# “Module Creator”

Shaun Walker, CTO, DotNetNuke Corporation

# Overview

DotNetNuke is a web application framework with a mission to make web software development simpler and more productive for the average developer. DotNetNuke provides a variety of specialized services that add value over what is available in the standard ASP.NET framework. These include, but are not limited to, a full framework for developing modular web applications, an extensibility model to add extensions to customize the platform to meet your needs, a robust security model which allows granular control over the activities a user can perform in the application, user interface patterns and controls to create a consistent modern user experience, and a skinning engine that allows you to transform the appearance of the application to match your branding.

Every software development platform has its own unique characteristics and conventions which new developers need to understand in order to be effective. The majority of developers are familiar with developing web applications by creating new code assets on the file system, editing the source code in an IDE, and accessing these code assets by direct URL paths in a web browser.

However, because DotNetNuke is a modular CMS application, there is both a file system and database configuration required in order to develop and expose extensions. The file system and the database are wired up based on some unique conventions which are specific to DotNetNuke, which imposes an initial barrier to entry for new developers.

# History

If we go back in time we can evaluate the characteristics of the original application which DotNetNuke grew out of, the IBuySpy Portal. The IBuySpy Portal was released at a time when Microsoft was making a revolutionary platform transition from previous technology to ASP.NET. The intent of the IBuySpy Portal was to provide developers with a fully functional application, including source code, which was simple and intuitive and provided a reference example of how to construct an ASP.NET application. Thousands of developers appreciated the simplicity of this application and proceeded to customize it to meet their specific business needs.

In terms of customization, the architecture was extremely straightforward and intuitive. To create a new module you needed to create the following files ( within folder structures which already existed in the IBuySpy Portal application ):

/Components  
 AnnouncementsDB.vb <- this contained your data access methods using simple ADO.NET methods  
/DesktopModules  
 Announcements.ascx <- your “view” control exposed through the portal user interface   
 EditAnnouncements.aspx <- your “edit” page for managing content

The registration of the module was simple as well as you only needed to add a row to a single database table, ModuleDefinitions, and set the DesktopSrc field to point to the \*.ascx file.

The IBuySpy Portal application was highly successful at providing a simple “getting started” experience for developers who wanted to get up to speed on ASP.NET. The application was simple and “hackable” by developers with limited experience, yet it was still capable of being utilized in enterprise scenarios as well. This encouraged rapid adoption of ASP.NET and in fact, provided the initial momentum for the DotNetNuke ecosystem growth.

This simple, intuitive approach has been replicated in other non-Microsoft platform products. Most of the open source CMS products on the LAMP stack today exemplify many of these traits. Wordpress, Drupal, and Joomla! have managed to build massive developer communities by providing applications which are extremely “hackable” and intuitive for inexperienced developers.

Although DotNetNuke grew out of the IBuySpy Portal, years of enhancements to the architecture to make it more enterprise-grade, more extensible, and more maintainable have resulted in the platform losing some of its “hackable” nature. Ironically, all of the simple techniques for developing modules are still supported; there are just very few examples on how to take advantage of them effectively. For example, the HTML/Text module was traditionally developed in the simple, “dynamic” Module pattern and many developers used it as a reference example of how to build their first module. In 2010 this module was converted to the compiled/WAP style which reduced its effectiveness as a simple reference example.

For developers on the Microsoft platform who are familiar with ASP.NET we usually tell them that the DotNetNuke development model is consistent with what they are used to – that they just need to create User Controls ( \*.ascx ) instead of Pages ( \*.aspx ). The reality is that it is significantly more involved than that, with the initial challenge being where to start in terms of coming up to speed on the specifics of the DotNetNuke architecture, environment, and conventions. It is only after a developer navigates past this first step, that they can then utilize their ASP.NET skills productively and understand the value of the platform.

# Business Need

Traditionally, developers have had to rely on documentation and sample modules in order to obtain the knowledge required to become productive on the platform. Unfortunately, the documentation and sample modules are not well organized, nor are they focussed on specific developer personas or why one approach is better than another depending on the business scenario. This is not an optimal model. Over the years, there have been a number of different approaches to attempt to deal with this problem.

In DotNetNuke 4 we introduced Project Item Templates which could be utilized within Visual Studio to create new modules. The problem is that the Visual Studio template system is not discoverable or intuitive for new developers, the tokenization capabilities are limited, and the packaging format is specific to Visual Studio and has changed multiple times. In addition, it does not deal with the wiring up of the file assets with the database; meaning that a developer still needs to figure out how to register the module and make the DotNetNuke application “aware” that it exists.

