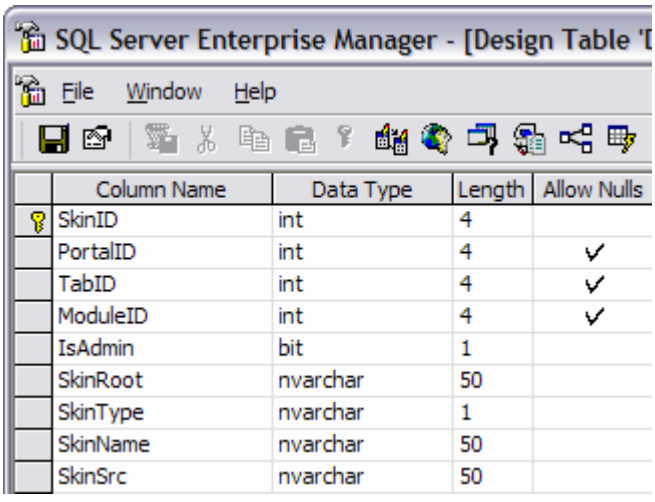
Control Panel

In order to achieve WYSIWYG usability, a Control Panel control is dynamically injected into the page when Administrators identify themselves. The Control Panel contains a variety of administrative options for managing tabs and modules. By default, the Control Panel is displayed as a horizontal bar at the top of your browser window. The placement of this bar at the top of the page is intended to minimize the distortion to the overall appearance of the page.

The Skinning solution also offers the ability for you to override the default Control Panel behaviour by including a pane named “ControlPanel” in your skin. This pane could be anywhere on your page – but you need to remember that the Control Panel control will only be displayed for the Administrator or Host. You also have the ability to create and leverage a custom Control Panel through the Host Settings.

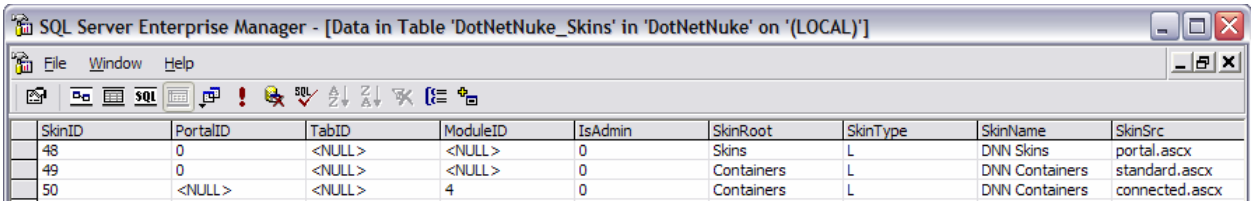
Data Model

The assignment of skins to various application levels must be persistent and is therefore stored in the database. To minimize the impact on core portal database tables, a separate mapping table was created with nullable foreign keys. The mapping table stores skin assignment information for both page skins and containers and can differentiate between public portal and private admin skins. The hierarchical relationship of Portal, Tab, and Module is represented in a flat format which allows for a single query to determine the appropriate skin assignment for any object.



	Column Name	Data Type	Length	Allow Nulls
🔑	SkinID	int	4	
	PortalID	int	4	✓
	TabID	int	4	✓
	ModuleID	int	4	✓
	IsAdmin	bit	1	
	SkinRoot	nvarchar	50	
	SkinType	nvarchar	1	
	SkinName	nvarchar	50	
	SkinSrc	nvarchar	50	

The data stored in the mapping table is fairly straightforward – the only complexity is in the ID area. A PortalID of NULL refers to a page skin defined at the host level. A PortalID which is not NULL refers to page skin for a specific portal. A TabID refers to a page skin for a specific tab. And a ModuleID refers to a container skin for a specific module. If no records exist in the mapping table, the application will use the default application skin and container (_default).



SkinID	PortalID	TabID	ModuleID	IsAdmin	SkinRoot	SkinType	SkinName	SkinSrc
48	0	<NULL>	<NULL>	0	Skins	L	DNN Skins	portal.ascx
49	0	<NULL>	<NULL>	0	Containers	L	DNN Containers	standard.ascx
50	<NULL>	<NULL>	4	0	Containers	L	DNN Containers	connected.ascx

Skin Objects

Skins Objects are active elements which can be included in the static HTML markup of your skin file to produce a dynamic user interface. There are a number of default skin objects included with DotNetNuke (outlined in Appendix B) for common portal functions such as login, status, and navigation. However, the fact is, any good technical solution falls short unless it provides at least a minimal degree of extensibility. Based on this requirement, a feature was added which allows you to create and install your own custom skin objects.

Custom skin objects are packaged and installed using the same process as Custom Modules (Private Assemblies). All of the necessary skin object resource files are combined with a DotNetNuke manifest file (* .dnn) and packaged into a compressed ZIP file.

A sample skin object created as a Private Assembly has been included in the default installation at /DesktopModules/PageTitle. The following *.dnn manifest file defines the package:

```
<?xml version="1.0" encoding="utf-8" ?>
<dotnetnuke version="2.0" type="SkinObject">
  <folders>
    <folder>
      <name>CompanyName - PageTitle</name>
      <modules>
        <module>
          <controls>
            <control>
              <key>PAGETITLE</key>
              <src>PageTitle.ascx</src>
              <type>SkinObject</type>
            </control>
          </controls>
        </module>
      </modules>
      <files>
        <file>
          <name>PageTitle.ascx</name>
        </file>
        <file>
          <name>YourCompanyName.PageTitle.dll</name>
        </file>
      </files>
    </folder>
  </folders>
</dotnetnuke>
```

Credits

The skinning solution implemented in DotNetNuke represents the combined efforts of many talented individuals. I would like to give credit to Bandar Naghi of HostIsIt Networks (http://www.naghi.net) – Multi-Layouts, Phil Beadle of Nexxus Systems International (http://www.nexxus.com.au) - XML Skins, David Haggard and Erik France of NewCovenant Evangelistic Ministries (http://www.newcovenant.com) – HTML Themes, Snapsis Dynamic Website Services (http://www.snapsis.com) – CSS Skins, Joe Brinkman of TAG Software (http://www.tag-software.net) – Module Title, and Steve Fabian of Gooddogs.Com (http://www.gooddogs.com/dnn) - Enhanced Containers. I would also like to acknowledge the SMARTY template engine (http://smarty.php.net/) for design concepts and inspiration. And finally I would like to thank the Microsoft ASP.NET project team for providing such a powerful and robust web application framework.

