

TOP 60 CUSTOM SOLUTIONS

BUILT ON MICROSOFT SHAREPOINT SERVER 2010

YAROSLAV PENTSARSKYY

Opinions expressed in this books are of the author only and do not reflect opinions of product vendors, or companies the author worked/ works for. Information in this book is distributed on an "as-is" basic, without warranty. Although every effort has been made in the preparation of this work, neither the author nor publisher or any other party affiliated with the the production of this book shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

This book is an independent publication and is not affiliated with, nor has it been authorized, sponsored, or otherwise approved by Microsoft Corporation.

Microsoft, SharePoint®, and Windows® are trademarks of the Microsoft group of companies

Cover images used for this book are taken from from nasa.com and istock photo.com.

Cover Page Artwork by: Luis Ponce

Copyright © 2010 Yaroslav Pentsarskyy

All rights reserved.

ISBN: 145287736X

ISBN-13: 9781452877365

Library of Congress Control Number: 2010907231

INTRODUCTION

Many of you are already familiar with functionality and features available in past versions of SharePoint. However, the new version of SharePoint has many great improvements and new features to offer. As with every technology, many features are backwards compatible, and you can continue building solutions as you do now, as most of those are still going to be understood in the new version. The best approach, however, is to get complete coverage of what's available for you in the new release, especially components that you had to build from the grounds up before. Every time a new technology such as SharePoint is released I see new books and materials available in the market. Most of the time, books are released few weeks before the technology reaches the release to manufacturing stage, which is understood, because many of us have access to the pre-release version of various tools. The pre-release version of tools usually is almost complete and usually resembles the release to manufacturing version ... right? I mean what major features could change in next three months? Well, if you're like me, you probably know that there are many things that usually change before the product reaches release to manufacturing stage. In fact, the time span from beta to release to manufacturing is when the product team hopes to fix all of the issues reported by the early adopters. This book has been in its initial stage since the latest beta of SharePoint to early adopters. After new version of SharePoint was released, all of the examples you will see in chapters here were reviewed for changes to their original scenarios. I do hope you will find this book resourceful and complete with examples that closely resemble your scenarios and actually work from the first time.

WHOM THIS BOOK IS FOR

This book is intended for two types of audiences:

A .NET developer with little experience in SharePoint looking to start developing using the latest release of SharePoint.

A SharePoint developer gearing up for the new release of the product.

Whoever from the two above you might be, here are some of the assumptions I made about why you would buy this book. If I'm going to buy a book that will tell me everything that I can read by pressing F1 in Visual Studio, or everything I can ready by going to the "What's new" section of the product page, I probably wouldn't buy it. Also, I don't want to buy a book to learn abstract scenarios that I will never work with. I assume you're thinking the same.

Here are the three objectives that this book is trying to achieve:

Get you familiar with technical and business scenarios rather than features of the product.

When you're in the middle of reading vague requirements on what you actually need to implement, it's hard to see how all of those great new features you're reading about will fit together to solve your problem on time. This book navigates through business and technical scenarios and, hopefully, it will get you to the correct solution right from the table of contents.

Give you a real code with all of the steps you need to do to run it.

We've all been there. You have a technical problem, you find the solution someone has posted online, you put it together ... - it doesn't work. The last thing you want to see is a sample that's broken, and then few hours later find that post author forgot to include one step that you never thought you would be required to do. This book comes with references to downloadable solution files that have been compiled and tested.

Get you thinking about what else you can do with features described and scenarios used.

Association is a powerful thing. You can read about a feature or a sample that is not related to your case in any way, yet sometimes you will get enough information to get started on your own solution. Through-out each sample you will find pointers that will get you thinking how else you could use the feature discussed.

With this set of objectives, I really do hope you will find this book everything you ever wanted a developer book to be and nothing less, nothing more. I understand you may not be as excited reading it, as I was writing it; after all, it's just a manual organized a bit differently to help you get things done easily.

With that, I hope you have fun reading this book and if you want to give me your feedback – visit my blog and drop me a note: www.sharemuch.com.

ABOUT THE AUTHOR

Yaroslav Pentsarsky has been architecting and implementing SharePoint solutions since its 2003 release. Yaroslav has extensive .Net and SharePoint development experience working with medium-sized businesses, non-profits, and government organizations.

As a recipient of the Microsoft Most Valuable Professional (MVP) 2009 Award, Yaroslav is also a developer audience leader for VanSPUG (Vancouver SharePoint Usergroup) and actively contributes to local and not-so-local technical communities by presenting at local events and sharing his findings in his almost-daily blog: www.sharemuch.com.

Outside of work, Yaroslav enjoys travelling and maintains a growing "places-to-visit" list.

ACKNOWLEDGEMENTS

Throughout my career in software development I have been involved with variety of organizations. I want to give a special thanks to my dynamic team at Habanero Consulting who actively supports and contributes to local and worldwide communities, and especially local SharePoint community here in Vancouver. A lot of my knowledge comes from interacting and exchanging ideas with my team.

My thanks also extend to local and worldwide usergroup leaders with whom I have been involved with over the last few years. Your involvement in the community helps people tremendously, please keep doing what you're doing.

I also would like to mention and thank to all of the bloggers out there who share their ideas online. It takes a lot of effort and dedication to post all of those tips and innovations on your blogs almost every day. Without those posts we wouldn't have such a great community and so many innovative ideas and solutions.

BOOK SOURCE CODE

This book comes with a source code for each chapter. As you go through the book – you will see many examples and source code. The source code in the book is for you to follow the flow of the logic that is referenced, not to type it out into your Visual Studio. If you would like to see actual example in action, download the corresponding chapter source code that is ready for you to compile and use. The source code samples assume you're running the system with:

- Microsoft SharePoint Server 2010 installed.
- Visual Studio 2010 Professional or higher installed.
- PowerShell enabled.
- Sufficient account permissions to access SharePoint 2010 Central Administration and Service Applications.