DotNetNuke 4 shipped the same time as ASP.NET 2.0 and allowed developers to build dynamic modules in the Web Site Project (WSP) model in the new free tool Visual Web Developer. DotNetNuke platform usage by developers soared as a result of this – with the project experiencing its highest levels of growth in 2005-2007. In DotNetNuke 4 we also introduced the Module Packager which provided a wizard for generating the manifest and packaging all of the assets related to a module. The packager was well suited for packaging dynamic modules for redistribution and it got a lot of usage.

In DotNetNuke 5 we overhauled the “Create New Module” user interface to reduce the number of steps required to register a module in the framework and create some basic module assets on the file system. This was done in the application itself which made it slightly more discoverable and it did deal with both the file system and database configuration. However, the template aspect was extremely limited and still did not go far enough to shield a developer from the unique configuration and convention requirements of the DotNetNuke platform. It also resulted in a fragmented experience where the developer needed to navigate to too many different areas, and provide too much information, in order to get up and running.

We have the opportunity to improve the developer experience and solve the problems outlined above.

# Goals

* Improve the “Getting Started” experience for new developers
* Shield developers from the specific configuration and convention details required to create extensions
* Improve the discoverability of development options and make them more intuitive and integrated within the CMS environment
* Make development possible in a variety of scenarios, including on-premise or in the cloud, and with a variety of IDEs and code editors
* Make development more accessible to non-Microsoft developers
* Ensure the solution is flexible to allow developers to customize it
* Get DotNetNuke “out of the way” of developers

# Use Cases

There are a variety of different use cases for different audiences which this solution could cater to:

* Experienced DotNetNuke developers.
  + Very often a module has requirements for nothing more than a single user control. In these cases, it may be overkill to set up an entire project with compilation, etc… Modules created in the dynamic style are fully capable of satisfying the business requirements ( in fact, all of the core Admin/Host functionality is developed and maintained in this manner ).
  + Experienced DotNetNuke developers can also utilize this solution to generate the initial assets they need as part of their module, and then switch over to the IDE of their choice to perform the actual software development. The benefit is that this solution creates all of the basic files and wires them up with the DotNetNuke application so that the developer can focus on writing code – not configuring the environment.
* Experienced ASP.NET developers who have no familiarity with DotNetNuke.
  + This solution helps developers who are unfamiliar with DotNetNuke create the initial environment for building a module. They will be able to see where the code files are created, the conventions in the code assets, and the database configuration. This will make them more immediately productive.
* Inexperienced ASP.NET developers
  + This solution is perfect for web agencies and part-time developers in I.T. departments who are tasked with adding additional functionality to the website and who do not have a thorough software development background. These “cut-and-paste”-style developers will be able to create custom modules in minimal time.
* QA Testing
  + When developing APIs you often need a way to test the API methods to determine if they are exhibiting the expected behavior. This solution allows you to create simple test harnesses for calling API methods.
* Prototyping
  + When developing solutions for clients you often want to create a simple proof of concept to demonstrate specific functionality. This solution allows you to build prototypes very quickly and simply.
* Non-Microsoft developers.
  + It is possible to build modules using basic HTML,CSS,Javascript techniques. This solution can allow developers with this skillset to build modules and not be concerned with the underlying technology platform ( ie. Microsoft ASP.NET ).

# Scope Definition

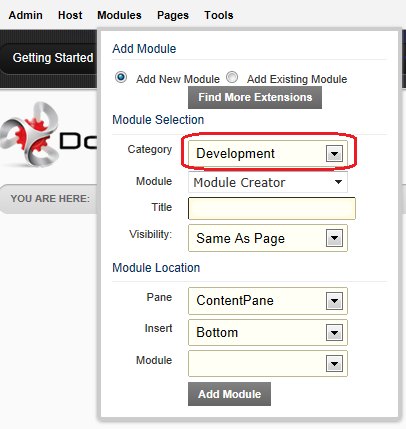
This is NOT a templating/scaffolding solution to enable developers to generate large amounts of source code. It is a solution to abstract the details of the development environment, enforce best practice configuration through convention, and very quickly get a developer to the point where they can begin to write code.

This solution focussed on dynamic module development techniques which are more applicable to the entry level or productivity developer persona who are building modules for internal consumption. Developers who are building commercial-grade modules or want to follow more “enterprise” development practices will be better served by Visual Studio templates which allow you to create pre-compiled modules.

