# Best Practices in Branding SharePoint 2007 Sites

## [[1]](#footnote-2)#Introduction

Ted Pattison, Ted Pattison Group

David Mann, Mann Software

May 2008

Applies to: Windows SharePoint Services 3.0, Microsoft Office SharePoint Server 2007

Summary: SharePoint

Contents

<table of contents will go here>

Main Subhead#ProductVersionKeyword\_MainSubhead

## [[2]](#footnote-3)#Introduction to Branding with SharePoint Technologies

"I want to use SharePoint, but I don't want it to look like SharePoint."

How many times have you heard that?

Or this:

"SharePoint sites all look the same. They are all so boxy "

Unfortunately, SharePoint is guilty as charged. Out of the box, all SharePoint sites look, well, like SharePoint.

This is problematic because most companies want to have their SharePoint environment use the same colors, fonts, images, layouts, etc. (hereafter called the brand), as the rest of their web-based initiatives. They have likely invested significant resources in developing that brand and applying it to other initiatives and so for SharePoint not to adhere to it would be a problem.

Fortunately, the 2007 release of SharePoint Products and Technologies makes it easy to apply a custom brand to SharePoint sites. This paper will demonstrate how to take your SharePoint environment from the default brand:

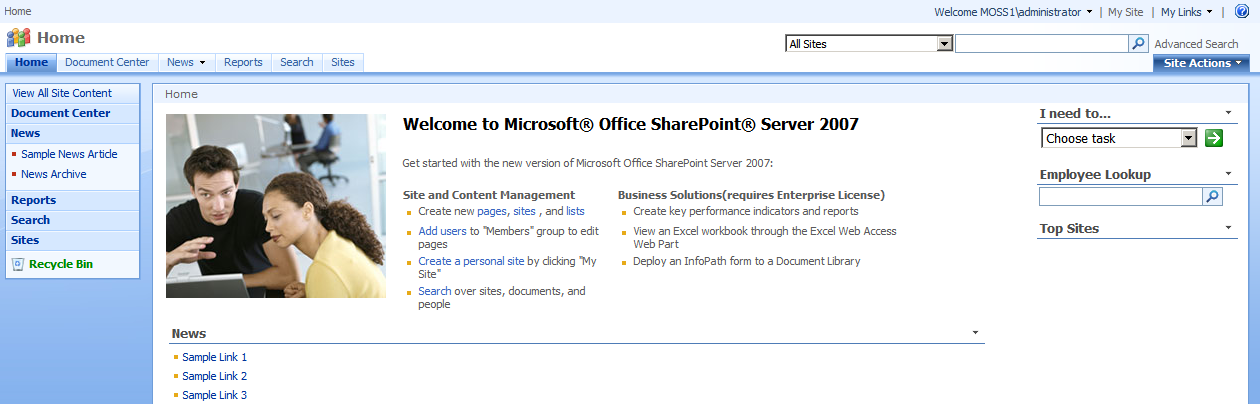


Figure : The default brand - boring and boxy

to, for example, something with a little more pizzazz:



Figure : The sample Liteware brand - just a little more adventuresome...

NOTE: The intent of this article is not to go through the specific details of how to build a new brand (creating MasterPages, modifying style sheets, etc.) but rather to show how to manage, deploy and apply the new brand to any or all sites in your SharePoint farm. Please see the Additional Resources section at the end for details on how to build the MasterPage, CSS and other elements necessary to create your brand.

The ultimate goal of a fully-realized branding implementation is to have a solution that delivers the brand elements you need in a way that:

* Follows best practices for SharePoint development and packaging
* Is deployable across site collections, web applications and farms
* Is manageable
* Integrates with standard source control management systems for versioning, check-in/check-out, etc.
* Is loosely coupled with the implementation (meaning it can be applied equally well to any implementation)

We'll make our way to this branding utopia through the course of this article.

## Getting Started

Depending on your goals, which edition of SharePoint Products and Technologies you have installed, and which tools you have available, there are several options available to you to help reach your goals. Ranging from simple browser-based customizations to SharePoint Designer customizations to custom programming with Visual Studio, and covering both Windows SharePoint Services (WSS) and Microsoft Office SharePoint Server (MOSS), Table 1 lists your options.

Table 1. Options for changing a site's look and feel

|  |  |  |  |
| --- | --- | --- | --- |
| Technique | Scope | Application | Deployment |
| Apply theme on team site | Site | Browser | WSS, MOSS |
| Customize theme on a team site | Site | SharePoint Designer | WSS, MOSS |
| Customizing master page of team site | Site | SharePoint Designer | WSS, MOSS |
| Customize master page of Publishing portal | Site Collection | SharePoint Designer | MOSS |
| Upload custom CSS to Publishing portal | Site Collection | SharePoint Designer | MOSS |
| Develop custom solution | Farm | Visual Studio | WSS, MOSS |

Themes are the simplest, but least powerful option. They are, however, an option, so let's take a quick peak at what they offer.

## Working with Themes

Windows SharePoint Services 3.0 (WSS) provides for the option of branding through themes. Themes are a collection of Cascading Style Sheet (CSS) files that can be used to change the presentation of a site. Unlike the branding solutions we will talk about shortly, themes cannot change anything but colors, fonts and other aspects that can be manipulated via CSS. Essentially this means that themes can only change existing elements, they cannot add new elements.

Note that themes in WSS are entirely different than themes in ASP.NET. WSS implements its infrastructure for themes by copying a CSS file from the file system of the Web server into the context of the current site and dynamically linking to this CSS file from each page.

A user (with proper permissions) can apply a WSS theme to a site to change the look and feel of that particular site. This change does not require that the users know anything about web programming, CSS, etc. All they need to do is pick the theme they wish to apply from a list. Some of the default themes are shown below in Figure 3. As you can see, they change the look of the site but none of the structure.

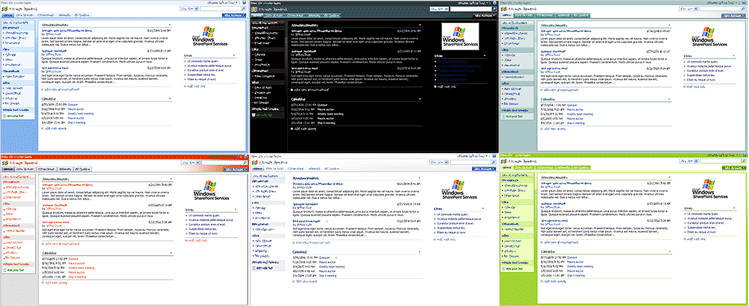


Figure : Themes can change the look of a site

A new theme can be applied to a site from the Site Settings page by clicking on the Site Themes link within the Look and Feel section.