Appendix A: Sample Skin

The following section includes technical details on creating skins and containers for DotNetNuke. To demonstrate best practices we will create our skins as HTML to get the benefit of abstraction.

First we will create our page skin:

Is comprised of the following simple HTML markup (notice the use of [TOKEN]s for skin objects and content panes as well as the attribute specification for graphics files):

```
<TABLE style="BORDER-COLLAPSE: collapse" cellPadding="0" width="100%" height="100%"
border="0">
  <TR>
    <TD vAlign="top">
      <TABLE style="BORDER-COLLAPSE: collapse" cellPadding="0" width="100%" border="0">
        <TR>
          <TD background="tile top.gif" vAlign="top" height="10"><IMG src="top left.gif"
height="10" width="10" border="0"></TD>
          <TD background="tile_top.gif" vAlign="top" align="right" colSpan="2"
height="10"><IMG src="top_right.gif" height="10" width="10" border="0"></TD>
        </TR>
        <TR>
          <TD vAlign="bottom" width="100%" colSpan="2">
            <TABLE style="BORDER-COLLAPSE: collapse" cellPadding="0" width="100%"
border="0">
              <TR>
                <TD vAlign="top" width="100%"><P dir="ltr"><IMG src="header_logo1.jpg"
height="35" width="130" border="0"></P></TD>
                <TD vAlign="top" noWrap width="181"><IMG src="header_metalbar_top.jpg"
height="35" width="181" border="0"></TD>
              </TR>
              <TR>
                <TD colSpan="2"><IMG src="header_logo2.jpg" height="13" width="243"
border="0"></TD>
              </TR>
            </TABLE>
          </TD>
          <TD>
            <TABLE style="BORDER-COLLAPSE: collapse" cellPadding="0" width="100%"
border="0">
              <TR>
                <TD vAlign="top" align="right" bgColor="#ffffff" colSpan="3"><IMG
src="login_line_right.gif" height="10" width="60" border="0"></TD>
              </TR>
              <TR>
                <TD rowSpan="3"><IMG src="login_lefttop.gif" height="38" width="27"
border="0"></TD>
                <TD background="tile_login_top.gif" height="7"><IMG src="spacer.gif"
height="7" width="7" border="0"></TD>
                <TD rowSpan="3"><IMG src="login_righttop.gif" height="38" width="61"
border="0"></TD>
              </TR>
              <TR>
                <TD noWrap bgColor="#000000" height="25"><P><font
color="#FFFFFF">[USER]  <span
class="OtherTabs">|</span>  [LOGIN]</font></P></TD>
              </TR>
              <TR>
                <TD background="tile login bottom.gif" height="6"><IMG src="spacer.gif"
height="6" width="6" border="0"></TD>
              </TR>
            </TABLE>
          </TD>
        </TR>
      </TABLE>
    </TD>
```

```
  </TR>
  <TR>
    <TD vAlign="top">
      <TABLE style="BORDER-COLLAPSE: collapse" cellPadding="0" bgColor="#ae2726"
border="0">
        <TR>
          <TD background="tile_main.jpg" vAlign="top" width="100%"><IMG
src="header_logo3.jpg" height="74" width="620" border="0"></TD>
          <TD align="right"><IMG src="header_right.jpg" height="74" width="61"
border="0"></TD>
        </TR>
      </TABLE>
    </TD>
  </TR>
  <TR>
  <TD vAlign="top">
      <TABLE style="BORDER-COLLAPSE: collapse" cellPadding="0" width="100%"
bgColor="#000000" border="0">
        <TR>
          <TD vAlign="top" noWrap width="369"><IMG src="menu_left.jpg" height="40"
width="369" border="0"></TD>
          <TD background="tile_menu.jpg" nowrap width="100%"><font
color="#FFFFFF">[SOLPARTMENU]</font></TD>
          <TD vAlign="top" noWrap width="61"><IMG src="menu right.jpg" height="40"
width="61" border="0"></TD>
        </TR>
        <TR>
          <TD vAlign="top" noWrap width="369"><IMG src="breadcrumbs_left.jpg" height="33"
width="369" border="0"></TD>
          <TD width="100%" nowrap bgColor="#000000"><font
color="#FFFFFF">[BREADCRUMB]</font></TD>
          <TD vAlign="top" noWrap width="61"><IMG src="breadcrumbs_right.jpg" height="33"
width="61" border="0"></TD>
        </TR>
        <TR>
          <TD vAlign="top" noWrap bgColor="#FFFFFF" width="369">  <A
HREF="/default.aspx?tabid=125"><IMG src="download.gif" height="32" width="191" border="0"
alt="Download the Code!"></A></TD>
          <TD width="100%" nowrap bgColor="#FFFFFF"> </TD>
          <TD vAlign="top" noWrap bgColor="#FFFFFF" width="61"> </TD>
        </TR>
      </TABLE>
    </TD>
  </TR>
  <TR>
    <TD height="100%" valign="top">
      <TABLE style="BORDER-COLLAPSE: collapse" cellPadding="0" width="100%" border="0">
        <TR>
          <TD vAlign="top" align="middle" colspan="3">[CONTENTPANE:3]</TD>
        </TR>
        <TR>
          <TD vAlign="top" align="middle">[CONTENTPANE:1]</TD>
          <TD vAlign="top" align="middle">[CONTENTPANE]</TD>
          <TD vAlign="top" align="middle">[CONTENTPANE:2]</TD>
        </TR>
        <TR>
          <TD vAlign="top" align="middle" colspan="3">[CONTENTPANE:4]</TD>
        </TR>
      </TABLE>
      <BR>
    </TD>
  </TR>
  <TR>
```

```
    <TD vAlign="bottom">
      <TABLE style="BORDER-COLLAPSE: collapse" cellPadding="0" width="100%" border="0">
        <TR>
          <TD align="middle" bgColor="#000000" colSpan="3" height="25"><font
color="#FFFFFF">[LINKS]</font></TD>
        </TR>
        <TR>
          <TD background="tile_footer.jpg" align="middle" colSpan="3" height="25"><font
color="#FFFFFF">[HOSTNAME]  <span
class="OtherTabs">|</span>  [TERMS]  <span
class="OtherTabs">|</span>  [PRIVACY]</font></TD>
        </TR>
        <TR>
          <TD background="tile_body_bottom.jpg" vAlign="top" align="left"><IMG
src="body_corner_left_bottom.jpg" height="26" width="20" border="0"></TD>
          <TD background="tile_body_bottom.jpg" align="middle"><font
color="#000000">[DOTNETNUKE]</font></TD>
          <TD background="tile body bottom.jpg" vAlign="top" align="right"><IMG
src="body_corner_right_bottom.jpg" height="26" width="20" border="0"></TD>
        </TR>
      </TABLE>
    </TD>
  </TR>
</TABLE>
```