# Solution

Developers who want to build extensions for the DotNetNuke platform must first be familiar with the basic CMS user experience for creating pages, module instances, and content. With this in mind, creating a new custom module should be no more difficult than it is to add an existing module to a page.

Leveraging the standard user experience, we will create a new “Development” module category in the Add Module menu in the Control Panel. A new “Module Creator” module will be available in this category. This module is only relevant to Host users. The should make the option very discoverable and accessible for developers ( ie. they do not have to navigate the various administrative menus ).



This Module Creator module can be added to any page just like a normal DotNetNuke module. It exposes a number of basic options which a developer will need to provide in order to create a custom module:

**Owner Name:** DotNetNuke requires a unique top level namespace to ensure uniqueness. The owner name could be an organization name, brand name, developer’s name, etc... The system will save the value provided as a Host Setting so that it can be used as a default for any modules created in the future.

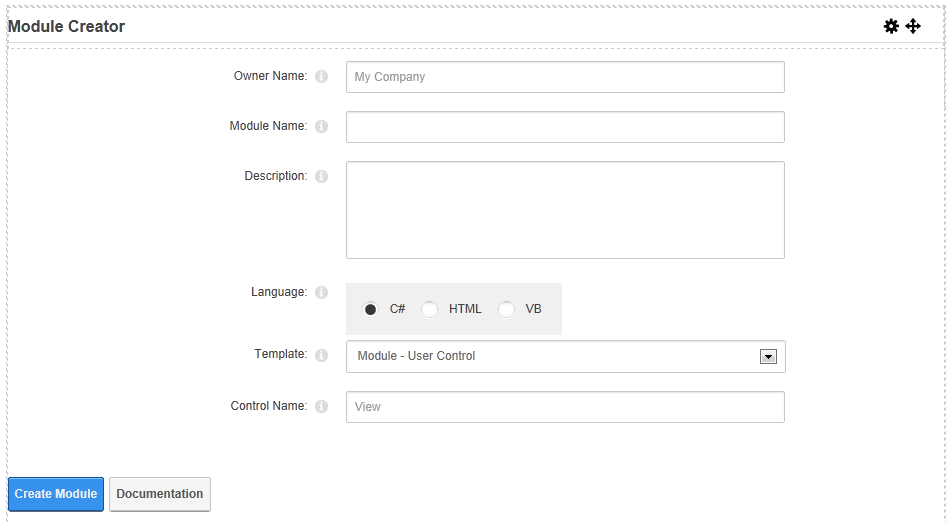
**Module Name:** A short name to identify the purpose of the module

**Description:** Optional. A description of the module

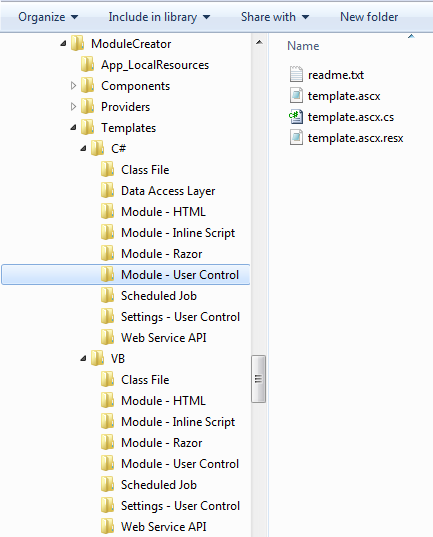
**Language:** C# or VB. The default is C#. Changing the selection reloads the list of Templates.

**Template:** A list of templates loaded dynamically based on the Language selected. It will only display templates of type “Module”. There are 3 different dynamic module models which can be utilized based on preference – Inline Script, User Control, and Razor. The default will be “Module – User Control”. When a user selects a template, it will load and display an associated “readme.txt” file ( if one exists ). The readme.txt file can contain a description of the template files, but it can also include code samples which could be cut and pasted into the module control to add additional functionality.