All source code can be downloaded from the 'Downloads' section at www.sharemuch.com. Below is the source code index for each sample:

CHAPTER 1

Solution Packaging
Custom Solution Deployment Script
Referenced Assemblies in Your SharePoint Solution

CHAPTER 2

- List Item Validation
- List Item Security
- Excluding List Items from Search Crawl
- Creating Custom Permissions Levels
- Enforce SharePoint List Relationship Behavior
- Working with SharePoint List Event Receivers
- Aggregating Contents of Lists, Queries, and Rollups

CHAPTER 3

- Creating Custom List Item Detail Forms
- Field Level Security in Your SharePoint List Forms
- Manage Behavior of SharePoint 2010 Composite Fields
- Defining List View Look and Feel in Your Custom List Schema
- Adding Web Parts to Item Detail View Form

CHAPTER 4

- Creating External Content Types with Visual Studio
- Exporting and Importing Your BDC Model
- Importing BDC Models into Visual Studio
- Provisioning SharePoint External List Schema Programmatically
- Executing Queries on External Lists

CHAPTER 5

- Process Automation and Scheduling Long Running Operations

CHAPTER 6

- Creating Your Own User Profile Properties
- Creating SharePoint User Profiles Programmatically
- Add New Terms to SharePoint User Profile Properties
- Retrieving Taxonomy Types Properties from SharePoint User Profile
- User Profile Integration with Out-of-the-box Features: Update MySite Status Message Programmatically
- Programmatically Enable Rating on SharePoint Lists and Libraries
- Adding Item Rating Control to Your Custom List Forms
- Tagging Content Programmatically with Managed Metadata Service

CHAPTER 7

- Creating Basic SharePoint Ribbon Controls
- Creating a Fly out Anchor on Your Ribbon
- What if your Ribbon Java Script is Too Large for One File
- Working with Ribbon Groups and Tabs
- Creating Site Level Ribbon Tabs
- Determining the State of Ribbon Tabs and Hiding Ribbon
- Opening Modal Windows upon Ribbon Control Clicked

CHAPTER 9

- Defining Site Templates and Driving Site Content
- Provisioning Page Content to Your Pages Programmatically
- Provisioning Other Web Parts and Views on to the Page
- Provisioning Several Pages with One Module
- Provisioning Web Parts directly to page layouts
- Rendering Additional Page Specific Metadata during Page Edit
- Programmatically Hide SharePoint Web from default Navigation
- Limiting Allowed SharePoint Page Layouts on a Desired Web
- Setting Automatic Page Title for SharePoint Default Pages

CHAPTER 10

- Limiting the List of Available Page Layouts with an Application Page
- Displaying SharePoint “Processing” Page during Your Long Running Operations

CHAPTER 11

- Extending Visual Studio Server Explorer Window with New Nodes
- Creating Visual Studio Project and Item Templates

TABLE OF CONTENTS

Introduction	iii
Whom this book is for	v
About the Author	vii
Acknowledgements	ix
Book source code	xi
Chapter 1 Setting Up for Success: Visual Studio 2010 Solution Structure and Deployment Scripts	1
Solution Packaging	7
Custom Solution Deployment Script	9
Referenced Assemblies in Your SharePoint Solution	18
Debugging Your SharePoint Applications	23
Chapter 2 Lists and Libraries: List Rollups, Security, and Integration with the Rest of SharePoint 2010 Components	27
List Item Validation	xx
List Item Security	xx
Excluding List Items from Search Crawl	xx
Creating Custom Permissions Levels	xx
Enforce SharePoint List Relationship Behavior	xx
Working with SharePoint List Event Receivers	xx
Aggregating Contents of Lists, Queries, and Rollups	xx

Chapter 3 Lists and List Items: Changing the Look of Forms xx and Incorporating Custom Logic into Item Forms

Creating Custom List Item Detail Forms	xx
Field Level Security in Your SharePoint List Forms	xx
Manage Behavior of SharePoint 2010 Composite Fields	xx
Dynamically Changing SharePoint 2010 List Form Rendering Templates.	xx
Making Changes to List View and List Item Detail View Using XSL.	xx
Defining List View Look and Feel in Your Custom List Schema.	xx
Adding Web Parts to Item Detail View Form	xx

Chapter 4 Using External Data with SharePoint 2010 Out-of-the-Box xx Components and Custom Features

Connecting to SQL Server Data Source	xx
Creating External Content Types with Visual Studio	xx
Exporting and Importing Your BDC Model	xx
Importing BDC Models into Visual Studio	xx
Provisioning SharePoint External List Schema Programmatically.	xx
Executing Queries on External Lists	xx
SharePoint External List Item Throttling and Limits	xx

Chapter 5 Process Automation and Scheduling Long Running Operations . . . xx

Chapter 6 Metadata, Tags, Rating: Working with and xx Extending Social Features of SharePoint 2010

Creating Your Own User Profile Properties	xx
Creating SharePoint User Profiles Programmatically	xx
Add New Terms to SharePoint User Profile Properties	xx
Retrieving Taxonomy Types Properties from SharePoint User Profile	xx
User Profile Integration with Out-of-the-Box Features:.	xx
Update MySite Status Message Programmatically	
Getting Started with SharePoint Social Rating Feature.	xx
Changing Update Frequency of SharePoint Rating	xx
and Social Data Synchronization	
Programmatically Enable Rating on SharePoint Lists and Libraries	xx
Adding Item Rating Control to Your Custom List Forms.	xx
Working with Managed Metadata Service and Tagging Features.	xx
Tagging Content Programmatically with Managed Metadata Service	140

Chapter 7 Creating SharePoint 2010 Ribbon Components xx and Managing Existing Ribbon Elements

Creating Basic SharePoint Ribbon Controls	xx
Creating a Fly Out Anchor on Your Ribbon	xx
What If Your Ribbon Java Script Is Too Large for One File	xx
Working with Ribbon Groups and Tabs.	xx
Creating Site Level Ribbon Tabs	xx
Determining the State of Ribbon Tabs and Hiding Ribbon.	xx
Where is SharePoint Out-of-the-box Ribbon Defined?	xx
Opening Modal Windows upon Ribbon Control Clicked.	xx

Chapter 8 Search: Extending Search Components xx and Incorporating Search Features in Your Portal

Add Your Own Search Refinement Categories.	xx
Adding New Metadata to Your Search Results View	xx
Adding Graphic Representation of Item Rating to Your Search Results.	xx

Chapter 9 Working with SharePoint 2010 Publishing and Custom Pages. xx

Getting Started with Creating Custom SharePoint Pages	xx
Defining Site Templates and Driving Site Content	xx
Provisioning Page Content to Your Pages Programmatically	xx
Provisioning Other Web Parts and Views onto the Page	xx
Provisioning Several Pages with One Module	xx
Provisioning Web Parts Directly to Page Layouts.	xx
Rendering Additional Page Specific Metadata during Page Edit	xx
Using SharePoint Publishing Site Navigation Properties.	xx
Programmatically Hide SharePoint Web from Default Navigation	xx
Hide Unused SharePoint Site Templates	xx
Limiting Allowed SharePoint Page Layouts on a Desired Web	xx
Setting Automatic Page Title for SharePoint Default Pages.	xx

Chapter 10 Adding Custom Logic to Your Site Using Application Pages. xx

Limiting the List of Available Page Layouts with an Application Page	xx
Displaying SharePoint "Processing" Page during.	xx
Your Long Running Operations	

Chapter 11 Extending Visual Studio 2010 to Speed Up	xx
and Standardize Your SharePoint 2010 Projects	
Extending Visual Studio Server Explorer Window with New Nodes	xx
Creating Visual Studio Project and Item Templates	xx

CHAPTER 1

Setting Up for Success: Visual Studio 2010 Solution Structure and Deployment Scripts

Getting Visual Studio 2010 solution structure right for your SharePoint 2010 projects is a crucial element for successful solution deployment, future maintenance, and enhancements a few months from now. With Visual Studio 2010, you have much better support in terms of creating solution elements such as modules, event receivers, and features. With that support, you can create solution elements almost anywhere and really confuse your sustainment team (even if it includes you) three months down the road, or confuse other developers trying to pick up where you left off.