As mentioned previously, themes are the least powerful option. This is because themes have limitations that make them less useful in an enterprise implementation:

* Themes cannot integrate a custom master page to customize the page layouts used within a site.
* Themes cannot add new CSS files, only make changes to the theme CSS files
* A theme can only be applied on a per-site basis as opposed to a wider scope. There is no support anywhere in SharePoint to apply a theme to all the sites within a site collection in a single action. Instead, a user would have to go to each site separately and apply the same theme to maintain a consistent look and feel across a site collection with multiple sites.

Each of these is problematic for the usefulness of themes for any significant deployment of SharePoint. Lets move on to some options that begin to overcome these issues and also present some additional benefits.

## Working with SharePoint Designer

Like themes, Microsoft Office SharePoint Designer 2007 (SPD) provides another means for branding sites in WSS or MOSS. SPD overcomes three of the problems with themes in that you can:

* customize a site’s master page
* add a new master page to change page layouts
* add and integrate new CSS files.

However, SPD doesn’t provide a way to apply a brand across an entire site collection. This means that we're not quite to the point we need to be. We can do more with regard to branding from SPD, but we are still hampered by our deployment options. We are still limited to working with one site at a time with no way to move changes across multiple sites, site collections, web applications or farms . This can lead to scenarios where you are forced to copy and paste your edits across multiple sites.

For WSS-only implementations, the story stops there. We can go no further with SPD and WSS.

For MOSS implementations, the SPD story gets a little better because MOSS provides a Web Content Management (WCM) infrastructure through the use of Publishing portals. A MOSS Publishing portal is a site collection containing Publishing sites which can be configured to have all sites within the collection use the same Master Page and CSS files.

Within the confines of a single site collection, this solves the problem for all site pages. However, across site collections, web applications or farms, we're still stuck with the same deployment problem. We also run into a problem when we attempt to apply a brand to application pages.

One other issue presented with SPD (that can't even come into play with themes) is that it’s impractical to integrate site customizations made with the SharePoint Designer into a source code management system such as Team System.

All of this makes SPD more attractive for branding than themes, but it means that we're still only part way to the goal we laid out at the beginning of this article. To arrive at our goal, we need to leave SPD behind and step into Visual Studio.

## Developing A Branding Solution

The rest of this whitepaper is going to walk through developing a custom branding solution for SharePoint 2007 sites in Visual Studio. We will walk through the code within a sample branding solution named LitwareBranding that has been developed using a Visual Studio project and that is deployed using a solution package.

Right from the start, generating a solution package makes it possible to deploy our branding solution to any farm running WSS 3.0 or MOSS. This means that when we're done, we'll have achieved our deployability goal. We'll have a single package (WSP file) that contains a single source file for each master page and CSS file which can be reused across multiple site collections or across farms.

Furthermore, the Visual Studio approach can be far more appealing than using the SharePoint Designer because it allows you to check all your source files into a source code management system and also to move your development efforts from a development farm to a staging farm for quality assurance testing and then into production.

## Introducing the LitwareBranding solution

This whitepaper is accompanied by a sample Visual Studio project named LitwareBranding. Figure 4 provides a screenshot which shows the structure of the LitwareBranding project within Visual Studio.

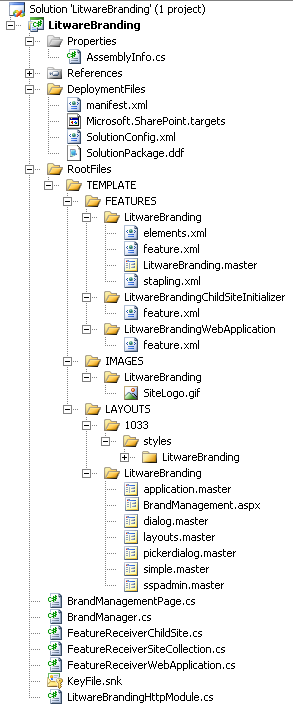


Figure : The high-level structure of the LitwareBranding solution

As you can see, the LitwareBranding solution has been designed using 3 different Features, as summarized in Table 2.

|  |  |  |
| --- | --- | --- |
| Feature | Scope | Description |
| LitwareBranding | Site | This is the main Feature for the solution. It is responsible for applying all of the branding techniques we will discuss to existing sites |
| LitwareBrandingChildSiteInitializer | Web | Applies custom brand to new child sites when they are created |
| LitwareBrandingWebApplication | WebApplication | Manages changes required to the web.config file for the web application that contains the site collection(s) to which our brand will be applied |

Table 2: The Features that make up our solution

Each of these Features will be discussed in more detail in the following sections.

At a very high level, programmatically applying a brand to a site requires the following changes:

1. Using the proper master page
2. Setting the AlternateCssUrl property
3. Updating the SiteLogoUrl property

To apply the brand to all sites within a site collection simply means iterating through the child sites and making the above changes. There are some other requirements related to placing files within the environment and modifying the web.config file which we'll cover as we examine each Feature.

### Exploring the LitwareBranding Feature

Let’s begin our code walkthrough by looking at the feature.xml file for the central Feature named LitwareBranding.

<Feature

Id="065E2243-B968-4F14-BAAE-610BB975EFB7"

Title="A sample feature: LitwareBranding"

Description="Demoware created to demonstrate branding solution"

Hidden="FALSE"

Scope="Site"

ImageUrl="LitwareBranding\AfricanPith32.gif"

ReceiverAssembly="LitwareBranding, [4-part assembly name]"

ReceiverClass="LitwareBranding.FeatureReceiver"

xmlns="http://schemas.microsoft.com/sharepoint/">

<ElementManifests>

<ElementManifest Location="elements.xml"/>

<ElementManifest Location="stapling.xml" />

<ElementFile Location="LitwareBranding.master"/>

</ElementManifests>

</Feature>

The LitwareBranding Feature has been defined with Scope attribute value of Site which means it is scoped to the site collection level. The idea is that a site collection owner should be able to activate a single Feature to apply the Litware corporate branding to all the sites within the current site collection. The LitwareBranding Feature contains a master page template named LitwareBranding.master, and two element manifests: elements.xml and stapling.xml.

Within elements.xml there is declarative XML provisioning logic to create an instance of this master page template. This logic is defined by a Module element with an inner File element.

<Module Name="MasterPages" Path="" List="116"

Url="\_catalogs/masterpage" >

<File Url="LitwareBranding.master" Type="GhostableInLibrary" />

</Module>

During Feature activation, this Module element causes WSS to provision an instance of a master page template named LitwareBranding.master in the master page gallery of the top-level site. After this master page instance has been provisioned, we can programmatically force pages throughout the site collection to utilize it.

The last element manifest from the Litwarebranding Feature, stapling.xml, makes use of a capability of WSS, known as Feature stapling, to attach the LitwareBrandingChildSiteInitializer Feature to the Global and Blank site definitions. We'll cover the details of that Feature later, but this causes any site created within the site collection to have the LitwareBrandingChildSiteInitializer Feature attached to it.

#### Creating a utility class for brand management

The LitwareBranding project contains a class named BrandManager (within the BrandManager.cs file). This utility class encapsulates all of the code that programs against the WSS object model to apply and remove various branding elements. There are static methods exposed by the BrandManager class that accomplishes the following tasks.

* Synchronizing site pages to link to LitwareBranding.master
* Synchronizing all pages to use an alternate CSS file
* Synchronizing all pages to use a custom graphic for the site logo
* Adding support to swap out the master page for application pages

Before drilling down in specific members, you should examine this high-level listing which shows all of the members of the BrandManager class:

public class BrandManager {

// read-only properties

public static string SiteCollectionUrl

public static string DefaultMasterPageUrl

public static string CustomMasterPageUrl

public static string CustomCssUrl

public static string CustomSiteLogoUrl

// utility branding methods

public static void ConfigureMasterUrl(bool ApplyMasterUrl) {}

public static void ConfigureCustomMasterUrl(bool ApplyCustomMasterUrl) {}

public static void ConfigureAlternateCss(bool ApplyCustomCss) {}

public static void ConfigureSiteLogo(bool ApplySiteLogo) {}

public static void ConfigureApplicationPageMaster(bool ApplyApplicationPageMaster) {}

}

As you can see, the BrandManager class contains several static properties. These properties provide URLs which point to various resources such as master pages and a CSS file. For example, the SiteCollectionUrl property returns the Web Application-relative path to the current site collection. This serves as the base for two other properties: DefaultMasterPageUrl and CustomMasterPageUr.

The DefaultMasterPageUrl property returns a Web Application-relative path to the default master page (named default.master) in the Master Page Gallery of the current site collection’s top-level site. This path is put together by combining the SiteCollectionUrl property with the following site-relative path:

\_catalogs/masterpage/default.master

The CustomMasterPageUrl property returns a Web Application-relative path to the master page instance named LitwareBranding.master in the Master Page Gallery of the current site collection’s top-level site. This path is put together using the SiteCollectionUrl property together with the following site-relative path:

\_catalogs/masterpage/litwarebranding.master

The other two properties, CustomCssUrl and CustomSiteLogoUrl, simply return fixed strings pointing to other files deployed as part of our solution: /\_layouts/1033/STYLES/LitwareBranding/styles.css and /\_layouts/images/LitwareBranding/SiteLogo.gif respectively. As you can see, we're instructing WSS to look for each of these files inside a LitwareBranding folder within the appropriate parent folder of the WSS Root (\Program Files\Common Files\Microsoft Shared\Web Server Extensions\12) file system structure.

Now, let’s examine the ConfigureMasterUrl method which enumerates through every site in the current site collection and updates each site’s MasterUrl property to point to either the custom master page or the default master page. The following listing shows this functionality:

public static void ConfigureMasterUrl(bool ApplyMasterUrl) {

// determine MasterUrl property setting

string MasterUrlPath = (ApplyMasterUrl ?

CustomMasterPageUrl :

DefaultMasterPageUrl);

// update MasterUrl property for all sites

foreach (SPWeb site in SPContext.Current.Site.AllWebs) {

site.MasterUrl = MasterUrlPath;

site.Update();

}

}

Note that changing the MasterUrl property affects all site pages that have been created with a MasterPageFile property setting of ~masterurl\default.master. This includes the standard default.aspx page template that provides the home page for many SharePoint site templates including Team Site and Blank site. It also includes the view and form pages such as AllItems.aspx and NewItem.aspx that are used by standard WSS lists.

As mentioned previously, we are updating both the MasterUrl property, which we have just seen, as well as the CustomMasterUrl property. This is because page layouts in a MOSS Publishing site will not be affected when you update only the MasterUrl property. Instead, these content pages are designed to use a dynamic token for the MastePageFile attribute which follows the form of ~masterurl\custom.master. This token is different from the ~masterurl\default.master token because it is switched out using the CustomMasterUrl property instead of the MasterUrl property. This effects all of the content pages within the Pages library of a MOSS publishing site. Therefore, the BrandManager class provides a second method named ConfigureCustomMasterUrl which can be used to switch out the master page for content pages created in the Pages library of MOSS Publishing sites.

public static void ConfigureCustomMasterUrl(bool ApplyCustomMasterUrl) {

// determine MasterUrl property setting

string CustomMasterUrlPath = (ApplyCustomMasterUrl ?

CustomMasterPageUrl :

DefaultMasterPageUrl);

// update MasterUrl property for all sites

foreach (SPWeb site in SPContext.Current.Site.AllWebs) {

site.CustomMasterUrl = CustomMasterUrlPath;

site.Update();

}

}

Modifying either MasterUrl property has no effect if the corresponding page type (standard WSS or publishing layouts) are not in use on the site.

In addition to swapping out master pages, the LitwareBranding solution also integrates a custom CSS file named styles.css. Following best practices, the LitwareBranding solution uses a strategy of deploying styles.css to a directory nested inside the LAYOUTS directory on the file system of each front end Web Server. The value of this deployment model is that the file can be deployed once on the Web server’s file system and yet still be accessible from any site within the current farm using a fixed URL. If you remember back to the static properties we discussed previously, this URL is returned by the CustomCssUrl property.

public static string CustomCssUrl {

get {

return "/\_layouts/1033/STYLES/LitwareBranding/styles.css";

}

}

The BrandManager class provides a method named ConfigureAlternateCss that uses the CustomCssUrl property to assign the URL to the AlternateCSS property of each site within the current site collection. Passing true to this method will cause the CustomCssUrl property value to be written to the AlternateCssUrl property of the SPWeb. Passing false will cause the AlternateCssUrl property value to be cleared out.

public static void ConfigureAlternateCss(bool ApplyCustomCss) {

// determine MasterUrl property setting

string AlternateCssUrl = (ApplyCustomCss ?

CustomCssUrl :

string.Empty);

// update AlternateCssUrl for all sites

foreach (SPWeb site in SPContext.Current.Site.AllWebs) {

// make sure no theme is enabled

site.ApplyTheme(string.Empty);

// apply custom CSS file

site.AlternateCssUrl = AlternateCssUrl;

site.Update();

}

}

The branding capabilities built into WSS also provides an easy way for you to replace the standard WSS site logo that it shows (by default) on the top left portion of the page. The LitwareBranding solution takes advantage of this capability by including a custom graphic named SiteLogo.gif which is deployed in a directory nested inside the IMAGES directory. Like the styles.css file, SiteLogo.gif is also deployed in such a way that it is accessible from any site in the current farm using the URL returned by the CustomSiteLogoUrl property.

public static string CustomSiteLogoUrl {

get {

return "/\_layouts/images/LitwareBranding/SiteLogo.gif";

}

}

The ConfigureSiteLogo method of the BrandManager class has been written to enumerate through each site of the current site collection and to update the SiteLogoUrl property. As before, passing true will cause the CustomSiteLogoUrl property to be used; passing false will cause the property to be cleared.

public static void ConfigureSiteLogo(bool ApplySiteLogo) {

// determine SiteLogoUrl property setting

string SiteLogoUrl = (ApplySiteLogo ?

CustomSiteLogoUrl :

string.Empty);

// update SiteLogoUrl for all sites

foreach (SPWeb site in SPContext.Current.Site.AllWebs) {

site.SiteLogoUrl = SiteLogoUrl;

site.Update();

}

}

### Swapping out the Master Page for application pages

While SharePoint 2007 makes it easy (as we've seen) to swap out the master page for site pages, it does not provide an equivalent way to switch to a new master page for application pages. This is because the two different types of pages (see definitions, below) make use of different master pages and only one (site pages) has support for customization. This is not to say that it is impossible to customize the master page for application pages, just that it is more difficult and requires a different approach. We'll cover all of the steps required in this section.

Definitions

Site page: ASPX pages that are part of the regular, user-browsable site. Includes Default.aspx, AllItems.aspx and other view pages as well as NewItem.aspx and other form pages. As the name implies, site pages belong to one and only one site.

Application page: ASPX pages typically used for site administration and other common functionality. These pages are located in the LAYOUTS directory of each web front end server and include pages such as the standard WSS Site Settings page (/\_layouts/settings.aspx). Application pages are addressable from any site in the farm.

Depending on the nature of your branding customizations, it is important to know that some changes will be reflected in application pages even without the steps we will cover in this section. These changes will be those that are handled by modifying CSS classes to update colors, fonts, and some existing images. Figure 5 shows an application page partially updated with only CSS changes (top) and the same application page fully updated with the new brand (bottom).

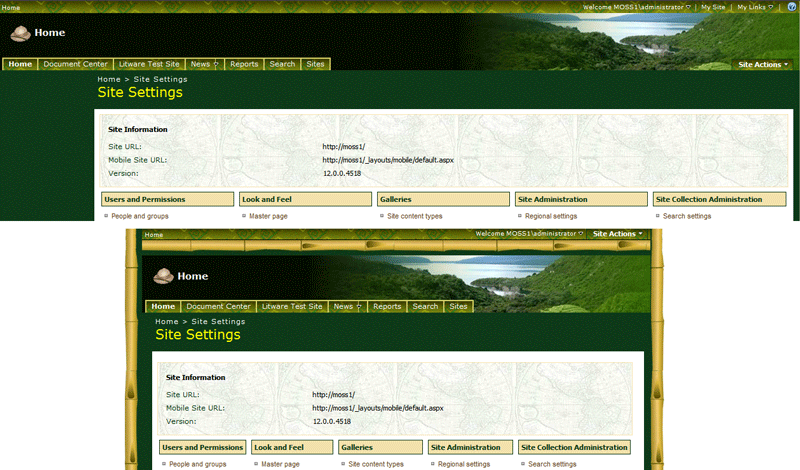


Figure : A partially branded application page (top) and a fully branded application page (bottom)

The changes from the partially-branded to the fully-branded example range from the obvious: the page width and the addition of the bamboo framing, to the subtle: the removal of the "My Site", "My Links" and Help icon links from the top of the page and the relocation of the "Site Actions" menu. This example gives a good idea of what is possible with each piece of the solution. If all that your brand requires for application pages can be achieved with CSS changes, then you do not need to implement the rest of the solution covered in this section.

Before going any further, we should discuss a little bit about why we are going to all of this trouble. If application pages are mostly used by administrators, can't we just give them the out of the box brand or get "pretty close" with CSS-only changes and not worry that their experience is different? After all, as someone with adminstrative access, they are a higher level of user and will probably be OK with a different user interface.

The short answer is that administrators are users, too, and you want to make their experience as consistent and enjoyable as possible. The long answer is that not all application pages are used only by administrators. Remember, I've always said that application pages are "typically" or "mostly" used by administrators. There are some notable exceptions. For example, the following application pages are just a few of the pages that are fully accessible to and will be used by visitors with less than administrator access to the site:

* Add New Alert
* User Information
* Recycle Bin
* Online Help
* View All Site Content
* Document Upload
* Workflows

There are plenty of other application pages that regular users will visit as well. For that reason alone, it is important to apply any custom brand fully across all application pages in your site, so let's get started.

By default, application pages are linked to a standard WSS master page named application.master that WSS deploys within the LAYOUTS directory. Unlike the way that site pages are processed, this entry is not used as a token to be swapped out with the value of an SPWeb property. (Remember, application pages are not associated with a particular SPWeb, so there is nowhere to go for a value to replace, even if the processing called for a replacement).

So what can we do?

The LitwareBranding solution included with this article demonstrates a technique for swapping out the master page for all application pages that link to the standard application.master. This technique involves implementing a custom HttpModule which swaps out the master page during the processing of a page request.

Examine the following skeleton listing for the class named LitwareBrandingHttpModule which provides the HttpModule used in the LitwareBranding solution to swap out the master page for application pages. We'll add the functionality to this class as we proceed.

using System;

using System.Web;

using System.Web.UI;

using Microsoft.SharePoint;

namespace LitwareBranding {

public class LitwareBrandingHttpModule : IHttpModule {

public void Init(HttpApplication context) {

context.PreRequestHandlerExecute

+= new EventHandler(context\_PreRequestHandlerExecute);

}

void context\_PreRequestHandlerExecute(object sender, EventArgs e) {

Page page = HttpContext.Current.CurrentHandler as Page;

if ( (page != null ) &&

(page.Request.Url.AbsolutePath.Contains("\_layouts"))) {

// register handler for PreInit event

page.PreInit += new EventHandler(page\_PreInit);

}

}

void page\_PreInit(object sender, EventArgs e) {

// if requested page links to \_layouts\application.master

// then modify Page to link to custom master page instead

}

public void Dispose() {}

}

}

As you can see, this HttpModule class registers a handler for the PreRequestHandlerExecute event inside the Init method. Within the method implementation of the PreRequestHandlerExecute event handler, the HttpModule class determines whether the request is based on an HttpHandler object that derives from the ASP.NET Page class. Only in cases where the request is based on a Page-derived object will the HttpModule class register an event handler from the PreInit event. The code in the PreRequestHandlerExecute event handler also checks to make sure the incoming request targets a page inside the \_layouts directory which will always be the case when processing an application page in WSS.

It’s important to remember that an HttpModule cannot be deployed in a WSS farm for an individual site collection. Instead, an HttpModule must be configured as an all-or-nothing proposition at the Web Application level. However, a Web Application may contain hundreds of site collections and only certain site collection might have enabled the LitwareBranding Feature. Therefore, the PreInit event handler for the HttpModule class must be able to determine whether the current site collection has been configured with the behavior to swap out the master page for its application pages.

The LitwareBranding solution solves this problem by creating a custom property in the site collection’s top-level site to indicate that swapping out the master page for application pages should be enabled. Examine the implementation of the ConfigureApplicationPageMaster method defined within the BrandManager class.

public static void ConfigureApplicationPageMaster(bool ApplyApplicationPageMaster) {

SPWeb TopLevelSite = SPContext.Current.Site.RootWeb;

if (ApplyApplicationPageMaster) {

TopLevelSite.Properties["UseCustomApplicationPageMaster"] = "True";

}

else {

TopLevelSite.Properties["UseCustomApplicationPageMaster"] = "False";

}

TopLevelSite.Properties.Update();

}

As you can see, this method creates a custom property named UseCustomApplicationPageMaster on the top-level site and assigns the property a value of True. Now, back in the HttpModule, the method implementation for the PreInit event handler can look for this property within the current site collection to see whether it has been configured to enable swapping out the application page master. Note that the PreInit event handler also performs several other checks to determine whether it is appropriate to swap out the master page within the context of the current request. Examine the following code snippet for the fully functional page\_PreInit method.

void page\_PreInit(object sender, EventArgs e) {

Page page = sender as Page;

if ((page != null) &&

(page.MasterPageFile != null) &&

(SPContext.Current != null)) {

// inspect UseCustomApplicationPageMaster property

SPWeb site = SPContext.Current.Site.RootWeb;

string UseCustomApplicationPageMaster =

site.Properties["UseCustomApplicationPageMaster"];

if ((!string.IsNullOrEmpty(UseCustomApplicationPageMaster)) &&

(UseCustomApplicationPageMaster.Equals("True"))) {

// now replace application.master with customized version

if (page.MasterPageFile.Contains("\_layouts/application.master "))

{

page.MasterPageFile =

"/\_layouts/LitwareBranding/application.master";

}

}

}

}

Once the PreInit event handler determines the current site collection has been configured to enable swapping out the master page for application pages and also that the current page links to application.master, it modifies the MasterPageFile property of the current ASP.NET Page object to use a custom master page located within a solution-specific directory within the LAYOUTS directory at the following path:

/\_layouts/LitwareBranding/application.master

## Swapping Out the Master Page for Other Pages

If all we needed to do was change site pages and application pages that used application.master we'd be done now. Unfortunately, SharePoint makes use of quite a number of different master pages to cover the various scenarios and types of pages it serves up. A quick search of the TEMPLATE directory for \*.master files reveals ten master pages deployed out of the box:

* Application.master
* Default.master
* Admin.master
* Popup.master
* MWSDefault.master
* Dialog.master
* Layouts.master
* PickerDialog.master
* Simple.master
* SSPAdmin.master

This does not include the eight deployed by the MOSS Publishing functionality. The purpose of some of these is fairly obvious - popup.master is used for popup pages and dialog.master for pages shown as dialog boxes. But what about the others, and what if you need to make changes to them? How do you make sure that they are served up correctly?

Whether you need to create a custom copy of each of these master pages is entirely dependent upon the needs of your brand. How you serve them up, though, is a matter of a simple change to the Http odule we introduced earlier.

Simpy replace the "if" block we used in the HttpModule above:

if (page.MasterPageFile.Contains("\_layouts/application.master "))

{

page.MasterPageFile =

"/\_layouts/LitwareBranding/application.master";

}

with the following code:

switch (page.MasterPageFile.ToLower())

{

case "/\_layouts/application.master":

page.MasterPageFile = "/\_layouts/LitwareBranding/application.master";

break;

case "/\_layouts/simple.master":

page.MasterPageFile = "/\_layouts/LitwareBranding/simple.master";

break;

default:

break;

}

In this case, we are only handling one other master page - simple.master - but you can easily expand upon this model to handle the other master pages if you need to.

This simple change allows us to see our brand applied to more pages. In the case of simple.master, it is used for the default Access Denied page so we see a page that matches our brand. Figure 6 shows the default Access Denied page (top) and our properly branded Access Denied page (bottom).

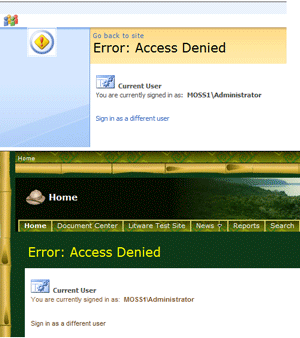


Figure : Our branding applied to other master pages (simple.master shown here)

## Registering an HttpModule in the web.config file

When deploying a business solution with WSS and MOSS, it is a best practice to distribute your development efforts within a solution package. For example, the Visual Studio project for LitwareBranding builds a single solution package named LitwareBranding.wsp which makes it easy and reliable to deploy this solution out to any farm running WSS 3.0 or MOSS. However, the LitwareBranding solution requires adding an HttpModule entry to the web.config file for each Web Application which will be running site collection that activate the LitwareBranding Feature.

