Beginning DotNetNuke 4.0 Website Creation in C# 2005 with Visual Web Developer 2005 Express

From Novice to Professional

Nick Symmonds

Apress®

Beginning DotNetNuke 4.0 Website Creation in C# 2005 with Visual Web Developer 2005 Express: From Novice to Professional

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-681-4

ISBN-10 (pbk): 1-59059-681-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: James Huddleston
Technical Reviewer: Adriano Baglioni
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editor: Damon Larson
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Pat Christenson
Proofreaders: Nancy Riddiough, Lori Bring
Indexer: Valerie Haynes Perry
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `www.apress.com` in the Source Code section.

CHAPTER 2

The Express and DotNetNuke Combination

Chapter 1 went through the basics of what you need in terms of a system and software. This chapter tells you why you need it. I'll tell you about some of the history of .NET and some of its major features. I'll also go into some of the background of DotNetNuke (DNN).

You may be thinking that this chapter really has nothing to do with making web pages, and technically you'd be correct. However, some background can increase your understanding of why things are being done they way they are, and what the advantages are. It makes you a more complete programmer.

Microsoft .NET

Microsoft .NET was first released in 2001. Just after the first release, I wrote a book called *Internationalization and Localization Using Microsoft .NET*. I had worked in programming for many years before, and once the .NET Framework was released to the world in beta form, I was all over it.

Before .NET

You see, before .NET, I was writing code in C, C++, and Visual Basic (VB) 5.0/6.0. I loved the power and raw capability of the C language. I loved the object-oriented slant of C++. I loved VB's ease of use and its slick visual development environment.

There were some problems with each of these, however. I disliked how difficult it was to create Windows programs in C. I disliked the nested code and obtuseness of C++. Sure, with C++, I could now write some Windows programs, but not quickly. (Try to find the `main()` function in a Visual C++ program. You'll be hunting for days.)

I also disliked the abstractness of VB. Although writing a Windows program with VB was simplicity in itself, trying to do any complicated drawing presented some pretty stiff barriers. Not to mention that even a Hello World program required quite a few DLLs (dynamic-link libraries) to be included. This made the simplest of VB programs very bloated in size, which was a problem, because these were not the days of 200 GB hard drives and $40 DVD burners. Computers have come a really long way in a few short years.

DLL Hell

There was also something else that all windows programmers were fighting back then. It was known as DLL hell. You may have heard of it. It is still around.

DLLs were invented as a way to save on memory and disk space—quite ingenious, actually. DLLs were useful when you would have a set of functions that could be common to many programs. Instead of linking the same code into many programs, you could make a DLL with common code that could be used by many programs, and not loaded until needed. This greatly reduced the size of all the executables on disk. DLLs could also be loaded into memory, which would also reduce the memory footprint of programs as well.

Sounds good, eh? Well, it was—that is, until the registry and COM (Component Object Model) came along. In order for a COM DLL to be registered, it needed a GUID (globally unique identifier). This is a unique 128-bit number. Your program would be required to use a DLL with a particular GUID. This assured no spoofing of code, and also assured that you got the proper version of the DLL. (As an aside, VB 6.0 is 100 percent COM. All the controls are COM; everything in it is COM.)

There is an unenforceable rule in the world of DLL programming that the signatures of all the functions within a DLL must not change between versions. It is OK to add new functions to a DLL, and you are allowed to change the logic in a function if you like between versions, but it is not OK to delete a function or change a function's name or arguments.

As I said, this is unenforceable, and it was frequently ignored. In fact, Microsoft was just as guilty of this as anyone else. It would not be unusual for two programs from different companies to use the same DLL. When one company wanted to change a function in a common DLL, they would do it—but they wouldn't necessarily bump up the version. Often, they would keep the GUIDs the same so that your unsuspecting program would think it was using the correct DLL, and they would frequently ship the new DLL with the new executable. You would install it on your machine, and all of a sudden your other program that used the same DLL would not work right. This happened frequently, and it is affectionately called DLL hell.