Each SharePoint solution will have few logically separated Visual Studio projects serving various functions.

For example, if part of your SharePoint solution will deliver content to a site, the other part will provision Web Parts and lists, and yet another part will provision workflow—it will be fairly confusing and irrational to place them all together as the same Visual Studio project.

Instead, it's best to create a:

- Platform project –to handle provisioning of core components; all the Web Parts and list definitions will go here.

- Content project – to provision default content to pages when the site is deployed –after all you don't want to have a site with nothing on it when customer receives it.
- Services project – to store all of your constant classes as well as any logic that will interact with non-SharePoint systems such as reading and writing to your **web.config** file.
- Branding project – to place all of the components that will take care of your solution branding –items like themes, along with their CSS files, images, and even any footer and header controls.

Here is a typical solution structure for a Platform project:

Platform Project Root

- Features
 - Master Page provisioning feature
 - Page layout provisioning feature
 - Page provisioning feature (s)
 - General Web Part provisioning feature
- Pages
 - Page Module
 - Page specific Web Part module
 - Page specific list instance
- Page layouts
- Master pages
- ControlTemplates *
 - Project specific custom user control
- Template *
 - 1033
 - Project site template definition
 - Site Templates
 - Project site template
- Controls
 - Custom controls

- Layouts *
 - Folder (project name)
 - Layouts ASPX page
- List definitions
 - List definition
- Lists
 - List Instance
- Web Parts
 - General Web Part

The rule of the thumb here is that you should keep all of your solution items hierarchical unless they are generic enough for other elements to reuse. For example, you see that I have "Page specific Web Part module" right under pages; this is mainly due to those custom Web Part modules being used on that page only, and nowhere else. If you anticipate using your Web Part modules throughout your site on multiple pages, the modules for such Web Parts should sit under Web Parts of the root of your SharePoint project (see General Web Part in my tree).

- Content Project Root
- Features
 - Page provisioning feature (s)
- Pages
 - Page module

You will notice that this Visual Studio project is much simpler than Platform, here we'll only place content provisioning XML files for your pages.

Services Project Root

- Constants
- List constants
 - Site constants
 - Web Part constants

- List Helpers
 - Service query helper

The above structure is for a Visual Studio project that is a type of a **Class Library** and not SharePoint project at all. The above project will produce a DLL with all of the classes you might create as you create artifacts for your solution, but in its essence, this is just a container for everything that helps your SharePoint solution but doesn't necessarily belong to it. There is one more benefit to keeping those artifacts separate; when it comes to bug fixes or functionality upgrades, it's easier to replace one DLL that keeps all of the supporting functionality than replacing core **Platform** DLL.

Branding Project Root

- Features
 - Theme installer
 - Theme setter
- Controls
 - Optional header control
 - Optional footer control
- Template *
 - Layouts
 - 1033
 - Styles
 - Themable
 - Project folder (containing images and style artifacts)

The above project will hold your entire theme related artifacts and elements. This way if you need to upgrade a few images or maybe a CSS markup, you can always do it separately without disturbing the rest of the **Platform** solution items.

The above four Visual Studio projects are not mandatory, of course, and if you have really small projects that require no branding or content provisioning, you would not include those respective Visual Studio projects.

Now that we know what goes where, let's go through a few small technicalities on how to create all of those projects in Visual Studio 2010.

We will start by firing up an instance of Visual Studio 2010 and creating a new SharePoint 2010 project as shown below.

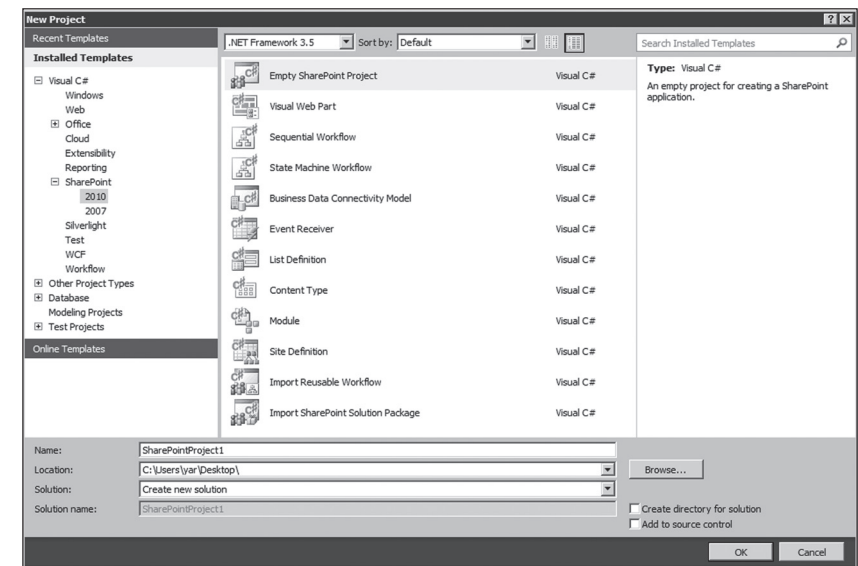


Figure 1-1 Creating SharePoint 2010 project in Visual Studio 2010

Remember, despite all of the .NET 4.0 goodness available in Visual Studio, we will use .NET 3.5 since this is the framework that SharePoint 2010 uses.

Next, you will be asked to choose whether your solution is a **Sandbox Solution** or **Farm Solution**. In this book, we will be deploying all of our applications as **Farm Solutions**. You will also be asked about the name of the site you wish to use for debugging. It is handy to specify the site that most closely resembles the site template you are creating solutions for—for example: **Team Site**, or **Publishing Site**.

After you picked the site and the type of solution, click **Finish** and your SharePoint project will be created. Your Visual Studio Solution Explorer will contain a single solution with a project in it.

NOTE:

One of the few advantages of creating **Sandboxed Solution** is when the administrator of the farm deploys your solution; they have a choice to decide what level of access to give to your application as well as define a threshold when to disable your solution if it reaches the limits defined in SharePoint 2010 configuration. Some types of project components—for example, Web Parts—require being deployed on a farm, and therefore, cannot be used in a Sandbox Solution.

The project you created will be your **Platform** project, not because it has to be in that sequence.