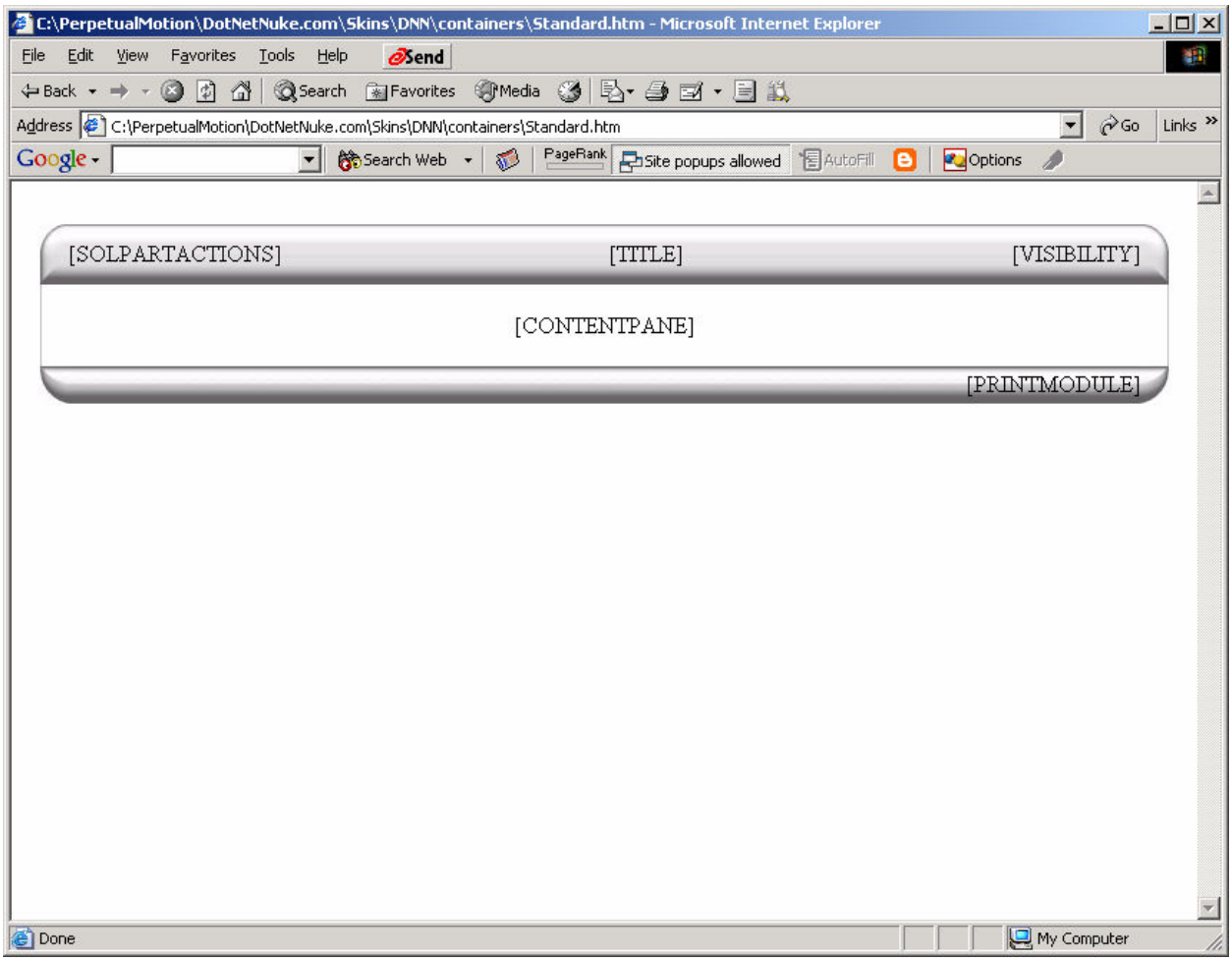
And the attributes file (skin.xml):

```
<Objects>
   <Object>
      <Token>[LINKS]</Token>
      <Settings>
         <Setting>
            <Name>Separator</Name>
            <Value><![CDATA[  |  ]]></Value>
         </Setting>
      </Settings>
   </Object>
   <Object>
      <Token>[CONTENTPANE:1]</Token>
      <Settings>
         <Setting>
            <Name>ID</Name>
            <Value>LeftPane</Value>
         </Setting>
      </Settings>
   </Object>
   <Object>
      <Token>[CONTENTPANE:2]</Token>
      <Settings>
         <Setting>
            <Name>ID</Name>
            <Value>RightPane</Value>
         </Setting>
      </Settings>
   </Object>
   <Object>
      <Token>[CONTENTPANE:3]</Token>
      <Settings>
         <Setting>
```

```
            <Name>ID</Name>
            <Value>TopPane</Value>
         </Setting>
      </Settings>
   </Object>
   <Object>
      <Token>[CONTENTPANE:4]</Token>
      <Settings>
         <Setting>
            <Name>ID</Name>
            <Value>BottomPane</Value>
         </Setting>
      </Settings>
   </Object>
</Objects>
```

We package our HTML skin file as well as all supporting graphics files into a single ZIP file.