**Control Name:** Modules can contain multiple controls. This option allows the user to specify the name of the file created. The default will be “View”



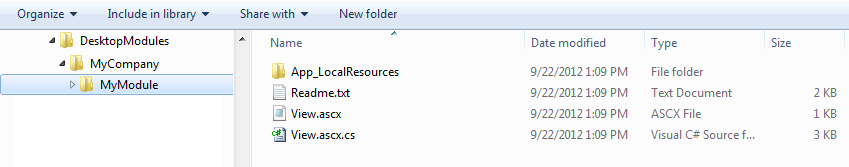
The templates are stored in the /DesktopModules/ModuleCreator/Templates folder based on Language. Additional child folders can be created within the Language folders and will be recognized within the Module Creator in the Template list. Files that are named “template".\* will be renamed to the Control Name when the module is created. These templates can contain tokens such as [OWNER], [MODULE], [CONTROL], [YEAR] which will be substituted at run-time with the parameter values supplied. Template files must contain at least enough content to make the asset usable within the environment, but there is no limit to the amount of content they can contain. In the Module templates provided, they contain the UI script for the standard Form Pattern in DotNetNuke including structure and default CSS classes.



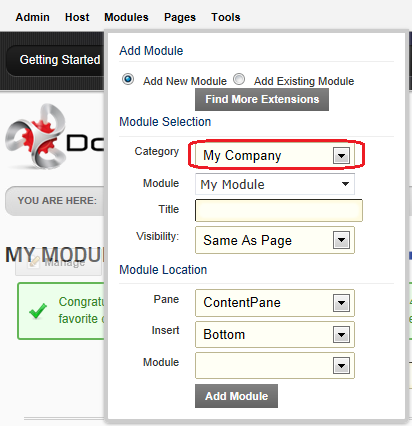
When a user selects Create Module, the system will utilize the parameters entered to create the module folder within /DesktopModules, copy the files from the Template folder selected and rename them if necessary, substitute the tokens embedded in those files, register the module in the database, create a module category for the Owner and associate it to the module, and update the ModuleDefID of the module instance with the ModuleDefID of the newly registered module. The end result is that the Module Creator module will be “replaced” on the page with the new module created. At this point the module is fully usable within the environment and the developer can browse to the page where it is located using the standard website navigation.



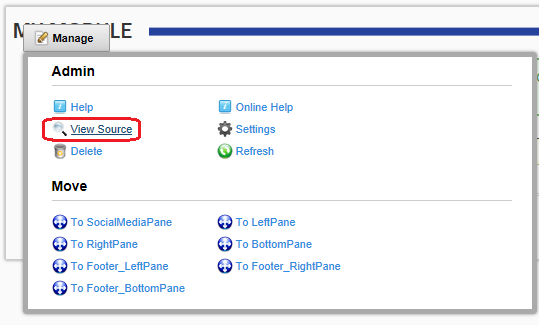
Behind the scenes, the system has created the module in the /DesktopModules folder and created the various files from the Template.



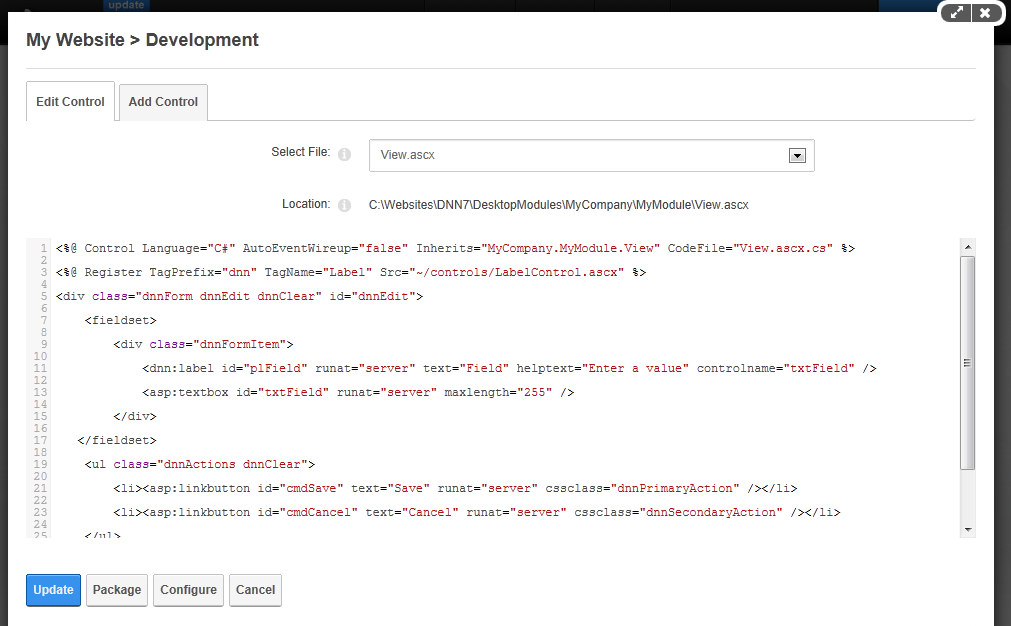
A module category is created automatically for the Owner Name so that all of the custom modules created by the developer are easy to locate later in the Add Module menu in the Control Panel:



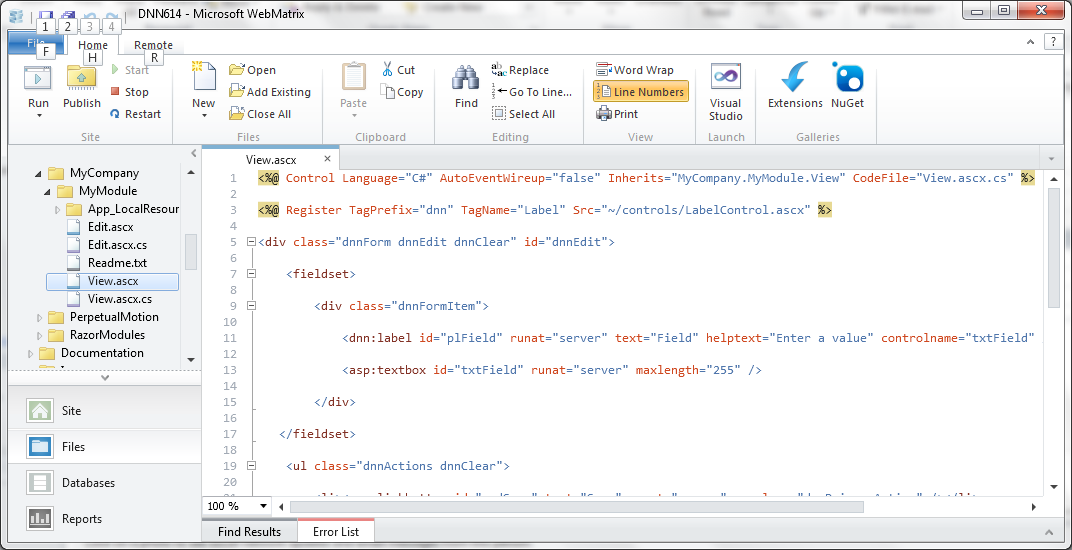
Once the module is created, it is possible to edit the source code of the module using the Module Actions menu, View Source option:



The View Source option is enhanced to do much more than it does today. The Edit Control tab allows the developer to view and edit the source code for any file associated to the module. By default the Select File: option selects the file identified as the “view” control. Selecting a file from the list will display the full path of the file as well as the contents of the file. The contents are displayed using an open source editor control – CodeMirror – which provides code highlighting and line numbers. Modifications can be made to the code and it can be saved in real-time for immediate results in the application. You also have the ability to Package the module directly from this user interface ( stepping you through the standard Extensions Packaging wizard ) and Configure the module in the Edit Extensions user interface.



Or you could use a more robust development environment such as Web Matrix or Visual Studio where the code assets created on the file system will be immediately accessible:



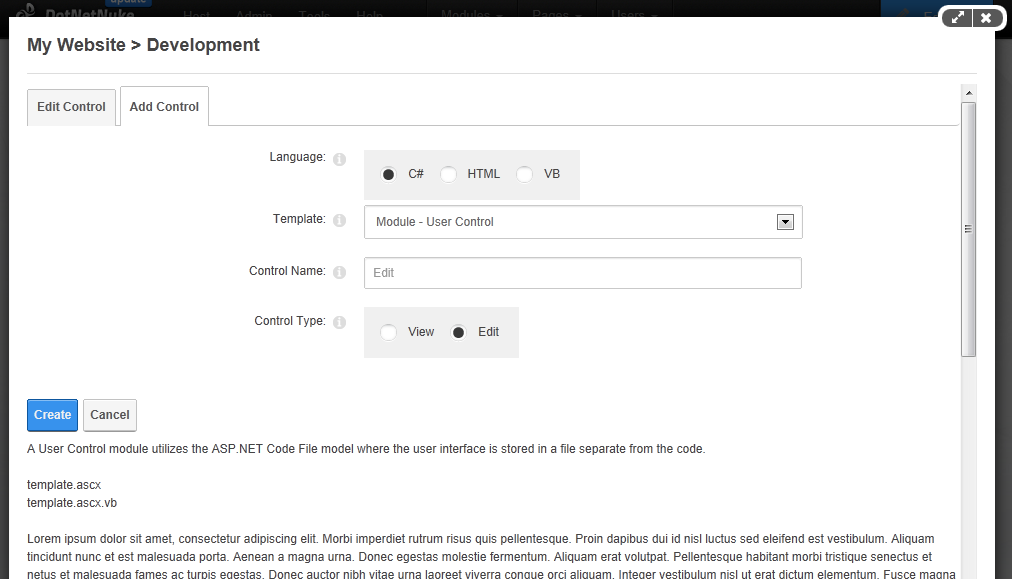
The View Source option within the DotNetNuke application also contains a new Add Control tab that displays some new options which allow you to create additional assets for this module. The goal of Add Control is to allow a developer to add individual web assets to their Module folder ( ie. user controls, class files, style sheets, javascript, etc… ):