Next, navigate to your **Platform** project properties and give a meaningful name to your assembly and namespace. You can see the convention that I recommend, below.

- Assembly Name: **SolutionName.Platform** – for **Platform** project.
- Default Namespace: **SolutionName.Platform** – assuming it's a **Platform** Visual Studio project.

If you remember our **Platform** project structure, it had number of folders and modules. There is a difference between a regular Visual Studio folder and a special mapped folder. A mapped folder is a folder that is mapped to a specific SharePoint 2010 directory under the SharePoint Root (aka: *[Drive]:\Program Files\Common Files\Microsoft Shared\Web Server Extensions\14*).

To create a mapped folder:

1. Right click on the Platform project.
2. Select Add.
3. Select SharePoint mapped folder.
4. Pick the folder you would like to map (TEMPLATE for example).
5. Click OK.

Now you will see a new folder in your solution structure. In the **Platform** project structure, all of the mapped folders are identified with an asterisk (*).

After I populate my solution structure with the hierarchy we discussed earlier, my Solution Explorer will look similar to what is shown below.

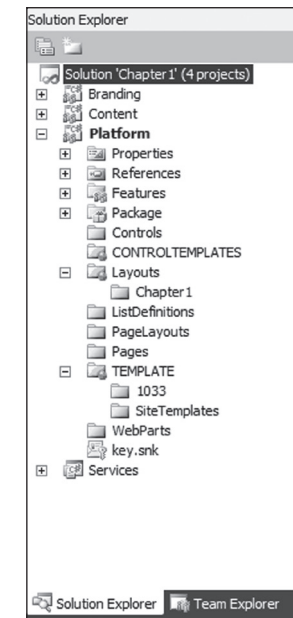


Figure 1-2 Platform project solution structure

Now the remaining pieces left are, at least, **Services** and **Branding** project structure, which follow the same principle.

Solution Packaging

You have created your Visual Studio solution structure and are eager to add components. You can perform manual solution deployment to your SharePoint 2010 farm using tools in SharePoint 2010 Central Administration. As an alternative, you can choose to automate your deployment steps so that every time you do a test deployment—we all know how often that happens—you don't have to go through a series of pages and clicks.

In Visual Studio 2010, you have a great new feature allowing you to build, package, and deploy your solution with a right click of all of the three options in Solution Explorer. In most cases, when you're dealing with a simple set of tasks, the separation between build and package

and deploy is OK; In more complex scenarios, you may want to build and package all at once or maybe even copy your resulting solution files (aka WSP) to a destination folder where your automated scripts pick it up and deploy. If the latter is the scenario that sounds like you, we're onto our first custom mini-solution: setting Visual Studio to package on build.

Each SharePoint project you create will come with a project definition file that will outline solution variables such as solution name and whether you're in configuration or release mode. You can modify the project configuration file on each of your SharePoint projects (**Platform, Branding**, etc) to perform your own build tasks when the solution is built in our case package task.

Build tasks are a set of commands you want to execute during build, and most of them are pretty standard. In fact, Visual Studio 2010 comes with a great set of SharePoint tasks, which in past many development teams created manually. You can find a set of tasks here, by opening the file in Notepad: `[Drive]:\Program Files (x86)\MSBuild\Microsoft\VisualStudio\v10.0\SharePointTools\Microsoft.VisualStudio.SharePoint.targets`.

NOTE:

On a topic of simple code editors I prefer Notepad ++ which is a free for download editor that also numbers your lines and colors your code keywords in all sorts of programming languages making it very easy to navigate through XML files and other code.

In this case, we're after a **CreatePackage** command that creates solution packages (aka WSP files). As you see, the **CreatePackage** is already defined in `Microsoft.VisualStudio.SharePoint.targets`. Here is how we add it as a build task.

To change the standard Visual Studio build order execution, we'll follow the steps below.

1. In your Visual Studio Solution Explorer, right click on the Platform project, and in the menu select Unload Project.
2. Right click on the project again and select Edit Platform.csproj. This will load the solution definition file in an XML compatible editor.

3. Scroll down to the very bottom until you see the following SharePoint targets are loaded.

LISTING 1-1

```
<Import Project="$(MSBuildExtensionsPath32)\Microsoft\
VisualStudio\v10.0\SharePointTools\Microsoft.VisualStudio.
SharePoint.targets" />
```

4. Right below is where you define your own build commands. To package, you add the contents below.

LISTING 1-2

```
<PropertyGroup>

<BuildDependsOn>$(BuildDependsOn);CreatePackage</
BuildDependsOn>

</PropertyGroup>
```

This means that whatever build targets are already defined, add **CreatePackage** to the list. If you're thinking about automating the development processes in your team, this is a great place to start. You can add as many of your own or existing targets here to complement a standard build sequence. Here, for example, is how you can copy the output WSP file from the BIN folder, where it usually gets generated to the main solution directory, and where your custom deployment scripts can pick it up.

LISTING 1-3

```
<TargetName="CopyPackage">

  <ExecWorkingDirectory="$(PackagePath)"

    Command="copy$(TargetDir)$(TargetName).wsp
    $(SolutionDir)$(TargetName).
    wsp"ContinueOnError="false"/>

</Target>

<PropertyGroup>

<BuildDependsOn>$(BuildDependsOn);CreatePackage;CopyPackage</
BuildDependsOn>

</PropertyGroup>
```

NOTE:

Ensure the package names for each Visual Studio SharePoint project (**Platform, Branding, Content**) follow below convention: **[solution name].[project name]**;

Example: '**Chapter1.Platform**'. This will make sure the solution files (WSP) are copied successfully to the root folder of your Visual Studio solution.

Once done, you can save your Visual Studio project definition file and reload the project again by right clicking on the **Platform** and choosing **Reload Project**.

Custom Solution Deployment Script

You've created your project structure and you followed all the rules, what else there is to the successful SharePoint custom solution deployment? The deployment comes next. I'm sure you have heard a lot about how easy it is to deploy Visual Studio projects with built-in deployment tools.

Depending on how you have set up your SharePoint and Visual Studio, you may run into an issue with deploying your solution right from Visual Studio. One of the most common issues you may get is the error message when trying to deploy your newly created Visual Studio solution using the **Deploy** command from within Visual Studio.

*Error occurred in deployment step 'Recycle IIS Application Pool':
The local SharePoint server is not available. Check that the server is running and connected to the SharePoint farm*

Or this:

*Error occurred in deployment step 'Recycle IIS Application Pool':
Cannot connect to the SharePoint site: http://localhost/. Make sure that this is a valid URL and the SharePoint site is running on the local computer. If you moved this project to a new computer or if the URL of the SharePoint site has changed since you created the project, update the Site URL property of the project*

Visual Studio uses the following process to deploy your solution:
vssphost4.exe.

Open your task manager and find the process in the list; take a note of the **User Name** under which this process is running—let's say it's **myadmin_account**.

Next, we'll ensure the SQL server your SharePoint install is using has proper permissions set up for the account Visual Studio is using, here are the steps:

1. Open your SQL Server Management Studio with the user account that is able to manage SharePoint databases.
2. Expand Object Explorer and drill down to **Security -> Logins**.
3. Locate the **myadmin_account**.
4. Right click on the username and select **Properties**.
5. Open **User Mappings** tab.
6. Ensure all three databases below have **myadmin_account** added as a **DBOWNER**:
 - SharePoint_Config
 - SharePoint_AdminContent_[guid]
 - SharePoint Site Content DB

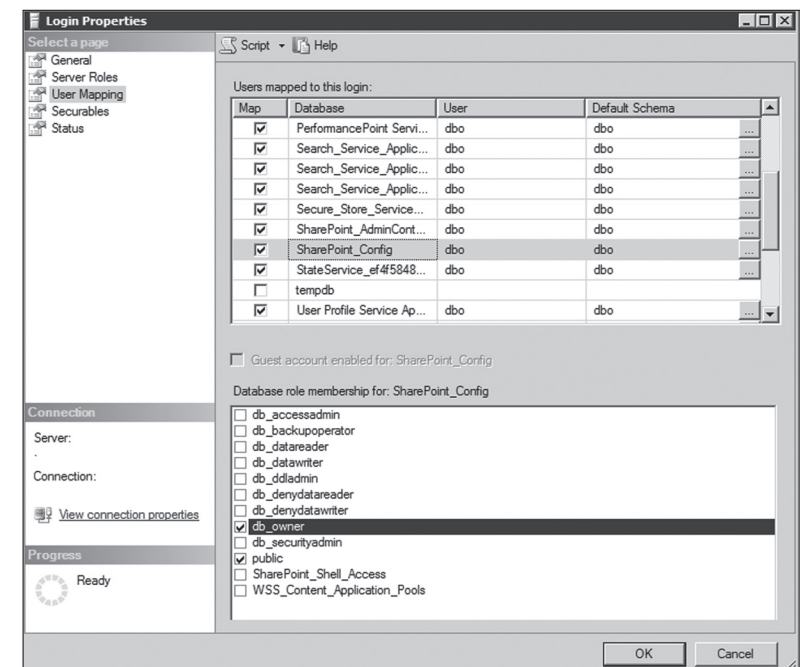


Figure 1-3 Setting up your install account as a DBOWNER

After the changes to the SQL login have been applied, close and launch your Visual Studio. This will ensure the deployment service is recycled, and consequently fix the deployment problem. If you're still seeing the same issue after restarting your Visual Studio, open your task manager and end the **vssphost4.exe** process manually; restart Visual Studio.

Although manual, the deployment method from within Visual Studio is quick and easy—there are more deployment scenarios requiring better approaches. Chances are that your client, and even your QA environment, will not have Visual Studio installed on the server. In order to have a solid deployment strategy and not a hundred pages of deployment documents, it really pays to create install and deployment script. With PowerShell available (which I assume you have by default or will install shortly while finishing reading this sentence) and SharePoint 2010 support of PowerShell commands, the deployment experience is more flexible and robust. After performing several SharePoint 2010 deployments, below is my generic script that will read your configuration and deployment preferences from an XML file to drive the deployment.

If you already have solution files (WSP) in your script directory, you can execute deployment commands with that assumption. The deployment script will consist of three files, plus any WSP files from your solution:

1. SetupSite.bat.
2. SolutionStructure.xml.
3. SetupSite.ps1.

We'll start with **SetupSite.bat**, as this is file's sole purpose is to call PowerShell script (PS1), which actually performs operations. Here is what the file looks like.

LISTING 1-4

```
@echo off
powershell -Command "& {Set-ExecutionPolicy bypass}" -NoExit
powershell -Command ".\SetupSite.ps1" -NoExit
pause
```

SolutionStructure.xml is the file that defines variables that will be used in PS1 script, such as Web Application names, features to be activated, and so on. Let's take a look at the contents.

LISTING 1-5

```
<SetupWebAppUrl="http://localhost">

<Solutions>

    <SolutionWebApplication="True">MyProject.Platform.wsp</
    Solution>

    <SolutionWebApplication="False">MyProject.Branding.
    wsp</Solution>

</Solutions>

<SiteCollectionName="MyNewSite"
Url="/sites/MyNewSite"OwnerAlias="administrator"
Template="STS#0">

    <Features>

        <Feature>MyCustomFeature</Feature>

    </Features>

    <SiteName="MySite" Url="MySite" Template="STS#0">

        <Feature>FeatureName</Feature>

    </Site>

    <SiteName="MySite1" Url="MySite1" Template="STS#0">

    </Site>

</SiteCollection>

</Setup>
```

Here are the variables we are using in the script.

1. **WebAppUrl** –is the Web Application URL. If you're deploying to a different Web Application than a root site, this is where you specify that exact URL.
2. **Solution** –is one of the solutions that will be deployed into the solution gallery in SharePoint Central Administration. You can list as many as you require. Solutions that require deployment on a

Web Application level specified in #1 will have **WebApplication** parameters set to **true**; otherwise, your solution will be deployed to all Web Applications.

3. **SiteCollection** –is a pretty descriptive element denoting a new site collection. Here, we define a site collection administrator user name as **OwnerAlias** and **Template** used for site collection root. If you have a custom template provisioned as a part of your solution, this is where you'd specify its respective ID. For example, **STS#0** is a template ID of a team site.
4. **Feature** –is one or more features that you'd like to see activated at the site collection. Those, for example, are features that will provision your content types.
5. **Site** –is one or more sites that will be created under the site collection. Unless your site is a test site that has nothing but site collection with many pages underneath it, you will have at least few sub-sites.
6. **Site -> Feature** –is the feature that will be activated under the sub-site. When you provision a generic site template—in this case we use team site (STS)—you are likely to provision additional pages under that site. Since your pages will be contained in **Modules**, you will need a feature that will provision them to the proper destination sub-site and not to all of the sites. For that same purpose, you will dedicate a page provisioning feature and activate it under the site where pages should be delivered.

Now that we've looked at the XML file that holds configuration settings, let's take a look at the script that reads all of those settings and makes proper provisions.

LISTING 1-6

```
Write-Host

#define variables for script

[xml]$SiteStructure=get-content SolutionStructure.xml

$WebAppUrl=$SiteStructure.Setup.Attributes.Item(0).Value

$SiteCollectionUrl=$SiteStructure.Setup.SiteCollection.
Attributes.Item(1).Value
```

```
$SiteUrl=$WebAppUrl+$SiteCollectionUrl

#check to ensure Microsoft.SharePoint.PowerShell is loaded

$snapi=Get-PSSnapi | Where-Object{$_Name-eq 'Microsoft.
SharePoint.Powershell'}

if($snapi -eq $null){
Write-Host "Loading SharePoint PowerShell Snapi"
Add-PSSnapi "Microsoft.SharePoint.Powershell"
}

#delete any existing site found at target URL

$targetUrl=Get-SPSite | Where-Object{$_Url -eq $SiteUrl}
if($targetUrl.Url.Length -gt 0){
Write-Host "Deleting existing site at " $SiteUrl
Remove-SPSite-Identity $SiteUrl -Confirm:$false
}

#add the solution package

$solutions=$SiteStructure.Setup.Solutions
foreach($solutionInstance in $solutions.ChildNodes)
{
if($solutions.Solution.Count -gt 0)
{
$targetSolution=Get-SPSolution | Where-Object{$_Name -eq
$solutionInstance.InnerText}
if($targetSolution.Deployed -eq "True")
{
Write-Host "Uninstalling existing solution package: "
$targetSolution.Name

$WebAppInstallTarget=$solutionInstance.Attributes.Item(0).
Value

if($WebAppInstallTarget -eq "True")
{
```

```

Uninstall-SPSolution-Identity $targetSolution.Name -
WebApplication $WebAppUrl-Confirm:$false
}
else{Uninstall-SPSolution -Identity $targetSolution.Name-
Confirm:$false}
do
{
$targetSolution=Get-SPSolution | Where-Object{$_.Name -eq
$solutionInstance.InnerText}
} while($targetSolution.JobExists -eq "True")
Write-Host "Removing existing solution packages"
Remove-SPSolution -Identity $solutionInstance.InnerText
-Confirm:$false
}
Write-Host "Adding solution packages"
Add-SPSolution -LiteralPath $solutionInstance.InnerText
Write-Host "Installing solutions"
$WebAppInstallTarget=$solutionInstance.Attributes.Item(0).
Value
if($WebAppInstallTarget -eq "True")
{
Install-SPSolution -Identity $solutionInstance.InnerText
-WebApplication $WebAppUrl -GACDeployment -force
}
else{Install-SPSolution -Identity $solutionInstance.InnerText
-GACDeployment -force}
do
{
$targetSolution=Get-SPSolution | Where-Object{$_.Name -eq
$solutionInstance.InnerText}
}while($targetSolution.JobExists -eq "True")
}

```

```

}
#creating site structure
$SiteCollectionName=$SiteStructure.Setup.SiteCollection.
Attributes.Item(0).Value;
$SiteCollectionOwner=$SiteStructure.Setup.SiteCollection.
Attributes.Item(2).Value;
$SiteCollectionTemplate=$SiteStructure.Setup.SiteCollection.
Attributes.Item(3).Value;
Write-Host "Creating new site collection at" $SiteUrl
$NewSite=New-SPSite -URL $WebAppUrl $SiteCollectionUrl
-OwnerAlias $SiteCollectionOwner -Template
$SiteCollectionTemplate
-Name $SiteCollectionName
$RootWeb=$NewSite.RootWeb
$features=$SiteStructure.Setup.SiteCollection.Features
if($features.Feature.Length -gt 0)
{
foreach($SiteColFeature in $features.Feature)
{
$ActivatedFeature=Enable-SPFeature $SiteColFeature -url
$RootWeb.Url
Write-Host "Enabled Feature:" $SiteColFeature-
foregroundcolorGreen
}
}
Write-Host "Site collection created successfully"
Write-Host "Title:" $RootWeb.Title -foregroundcolor Green
Write-Host "URL:" $RootWeb.Url -foregroundcolor Green
Write-Host "-----"
for($i=1; $i -lt $SiteStructure.Setup.SiteCollection.
ChildNodes.Count; $i++)
{

```

```

$childsite=$SiteStructure.Setup.SiteCollection.ChildNodes.
Item($i);

$WebName=$childsite.Attributes.Item(0).Value
$WebUrl=$childsite.Attributes.Item(1).Value
$WebTemplate=$childsite.Attributes.Item(2).Value
Write-Host "Creating new web at " $SiteUrl/$WebUrl

$NewWeb=New-SPWeb $SiteUrl/$WebUrl -Template $WebTemplate -
Addtotopnav -Useparenttopnav -Name $WebName

Write-Host "Web created successfully"

Write-Host "Title: " $NewWeb.Title -foregroundcolor Green
Write-Host "URL: " $NewWeb.Url -foregroundcolor Green

$features=$SiteStructure.Setup.SiteCollection.ChildNodes.
Item($i)

if($features.Feature.Length -gt 0)
{
    foreach($WebFeature in $features.Feature)
    {
        $ActivatedFeature=Enable-SPFeature $WebFeature -url $NewWeb.
        Url

        Write-Host "EnabledFeature: " $WebFeature -foregroundcolor
        Green
    }
}

Write-Host "-----"

}

start-process -filepath iexplore -argumentlist $SiteUrl

```

The main purpose for dropping this script in front of you is not so you can practice your typing skills. Rather it is to take a look at some of the sections in the script and take note of the syntax and how to access objects and process logic. The next time you need to extend this PowerShell script to include new scenarios, you will have a better idea where to start and what to copy where. This is definitely not

the catchall scenarios script, but it will work for most of your small to medium site solution deployments, and definitely reduce your deployment time from one environment to another.

Referenced Assemblies in Your SharePoint Solution

If you're developing a SharePoint 2010 solution that works with third party components or even if it's your own Service project that produces a separate DLL, you will have to instruct SharePoint to deploy those libraries. You may think that just by providing a project reference in your solution, SharePoint project will take care of your extra DLLs and use them in WSP it generates; however, there are few extra steps you need to take to make that happen.

Assuming you are referencing an internal assembly from another project residing in the same Visual Studio solution, here are the steps you need to take:

1. Create a reference in the SharePoint project that uses your **Service** project
 - Right click on your SharePoint project and select **Add a Reference**.
 - Select **Projects** tab.
 - Add a new reference.
2. In the Solution Explorer of your SharePoint project locate **Package** and expand it.
3. Double click on expanded child element of **Package**.
4. Click **Advanced** from the newly opened window and click **Add**.
5. Here pick the option **Add Assembly from Project Output**.
6. Pick the assembly from the drop down and click **OK**.

NOTE:

The assembly you will be referencing from your project must be signed with a strong name key if you're planning to deploy it to Global Assembly Cache.

Now, if you're referencing third party assemblies that you have already copied into your solution structure, you'd perform the steps below to make your DLLs part of the SharePoint project package.

In your Solution Explorer, in SharePoint project, locate **Package** and expand it.

Double click on expanded child element of **Package**.

Click **Advanced** from the newly opened window and click **Add**.

Here pick the option **Add Existing Assembly**.

Locate an assembly from the disk and click **OK**.

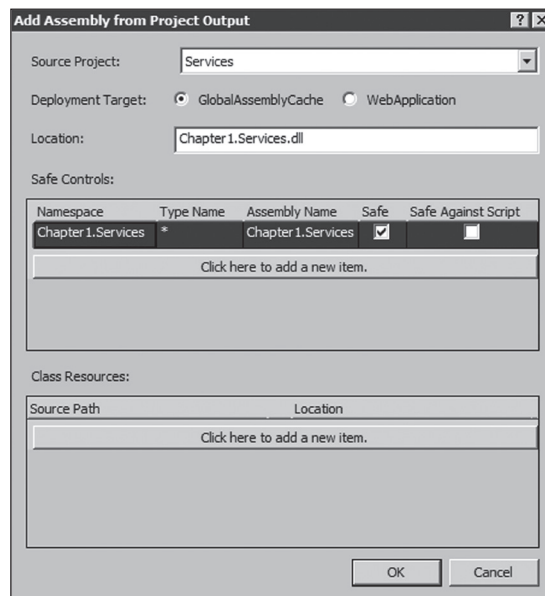


Figure 1-4 Adding a project output assembly to the package of SharePoint project

A nice feature about Visual Studio is that it will provision all of the necessary Safe Control attributes into your IIS site **web.config** file to make sure your assembly is loaded properly. In some cases, you may want to make additional changes to web.config at the time of your solution deployment. You may want to set variables in your **web.config** so that your custom application will consume, connection strings, custom error pages, custom login pages, authentication providers, and so on. You can either write a one – or two—page deployment

instruction for your administrators on all of those settings or you can programmatically provision those changes and save yourself and everyone user errors. Here, we'll see how to create a SharePoint Web Application scoped feature allowing to provision variables to **web.config** at the time of deployment.

The reason why our feature will be activated on Web Application scope is so that application changes can take place before other potentially dependent features on site collections and sites are deployed.

We'll start by creating new **Web Application** scoped feature in your Solution Explorer in **Platform** project.

1. Locate **Features** node in your Solution Explorer; right click on it and select **Add Feature**.
2. Specify feature title.
3. Select **Scope** to be a **Web Application** and save the feature.
4. Right click on newly create feature from Solution Explorer and select **Add Event receiver**.
5. Locate **FeatureActivated(SPFeatureReceiverProperties properties)** and uncomment the section.
6. Right above the definition for the **FeatureActivated** you uncommented add the following security identifier on a class:

LISTING 1-7

```
[SharePointPermission(SecurityAction.LinkDemand, ObjectModel = true)]
```

7. Add the following namespace reference section:
 - using System.Collections.Generic;
 - using Microsoft.SharePoint.Administration;
8. Define the class variable that will hold all of your configurations:

```
List<SPWebConfigModification> webConfigModifications =
new List<SPWebConfigModification>();
```

9. Add the helper methods (below) right under your **FeatureActivated**; those will serve as helper methods entering various elements and attributes in the **web.config**

LISTING 1-8

```
protected void SaveConfig(SPWebApplication app)
{
    foreach (SPWebConfigModification mod in webConfigModifications)
    {
        app.WebConfigModifications.Add(mod);
    }
    webConfigModifications.Clear();
}

protected void AddSection(string name, string xpath)
{
    SPWebConfigModification mod = new SPWebConfigModification(name,
        xpath);
    mod.Sequence = 0;
    mod.Type = SPWebConfigModification.SPWebConfigModificationType.
        EnsureSection;
    webConfigModifications.Add(mod);
}

protected void AddNodeValue(string name, string xpath, string
    resource)
{
    SPWebConfigModification mod = new SPWebConfigModification(name,
        xpath);
    mod.Sequence = 0;
    mod.Type = SPWebConfigModification.SPWebConfigModificationType.
        EnsureChildNode;
    mod.Value = resource;
    webConfigModifications.Add(mod);
}
```

```
}

protected void AddAttributeValue(string name, string xpath,
    string value)
{
    SPWebConfigModification mod = new SPWebConfigModification(name,
        xpath);
    mod.Sequence = 0;
    mod.Type = SPWebConfigModification.SPWebConfigModificationType.
        EnsureChildNode;
    mod.Value = value;
    webConfigModifications.Add(mod);
}
```

Lastly, we'll enter our **web.config** changes to illustrate the syntax we use. Place the listing below into the **FeatureActivated** method body

LISTING 1-9

```
SPWebApplication app = properties.Feature.Parent as
    SPWebApplication;

string customErrorPath = "configuration/system.web/*[local-
    name()='customErrors']";

AddNodeValue("error", customErrorPath, "<error
    statusCode='404' redirect='~/Pages/404.aspx' />");

AddAttributeValue("mode", "configuration/system.web/
    customErrors", "On");

string authenticationPath = " configuration/system.
    web/*[local-name()='authentication']";

AddNodeValue("forms", authenticationPath, "<forms
    name='MySignIn' loginUrl='SignIn.aspx' path='/Pages/' />");

SaveConfig(app);
```



```
app.Farm.Services.GetValue<SPWebService>().
ApplyWebConfigModifications();

app.Update();
```

As you can see, we're calling the helper functions we created earlier to specify:

- Custom 404 error page
- Turn on custom errors
- Set forms authentication
- Set custom form to handle forms authentication

Now, when this solution is deployed the feature will be activated and all of the defined changes will automatically get provisioned to the relevant configuration file.

Debugging Your SharePoint Applications

So far, you have learned how to create your solutions and deploy them automatically. What else could you possibly need before starting your custom component development? Debugging is next. Knowing how to debug your application in various scenarios can make a difference between painful guessing, traversing through thousands of lines of logs and stepping through the code and identifying the problem in just few minutes.

In SharePoint 2010, there has been significant improvement in terms of debugging information that is emitted when errors happen. By default, SharePoint holds all of its logs in the following location:
`[Drive]:\Program Files\Common Files\Microsoft Shared\Web Server Extensions\14\LOGS.`

If you open this location you will see at least few log files that log the activity happening on your SharePoint install. If you open one of the files, you will be instantly overloaded with the volumes of information those logs contain. One of the useful pieces of information SharePoint provides when errors occur is the **Correlation ID** of the error. **Correlation ID** is the number in a GUID format that links the error with the corresponding record in the log file. If you copy the ID from the

error message on the screen and search for it in the log file, you will narrow down troubleshooting from a few thousand lines to ten or so.

One of the other issues you might be having is that SharePoint doesn't collect enough logging information in the log files and, therefore, they become useless. The good news is that you can adjust the verbosity and type of the logs in SharePoint Central Administration; here is how.

1. Navigate to your SharePoint Central Administration with the install account credentials.
2. Click on the **Monitoring** link on the left side navigation.
3. Under the **Reporting** section, click **Configure Diagnostic Logging**.
4. Here, pick the category of SharePoint activity you wish to monitor.
5. Set the **Least critical event to report to the event log** to **Warning** or other appropriate level.
6. In the next dropdown select **Verbose** to get the most detailed log information. Keep in mind that this will cause general system slowdown and produce large volume of logs.

<input type="checkbox"/> Web Content Management <input type="checkbox"/> Word Automation Services Least critical event to report to the event log <input type="text" value="Warning"/> Least critical event to report to the trace log <input type="text" value="Verbose"/>	
Event Log Flood Protection Enabling this setting allows detection of repeating events in the Windows event log. When the same event is being logged repeatedly, the repeating events are detected and suppressed until conditions return to normal.	<input checked="" type="checkbox"/> Enable Event Log Flood Protection
Trace Log When tracing is enabled you may want the trace log to go to a certain location. Note: The location you specify must exist on all servers in the farm. Additionally, you may set the maximum number of days to store log files and restrict the maximum amount of storage to use for logging. Learn about using the trace log.	Path <input type="text" value="%CommonProgramFiles%\Microsoft Shared\Web Server Extensions\14\LOGS\"/> Example: C:\Program Files\Common Files\Microsoft Shared\Web Server Extensions\14\LOGS Number of days to store log files <input type="text" value="14"/> Restrict Trace Log disk space usage <input type="checkbox"/> Restrict Trace Log disk space usage Maximum storage space for Trace Logs (GB) <input type="text" value="1000"/>

Figure 1-5 Diagnostic logging settings for your SharePoint site

7. In the **Trace Log** section, you can also pick where the log will be stored if different from the current location.

Setting up verbose logging may be helpful at the time you have a problem but as soon as you don't need the option to be available, give your SharePoint server a break and set the logging to be less verbose and you will notice an increase in performance.

Information in logs is only the data that SharePoint was able to catch from its modules. If you're trying to find an issue with your customizations, SharePoint logs may not be the first place you want to look.

We have all seen generic error pages and know how frustrating it can be to determine the cause of the problem.

At first, it's quite clear that you have to disable custom error reporting that IIS site SharePoint is running. However, there is at least site collection you're working with, as well as the Central Administration site, not to mention the entire out-of-the-box virtual directories have their own configuration files. So which **web.config** file do you have to modify to get to the bottom of the error message clarity issue?

If your error occurred while performing out-of-the-box activities in SharePoint or custom Web Parts loaded into the page, you need to make the following changes.

1. Open **IIS Manager** on your SharePoint server with an install account credentials.
2. Expand **Sites** node and locate the site that represents your SharePoint site.
3. Right click on the Site node, select **Manage Web Site -> Advanced Settings**.

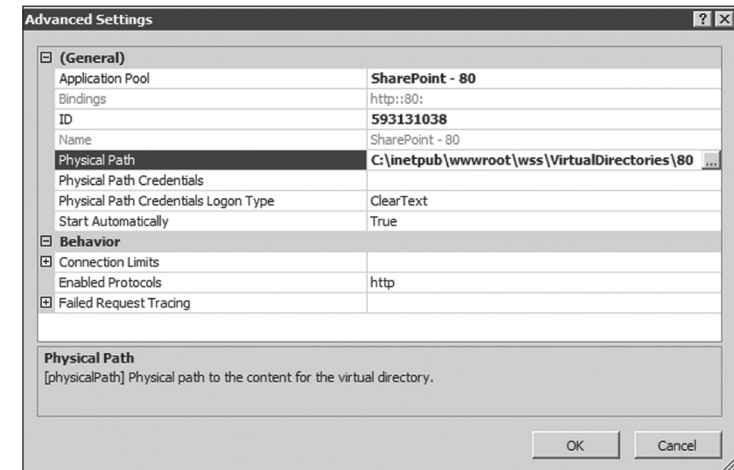


Figure 1-6 Determining the physical path of of your SharePoint site using IIS

4. Locate the physical path and copy the location. Your path is likely to be something like this: `[Drive]:\inetpub\wwwroot\wss\VirtualDirectories\80`.
5. Locate the **web.config** file and open it in **Notepad**.
6. Find the following in the file: **customErrors**.
7. Replace the value of the **mode** attribute to **Off**.
8. Find the following in the file: **debug** and change its value to **true**.
9. Find the following in the file: **CallStack** and change its value to **true**.
10. Save the file and refresh the page on which you received an error originally.

Now, you should get a more detailed description of the error and debug your application.

If you have received an error while executing custom functionality in SharePoint Central Administration, your steps of troubleshooting would be exactly as above except in your IIS Manager, you would located the **Physical Path** of your SharePoint Central Administration site.

Things can be a bit confusing to most folks who have enabled those options in their corresponding **web.config** files and still get a cryptic

error when trying to debug their custom page in the **_layouts** folder or a custom web service in the **_vti_bin** folder.

As mentioned before, those are virtual folders in IIS and, therefore, they have their own **web.config** which you guessed right ... has its own values set for debugging and error handling.

As you've see how to extract the path to the appropriate **web.config** location in IIS for web sites, the same approach applies to virtual directories such as **_layouts** and **_vti_bin**. Once you have the physical location of the file, go ahead and make modifications that apply to your scenario.

Now, when you open **web.config** files in your virtual directories, they will look much smaller and with less detail. This doesn't mean you have to specify debugging parameters here. If you specified **debug='true'** in your site **web.config**, this value will be inherited. Only the values that are defined in the virtual directory explicitly need to be overwritten—usually it's **CustomErrors**.

Options above turn on heavy logging on your SharePoint server and produce significant overhead in processing, meaning it's not OK to have those on by default in your production environment.

CHAPTER 2

Lists and Libraries: List Rollups, Security, and Integration with the Rest of SharePoint 2010 Components

Working with lists is one of the most common tasks you will do in SharePoint. After all, lists are nearly everything in SharePoint. Blog articles, discussions, calendars, document libraries are all lists. Understanding some of the enhancements in how lists work will make a difference when creating your custom applications.

We'll start by adding a simple list to the Visual Studio structure we have created before. The concept of list includes list definition and list instance. As the name suggests, the definition will define what fields your list is going to have, views, and queries. The instance will create a new instance of this list and possibly set some initial data into the list. In most cases many lists are already defined, and in fact, SharePoint comes with variety of available list types. If you're happy with any of the list definitions that SharePoint comes with, the only task to do is to create an instance of the list. Let's see what involved in creating a list instance using Visual Studio. Since our list instance is going to be a generic instance not tied to any specific page, we'll create a new Visual Studio folder under the root of our **Platform** project and call it **Lists**. Following the steps below, we create a list instance.