In order to update the web.config file within specific Web Applications, the LitwareBranding solution uses a second Feature named LitwareBrandingWebApplication. This Feature is scoped to the Web Application level and is configured with a Feature receiver class to fire event handlers as it is activated and deactivated within the scope of a particular Web Application.

<Feature

Id="FF739C76-0B08-4bc2-A3A2-F61524B492D8"

Title="Litware Branding Support Feature (WebApplication)"

Scope="WebApplication"

Hidden="False"

ReceiverClass="LitwareBranding. FeatureReceiverWebApplication"

ReceiverAssembly="LitwareBranding, [4-part assembly name]"

xmlns="http://schemas.microsoft.com/sharepoint/">

<!-- no declarative elements -->

<ElementManifests />

</Feature>

Like any other Feature receiver class, the FeatureReceiverWebApplication class inherits from the SPFeatureReceiver class and it overrides the four handler methods named FeatureInstalled, FeatureActivated, FeatureDecactivating and FeatureUninstalling. The implementation for FeatureActivated adds the HttpModule entry to the web.config file for the current Web Application. The implementation for FeatureDeactivating reverses that operation by removing the HttpModule entry. The other two events must be implemented but they do not need to do anything.

When you need to add an entry the web.config file, it is a best practice to use the SPWebConfigModification class from the WSS object model. The FeatureReceiverWebApplication class contains a utility method named CreateHttpModuleModification which creates and initializes an instance of SPWebConfigModification and passes it back as its return value.

public SPWebConfigModification CreateHttpModuleModification() {

SPWebConfigModification modification;

string ModName = "add[@name='LitwareBrandingModule']";

string ModXPath = "configuration/system.web/httpModules";

modification = new SPWebConfigModification(ModName, ModXPath);

modification.Owner = "LitwareBranding";

modification.Sequence = 0;

modification.Type =

SPWebConfigModification.SPWebConfigModificationType.EnsureChildNode;

modification.Value =

@"<add name=""LitwareBrandingModule"" “ +

@ “type=""LitwareBranding.LitwareBrandingHttpModule, [4-part assembly name]"" />";

return modification;

}

Once you have a utility method such as CreateHttpModuleModification which returns an initialized SPWebConfigModification object, you can simply call this method from event handlers such as FeatureActivated and FeatureDeactivating to add or remove the required HttpModule entry to or from the web.config file for the current Web Application.

public override void FeatureActivated(SPFeatureReceiverProperties properties) {

SPWebApplication WebApp = (SPWebApplication)properties.Feature.Parent;

WebApp.WebConfigModifications.Add(CreateHttpModuleModification());

WebApp.WebService.ApplyWebConfigModifications();

WebApp.WebService.Update();

}

public override void FeatureDeactivating(SPFeatureReceiverProperties properties) {

SPWebApplication WebApp = (SPWebApplication)properties.Feature.Parent;

WebApp.WebConfigModifications.Remove(CreateHttpModuleModification());

WebApp.WebService.ApplyWebConfigModifications();

WebApp.WebService.Update();

}

### Initializing Branding During Feature Activation

Now, it’s time to put all of these pieces together. When a user activates the LitwareBranding Feature within a specific site collection, there is a Fseature receiver with a FeatureActivated event handler that uses the BrandManager class to apply all the various branding elements. There is also a FeatureDeactivating event handler that removes all the branding elements during Feature deactivation.

public override void FeatureActivated(SPFeatureReceiverProperties properties) {

EnsureWebApplicationFeatureEnabled();

BrandManager.ConfigureMasterUrl(true);

BrandManager.ConfigureCustomMasterUrl(true);

BrandManager.ConfigureAlternateCss(true);

BrandManager.ConfigureSiteLogo(true);

BrandManager.ConfigureApplicationPageMaster(true);

}

public override void FeatureDeactivating(SPFeatureReceiverProperties properties) {

BrandManager.ConfigureMasterUrl(false);

BrandManager.ConfigureCustomMasterUrl(false);

BrandManager.ConfigureAlternateCss(false);

BrandManager.ConfigureSiteLogo(false);

BrandManager.ConfigureApplicationPageMaster(false);

}

Note the call to the EnsureWebApplicationFeatureEnabled method at the beginning of the FeatureActivated event handler. This method has been written ensure that the Web Application-level Feature named LitwareBrandingWebApplication has been activated so that the HttpModule which swaps out the master pages for application pages is properly registered with ASP.NET.

public void EnsureWebApplicationFeatureEnabled() {

// make sure feature which adds HttpModule to web.config is active

SPSecurity.RunWithElevatedPrivileges(delegate() {

using (SPSite siteCollection = new SPSite(SPContext.Current.Site.ID)) {

try {

Guid FeatureId = new Guid("FF739C76-0B08-4bc2-A3A2-F61524B492D8");

siteCollection.WebApplication.Features.Add(FeatureId);

}

catch { }

}

});

}

}

It makes use of the GUID from the LitwareBrandingWebApplication's feature.xml file to add that Feature to the site collection. If you're following along at home, make sure you use the right value for your Feature. If the Feature is already added to the site collection, an error will be thrown, which the empty catch statement will trap and swallow so as to not halt our processing.

## Auto-initializing Branding in Child Sites Using Feature Stapling

So far, we've applied our branding to both site pages and application pages for every site within a site collection where our Feature is activated. Great, we're done right? Unfortunately, no. WSS and MOSS pose another design problem when creating a site collection-scoped branding solution. You must decide how to deal with properly branding new child sites as they are created. As we saw in looking at the code, the BrandManager class only affects the branding properties of existing sites. Therefore, the LitwareBranding solution includes a third Feature named LitwareBrandingChildSiteInitializer which handles applying the custom branding elements to new sites as they are created anywhere in the site collection where our main Feature is activated. The LitwareBrandingChildSiteInitializer Feature is very basic, containing only a Feature receiver and an event handler for the FeatureActivated event:

// fired whenever a new site is created

public override void FeatureActivated(SPFeatureReceiverProperties properties) {

SPWeb ChildSite = (SPWeb)properties.Feature.Parent;

SPWeb TopLevelSite = ChildSite.Site.RootWeb;

ChildSite.MasterUrl = TopLevelSite.MasterUrl;

ChildSite.CustomMasterUrl = TopLevelSite.CustomMasterUrl;

ChildSite.AlternateCssUrl = TopLevelSite.AlternateCssUrl;

ChildSite.SiteLogoUrl = TopLevelSite.SiteLogoUrl;

ChildSite.Update();

}

As you can see, this event handler fires during Feature activation and copies the top level site’s branding properties into the current child site. Now you must figure out how to get this Feature to activate automatically whenever a new child site is created inside a site collection in which the LitwareBranding Feature has been activated. We looked at this breifly earlier, but the answer is Feature stapling. You'll recall from our discussion of the LitwareBranding Feature that it provides a FeatureSiteTemplateAssociation element in its feature.xml which staples the LitwareBrandingChildSiteInitializer Feature to the GLOBAL site definition.

<!-- staple GLOBAL site defition to LitwareBrandingChildSiteInitializer -->

<FeatureSiteTemplateAssociation

Id="1204A425-D105-46c5-BB2C-473A2F27B563"

TemplateName="GLOBAL" />

Notice that we are attaching, or stapling, a Feature with an ID of 1204A425-D105-46c5-BB2C-473A2F27B563, which happens to be the ID of our LitwareBrandingChildSiteInitializer Feature, to the GLOBAL site definition. If you're following along at home, make sure you use the right value for your Feature.

GLOBAL is a special site definition that is used automatically for any site created, regardless of what template or site definition the user selects. GLOBAL is applied first, and then the specific definition chosen by the user is applied. This allows developers to easily set up elements that are used across WSS, regardless of what definition is used.

This stapling technique is what forces automatic Feature activation on a newly created child site. By stapling the LitwareBrandingChildSiteInitializer Feature to the GLOBAL site definition, you are in effect configuring the Feature to activate automatically whenever a new site is created no matter what site template or definition has been used.

Note that there is one exception to the GLOBAL rule. Sites created from the standard WSS site definition named Blank Site do not get GLOBAL elements applied to them. This behavior was added to SharePoint 2007 so that sites created from the Blank Site template will work with the MOSS content deployment strategy. If you want to automate the activation of the LitwareBrandingChildSiteInitializer Feature in sites created from the Blank Site template, you must add explicit stapling instructions for that Site Definition and configuration as well, as we have in our sample solution.

<!-- staple blank site template to LitwareBrandingChildSiteInitializer -->

<FeatureSiteTemplateAssociation

Id="1204A425-D105-46c5-BB2C-473A2F27B563"

TemplateName="STS#1" />

As before, if you're following along at home, make sure you use the right ID value for your Feature.

## Managing Branding Elements

The solution we have covered so far is an all-or-nothing proposal. This works great for single, monolithic implementations where a single brand is being centrally developed, managed and applied in one piece. However, what if you need more flexibility? What if you need to be able to turn on and off individual elements of your brand?

In that case, having all branding elements applied or removed in a Feature receiver is not the best option. Instead, you will need an application page that can turn on or off each specific element. The Visual Studio solution included with this article includes a rudimentary version of just such a page - BrandManagement.aspx, shown in Figure 7.

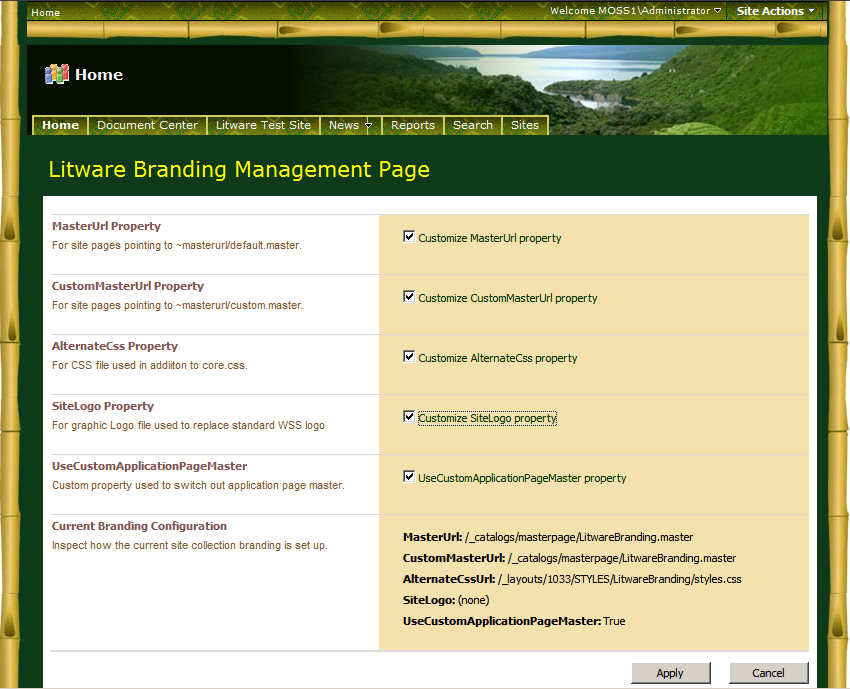


Figure : The Liteware Brand Management page allows for more granular control over branding elements

The code for this page is extremely simple. It makes use of the same BrandManager utility used in our Feature receiver, as seen here in the handler for the Apply button click event:

protected void cmdApplyCustomBrand\_Click(object sender, EventArgs e) {

bool ApplyMasterUrl = chkCustomizeMasterUrl.Checked;

bool ApplyCustomMasterUrl = chkCustomizeCustomMasterUrl.Checked;

bool ApplyAlternateCss = chkCustomizeAlternateCss.Checked;

bool ApplySiteLogo = chkCustomizeSiteLogo.Checked;

bool ApplyCustomApplicationPageMaster = chkUseCustomApplicationPageMaster.Checked;

BrandManager.ConfigureMasterUrl(ApplyMasterUrl);

BrandManager.ConfigureCustomMasterUrl(ApplyCustomMasterUrl);

BrandManager.ConfigureAlternateCss(ApplyAlternateCss);

BrandManager.ConfigureSiteLogo(ApplySiteLogo);

BrandManager.ConfigureApplicationPageMaster(ApplyCustomApplicationPageMaster);

SPUtility.Redirect(Request.RawUrl,

SPRedirectFlags.Default,

HttpContext.Current);

}

While the sample page included with the solution is very basic, it hints at some of the extensibility points available for this solution. Using this page as a launchpad, it would be very easy to:

* Allow administrators to specify a master page (or a CSS file, site logo, or any combination of these) to be used rather than having it hardcoded in our solution by presenting textboxes on this page to collect the information
* Allow administrators to select which elements of our brand would be applied
* Build our own skinning engine that would allow administrators or users to specify which brand, or which elements of a brand, to apply from a list of approved choices. This would be similar to themes, but would maintain our brand and could be deployed across environments and applied to all page types.

The last element to know if you are making use of a brand management page such as this one is how to provide access to it. In our sample solution we do this by simply adding a new menu item to the Look and Feel section of the Site Settings page, as shown in 8.

We do this with a simple entry in the elements.xml file of our LitwareBranding Feature, as shown below.

<CustomAction

Id="SiteActionsToolbar"

GroupId="Customization"

Location="Microsoft.SharePoint.SiteSettings"

Sequence="30"

Rights="ManageWeb"

Title="Litware Branding Manager" >

<UrlAction Url="~sitecollection/\_layouts/LitwareBranding/BrandManagement.aspx"/>

</CustomAction>

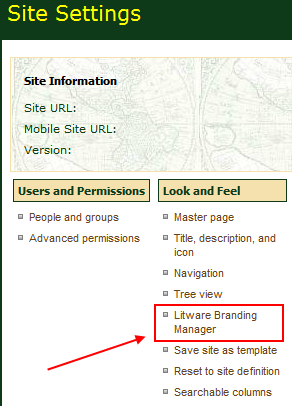


Figure : Providing access to our brand management page

One thing that is important to note is that we do not want just anyone who happens to make their way to the Site Settings page (assuming they have some level of permissions to the Site Settings page) to be able to click on the link and be able to change the branding on our site. What we need is some way to secure the branding page.

If you look at the CustomAction entry above, you'll see a Rights attribute which has a value of ManageWeb. This does exactly what you would expect - it only shows the menu to users who have the ManageWeb permission for the current site. Valid values for this attribute are any elements from the [SPBasePermissions](http://msdn.microsoft.com/office/) enumeration. That takes care of one part - it hides the menu option if users don't have the proper permissions; but it doesn't actually secure the page itself. If someone happens to know the URL, they can still navigate to it and change the branding.

To secure the page itself, we need to make some changes to the code file. If you look at the BrandManagementPage.cs file, you'll see the following code:

protected override SPBasePermissions RightsRequired

{

get

{

return SPBasePermissions.ManageWeb;

}

}

Like the Rights attribute in the elements.xml file, this code will check that users have the proper permissions before showing them the page. User who navigate directly to the page but do not have the proper permissions will see the access denied page - properly branded because of the changes we made earlier to the simple.master page.

## Disabling Themes

If you've gone to all of the trouble of creating a custom brand and applying it to your sites, you almost certainly don't want people to muck about with it by applying their own theme. It is also highly likely that if your custom master page has changed some of the page elements, that the default themes would not render properly anyway. For these reasons it is advisable to disable theme support if you have applied a custom brand. There are two pieces to this:

1. Removing the Site Theme menu item from the Site Settings page

2. Changing security on our site so that theming is not available

Removing the Site Theme menu item from the Site Settings page requires another entry in the elements.xml file of our LitwareBranding Feature:

<HideCustomAction

Id="HideTheme"

HideActionId="Theme"

GroupId="Customization"

Location="Microsoft.SharePoint.SiteSettings" />

As you might guess from our previous coverage of the elements.xml file, this entry will hide the Site Theme link from the Look and Feel section of the Site Settings page. This is part of the battle, but it won't stop users from navigating directly to the page and changing the site theme from there.

To accomplish that, we need to make some permission changes. One of the values of the SPBasePermissions enumeration is ApplyThemeAndBorder, and as you can likely guess, this is the one we are interested in. In Central Administration, turning off this permission is available via the User Permissions for Web Application link in the Application Security section of Application Management, as shown in Figure 9.

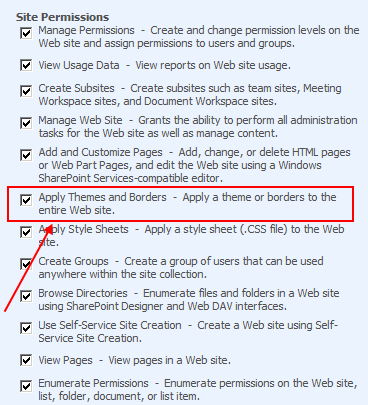


Figure 9: Removing the ApplyThemeAndBorder permission through Central administration

Ideally, turning off this permission would remove the link from Site Settings, but alas, SharePoint does not make it that easy for us. We still need the HideCustomAction element covered above.

Figure 9 shows how you would disable this permission manually through Central Administration, but we would like to do so programmatically. To disable the ApplyThemeAndBorder permission with code, we need to add the following lines to the FeatureActivated event of our LitwareBrandingWebApplication Feature:

//block users from applying themes

SPBasePermissions perm = SPBasePermissions.ApplyThemeAndBorder;

WebApp.RightsMask = WebApp.RightsMask & ~perm;

WebApp.Update();

To make things easy, I add it right at the end of the method, after the code that updates the web.config file.

To wrap things up nicely, we ought to re-enable this permission whenever our branding is removed, so add the following to the FeatureDeactivating event, again, at the end after we update the web.config:

//allow users to apply themes

SPBasePermissions perm = SPBasePermissions.ApplyThemeAndBorder;

WebApp.RightsMask = WebApp.RightsMask | perm;

WebApp.Update();

The net effect of these final changes is to hide the Site Themes link from the Site Settings page and stop people from applying a theme if they navigate directly to the /\_layouts/themeweb.aspx page. It is important to note that the user will be able to see the Site Theme page, will be able to select a theme, and will be able to click the Apply button. However, at that point, a permission check will be done and they will get an access denied error. It is perhaps not the best user experience, but it is what we have, and after all, they manually navigated to the page despite the fact that we had removed the link, so what do they expect?

## Summary

This article has walked through a complete branding solution for changing the look and feel of sites in SharePoint 2007. The LitwareBranding solution has been designed to apply its branding using custom master pages and a custom CSS file on a site collection-wide basis. Along the way you also saw some advanced techniques for swapping out the master page for application pages and for using Feature stapling to automatically apply branding to child sites as they are created. It is also noteworthy that the entire LitwareBranding solution and all of its internal components are deployed using a single solution package. This ensures the easiest and most reliable path to deploy your development efforts across multiple farms running either WSS 3.0 or MOSS.

## Additional Resources

For more information, see the following resources:

* Add link to related content on MSDNAKeyword
* Another link to more content
* Microsoft Office Developer Centerhttp://msdn.microsoft.com/office/

## See Also

1. # ProductVersionKeyword\_Introduction [↑](#footnote-ref-2)
2. # ProductVersionKeyword\_MainSubhead [↑](#footnote-ref-3)