After much hand wringing and debugging, you might find that a function was changed to add a new argument, or perhaps an argument was changed from a long to a short. A difference like this would not show up until you tried to pass in a number greater than 65,535 (the max number for a short). A bug like this would be infuriating to find and fix.

Other Problems

There are some other quite significant problems with using raw languages such as C and C++, and they have to do with memory usage.

These languages allow you to manipulate memory at will. They allow you to allocate and de-allocate memory as you need for buffer space. Often, this results in a very common issue called a memory leak, in which you allocate memory for use and forget to de-allocate it. If you did this in a loop, you would eventually run out of memory. This can be a very sticky bug to find and fix.

The second most common type of memory problem is the buffer overrun. In C, for instance, a string is a denoted by a start position and special character at the end of the string. When iterating through the characters of a string, you are supposed to look for this special end character. Oftentimes, it is easy to go beyond this character and into other memory space without the operating system complaining at all. A few years ago, buffer overruns were a serious security leak in some Microsoft products. Hackers were intentionally exploiting buffer overruns and putting malicious code into unprotected memory space. All it required to get a virus running was a simple instruction pointer redirect.

Another problem is related to threads. A thread is a piece of code that (appears) to run at the same time as other code. Threads are used for printing and other background tasks such as modem communications. Threads are used everywhere in Windows programs.

A thread has to communicate with the program that spawned it. It does this through common variables that are available to the main thread and any worker threads. What do you think happens when two threads tried to change a variable at the same time? Chaos happens. Thread synchronization issues, deadlocks, and race conditions are easy to introduce and very hard to debug. It takes a great deal of knowledge to program threads properly.

What .NET Fixes

I have told you some of the neat aspects of other languages and some of the pitfalls. Depending on what you need to do, I feel the pitfalls outweigh the advantages. The business of testing and debugging code is expensive. However, it is not nearly as expensive as letting a bug escape to the customer.

It is this expense in both time and money that .NET addresses. When .NET was introduced, it offered the following solutions to programming pitfalls:

- *Garbage collection*: No more memory leaks
- *Safe code:* No more buffer overruns
- *Versioned assemblies*: No more DLL hell
- *Complete classes*: Almost no need to call a low-level Windows API directly
- *Common data types*: Ability for multiple programming languages to be used in writing parts of the same program
- *.NET Remoting*: No more COM
- *Reversion to configuration files*: No more using the registry to store settings; XML configuration files in .NET (used to be INI configuration files before Windows 95)
- *Discontinued use of pointers*: No more pointers, which are confusing and a big source of memory leaks

Let's look at some of these in detail.

Garbage Collection

When Java was introduced, it had the great ability to do garbage collection—automatically releasing memory once it was no longer needed.

.NET has a low-level garbage collection thread that gets run every so often. It is low-level so that it does not interfere with any of the horsepower you require from the machine. It runs during your CPU's idle time.

THREADS AND PROCESSING TIME

Windows uses a time-slice threading model. In this model, the operating system takes chunks of time and gives it to threads in round robin order. This time slices are very small, so all threads look as if they are running in real time.

Threads have priority. A low-priority thread is not given any time if a higher-priority one needs it. While this seems unfair, think about how much time is needed to actually perform a task. I am sure you have had many programs running on your machine at the same time. This might include something like a word processor, a streaming audio program, and maybe a mail program, all of which run threads that have both low and high priorities. The computer is so fast that it is able to handle all these threads in round robin order and still have a ton of time and processing power left over for other tasks. Such tasks could include the .NET garbage collector.

To get a sense of how fast the computer is, you can take a look at the Task Manager. To do so, right-click the taskbar and choose Task Manager. Figure 2-1 shows the menu you will get.

Multiple processors and better operating systems can handle this kind of massively multithreaded program, but the normal Windows computer can't.

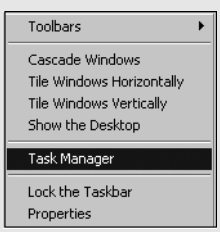

Figure 2-1. *Menu to pull up the Task Manager*

Once you get the Task Manager running, choose the Processes tab and click the CPU table column header. This will organize the entries by CPU time from greatest to least. Note that in Figure 2-2 I have many processes open, but the system idle process takes 99 percent of the computer's time.

You will notice that my computer with all these things running is doing absolutely nothing most of the time. I have hundreds of threads running and all of them take up less than one percent of the computer's time.

There is one thing to note about threads. If you get thread-happy in your program and spin off hundreds of threads, the overhead to manage those threads soon takes up a good portion of the time allotted to your program. There is a certain amount of time needed for things like switching threads, saving thread state, and so on. If you have too many threads, this management time could overwhelm the time allowed for your

threads. This does not even take into account thunking. (*Thunking* is a great word, don't you think? Thunking is what happens when a 32-bit program has to step down to a 16-bit program. It largely has to do with memory management. Basically, your program is running along at light speed, and then it goes *thunk*!)

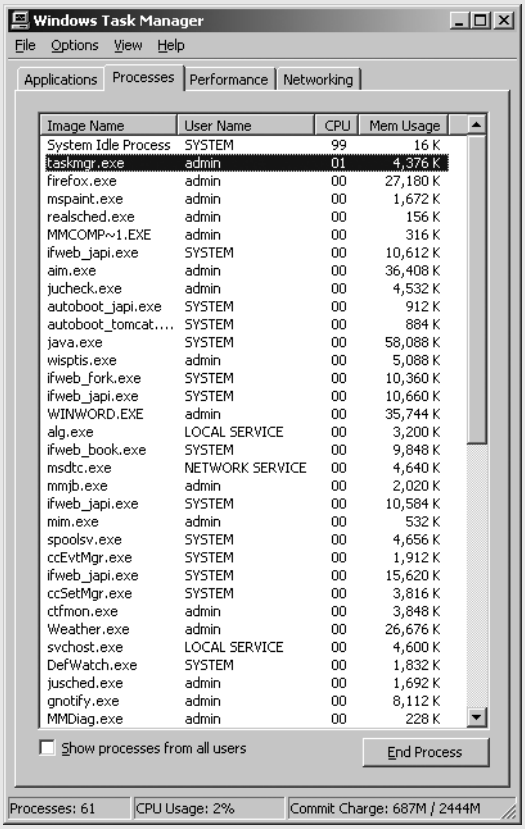

Figure 2-2. *Task manager showing system idle time*

The garbage collection thread walks the heap for any memory that seems to be unclaimed. It also finds thread objects that have stopped and are no longer really connected to anything. Once it finds some memory to be reclaimed, it marks it as such. Your object will then fire an event that you can listen to. This event says "I don't think this is being used anymore. Unless you change a flag, I will delete it the next time I see it." You get a chance to stop the garbage collection by resetting a flag. This process is used just in case you haven't lost connection with an object and you really want it around.

If this flag is not reset, then the next time the thread runs and sees this object, it will de-allocate the memory and delete the object. But that's not all.

Consider a case in which you've instantiated objects 1, 2, 3, and 4. Now, let's say you dispose of objects 2 and 4. This leaves 1 and 3. This also leaves a hole between objects 1 and 3. If

you now want to instantiate another object that is slightly larger than object 2, the system will be unable to use the dead space between objects 1 and 3.

In this way, the garbage collector has another job, which is to create contiguous space where there was none. In this case, it would move the contents of object 3's memory to where object 2 was. This will open a contiguous space that's the size of objects 2 and 4. The garbage collector is a neat freak.

You might be asking, why should I get rid of memory when the garbage collector does it for me? Well, for small programs that do not run for long, you don't really need to. Realize, however, that even these days memory is scarce. There is a threshold of memory usage that the garbage collector will tolerate. Beyond this point, it starts running at a higher priority and for longer. Since garbage collection takes time, your program could slow down, and could slow down significantly. Besides, it is just good etiquette to clean up after yourself.

I will teach you about proper object disposal in Chapter 4, when we delve into some C# programming. Don't worry though—it is not terribly geeky or difficult to do. And remember, if you forget, the garbage collector will clean up after you.

Safe Code

Safe code in .NET parlance is called managed code. This is code that is within the control of the .NET memory manager and security apparatus.

As far as the memory manager goes, this means that if you instantiate an object using .NET, then .NET will take care of the memory management of that object, including garbage collection. If you use "unsafe" code, then all bets are off. .NET will not be able to manage this code for you, and you are back to all the potential problems you had before.

The security apparatus I refer to does not mean keeping out the hackers. It means not letting you do anything that will compromise the system. .NET has many rules concerning what you can and can't do. For instance, it will not let you accidentally write into memory that is not yours. It will not let you stuff a 50-character string into a 30-character space. C will be more than happy to let you do this.

Versioned Assemblies

You can still create DLLs in .NET. However, they are not your father's DLLs.

Microsoft realized when designing .NET that memory was no longer the scarce resource it once was. It is no longer necessary to have a single DLL for many executables. To this end, you can now create a DLL for your program that resides in that program's folder on the machine. You can also have the same DLL for another .NET program that resides in that other program's folder on the machine. Start both programs up, and they will both use their own respective DLL. Change one DLL and it will not affect the other program like it used to.

Each program is forced to use the DLL that is assigned to it. It is possible to have two versions of the same DLL in memory at the same time. With one stroke, DLL hell is a thing of the past.

So is the commonality of code lost? No. If you want, you can sign your DLL (for security reasons) and put it into the GAC. But keep in mind that versioning is enforced here, and if you change a DLL and put a new one in the GAC, then both versions will be in there even though both DLLs have exactly the same name.

WHAT IS THE GAC?

The GAC is the global assembly cache. It is a common area to store DLLs that may be used by more than one .NET program. All the .NET Framework is in the GAC.

You can find the GAC using Windows Explorer. In Windows XP, you will find it in `C:\WINDOWS\assembly` (provided that the .NET Framework is installed on your machine). Open up Windows Explorer and look in there. Figure 2-3 shows my GAC.

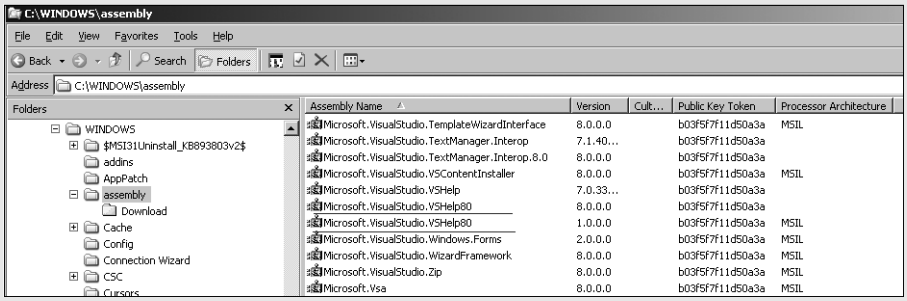

Figure 2-3. *The GAC, showing multiple files with same name*

Notice in this screenshot I have underlined two files in the same directory with the exact same name. Try to give two files the same name in any other directory and you will get an error. These two files are distinguished by their version number and public key token. When you install the .NET Framework onto your machine, it loads an add-in to Windows Explorer that enables it to see the GAC this way. If you were to go to a DOS box and do a `DIR` command, you would not see anything like what you see in Figure 2-3.

There is one thing to note about putting assemblies in the GAC. .NET allows you to do an `XCOPY` deployment. It does not need to register anything. If you need to put something in the GAC, then you lose this capability. This is something to remember when considering an install for your program.

Your program will know, for example, that it wants version 2.0.3.4.5 of some DLL, and another program will know it that it wants version 2.0.3.4.6. The point here is that DLLs can no longer be overwritten, and again DLL hell is avoided. This feature is a major reason why I was so anxious for .NET to come along.

By the way, there is a signing process that goes along with your program and the DLL that it uses. This process uses encryption to make sure that the DLL it gets is the one it wants. Microsoft has gone to great lengths to make sure that bad DLLs cannot be introduced onto your system and spoof a DLL that you are trying to use.

Complete Classes

VB suffered from a severe lack of performance. It is a great language and development environment for writing Windows programs that do not require extensive use of system resources—but some of its features are lacking indeed.

One such feature is the drawing capability of VB. To put it bluntly, it is pathetic. Any rendering of complicated shapes becomes impossible in VB without resorting to the Windows API.

The Windows API is unsafe code. In fact, it is downright scary and really complicated. However, if you want to create any kind of usable and professional program in VB, you will need to resort to the Windows API.

I have a book on the Windows API that is a few thousand pages long. It is a few years old and very worn. When I was working heavily in VB, I knew quite a few API commands by heart, and how to use them.

VB is like an overbearing parent. It protects you from the big bad operating system and does not allow you to do anything that might hurt you. However, VB does allow you to make API calls, which become the back door out to the wild world. Using these API calls can crash your system if you are not careful.

Like I said, though, if you wanted to write any kind of complicated system, you needed to become familiar with the Windows API.

Then along comes .NET. I had heard that VB .NET and C# were on a par as far as what they could do. This is true. VB .NET can now do some incredibly complicated drawing and other neat things that it could never do before. .NET allows this because it has wrapped all the API calls you would need in .NET classes and calls.

.NET allows you to dig deep into the Windows API using safe code. You will not get into trouble like you could by using the raw API.

This was so cool to me that I decided to try some serious GDI work in .NET. (GDI is the graphics device interface, and is probably the most common set of raw API calls).

Microsoft has come up with a set of classes called the GDI+. While I was trying this out, I wrote my second book on .NET, called *GDI+ Programming in C# and VB .NET*. This book is all about graphics in .NET and how to use the classes to do some amazing things. I think that I used direct API calls only once or twice throughout all the examples in the book. I was very hard-pressed to find something that the GDI+ classes could not do.

The important point is that .NET has a complete set of classes that allow you to do almost anything you could want to do without needing to go to the API.

Common Data Types

In C, the size of an Integer data type is compiler dependent. Most times, however, it is 4 bytes. In VB 6.0, an Integer is something different altogether. In C, a string is a starting memory position and an end character. In VB, a string is totally different and has a size characteristic to it.

Likewise, if you wanted to write a program in C++, all your code for that program would need to be in C++. There is no way to pass data directly from a part of a program written in VB to one written in C++. You can pass data from VB to a C++ COM DLL using marshaling, but that is very complicated. So you end up having to have your programming team write code using the same language. You have no chance to leverage the talents of your best VB programmer.

.NET enforces a common data type set throughout the framework. It also compiles the code you write to something called intermediate code. This intermediate code ends up being the same regardless of whether you wrote it in C# or VB .NET or even COBOL .NET.

These two things allow you to write a program using assemblies from any of the .NET languages. The VB programmer can write a complicated set of classes in VB, and the C# programmer can pass data back and forth and use the interfaces with no problems. All this is native and requires no extra marshaling of data.

This allows you to use programmers with knowledge in several different languages on the same project. Your VB programmer no longer needs to feel left out of the “real” projects.

.NET Remoting

Ah, remoting! Although Microsoft will deny it, this is where they got rid of DCOM (Distributed COM). It is a way for a client program to talk to a server. It is not the browser/web server combination, but an executable on one machine instantiating and talking to an executable on another machine. These days, this type of client is called a fat client. Unlike the browser, this client can make full use of the client machine's operating system and is in fact operating system dependent.

DCOM was, and is, a nightmare. It is difficult to set up and even more difficult to use properly. It can be slow and it is not firewall-friendly.

DCOM is also dependent upon GUIDs being in synch. Often, if you changed a server, you would change its set of GUIDs. Then the client would no longer recognize the server and couldn't work with it. So, if you changed the server, you would need to recompile the client to work with the new server. You would need to redistribute the new client whenever a new server came along. This could be avoided in C++; but in VB 6.0, you were hosed.

.NET Remoting changed all that. There are two kinds of remoting available to your .NET programs: HTTP remoting using SOAP, and binary remoting.

Binary remoting is the fastest, but it may not be able to pass through firewalls. HTTP remoting is XML serialization, and passes through on port 80. HTTP remoting is much slower than binary remoting. There is a third kind of remoting as well, which is a combination of the two mentioned here. It is HTTP remoting using binary data.

.NET has made changing between remoting types very easy. There is no recompiling of any program. It is just a value change in a configuration XML file.

Remoting is different from DCOM because it uses a leased lifetime for an object. DCOM relies on pinging. If the objects cannot ping each other, then the remote object is destroyed.

Remoting has divorced the tight coupling between the client and the server, which makes updating one part or the other much easier.

Reversion to Configuration Files

A far as .NET is concerned, the registry is a thing of the past. All configuration options and persisted values are kept in XML configuration files.

These configuration files allow your .NET program to be installed on a computer just by copying it to a folder. You then invoke the executable and you are running. Think of this compared to installing something like Microsoft Word.

Back in the DOS days, this was how all programs were installed. Just copy them from one machine to another. Along came Windows 95/98/ME/NT/2000/XP with its much heralded registry. Now we are back to the original method. (I am not the only one who finds this amusing.)

Discontinued Use of Pointers

Ah, pointers. No self-respecting C or C++ programmer would ever admit to not being an expert in pointer arithmetic, right? Single indirection I could handle just fine, but sometimes I would see double and triple indirection in code, and I would just throw up my hands. Some programmers took great joy in producing abstruse C code.

Well, Java came along and changed all that. Java is very object oriented and has no provision for pointers. This alone reduced the amount of bugs by an order of magnitude.

In case you are wondering, here is a little explanation of pointers. A pointer is a reference to a memory location. If you wanted a function to work on a very large string, the efficient thing to do would be to pass a pointer to the string into the function instead of passing in the string itself. The function would then reference the string and work on it. This had the added advantage (or disadvantage) of permitting you to change a variable directly in the calling program. If you were to pass in the whole string, the function would work on a copy of the string, and nothing in the main program would change. While pointers may seem like a cool thing, they are a major source of bugs. The memory referenced by pointers is not protected well. It is very easy to inadvertently change something you should not have access to.

Everything in .NET is an object. .NET does not allow you to pass things by passing pointers. It certainly does not allow you to walk through memory one byte at a time like "C" does with pointers.

.NET is very safe. There is a way, however, to pass a reference to an object into a function. This allows you to use a function to change an object in a calling function. The .NET method of passing a reference is explicit. You must explicitly say that the argument in a function call is a `byref` argument. Type safety is still enforced in .NET even when passing a variable by reference.

The Evolution of DotNetNuke

Like anything new from Microsoft, .NET came with a whole host of help files and examples. One of the first examples to come out was a starter kit for ASP.NET called IBuySpy. This was a portal application that contained enough code to actually be useful. Microsoft released the code to the world, and the license agreement was such that anyone could release any derivation of it with no fees.

This application caught the eye of an ASP.NET programmer in Canada by the name of Shaun Walker. He took the program and altered it to fit an amateur sports web hosting environment. Along the way, he more than doubled the code—from 11,000 to over 25,000 lines.

The program worked fine for him, so he tried to sell it to the world. When this was not successful, he decided to release it to the open source community as a general purpose web application framework. It took off.

Within three months he had 5,000 registered users, and the product was dubbed DotNetNuke. It was named after an existing open source web portal product called PHP-Nuke.

DNN is free and its licensing scheme is similar to the BSD (Berkeley Software Distribution) license. Basically, you can use it, enhance it—whatever you need. The BSD license gives the most freedom of any licensing scheme.

Currently, DNN has over 40 core programmers and is over 200,000 lines of code. This is truly amazing.

DotNetNuke Features

DNN has many features that allow you to create websites and manage them easily. While VWD 2005 Express does have starter kits for individual websites, DNN goes far beyond this.

Virtualized Websites

DNN allows you to have *virtualized* websites. Many companies have multiple websites. Think of Microsoft. It has `www.microsoft.com`, `http://msdn.com`, `http://search.microsoft.com`, `http://hotmail.com`, and a few others.

While `www.microsoft.com` provides a way to get to some of these other websites through the main page, you can also get to these sites directly.

DNN allows you to set up multiple URLs that are accessible and manageable though a single URL. Your company may have one URL for sales, one for the help desk, and another for frequently asked questions. DNN allows you manage all these through a single portal.

Consistent Framework

Whether you are working on Microsoft Word, Excel, or PowerPoint, you can be assured that the menu structure for all three programs will be the same. You can be assured that the look and feel of the three programs is also the same. It is this consistency that makes these programs usable.

The framework in DNN is very consistent when it comes to adding pages, managing content, and so on. You will find that the modules that can be plugged into DNN are also familiar to you. This even extends to the folder structure and the files that are on your hard drive.

This consistent framework just may entice you to create your own module for public use in DNN. Who knows?

Modular Architecture

The framework of DNN is such that a single page can have several sections on it. Each of these sections can contain a module of your choice.

A module is a self-contained program that can run within this space. If you wanted a search engine, a shopping cart, and some text on a single page, you would normally create a single page and include the functionality of all these items on it. DNN allows you to separate the functionality of each item while still displaying a single page to the user. You will find this feature very powerful indeed.

Multilanguage Capabilities

ASP.NET uses the same type of resource files as a C# full-client program. The language resource files are XML files called ResX files.

There are many language packs that you can download and install into your DNN project. Every text string and word in DNN is inside one of these language resource files. All you need to do is download one and log in again using the new language.

It is also a simple matter to show a drop-down list of languages in your application to allow the user to choose his language as well.

Skinning

Skinning is the process through which you define the look and feel of a web page or website in an external file. The program looks to this file before the page renders, and applies this look and feel to the page.

DNN allows you to write and provide skins for your website, and to change them when you want. Also, the flexibility is such that you can even change the look and feel on a page-by-page basis if you want.

Skinning is probably the most used and coolest feature of DNN as a whole.

Membership Management

DNN has several roles that you may apply to your website. It has the ability to create roles such as guest, registered user, administrator, and so on.

When you create a page in DNN, you can specify whether that page is viewable by anyone visiting the website or only by registered users. This is a very powerful feature that is very easy to use. Managing role security without this feature takes quite a bit of work.

Tested Code

While there are many more DNN features not mentioned here, there is one that is perhaps more important than all the rest: proven code.

DNN has been around for a while now, and it has been used by thousands of people in thousands of websites.

DNN is thoroughly tested, and all the kinks have been worked out by testers and users like you. You will be using a product that does what it says and works with no fuss. DNN is a proven product.

Summary

This chapter has provided some information on why the combination of Microsoft .NET and DNN is such a powerful one.

First of all, the complete software package is free. The VWD 2005 Express development environment is free, and the DNN framework is free. This brings professional website development to more non-programmers.

If you are a programmer or manage a programming department, the next advantage is important to you. You can leverage the programming expertise of coders with different language backgrounds. Your website can be written in VB .NET or, as is the case with this book, in C#. Your website can be written in a combination of these languages if you like.

The advantages of DNN enable you to get up and running with a professional website with almost no programming necessary. While this statement usually means “limited functionality,” in this case it does not. You will be able to use DNN with VWD to create a website with as little or as much functionality as you like. You can let the pluggable DNN modules do all the work, or you can go into the code and tweak it to your specifications.

The combination is powerful indeed.