Now we will create a container to match our skin:

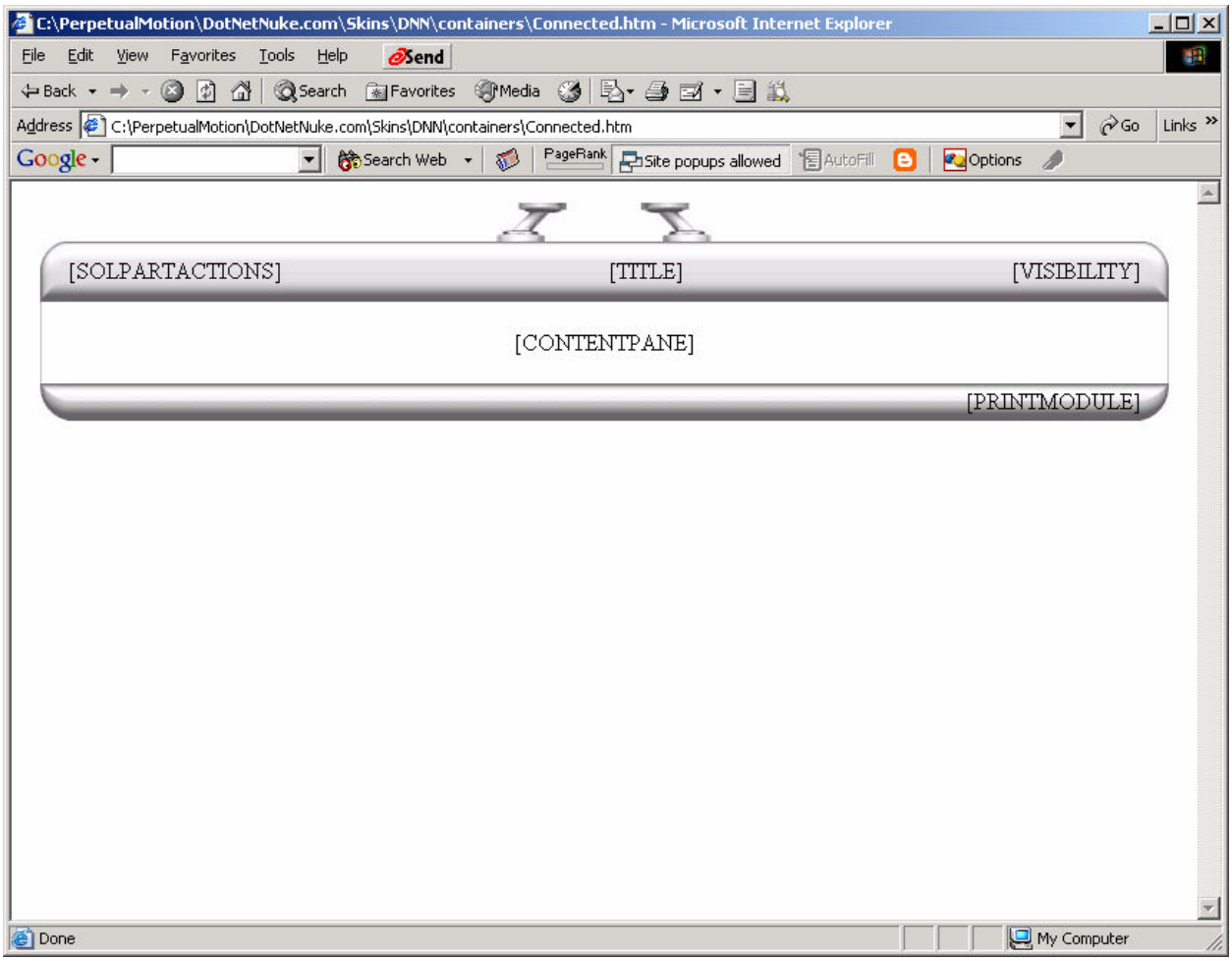


And the associated HTML markup:

```
<TABLE style="BORDER-COLLAPSE: collapse" cellPadding="0" width="100%" border="0">
  <TR>
    <TD vAlign="top" noWrap align="middle" colSpan="9"><IMG src="spacer.gif" height="15"
width="1" border="0"></TD>
  </TR>
  <TR>
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif" height="1" width="10"
border="0"></TD>
    <TD vAlign="top" noWrap width="20" colspan="2"><IMG src="body_corner_left.jpg"
height="42" width="20" border="0"></TD>
    <TD background="tile_body_top.jpg" align="middle" noWrap>[SOLPARTACTIONS]</TD>
    <TD background="tile body top.jpg" align="middle" width="100%" nowrap>[TITLE]</TD>
    <TD background="tile_body_top.jpg" align="middle" noWrap>[VISIBILITY]</TD>
    <TD vAlign="top" noWrap align="right" width="20" colspan="2"><IMG
src="body_corner_right.jpg" height="42" width="20" border="0"></TD>
```

```
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif" height="1" width="10"
border="0"></TD>
  </TR>
  <TR>
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif" height="1" width="10"
border="0"></TD>
    <TD vAlign="top" noWrap width="1" bgColor="#c0c0c0"><IMG src="spacer.gif" height="1"
width="1" border="0"></TD>
    <TD vAlign="top" noWrap width="19"> </TD>
    <TD colspan="3" align="middle"><br>[CONTENTPANE]</TD>
    <TD vAlign="top" noWrap width="19"> </TD>
    <TD vAlign="top" noWrap width="1" bgColor="#c0c0c0"><IMG src="spacer.gif" height="1"
width="1" border="0"></TD>
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif" height="1" width="10"
border="0"></TD>
  </TR>
  <TR>
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif" height="1" width="10"
border="0"></TD>
    <TD vAlign="top" noWrap width="1" bgColor="#c0c0c0"><IMG src="spacer.gif" height="1"
width="1" border="0"></TD>
    <TD colspan="5"> </TD>
    <TD vAlign="top" noWrap width="1" bgColor="#c0c0c0"><IMG src="spacer.gif" height="1"
width="1" border="0"></TD>
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif" height="1" width="10"
border="0"></TD>
  </TR>
  <TR>
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif" height="1" width="10"
border="0"></TD>
    <TD vAlign="top" noWrap width="20" colspan="2"><IMG src="body corner left bottom.jpg"
height="26" width="20" border="0"></TD>
    <TD background="tile_body_bottom.jpg" colspan="3" align="right">[PRINTMODULE]</TD>
    <TD vAlign="top" noWrap align="right" width="20" colspan="2"><IMG
src="body corner right bottom.jpg" height="26" width="20" border="0"></TD>
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif"height="1" width="10"
border="0"></TD>
  </TR>
</TABLE>
```

And a second container for some variety:



And the associated HTML markup:

```
<TABLE style="BORDER-COLLAPSE: collapse" cellPadding="0" width="100%" border="0">
  <TR>
    <TD vAlign="top" noWrap align="middle" colSpan="9"><IMG src="body_seperator_left.jpg"
height="27" width="50" border="0"><IMG src="spacer.gif" height="20" width="50"
border="0"><IMG src="body_seperator_right.jpg" height="27" width="50" border="0"></TD>

  </TR>
  <TR>
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif" height="1" width="10"
border="0"></TD>
    <TD vAlign="top" noWrap width="20" colspan="2"><IMG src="body_corner_left.jpg"
height="42" width="20" border="0"></TD>
    <TD background="tile_body_top.jpg" align="middle" noWrap>[SOLPARTACTIONS]</TD>
    <TD background="tile_body_top.jpg" align="middle" width="100%" nowrap>[TITLE]</TD>
    <TD background="tile_body_top.jpg" align="middle" noWrap>[VISIBILITY]</TD>
```

```
    <TD vAlign="top" noWrap align="right" width="20" colspan="2"><IMG
src="body_corner_right.jpg" height="42" width="20" border="0"></TD>
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif" height="1" width="10"
border="0"></TD>
  </TR>
  <TR>
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif" height="1" width="10"
border="0"></TD>
    <TD vAlign="top" noWrap width="1" bgColor="#c0c0c0"><IMG src="spacer.gif" height="1"
width="1" border="0"></TD>
    <TD vAlign="top" noWrap width="19"> </TD>
    <TD colspan="3" align="middle"><br>[CONTENTPANE]</TD>
    <TD vAlign="top" noWrap width="19"> </TD>
    <TD vAlign="top" noWrap width="1" bgColor="#c0c0c0"><IMG src="spacer.gif" height="1"
width="1" border="0"></TD>
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif" height="1" width="10"
border="0"></TD>
  </TR>
  <TR>
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif" height="1" width="10"
border="0"></TD>
    <TD vAlign="top" noWrap width="1" bgColor="#c0c0c0"><IMG src="spacer.gif" height="1"
width="1" border="0"></TD>
    <TD colspan="5"> </TD>
    <TD vAlign="top" noWrap width="1" bgColor="#c0c0c0"><IMG src="spacer.gif" height="1"
width="1" border="0"></TD>
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif" height="1" width="10"
border="0"></TD>
  </TR>
  <TR>
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif" height="1" width="10"
border="0"></TD>
    <TD vAlign="top" noWrap width="20" colspan="2"><IMG src="body_corner_left_bottom.jpg"
height="26" width="20" border="0"></TD>
    <TD background="tile body bottom.jpg" colspan="3" align="right">[PRINTMODULE]</TD>
    <TD vAlign="top" noWrap align="right" width="20" colspan="2"><IMG
src="body corner right bottom.jpg" height="26" width="20" border="0"></TD>
    <TD vAlign="top" noWrap width="10"><IMG src="spacer.gif"height="1" width="10"
border="0"></TD>
  </TR>
</TABLE>
```

We package our HTML container file as well as all supporting graphics files into a ZIP file.

After performing our skin upload, the system has pre-processed the HTML files into ASCX user controls which we can then select them in the Admin / Site Settings user interface:

And we can modify the Module Settings to override the skin assignment for the Sponsors module to use the "connected" container (look under the Page Settings – Basic Settings section):

To produce our new skinned portal:

Appendix B: Skin Objects

The following tables contain the default skin objects which can be used in your skins.

Skin Objects (stored in the ModuleControls database table)

Token	Control	Description
[BANNER]	< dnn:Banner runat="server" id="dnnBanner">	Displays a random banner ad
[BREADCRUMB]	< dnn:Breadcrumb runat="server" id="dnnBreadcrumb">	Displays the path to the currently selected tab in the form of TabName1 > TabName2 > TabName3
[CONTENTPANE]		Injects a placeholder for module content
[COPYRIGHT]	< dnn:Copyright runat="server" id="dnnCopyright">	Displays the copyright notice for the portal
[CURRENTDATE]	< dnn:CurrentDate	Displays the current date

Token	Control	Description
	runat="server" id="dnnCurrentDate">	
[DOTNETNUKE]	< dnn:DotNetNuke runat="server" id="dnnDotnetNuke">	Displays the Copyright notice for DotNetNuke (not required)
[HELP]	< dnn:Help runat="server" id="dnnHelp">	Displays a link for Help which will launch the users email client and send mail to the portal Administrator
[HOSTNAME]	< dnn:HostName runat="server" id="dnnHostName">	Displays the Host Title linked to the Host URL
[LANGUAGE]	<dnn:Language runat="server" id="dnnLanguage" />	Displays the language selector drop down list
[LINKS]	< dnn:Links runat="server" id="dnnLinks">	Displays a flat menu of links related to the current tab level and parent node. This is useful for search engine spiders and robots
[LOGIN]	< dnn:Login runat="server" id="dnnLogin">	Dual state control – displays “Login” for anonymous users and “Logout” for authenticated users.

Token	Control	Description
[LOGO]	< dnn:Logo runat="server" id="dnnLogo">	Displays the portal logo
[PRIVACY]	< dnn:Privacy runat="server" id="dnnPrivacy">	Displays a link to the Privacy Information for the portal
[SEARCH]	< dnn:Search runat="server" id="dnnSearch">	Displays the search input box
[SIGNIN]	< dnn:Signin runat="server" id="dnnSignin">	Displays the signin control for providing your username and password.
[SOLPARTMENU]	< dnn:SolPartMenu runat="server" id="dnnSolPartMenu">	Displays the hierarchical navigation menu (formerly [MENU])
[TERMS]	< dnn:Terms runat="server" id="dnnTerms">	Displays a link to the Terms and Conditions for the portal
[TREEVIEWMENU]	< dnn:TreeViewMenu runat="server" id="dnnTreeViewMenu">	Display a navigation Menu using the DNN Treeview Control to provide a Windows Explore like Menu.

Token	Control	Description
[USER]	< dnn:User runat="server" id="dnnUser">	Dual state control – displays a “Register” link for anonymous users or the users name for authenticated users.

Skin Attributes (used in Skin.xml)

Token	Attribute	Default	Description
[BANNER]	BorderWidth	0	The border width around the banner
[BREADCRUMB]	Separator	breadcrumb.gif	The separator between breadcrumb links. This can include custom skin images, text, and HTML (ie. <![CDATA[<img src=”file.gif”>]]>)
	CssClass	SelectedTab	The style name of the breadcrumb links
	RootLevel	1	The root level of the breadcrumb links. Valid values include: -1 - show word “Root” and then all breadcrumb tabs

Token	Attribute	Default	Description
			0 - show all breadcrumb tabs n (where n is an integer greater than 0) - skip n breadcrumb tabs before displaying
[CONTENTPANE]	ID	ContentPane	The content pane key identifier to be displayed in the user interface and stored in the database.
[COPYRIGHT]	CssClass	SelectedTab	The style name of portal copyright link
[CURRENTDATE]	CssClass	SelectedTab	The style name of date text
	DateFormat	MMMM dd, yyyy	The format of the date text
[DOTNETNUKE]	CssClass	Normal	The style name of DotNetNuke portal engine copyright text
[HELP]	CssClass	OtherTabs	The style name of help link
[HOSTNAME]	CssClass	OtherTabs	The style name of Host link (Powered By xxxxxxxxx)

Token	Attribute	Default	Description
[LANGUAGE]	CssClass		The style name of the drop down list
[LINKS]	CssClass	CommandButton	The style name of the links
	Separator		The separator between links. This can include custom skin images, text, and HTML. (ie. <![CDATA[<img src=”file.gif”>]]>)
	Alignment	Horizontal	The links menu style (“Horizontal” or “Vertical”)
	Level	Same	Determines the menu level to display (“Same”, “Child”, “Parent”, “Root”)
[LOGIN]	Text	Login	The text of the login link
	CssClass	OtherTabs	The style of the login link
	LogoffText	Logoff	The text for the logoff link
[LOGO]	BorderWidth	0	The border width around the logo

Token	Attribute	Default	Description
[PRIVACY]	Text	Privacy Statement	The text of the privacy link
	CssClass	OtherTabs	The style name of privacy link
[SEARCH]	Submit	Submit	HTML to activate the search lookup (i.e “Search” or “Go” or <img src=“mybutton.gif”>
	CssClass		Css class for the search control
[SIGNIN]			
[SOLPARTMENU]	separatecss	true	Use CSS defined in a style sheet (values: true, false)
	backcolor	#333333	Background color
	forecolor	white	Fore color of menu item when selected
	highlightcolor	white	Color of top and left border to give a highlight effect
	iconbackgroundcolor	#333333	Background color in area

Token	Attribute	Default	Description
			where icon is displayed
	selectedbordercolor		Color of border surrounding selected menu item
	selectedcolor	#CCCCCC	Background color of menu item when selected
	selectedforecolor	white	Fore color of menu item when selected
	display	horizontal	Determines how the menu is displayed, horizontal or vertical (values: vertical, horizontal)
	menubarheight	16	Menu bar height in pixels
	menuborderwidth	1	Menu border width in pixels
	menuitemheight	21	Menu item height in pixels
	forcedownlevel	false	Flag to force the downlevel menu to display (values: true, false)

Token	Attribute	Default	Description
	moveable	false	Flag to ☐etermine if menu can be moved (values: true, false)
	iconwidth	0	Width of icon column in pixels
	menueffectsshadowcolor	dimgray	Color of the shadow
	menueffectsmouseouthidedelay	500	Number of milliseconds to wait until menu is hidden on mouse out. (0 = disable)
	mouseouthidedelay	1	Number of milliseconds to wait until menu is hidden on mouse out. (0 = disable)
	menueffectsmouseoverdisplay	Highlight	Adjusts effect when mouse moves over menu bar item (Values: Outset, Highlight, None)
	menueffectsmouseoverexpand	true	Makes menu expand on mouse over (unlike any menu found within the Windows environment) (Values: true, false)

Token	Attribute	Default	Description
	menueffectsstyle	filter:progid:DXImageTransform.Microsoft.Shadow(color='DimGray', Direction=135, Strength=3) ;	IE only property for SubMenu styles and transitions
	fontnames	Arial	
	fontsize	12	
	fontbold	false	
	menueffectsshadowstrength	3	Determines how many pixels the shadow extends
	menueffectsmenutransition	None	Determines which direction the shadow will fall (Values: None, AlphaFade, AlphaFadeBottomRight, Barn, Blinds, Checkerboard, ConstantWave, Fade, GradientWipe, Inset, Iris, RadialWipe, Random, RandomBars, Slide, Spiral, Stretch, Strips, Wave, Wheel, Zigzag)

Token	Attribute	Default	Description
	menueffectsmenutransitionlength	0.3	Number of seconds the transition will take
	menueffectsshadowdirection	Lower Right	Determines which direction the shadow will fall (Values: None, Top, Upper Right, Right, Lower Right, Bottom, Lower Left, Left, Upper Left)
	menucontainercssclass	MainMenu_MenuContainer	Menu Container CSS Class
	menubarcssclass	MainMenu_MenuBar	Menu Bar CSS Class
	menuitemcssclass	MainMenu_MenuItem	Menu Item CSS Class
	menuiconcssclass	MainMenu_MenuIcon	Menu Icon CSS Class
	menuitemselcssclass	MainMenu_MenuItemSel	Menu Item CSS Class for mouse-over
	menubreakcssclass	MainMenu_MenuBreak	Menu Break CSS Class

Token	Attribute	Default	Description
	submenucssclass	MainMenu_SubMenu	SubMenu CSS Class
	menuarrowcssclass	MainMenu_MenuArrow	Menu Arrow CSS Class
	menurootarrowcssclass	MainMenu_MenuRootArrow	Menu Root Arrow CSS Class
	forcefullmenulist	false	Displays the full menu as an indented list of normal hyperlinks (like a sitemap) {true|false}
	useskinpatharrowimages	false	Use arrow images located in the skin and not those in the /images folder {true|false}
	userootbreadcrumbarrow	true	Use a breadcrumb arrow to identify the root tab that is listed in the breadcrumb array list {true|false}
	usesubmenubreadcrumbarrow	false	Use a breadcrumb arrow to identify the submenu tabs that are listed in the breadcrumb array list {true|false}

Token	Attribute	Default	Description
	rootbreadcrumbarrow		image used for root level menu breadcrumb arrows – i.e file.gif
	submenubreadcrumbarrow		image used for submenu menu breadcrumb arrows – i.e file.gif
	usearrows		use arrows to indicate child submenus
	downarrow	menu_down.gif	arrow image used for downward facing arrows indicating child submenus
	rightarrow	breadcrumb.gif	arrow image used for right facing arrows indicating child submenus
	level	Root	Root level of the menu in relationship to the current active tab {Root|Same|Child}
	rootonly	false	indicator to turn off submenus {true|false}
	rootmenuitembreadcrumbcssclass		CSS Class used for root menu items when they are found in

Token	Attribute	Default	Description
			the breadcrumb array list
	submenuitembreadcrumbcssclass		CSS Class used for sub menu items when they are found in the breadcrumb array list
	rootmenuitemcssclass		CSS Class used for root menu items
	rootmenuitemactivecssclass		CSS Class used for root menu items when they are the active tab
	submenuitemactivecssclass		CSS Class used for sub menu items when they are the active tab
	rootmenuitemselectedcssclass		CSS Class used for root menu items when they moused-over
	submenuitemselectedcssclass		CSS Class used for sub menu items when they moused-over
	separator		The separator between Root level menu items. This can include custom skin images, text, and HTML (ie. <![CDATA[<img

Token	Attribute	Default	Description
			src=”file.gif”>]]>)
	separatorcssclass		CSS class used for the root level menu item separator
	rootmenuitemlefthtml		HTML text added to the beginning of the Root menu items
	rootmenuitemrighthtml		HTML text added to the end of the Root menu items
	submenuitemlefthtml		HTML text added to the beginning of the sub menu items
	submenuitemrighthtml		HTML text added to the end of the sub menu items
	tooltip		Tooltips added to the menu items. These come from the tab object properties which are filled from the tabs table {Name|Title|Description}
	leftseparator		The separator used just before a root level menu item. A use for this might be a left edge of a tab image for

Token	Attribute	Default	Description
			example
	rightseparator		The separator used just after a root level menu item. A use for this might be a right edge of a tab image for example
	leftseparatoractive		The separator used just before an active root level menu item
	rightseparatoractive		The separator used just before an active root level menu item
	leftseparatorbreadcrumb		The separator used just before a root level menu item found in the breadcrumb array list
	rightseparatorbreadcrumb		The separator used just before a root level menu item found in the breadcrumb array list
	leftseparatorcssclass		CSS class used for leftseparator

Token	Attribute	Default	Description
	rightseparatorcssclass		CSS class used for leftseparator
	leftseparatoractivecssclass		CSS class used for leftseparatoractive
	rightseparatoractivecssclass		CSS class used for rightseparatoractive
	leftseparatorbreadcrumbcssclass		CSS class used for leftseparatorbreadcrumb
	rightseparatorbreadcrumbcssclass		CSS class used for rightseparatorbreadcrumb
	menualignment	Left	Alignment of the menu within the menu bar. {Left|Center|Right|Justify}
	cleardefaults	false	If true, this setting will clear/empty the default color settings of the menu so that they can be left empty and not just overridden with another value
	delaysubmenuload	false	If true this setting delays the loading of the menu until the rest of the page has rendered.

Token	Attribute	Default	Description
			This avoids the IE error "Operation Aborted" {true|false}
[TERMS]	Text	Terms of User	The text of the terms link
	CssClass	OtherTabs	The style name of terms link
[TREEVIEWMENU]	BodyCssClass		Css class for the body of the treeview menu
	CssClass		Css class for the treeview control
	HeaderCssClass		Css class for the header
	HeaderTextCssClass	Head	Css class for the header text
	HeaderText		Text for the header of the tree menu
	IncludeHeader	True	
	Level		Indicates the root level of the tree menu (blank = root) [parent|child|same]

Token	Attribute	Default	Description
	NodeChildCssClass	Normal	Css class for a child node
	NodeClosedImage	folderclosed.gif	Image for a closed (not in current breadcrumbs but has children) node
	NodeCollapseImage	min.gif	Image to shown that will activate a collapse of the menu node
	NodeCssClass	Normal	Css class for the nodes
	NodeExpandImage	max.gif	Image to shown that will activate an expansion of the menu node
	NodeLeafImage	file.gif	Image used for a “leaf” node (no children)
	NodeOpenImage	folderopen.gif	Image for a opened (in current breadcrumbs and has children) node
	NodeOverCssClass	Normal	Css class for a node on mouseover
	NodeSelectedCssClass	Normal	Css class for the selected node

Token	Attribute	Default	Description
	NoWrap	false	Replaces spaces in the text of the node with non breaking spaces
	ResourceKey		Resource Key to localize the Title text. If blank Text property will be used
	RootOnly	false	Show only the root menu (no children)
	TreeCssClass		Css class for the tree
	TreeGoUpImage	folderup.gif	Image to go up a level on the tree
	TreeIndentWidth	10	Addition width to indent each tree level
	Width	100%	With of tree control
[USER]	Text	Register	The text of the register/user link
	CssClass	OtherTabs	The style name of register/user link

Container Objects (stored in the ModuleControls database table)

Token	Control	Description
[ACTIONBUTTON]	<dnn:ActionButton runat=”server” id=”dnnActionButton”>	Generic Button control used for various actions involving a module
[CONTENTPANE]		Injects a placeholder for module content
[DROPDOWNACTIONS]	< dnn:DropDownActions runat=”server” id=”dnnDropDownActions”>	Simple dropdown combobox for module actions
[ICON]	< dnn:Icon runat=”server” id=”dnnIcon”>	Displays the icon related to the module
[LINKACTIONS]	< dnn:LinkActions runat=”server” id=”dnnLinkActions”>	Links list of module actions
[PRINTMODULE] Depreciated – use [ACTIONBUTTON] instead	< dnn:PrintModule runat=”server” id=”dnn PrintModule “>	Displays a new window with only the module content displayed.
[SOLPARTACTIONS]	< dnn:SolPartActions runat=”server”	Popup module actions menu (formerly

Token	Control	Description
	id=”dnnSolPartActions”>	[ACTIONS])
[TITLE]	< dnn:Title runat=”server” id=”dnnTitle”>	Displays the title of the module
[VISIBILITY]	< dnn:Visibility runat=”server” id=”dnnVisibility”>	Displays an icon representing the minimized or maximized state of a module.

Container Attributes (used in Container.xml)

Token	Attribute	Default	Description
[ACTIONBUTTON]	CommandName		maps to ModuleActionType in DotNetNuke.Entities.Modules.Actions: [AddContent| EditContent| ContentOptions| SyndicateModule| ImportModule| ExportModule| OnlineHelp| ModuleHelp| HelpText| PrintModule| ModuleSettings| DeleteModule| ClearCache| MoveTop| MoveUp| MoveDown| MoveBottom| MovePane| MoveRoot]
	CssClass	CommandButton	Cssclass for the button

Token	Attribute	Default	Description
	DisplayLink	True	Display the localized text for the command [True|False]
	DisplayIcon	False	Display the icon for the command [True|False]
	IconFile		File to use for command icon if not using the built in command icon (i.e. myicon.gif)
[CONTENTPANE]	ID	ContentPane	The content pane key identifier to be displayed in the user interface and stored in the database.
[DROPDOWNACTIONS]			
[ICON]	BorderWidth	0	The border width around the icon
[LINKACTIONS]			
[PRINTMODULE] Depreciated – Use [ACTIONBUTTON] instead	PrintIcon	print.gif	The custom print icon file located in the skin file
[SOLPARTACTIONS]			

Token	Attribute	Default	Description
[TITLE]	CssClass	Head	The style name of title
[VISIBILITY]	BorderWidth	0	The border width around the icon
	MinIcon	min.gif	The custom min icon file located in the skin file
	MaxIcon	max.gif	The custom max icon file located in the skin file

Appendix C: Container Conversion

The original DotNetNuke containers contained basic HTML for providing a custom border for content modules. The HTML was typically included in a container.txt file containing a few key [TOKENS] which were parsed on every request.

```
<table border="0" cellspacing="0" cellpadding="0" width="100%" Summary="Module Design
Table">
  <tr>
    <td width="23"><img border="0" src="top-left.gif" WIDTH="23" HEIGHT="31" Alt="Module
Border"></td>
    <td background="top-tile.gif"></td>
    <td width="23"><img border="0" src="top-right.gif" WIDTH="23" HEIGHT="31"Alt="Module
Border"></td>
  </tr>
  <tr>
    <td width="23" background="left-tile.gif"></td>
    <td[ALIGN]>
       [MODULE]
    </td>
    <td width="23" background="right-tile.gif"></td>
  </tr>
  <tr>
    <td width="23"><img border="0" src="bottom-left.gif" WIDTH="23"
HEIGHT="31"Alt="Module Border"></td>
    <td background="bottom-tile.gif"></td>
    <td width="23"><img border="0" src="bottom-right.gif" WIDTH="23"
HEIGHT="31"Alt="Module Border"></td>
  </tr>
</table>
```

Converting an original DotNetNuke container to the new skinning architecture involves making some simple modifications to the HTML:

```
<table border="0" cellspacing="0" cellpadding="0" width="100%" Summary="Module Design
Table">
  <tr>
    <td width="23"><img src="top-left.gif"  border="0" WIDTH="23" HEIGHT="31" Alt="Module
Border"></td>
    <td background="top-tile.gif"></td>
```

```
    <td width="23"><img src="top-right.gif" border="0" WIDTH="23" HEIGHT="31" Alt="Module
Border"></td>
  </tr>
  <tr>
    <td background="left-tile.gif" width="23"></td>
    <td runat="server" id="ContentPane">
      <table cellSpacing="0" cellPadding="0" width="98%" summary="Module Title Table">
   <tr>
         <td vAlign="bottom" nowrap align="left" height="34">
           <dnn:Actions runat="server" id="dnnActions" />
           <dnn:Icon runat="server" id="dnnIcon" />
         </td>
         <td vAlign="bottom" nowrap align="left" width="100%" height="34">
           <dnn:Title runat="server" id="dnnTitle" /> 
         </td>
         <td vAlign="bottom" nowrap align="right" height="34">
           <dnn:Visibility runat="server" id="dnnVisibility" />
         </td>
   </tr>
       <tr>
         <td colspan="3" width="100%">
           <hr noshade size="1">
         </td>
       </tr>
      </table>
    </td>
    <td background="right-tile.gif" width="23"></td>
  </tr>
  <tr>
    <td width="23"><img src="bottom-left.gif" border="0" WIDTH="23" HEIGHT="31"
Alt="Module Border"></td>
    <td background="bottom-tile.gif"></td>
    <td width="23"><img src="bottom-right.gif" border="0" WIDTH="23" HEIGHT="31"
Alt="Module Border"></td>
  </tr>
</table>
```

Additional Information

The DotNetNuke Portal Application Framework is constantly being revised and improved. To ensure that you have the most recent version of the software and this document, please visit the DotNetNuke website at:

http://www.dotnetnuke.com

The following additional websites provide helpful information about technologies and concepts related to DotNetNuke:

DotNetNuke Community Forums
http://www.dotnetnuke.com/tabid/795/Default.aspx

Microsoft® ASP.Net
http://www.asp.net

Open Source
http://www.opensource.org/

W3C Cascading Style Sheets, level 1
http://www.w3.org/TR/CSS1

Errors and Omissions

If you discover any errors or omissions in this document, please email marketing@dotnetnuke.com. Please provide the title of the document, the page number of the error and the corrected content along with any additional information that will help us in correcting the error.

Appendix D: Document History

Last Update	Version	Author(s)	Changes
Oct 2003	1.0	HTML Themes for DNN 1.10 Developers Guide	David Haggard, Erik France
Nov 2003	1.1	HTML Themes for DNN 2.0 Developers Guide	David Haggard
Dec 10, 2003	1.2	Skinning Whitepaper updated for DNN 2.0	Shaun Walker
Jan 26, 2004	1.3	Modified references to Skin Objects processing and install	Shaun Walker
Feb 19,2004	1.4	Added Resources section, skin upload permissions, edits	Shaun Walker
March 4, 2004	1.5	Updates to reflect Version 2.0.1 changes	Shaun Walker
March 18, 2004	1.6	Updates for final 2.0 release	Shaun Walker
June 4, 2004	1.7	Updates for Version 2.1.1	Jeremy White
June 11, 2004	1.8	Updates for Version 2.1.2	Jeremy White

Last Update	Version	Author(s)	Changes
Sept 9, 2004	1.9	Updates for version 3.0.1	Shaun Walker
Oct 20, 2004	1.10	Updates for version 3.0.3	Jeremy White
Mar 11, 2005	1.11	Updates for version 3.0.12	Jeremy White
Sep 29, 2005	1.12	Added [LANGUAGE] token	Vicenç Masanas