**Language:** C# or VB. The default will be the language that the module was originally created in. Changing the selection reloads the list of Templates.

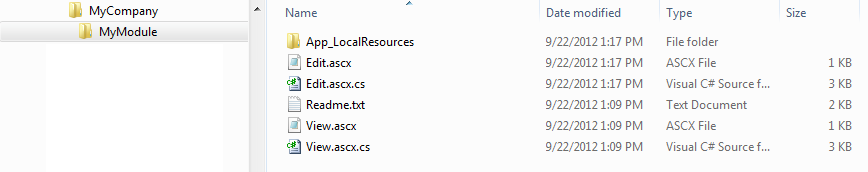
**Template:** A list of templates loaded dynamically based on the Language selected. It will display both files of type “Module” as well as other file types, including the ability to create Class Files. When a user selects a template, it will load and display an associated “readme.txt” file ( if one exists ).

**Control Name:** Modules can contain multiple controls. This allows the user to specify the name of the file created. The default will be “Edit”

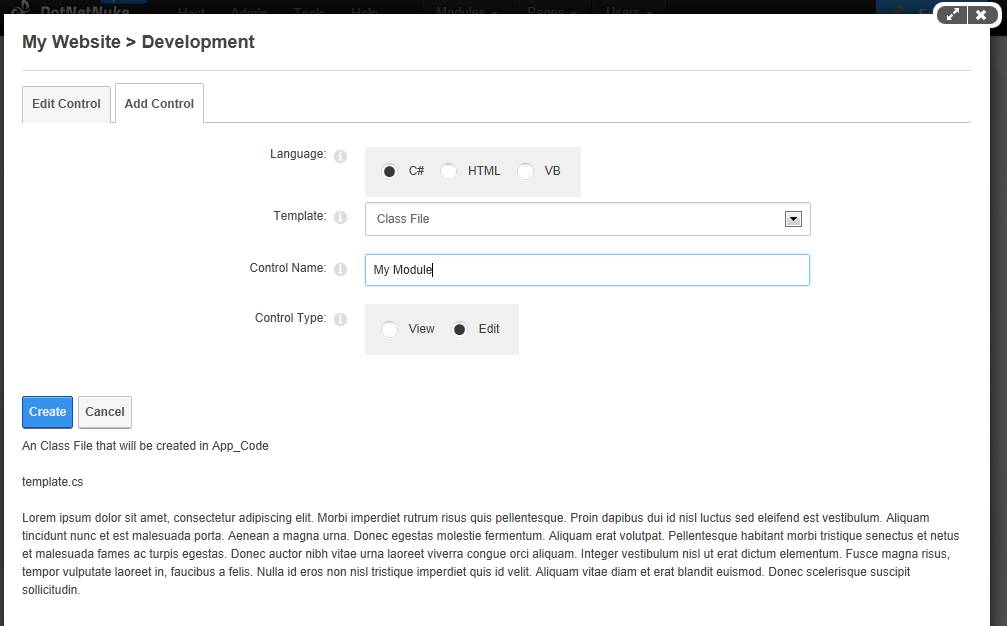
**Control Type:** Modules have different security modes which are supported by the platform and integrated with the user permissions. The default will be “Edit”



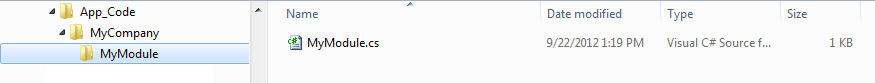
Selecting Create will copy the files from the template folder, replace any tokens, and register the new module control in the database.



You can also create dynamic Class Files which are copied to the App\_Code folder and can be edited through the Edit Control tab.



The Class File was created in App\_Code in a subfolder matching the same structure as /DesktopModules for the module.



All files related to the module can be viewed and edited in the Edit Control tab:

