DNN Extension Development

Best Practices Guide

Last Updated: July 16, 2015

IowaComputer GURUS

IowaComputer
GURUS
Custom Websites and Intranets
.NET Application Development
Expert Technology Support and Training
Performance Optimization
Technology Services and Support ... for the Life of Your Project

Contents

Technology Services and Support ... for the Life of Your Project

IowaComputer GURUS

Custom Websites and Intranets

.NET Application Development

Expert Technology Support and Training

Performance Optimization

Technology Services and Support ... for the Life of Your Project

The Purpose of This Document

This document has been created to provide a baseline set of information relating to DotNetNuke Extension development. The best practices listed here cover items that relate specifically to the creation of Modules, Authentication Providers, and Skin Objects and have been compiled based on information obtained over the 150+ custom development projects that have been completed by IowaComputerGurus in the past 3 years.

Not all aspects of each section will apply to every project. In addition there are specific solutions that even with the best of intentions must deviate from the practices listed in this guide. Therefore, we want to stress the importance that this is simply a set of basic best practices that should apply in most situations. Additionally, this document is NOT intended to be a "how to develop DotNetNuke modules" document. Section two provides information to community based and other resources that might be of benefit for those just starting out with DotNetNuke development.

Document Revision History

Revision Date	Module Version	Minimum DNN Version	Notes
2/7/2009	03.00.00	5.x	First public release of module
7/3/2012	06.00.00	6.x	Updated for DNN 6.x support and design patterns
5/15/2015	06.00.00	6.x	Updated for new document format

Disclaimer

This document is provided as an additional source of information on the usage of this module. Module content, features, and functionality are subject to change at any time and will be distributed to the public with unique version numbers. It is the reader's responsibility to ensure that this documentation matches the current version of the module in question. Additionally, the reader understands that by using this documentation and the module that they agree to the terms of use, posted on the IowaComputerGurus.com website and available from all module download pages.

Copyright and Trademark Notices

The information contained within this document is protected under international copyright laws with a content owner of IowaComputerGurus Inc. This document may be re-distributed to anyone, however, it must remain intact and with this disclaimer visible. DotNetNuke and DNN are both registered trademarks of DotNetNuke Corporation.

Getting Started with Module Extension Development

This guide was not created to provide an introduction to DotNetNuke module development, but to be a guide in creating modules that have the best chance of being stable across the multitude of DNN environments. The following sections provide links and information to helpful sites for obtaining project templates, tutorials and books relating to the "Getting Started" activities with DotNetNuke development. These are just a small sampling of the resources that are available.

Project Templates

Depending on your language of choice, you are presented a few options when it comes to obtaining project templates for creating DotNetNuke modules.

DotNetNuke Starter Kits

If you are a VB.NET developer or looking for dynamic module templates, the DotNetNuke Starter Kit packages are the best place to get your templates. You can obtain these downloads from the www.dotnetnuke.com website in the form of a simple .vsi installation package.

The starter kit comes with dynamic module (WSP) templates for both C# and VB as well as a compiled VB module template. Please see the section regarding WAP vs. WSP for recommendations on the two types of projects.

Chris Hammond's Template

One of the most commonly used templates is one created by Chris Hammond from DotNetNuke Corporation and can be found at: http://christoctemplate.codeplex.com. This is a C# module template that has a lot of common code-stubs to help make the initial project roll-out a success.

IowaComputerGurus.com

After developing multiple projects, we identified a small gap in the way that existing templates worked. Once experience was gained with module development, the "default" code, included in the templates, was not needed. An extra 15 minutes was spent removing code that should never have been in the template. As such, we created C# templates for Visual Studio 2010 that only have project structure and no default code. These templates help to reduce time necessary to go from project creation to development. Additionally, these templates adhere to all of the best practices as outlined in this document.

Online Resources

There is a plethora of information available online regarding DotNetNuke development. However, due to the massive amounts of information available, it can be hard to find the exact information that you are looking for. Therefore, the following are a few of the recommended online resources for those looking for basic development information.

DnnCreative.com

This is a paid video tutorial site, however, they provide a large collection of tutorials that can be helpful for the developer that is looking to start module development as well as individuals looking for basic DotNetNuke usage information.

DotNetNuke.com Blogs

The blogs on DotNetNuke.com are one of the most commonly overlooked information sources available. Often you will find core team members blogging about new functionality that has/will be included in the core, as well as many basic development tutorials are listed.

MitchelSellers.com

In addition to the above resources, Mitchel will often blog about various DotNetNuke and .NET development topics on his blog.

Books

There are many DotNetNuke books out on the market. There is only one book relating exclusively to DotNetNuke module development, "Professional DotNetNuke Module Development", written by Mitchel Sellers. This book covers the foundation of DotNetNuke module development and is a great starting point. Some of the other DotNetNuke books cover development as well as varying levels of detail.

Project Setup/Configuration Recommendations

The first section of "Best Practices" recommendations support the actual project setup, configuration, and packaging for deployment. These recommendations are very general in nature and apply to all types of extension development.

WAP vs. WSP Project Structures

It has been the experience of IowaComputerGurus that using the Web Application Project (WAP) development model is a much better approach when working with custom or commercial modules when compared to Web Site Projects (WSP). We have come to this conclusion for a number of compelling reasons:

- WAP allows for easy compilation to a single DLL for all code
- WAP allows for easy source control integration with a solution for each module
- WAP encourages working with just the DotNetNuke.dll, with a separate solution; any attempts at non-standard integration is discouraged simply due to the project structure
- WAP allows for post-build events that can help automate module packaging and speed up development efforts
- WAP allows for creation of multiple modules that have dependencies without installation order risks and other quirks associated with inter-module dependencies with WSP

For these reasons, we utilize and strongly suggest the regular use of the WAP model. Although some items may be tricky at first, when it comes to debugging, we have found a more stable development environment is created and developers can focus more easily on the task at hand.

Module Folder Naming and Code Namespaces

One of the most commonly overlooked items by developers is in relation to the name of the folders for their modules and/or the namespaces associated with their .NET code. Developing for DotNetNuke does introduce a few challenges that might not be immediately realized. A DNN user can install a module developed by any company. This can be a risk if using common naming for projects. As such, we have two recommendations for module folder naming: the Best Scenario, which provides for the best security against conflicts and the Best Compromise Scenario, which is slightly easier to implement, but can still have risks, although, minimized greatly. The final point in this section discusses the importance and recommendations around Namespace and Assembly definitions.

Best Scenario

From a Best Practices standpoint, we have found that placing all custom modules developed by a single company into a common folder is one way to easily prevent conflicts from a physical file point of view. For example, placing a "guestbook" module developed by IowaComputerGurus in the /DesktopModules/ICG/Guestbook folder would provide a level of separation between other modules that traditionally are stored directly inside the /DesktopModules folder and our module.

This naming convention helps ensure that the only file conflicts that would be introduced are related specifically to code that you deploy. The process of configuring this is slightly involved, requiring the changing of base path and other path values in the respective .dnn manifest files.

IowaComputer
GURUS
Custom Websites and Intranets
.NET Application Development
Expert Technology Support and Training
Performance Optimization
Technology Services and Support ... for the Life of Your Project

Best Compromise Scenario

If for one reason or another you cannot work with the "Best Scenario" laid out in the previous section, working with a naming strategy that incorporates your company name or initials into the module folder is a good place to start. For example, with the guestbook placing the module in an ICG_Guestbook folder would reduce the risk of conflict. This scenario is not as favorable simply because it does not group modules together.

Namespaces and Assembly Names

Aside from physical file naming within the module folder, it is important that namespaces and assemblies be created in a manner that will not conflict with others. Using a namespace structure that starts with Company.Modules.Module name or ICG.Modules.Guestbook for an example helps ensure that others will not be placing code inside a similar namespace where it might conflict with your code.

The last consideration is in the naming of the compiled dll. We recommend naming this file the exact same as the root namespace for the project. In our example we would generate a dll with a name of ICG.Modules.Guestbook.dll. This method ensures that we have the best chance of not conflicting with another module. If we simply used Guestbook.dll it could be easy for another module to have the same DLL name and overwrite our file.

Packaging and DNN Version Support

DotNetNuke has evolved greatly and there are various limitations and gotcha's for supporting specific versions. The final decision comes down to which versions you need to support your module. However, our current recommendation is to set a minimum requirement of 6.0.0 or later for all new modules. This ensures that DotNetNuke design patterns can be observed.

Always Set the Minimum Required Version

When making a package decision, ensure that users do not install a module on a non-supported version. If you build your module against DNN 5.5.0 and a user tries to install it to a DNN 5.0.0 website, as soon as the package is unzipped, their site will cease to function, throwing an error that it was not able to bind to the proper DLL.

To ensure that this does not happen all, DotNetNuke manifests support the specification of a version limiter. If the DotNetNuke installation is not at a set minimum value, the module will not install and the user will be notified that they do not yet meet the pre-requisites for installation. This can be handled by adding the following nodes under the main <package> node within your DotNetNuke manifest and is included by default with our templates.

```
<dependencies>
<dependency type="CoreVersion">06.00.00</dependency>
</dependencies>
```

It is important to note that the CoreVersion value must be a three part version number with two digits for each part as reflected above.

Being a Good DotNetNuke Citizen

The WAP vs. WSP Project section helps define the project structure and the beginnings of what it takes to build an extension that will play well inside the DotNetNuke environment. This section of the document takes the foundation of this a bit further and discusses practices that should be followed to ensure developed solutions play well across DotNetNuke sites with many different configurations.

Support DatabaseOwner and ObjectQualifier

One key configuration element of DotNetNuke that is often overlooked by module developers is the ability to specify custom DatabaseOwner and ObjectQualifier values. Items not heavily used, and then being omitted from Sql Data Provider files can prove to be problematic and a support nightmare.

Supporting these replacement tokens, {databaseowner} and {objectqualifier}, is incredibly simple from a developer standpoint. We have found that when creating scripts via an external system, it is easy to do a find and replace operation to add this support as long as you follow a specific naming standard. For example, if you name all objects using [dbo].[objectname], you can do a find on the value "[dbo].[" without the quotes and replace it with "{databaseOwner}[{objectQualifier}" again without the quotes and your entire script set is updated.

Avoid External Configuration and Web.Config Changes

Although the XML Merge functionality it is easy for developers to make web.config changes without requiring site administrators to make manual changes; it is our recommendation that solutions avoid any web.config changes if at all possible. By avoiding these configurations you have a module that will install in a more streamlined approach and there are less moving parts that can go wrong.

Even with the XML Merge functionality that comes with 5.x and later there are still risks associated with adding web.config items and conflicts with other modules. Although we fully understand that depending on the specific implementation this may not be possible. So this rule is something that should be considered carefully based upon what goal is being attempted with the custom solution.

Re-Use Common DotNetNuke Controls When Possible

One of the biggest advantages of leveraging the DotNetNuke platform is the plethora of built in controls, tools, utilities and functions that can be used. Using these built in defaults, users will be presented with interfaces that are familiar to them as well as reducing your overall time to development. Listed below are a few of the most common controls and their respective locations included for reference.

Control Name	Path	Functionality
Label	~/Controls/LabelControl.ascx	Provides a label and expanding help text icon.
UrlControl	~/Controls/UrlControl.ascx	Provides functionality for Url, Page, or file selection as well as the ability to upload files within the existing DotNetNuke file structure. Used across the DNN installation by default.
Various	DotNetNuke Web Controls Library	This assembly contains all of the wrappers to the various Telerik controls that are not contained as a default part of the DotNetNuke core. Items of interest include RadCaptcha for human verification.

If you are unfamiliar with the usage of any of these controls, or the others that are not included in the table above, please see the various resources listed in the section, Getting started with Module/Extension Development, as many implementation examples can be found via those resources.

Utilize DotNetNuke Standard Form Patterns for Layout

With DotNetNuke 6.x a consistent pattern for UI management was defined that can be easily followed by developers. Following these patterns makes it easy for individuals to create consistent, easy to style user interfaces for their custom extensions.

Taking a little bit of time to master learning these patterns, will reduce overall time to develop. Full information on the design patterns and processes can be found at: http://uxguide.dotnetnuke.com/.

Avoid Manipulating DotNetNuke Database Tables Directly

When reviewing custom solutions that have been developed for customers in the past, one of the biggest mistakes that we see are cases where existing DotNetNuke API's are not leveraged. Developers are either re-creating the wheel or they are actually manipulating DNN database tables and data directly. As part of this, for our best practice recommendation, you should AVOID doing anything to DNN tables directly. We help you understand how to compensate for this in the following sub-sections. They will give some needed insight to where you can find the API's necessary to leverage the DNN functionality. Also addressed are the reasons why direct database access is risky.

Determining Proper APIs

Sadly, one of the areas that are lacking within the DotNetNuke framework is in API documentation. There is a bit of a learning curve necessary to fully identify the various namespaces and locations that can be used to obtain the information needed. The following table lists a few of the most common namespaces that provide access into key DotNetNuke data that is commonly needed in custom solutions.

Namespace	*Description*
DotNetNuke.Entities.Portals	Classes in this namespace provide portal specific information. The PortalController is one of the key objects to investigate in this namespace.
DotNetNuke.Entities.Tabs	Classes in this namespace provide information on tabs, or pages, within the DotNetNuke portal. The TabController provides many helpful methods, especially for obtaining lists of pages in the portal, and similar items.
DotNetNuke.Entities.Modules	Classes in this namespace provide information on modules within the DotNetNuke portal. The ModuleController class provides access to module lists, settings and more. This is one of the more commonly known namespaces as the default "settings" functionality uses the ModuleController.
DotNetNuke.Entities.Users	Classes in this namespace provide information on users within the DotNetNuke portal. The UserController provides most of the basic functionality needed to retrieve and work with users.
DotNetNuke.Security.Roles	Classes in this namespace provide information about Roles. The RoleController allows you to get a list roles, add/remove users from roles and more.

The above reference serves as a good starting point and illustrates that DotNetNuke does expose a very robust API for programming. The only real limiting factor is that sometimes you may have to play with a method just to see what it does due to the lack of API comments.

Risks of Direct Database Manipulation

From a best practices standpoint, the true question here is "what is my risk"? Well we believe that the risks of direct database manipulation are actually very high, thus the inclusion of this topic in our best practices recommendation. Looking at risks we have a few vectors to investigate.

Upgrade Risks

The primary risk with direct database manipulation comes with future upgrades to DotNetNuke. There is no set criterion that prevents DotNetNuke from changing the structure, format, or inputs for core tables and procedures. Therefore, if you are calling database assets directly, your custom scripts can either fail, or result in unintended operation. This risk is very hard to quantify. There are some things to be concerned about, especially with the massive overhauls and improvements that have been coming with the 5.x series.

Operational Risks

Aside from the upgrade risks, bypassing the API's can have operational risks or effects. For example, directly modifying information on modules that support caching, or user information that is cached by DNN, will not update. If you use the API's provided the respective cache entries are purged and the information is updated.

Properly Tie Data for Multi-Portal and Multi-Module Installations

Another Best Practices area surrounds the whole concept of data isolation. What DotNetNuke data element should drive the storage of your data? Our Best Practices recommendation in this area are mostly focused on using common sense. The following table outlines the three most common data elements used:

DotNetNuke Data Element	***Use For Storing***
ModuleId	Individual settings for a module and data for a module. This data, if a module is "referenced", will still be visible for the referenced module.
TabModuleId	Individual settings for a module, typically no data should be isolated at this level. If a module is put on multiple pages, it will have one ModuleId value and a different TabModuleId value for each page that has been added to.
PortalId	Use this for storing data at a portal level.

Selecting the proper data element is really a matter of understanding your requirements and how long the data should stick around. Therefore, our recommendation here, use appropriate options.

Foreign Keys and DotNetNuke Tables

One extension to this Best Practices is how to handle foreign keys when it comes to custom tables and their respective data relations. There are two schools of thought here, input the values in the custom tables but don't tie anything specifically to the DNN tables or, tie the tables using Foreign Key constraints. Be sure to set ON DELETE CASCADE.

It has been our experience that the latter is the true Best Practices as it helps keep old data from cluttering up a system. When a module is deleted, as long as ON DELETE CASCADE is set, the risk of impacting a delete operation is minimal. However, if you forget this option, you can cause serious troubles within the DotNetNuke system.

Use Consistent Flow for Module Configuration

Many developers come up with various ways of configuring their modules. Some use the standard "Settings" control method in which you use the default "Settings" action menu item and a custom control with a key set to "Settings". Others add specific action item elements to the modules and require that users go through a separate process.

It has been our experience, that whenever possible, a centralized management point is critical for proper module configuration and ease of use. For this Best Practices we recommend using the standard Settings option. However, we fully understand that this is not the friendliest method if a number of settings are needed as screen real estate is at a premium. If another method of configuration is needed, be sure to add guidance to users, avoid requiring multiple settings forms to be filled out. If multiple configurations are needed, consider implementing a "Control Panel" style configuration to ensure ease of use.

Carefully Consider any Third-Party Inclusions

One of the most important actions to consider when looking at being a good DNN Citizen, is to carefully consider the inclusion of any third-party DLL's with your solution. The reason for this is that with DNN 4.x, there is not support in DNN to control if/when the dll should be added or updated from the system. Now, if your custom solutions are the only items installed on the site, you do not have a lot of risk. When utilizing something similar to the Telerik controls in a custom module, and the site also has a skin that uses the Telerik Rad Menu, if you include the Telerik dll with your module and the administrator removes the module selecting the "Delete Files" button the Telerik dll will be removed as well, rendering the site unusable.

Now, if you are developing specifically for DotNetNuke 5.x, this risk can be mitigated by using the assembly component type. With this component type, DNN will manage references to the assembly and prevent this type of issue.

Implement Needed DNN Interfaces for Development

One of the final pieces of being a good DNN citizen is to be sure to implement common interfaces that allow your module(s) to take advantage of the most common pieces of functionality within the DotNetNuke core. Although not all modules will need to implement each interface, it is important to remember that these interfaces are available and depending on the functionality desired for your module, it might make sense to implement one or more of these interfaces.

The following sections discuss the importance of three of the most common interfaces that are implemented by custom modules. These are considered especially important as they provide critical support features to the framework.

IPortable

The IPortable interface is implemented at the Business Controller class level and is a conduit to allow modules to import and export content. This functionality can be key for many module types. The important items to remember with this interface are that content is exported based on ModuleId. Any other import/export logic would need to be handled by the module in question.

Supporting this interface is critical in situations where individuals are using Portal Templates to setup additional portal sites.

Technology Services and Support ... for the Life of Your Project

ISearchable

The ISearchable interface is implemented at the Business Controller class level and is a conduit that allows modules to publish information to the DotNetNuke search indexer. This is one of the most commonly forgotten interfaces as the process to implement is not the straightest forward. However, this is typically one of the most needed interfaces as it is what allows users to search and find content entered into a custom module. For implementation assistance we recommend looking at this article: http://www.adefwebserver.com/DotNetNukeHELP/ISearchable/ or the Interfaces section of the "Professional DotNetNuke Module Programming" book.

IActionable

The IActionable interface is implemented at the .ascx user control level and is a method used to add additional actions to the DotNetNuke action menu for the module. Implementation of this interface for module actions is considered a Best Practices to provide a consistent set of functionality to users across modules.

Include License File and Release Notes with Package

When packaging your Extension for installation, it is important to include your software license information as well as release notes so that installing users can see the highlights regarding changes in the most current version. For ease of use these files should be separate files within the solution and referenced as such so that HTML formatting can be used to improve the user experience. The IowaComputerGurus', DotNetNuke module templates include this behavior by default for an example.

Cleanup Un-Needed Files

As your Extension grows in maturity, a time will come when you will need to make changes to the file structure and older files will become obsolete. Although not directly harmful to a user's installation, part of being a good citizen is ensuring that you cleanup after yourself.

The DotNetNuke manifest supports a "Cleanup" object type which, if used, will remove files from the installation. This will allow you to remove any old DLL's, images, and controls or other files that your application might have created.

IowaComputer GURUS

Custom Websites and Intranets

.NET Application Development

Expert Technology Support and Training

Performance Optimization

Technology Services and Support ... for the Life of Your Project

Code Specific Requirements

The following section illustrates a few key coding items that should be considered when working with DotNetNuke.

Set ValidationGroup Properties on ALL Validators

It is a very common occurrence for a DotNetNuke module to be placed on a page with multiple other modules. For this reason, it is important when working with any ASP.NET validator controls that you specify a value for the ValidationGroup. If this property is omitted, you can actually cause entire pages to stop functioning as your validators might be accidentally triggered by an action that occurs outside the scope of your module, specifically, but on the same page in the site.

Testing Scenarios

In addition to all of the above scenarios, it is important to note that following a consistent testing process will ensure your Extension has the best ability to play well with the others on an individual DotNetNuke installation. At a minimum basis, we have a few recommendations for testing scenarios for any module that will be distributed broadly.

Installation Types

The first item to think about when testing is to ensure that testing is completed both on a Parent Portal (http://www.mysite.com) as well as a Child Portal, (http://www.mysite.com/child) as both of these environments can exhibit different behaviors when creating links and other setup processes.

Test Scenarios

When looking at the two different types of installations, it is important for each release to not only test a 100% clean installation, but also to test an upgrade to ensure that the module properly upgrades and that the instances of the module on the site still function as they should. Un-installation should also be tested with each release to ensure that all data and files are properly removed and that users can re-install the module should they have a need.

Living Document Notice

This Best Practices guide is considered a living document and will be maintained by IowaComputerGurus. It will include the most up-to-date information possible. If you have specific suggestions for content additions, removals, or modifications, please use the contact information found at the end of the document to contact us. We are always looking for validation and additional information that can be added to this document.

Custom Websites and Intranets

.NET Application Development

Expert Technology Support and Training

Performance Optimization

Technology Services and Support ... for the Life of Your Project

Resources

The following resources will help you when researching/implementing items using the DotNetNuke Best Practices as outlined in this document. We would additionally like to thank Chris Paterra from DotNetNuke Corporation for the guidance he has provided to the greater DotNetNuke community.

- E Wiki Manifest Page - http://www.dotnetnuke.com/Resources/Wiki/Page/Manifests.aspx
- UX Guide - http://uxguide.dotnetnuke.com/
- DNN jQuery Plugins Wiki - http://www.dotnetnuke.com/Resources/Wiki/Page/Reusable-DotNetNuke-jQuery-Plugins.aspx

About IowaComputerGurus Inc.

IowaComputerGurus Inc, a Microsoft Certified Partner organization specializes in developing custom solutions using the Microsoft .NET development stack and often using the DotNetNuke application framework for web application development. Based in Des Moines, Iowa they provide services to customers all over the world and base their business on providing quality, affordable technology solutions with the best customer service. The company is led by Mitchel Sellers a Microsoft MVP, Microsoft Certified Professional and published author.

Contacting IowaComputerGurus

IowaComputerGurus, Inc.
5550 Wild Rose Lane, Suite 400
West Des Moines, Iowa 50266

Phone: (515) 270-7063
Fax: (515) 866-591-3679
Email: webmaster@iowacomputergurus.com
Website: http://www.iowacomputergurus.com

Feedback

IowaComputerGurus is always looking for feedback on our Best Practices guides. If you have suggestions of additional items for inclusion or any modifications to existing content, please use one of the above listed contact methods and we will be sure to update the document accordingly.