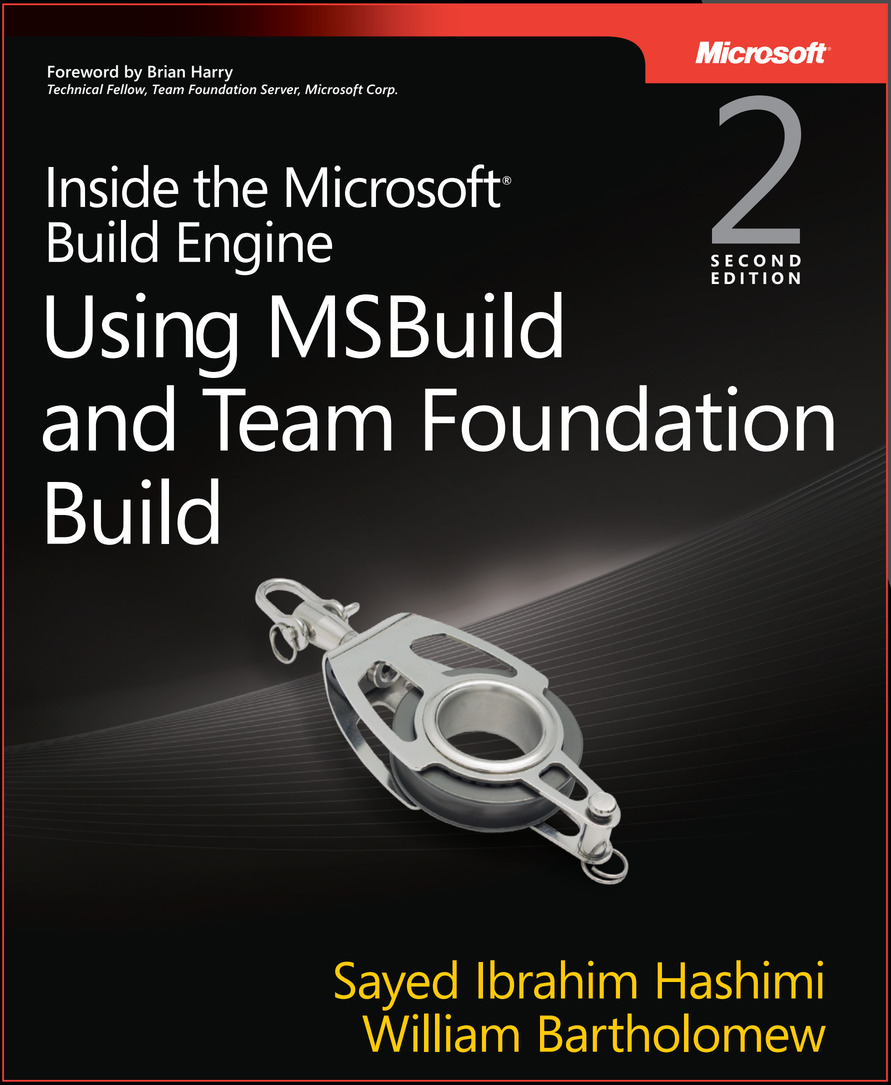
Foreword by Scott Hanselman,
Principal Community Architect, Microsoft

Microsoft

Supplement to



Foreword by Brian Harry
Technical Fellow, Team Foundation Server, Microsoft Corp.

Microsoft

2
SECOND EDITION

Inside the Microsoft® Build Engine
Using MSBuild and Team Foundation Build

Sayed Ibrahim Hashimi
William Bartholomew

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number (PCN): 2013935725
ISBN: 978-0-7356-7816-3

Acquisitions Editor: Devon Musgrave
Developmental Editor: Devon Musgrave
Project Editor: Valerie Woolley
Editorial Production: Christian Holdener, S4Carlisle Publishing Services
Technical Reviewer: Marc Young w/ CM; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Copyeditor: Susan McClung
Indexer: Jean Skipp
Cover: Twist Creative • Seattle and Joel Panchot

I would like to dedicate this book to my parents Sayed A. Hashimi and Sohayla Hashimi. Without them I wouldn't be here to write this book.

—Sayed

I dedicate this book to my parents Rosanna O'Sullivan and Roy Bartholomew for their unending support and to Lauren Taylor and Jillian Bartholomew for the joy they bring each day.

—William

Contents at a glance

Contents

Foreword

Ah, the thankless life of the Build Master. If they do their job well, you'll never know they exist! If they don't, well, everyone knows it's the Build Master's fault, right?

I've been a builder in one form or another since my first foray into managing the build. Nearly 15 years ago now, I worked on an extremely large system with a team of hundreds. When it came time to build, we used Fred's machine. Yes, I learned that day that we built and shipped large systems on Fred's laptop.

This is also how I came to find that learning a new build system is similar to boiling a frog. If you throw a frog into hot water, it jumps out. But if you turn the water up slowly, the frog doesn't realize it's getting hot, so it stays in the pot and gets boiled. The team didn't realize how big the system had become and how complex the build was getting.

I realized immediately, somewhat intuitively, that we needed build box. Fast-forward some years, and now every group I work with uses Continuous Integration. Groups I work with have build farms, one with a "Siren of Shame," a flashing light to effectively shame the build-breaker. We have build artifacts as complex and elegant as actual preconfigured virtual machines that pop out the end of our build.

All this was made possible by the power of automation and the surprising flexibility of MSBuild. Sayed and William have written what amounts to the "missing manual" for MSBuild. MSBuild, and its enterprise support counterpart Team Foundation Build, are almost unapologetically powerful. However, they need to be.

Today's software systems are multilayered, multitiered, and iterate at a speed previously unheard of. All our software development practices and team building comes together at one pinch point: the build.

This essential reference to MSBuild gives us the knowledge not only into how to create an adaptable and vigorous build system, but also valuable insights into the "why" of the product. William is a senior development lead on engineering systems within the Developer division at Microsoft, while Sayed is a program manager overseeing build and pushing for the Microsoft Azure Cloud and Web Tools. I could think of no better people to help me understand a large build system than the folks building large systems themselves.

Sure, we've all started with "Build.bat" and called it our build system. Perhaps we've put together a little schedule and called it an automated build. But these simple constructs don't scale across a large team or a large product. This book is what the documentation should have been—a guide that takes us through the humble beginnings of MSBuild as a supporting and unseen player in the .NET ecosystems to complete and sophisticated team build solutions.

More importantly, Sayed and Bill dig into the corners and edge cases that we all find ourselves bumping up against. They elaborate on the deceptively deep extensibility model that underlies MSBuild and give us the tools to bring both stock and custom components together into a complete team workflow.

MSBuild continues to evolve from version 2, to 3.5, and now to version 4 and beyond. This updated supplemental edition builds (ahem) on the good work of the previous editions and includes new sections on the updates to the MSBuild core, changes in Team Build, and even updates to Web Publishing in Microsoft Visual Studio 2012.

I'm glad that this book exists and that people who care about the build like Sayed and William exist to light the way. Now, if I can just find out what I did just now that broke my build . . .

— Scott Hanselman
Teacher, coder, blogger, podcaster
hanselman.com

Introduction

Build has historically been kind of like a black art, in the sense that there are just a few people who know and understand it and are passionate about it. But in today's evolving environment, that is changing. Now more and more people are becoming interested in build and making it a part of their routine development activities. Today's applications are different from those that we were building 5 to 10 years ago. Along with that, the process that we use to write software is different as well. Nowadays, it is not uncommon for a project to have sophisticated build processes that include such things as code generation, code analysis, unit testing, automated deployment, and so on. To deal with these changes, developers are no longer shielded from the build process. Developers have to understand the build process so that they can employ it to meet their needs.

Back in 2005, Microsoft released MSBuild, which is the build engine used to build most Microsoft Visual Studio projects. That release was MSBuild 2.0. Since that release, Microsoft has released three major versions of MSBuild—MSBuild 3.5, MSBuild 4.0, and now MSBuild 4.5. Along with the updates included in MSBuild 4.5, there are many build-related updates in related technologies. For example, with Visual Studio 2012, you now have the ability to share projects with Visual Studio 2010. Another great example is the usage of *NuGet*. In many ways, *NuGet* has changed how we develop and build applications. In this book, we will look at the updates included in MSBuild 4.5, as well as other related technologies.

Team Foundation Build (or Team Build as it is more commonly known) is now in its fourth version. Team Build 2005 and Team Build 2008 were entirely based on MSBuild, using it for both build orchestration and the build process itself. Team Build 2010 moved build orchestration to Microsoft Windows Workflow Foundation and continues to use MSBuild for the low-level build processes. Team Build 2012 continues this architecture but now supports building in the cloud using the Team Foundation Service, an updated task-focused user interface, gated check-in improvements to improve throughput, and better support for unattended installation.

When developing automated build processes, the next step in many cases is to automate the publish process. In Visual Studio 2010, the initial support for the Web Deploy tool was added. In Visual Studio 2012, there have been a lot of updates to how web projects are published, including first-class support for publish profiles from the command line, sharing of publish profiles with team members, database publishing, and many more. In this update, we will describe these updates and show you some real-world examples as well. You'll see how the process used in Visual Studio 2012 is much more straightforward than what was provided in Visual Studio 2010.

Who should read this book

This book is an enhancement to our *Inside the Microsoft Build Engine, Using MSBuild and Team Foundation Build, Second Edition* (Microsoft Press, 2011), a book whose content is still relevant and accurate. Rather than add these three chapters to that book and release it as a third edition, we decided to offer this shorter (and cheaper) supplement. Think of the three chapters in this supplement as an addition to *Inside the Microsoft Build Engine*.

The second edition and this supplement to it were written for anyone who uses or is interested in using MSBuild or Team Build. If you're using Visual Studio to build your applications, you're already using MSBuild. *Inside the Microsoft Build Engine* and its supplement are for all developers and build masters using Microsoft technologies. If you're interested in learning more about how your applications are being built and how you can customize this process, you need these books. If you are using Team Build or thinking of using it tomorrow, these books are must-reads. They will save you countless hours.

The second edition and this supplement will help the needs of enterprise teams as well as individuals. To get the most out of these materials, you should be familiar with creating applications using Visual Studio. You are not required to be familiar with the build process, but if you are not, make sure to begin with the second edition because it starts with the basics and goes on from there. Because one of the most effective methods for learning is through examples, both the second edition and this supplement contain many examples.

Assumptions

To get the most from this supplement, you should meet the following profile:

- You're familiar with MSBuild 4.0 and Team Foundation Build 2010.
- You should be familiar with Visual Studio.
- You should have experience with the technologies you are interested in building.
- You should have a solid grasp of XML.

Who should not read this book

This supplement to *Inside the Microsoft Build Engine* covers the new and changed functionality in MSBuild 4.5 and Team Foundation Build 2012, so it's not aimed at people new to MSBuild and Team Foundation Build. If you're new to MSBuild and Team Foundation Build, we highly recommend reading *Inside the Microsoft Build Engine* first.

Organization of this book

This book is divided into three chapters, each of which focuses on a different build technology. Chapter 1, "What's new in MSBuild 4.5," covers the new and changed functionality in MSBuild 4.5, including compatibility with previous versions, out-of-process tasks, and *NuGet*. Chapter 2, "What's new in Team Foundation Build 2012," covers the new and changed Team Foundation Build functionality, including the introduction of Team Foundation Online, a new customizable task-focused user interface, improved debugging and administration, and a number of gated check-in improvements. Chapter 3," What's new in web publishing", includes details on the updated web publish experience in Visual Studio 2012. This includes updates to publish profiles, database publishing support, web.config transform updates and more.

Conventions and features in this book

This book presents information using conventions designed to make the information readable and easy to follow.

- Each exercise consists of a series of tasks, presented as numbered steps (1, 2, and so on) that list each action you must take to complete the exercise.
- Boxed elements with labels such as "Note" provide additional information or alternative methods for completing a step successfully.
- Text that you type (apart from code blocks) appears in **bold. In code blocks code in bold indicates code added since the previous example.**

System requirements

You will need the following hardware and software to complete the practice exercises in this book:

- One of Windows 7 (x86 or x64), Windows 8 (x86 or x64), Windows Server 2008 R2 (x64), or Windows Server 2012 (x64).
- Visual Studio 2012, any edition (multiple downloads may be required if using Express edition products)
- Computer that has a 1.6 GHz or faster processor
- 1 GB (32-bit) RAM (add 512 MB if running in a virtual machine)
- 10 GB of available NTFS hard disk space
- 5,400 RPM hard disk drive
- DirectX 9 capable video card running at 1,024 x 768 or higher-resolution display
- DVD-ROM drive (if installing Visual Studio from DVD)
- Internet connection to download software or chapter examples

Depending on your Windows configuration, you might require Local Administrator rights to install or configure Visual Studio 2012.

You will need the following minimum level of hardware and software to use the virtual machine used for the practical exercises in Chapter 2 of this book:

- Either Windows Server 2008 R2 with the Hyper-V role enabled, Windows Server 2012 with the Hyper-V role enabled, or Windows 8 with Hyper-V enabled
- Intel VT or AMD-V capable processor (SLAT-compatible processor required if using Windows 8)
- 6 GB of free physical RAM (8 GB or more is recommended)
- 3 GB of RAM assigned to the virtual machine (4 GB or more is recommended)
- 50 GB of available NTFS hard disk space (more is recommended if using snapshots)

Code samples

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text. All sample projects, in both their pre-exercise and post-exercise formats, can be downloaded from the following page:

http://aka.ms/MSBuild2ESupp/files

Follow the instructions to download the MSBuild2ESupp_678163_Companion Content.zip file.

Note In addition to the code samples, your system should meet the System Requirements listed previously.

Installing the code samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book.

1. Unzip the MSBuild2ESupp_678163_CompanionContent.zip file that you downloaded from the book's website to C:\InsideMSBuild\.
2. If prompted, review the displayed end user license agreement. If you accept the terms, select the Accept option, and then click Next.

Note If the license agreement doesn't appear, you can access it from the same webpage from which you downloaded the MSBuild2ESupp_678163_CompanionContent.zip file.

Using the code samples

After extracting the samples you will see a folder for each chapter. Within each subfolder you will find all the samples for that chapter.

Acknowledgments

The authors are happy to share the following acknowledgments.

Sayed Ibrahim Hashimi

Wow. This will be my fourth book and my third with Microsoft Press. I'm not sure how I ended up here, but it certainly was not on my own. Throughout each book there were key contributors who helped us create the final product. Even though this book is much smaller than my previous ones, it's still no easy task. We still have to go through the entire process and involve basically the same number of people to help us. Being an author, I receive a majority of the credit for the result, but there are plenty of others who deserve credit as well.

I'd like to start by thanking my co-author, William Bartholomew. William also works at Microsoft. William is a known Team Build expert, and it shows in his writing. I've had so many people approach me and tell me how good the Team Build chapters are. Most times, I smile and quietly accept the praise—that's part of being a co-author—thanks, William! William has helped me since the first edition of this book. I love working with him and hope that I can do so again in the future.

In the second edition, we had a wonderful technical editor, Marc Young. Thankfully, we were able to convince him to come on board for this supplement as well. I'm really glad that we were able to do so. He did a brilliant job on the second edition and a great job for this supplement, too. Marc is not shy when it comes to letting authors know that they are wrong! He goes to great lengths to verify, or disprove, author statements and code samples. I appreciate all of his efforts, and readers should as well. His feedback is also critical in shaping the content of the book.

Devon Musgrave, the man behind the scenes at Microsoft Press, deserves a lot of credit here. I remember having dinner with Devon in Bellevue one night in the summer of 2012. We discussed the idea of an update to the book. We both knew that a full rewrite wasn't the best idea. The vast majority of the second edition is still relevant, so it would have been better if we could just publish what's new. We decided to try out a new format for Microsoft Press books: the supplement. This will be the first supplement Microsoft Press delivers. I'm really happy that we were able to make this work, and if it weren't for Devon, this wouldn't have happened.

Valerie Woolley, our Project Editor at Microsoft Press, was critical to the delivery of this book. She has helped us stay on track with our deliverables and ensure that things keep moving. Thanks for keeping us on track, Valerie, and I apologize for not turning in all my content on the dates you requested.

In addition to the people that I have listed here there are several others who contributed to this book. With any significant project there are names that go unknown. I truly appreciate all the efforts of everyone involved in this book. I wish that I could name them all here. Thank you.

Last but certainly not least, I'd like to thank all the readers. You guys have stuck by us for two editions. I appreciate all the support and kind words that have been expressed about the books. Because of your support, we were able to publish the second edition, as well as this supplement. Please continue to let us know how we are doing. Hopefully, you will enjoy this supplement as much as the second edition.

William Bartholomew

First, I'd like to thank my third-time co-author, Sayed, because without him, this book would not be as broad as it is. From Microsoft Press, I'd like to thank Devon Musgrave, Valerie Woolley, and the art team for their efforts (and tolerance) in converting our ideas into a publishable book. Thanks must go to Marc Young for his technical review efforts in ensuring that the procedures are easily followed, the samples work, and the book makes sense. Finally, I'd like to thank the Team Build Team, in particular Justin Pinnix and Patrick Carnahan, for their tireless support.

Errata & book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at oreilly.com:

http://aka.ms/MSBuild2ESupp/errata

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at mspinput@microsoft.com.

Please note that product support for Microsoft software is not offered through the addresses above.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at

http://aka.ms/tellpress

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

CHAPTER 3

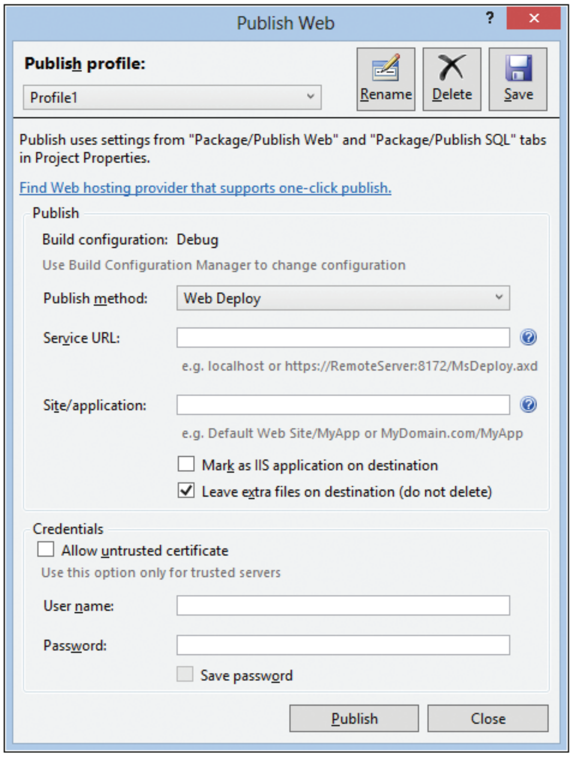
What's new in web publishing

In Microsoft Visual Studio 2010, the MSBuild-based web publishing experience was introduced for Web Application Projects (including both ASP.NET MVC and Web Forms). Along with that release came the foundation for the wave of updates that are included with Visual Studio 2012. The Visual Studio web team is also releasing regular updates to the web publishing experience, along with ASP.NET updates and changes to the Azure software development kit (SDK). The updated web publishing experience has been made available for Visual Studio 2010 SP1 as well. You can install the latest web publishing support, including both Visual Studio 2012 and 2010 SP1, from the Azure SDK, which you can find at *http://www.windowsazure.com*.

Since Visual Studio 2012 was initially released, there have already been a few updates to the web publishing experience. At this time, the latest Azure SDK is version 1.8. The content in this chapter has been written on the assumption that the Azure SDK 1.8 has been installed. We will highlight any new content that is not built into Visual Studio 2012. For the remainder of the chapter, we will discuss the updates in terms of Visual Studio 2012, but all the material also applies to Visual Studio 2010 with the Azure SDK.

Overview of the new Publish Web dialog box

In Visual Studio 2010, the Publish Web dialog box was pretty basic. It had some profile management options at the top and minimal publishing settings. You can see the Publish Web dialog box from Visual Studio 2010 in Figure 3-1.



Publish profile:

Profile1

Publish uses settings from "Package/Publish Web" and "Package/Publish SQL" tabs in Project Properties.

Find Web hosting provider that supports one-click publish.

Publish

Build configuration: Debug

Use Build Configuration Manager to change configuration

Publish method: Web Deploy

Service URL:

e.g. localhost or https://RemoteServer:8172/MsDeploy.axd

Site/application:

e.g. Default Web Site/MyApp or MyDomain.com/MyApp

☐ Mark as IIS application on destination

☑ Leave extra files on destination (do not delete)

Credentials

☐ Allow untrusted certificate

Use this option only for trusted servers

User name:

Password:

☐ Save password

FIGURE 3-1 The Visual Studio 2010 Publish Web dialog box.

In Visual Studio 2012, the Publish Web dialog box has been extensively updated. The dialog box now consists of several different tabs. Even though the dialog box has more functionality, the overall experience is simpler. This is especially the case when the Import functionality is used to populate the settings. In Figure 3-2, you can see the new Publish Web dialog box.

The Publish Web dialog box consists of four tabs. On the Profile tab, you can manage your profiles. To create a new profile, you can either click Import and select an existing *.publishSettings* file, or you can create a new profile manually by selecting the New option from the Select Or Import A Publish Profile drop-down list. A *.publishSettings* file is a simple XML file that contains the publishing information. This file is produced by many web hosting providers and can be used with Visual Studio or Web Matrix. If your hosting provider does not make these files available, you should demand that they do. These *.publishSettings* files are different from the *.pubxml* files created with Visual Studio. The *.pubxml* files contain the remote endpoint information, as well as values that are specific to the publishing requirements of your project. In contrast, the *.publishSettings* file just contains the publishing endpoint information. The other difference is that a *.publishSettings* file can contain more than one set of publish settings. For example, Windows Azure Web Sites includes both a Web Deploy profile and the File Transfer Protocol (FTP) settings.

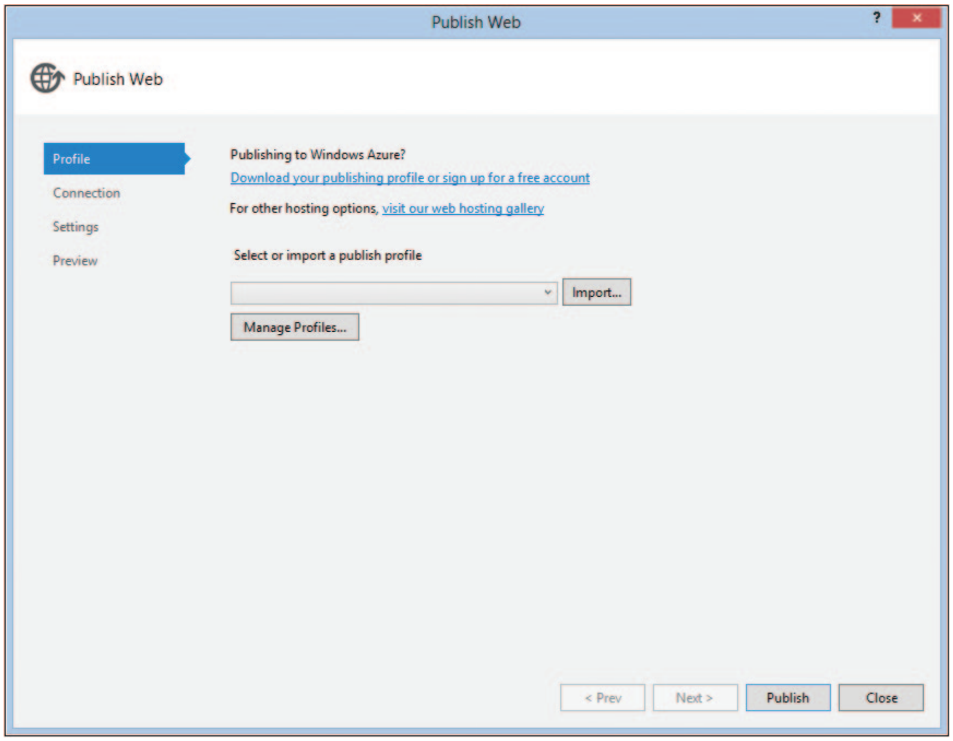
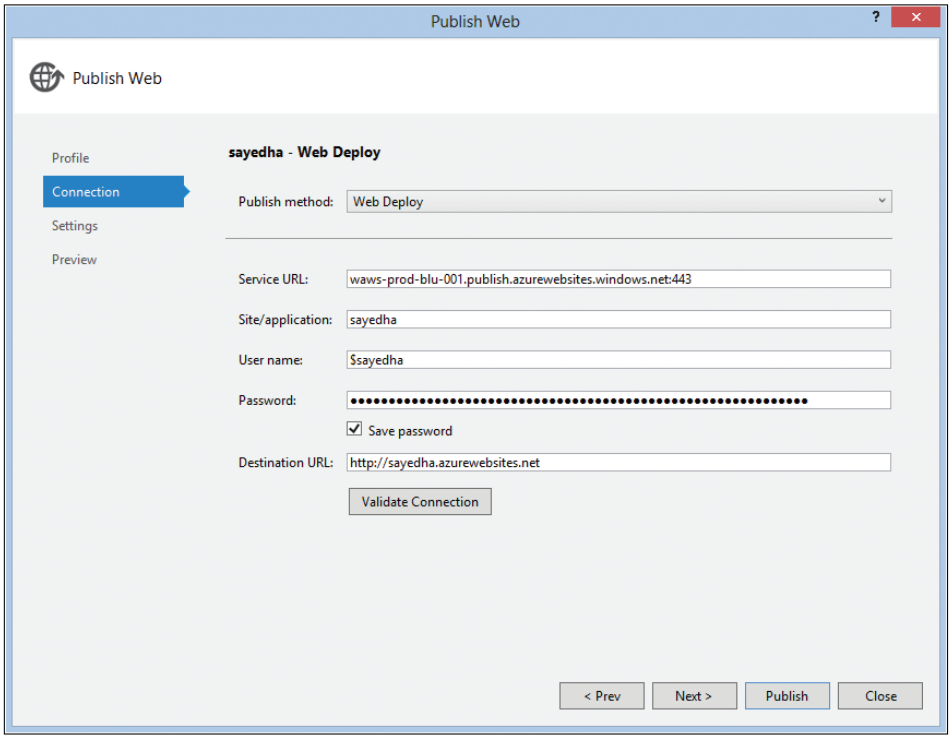

Publish Web
Publish Web
Profile
Connection
Settings
Preview
Publishing to Windows Azure?
Download your publishing profile or sign up for a free account
For other hosting options, visit our web hosting gallery
Select or import a publish profile
Import...
Manage Profiles...
< Prev
Next >
Publish
Close

FIGURE 3-2 The Visual Studio 2012 Publish Web dialog box.

Here is a basic publishing scenario: You have an existing ASP.NET project that you need to publish to a remote web host. Your host provides a *.publishSettings* file, which you can import into Visual Studio. In my case, I'm publishing to Windows Azure Web Sites, but this flow works for any hosting provider that supports *.publishSettings* files. To open the Publish Web dialog box, right-click the web project in Solution Explorer and select Publish, which will open the dialog box shown in Figure 3-2. You can use the Import button to import the *.publishSettings* file. After importing the file, you will be brought to the Connection tab automatically. You can see this tab in Figure 3-3.



sayedha - Web Deploy

Publish method: Web Deploy

Service URL: waws-prod-blu-001.publish.azurewebsites.windows.net:443

Site/application: sayedha

User name: $sayedha

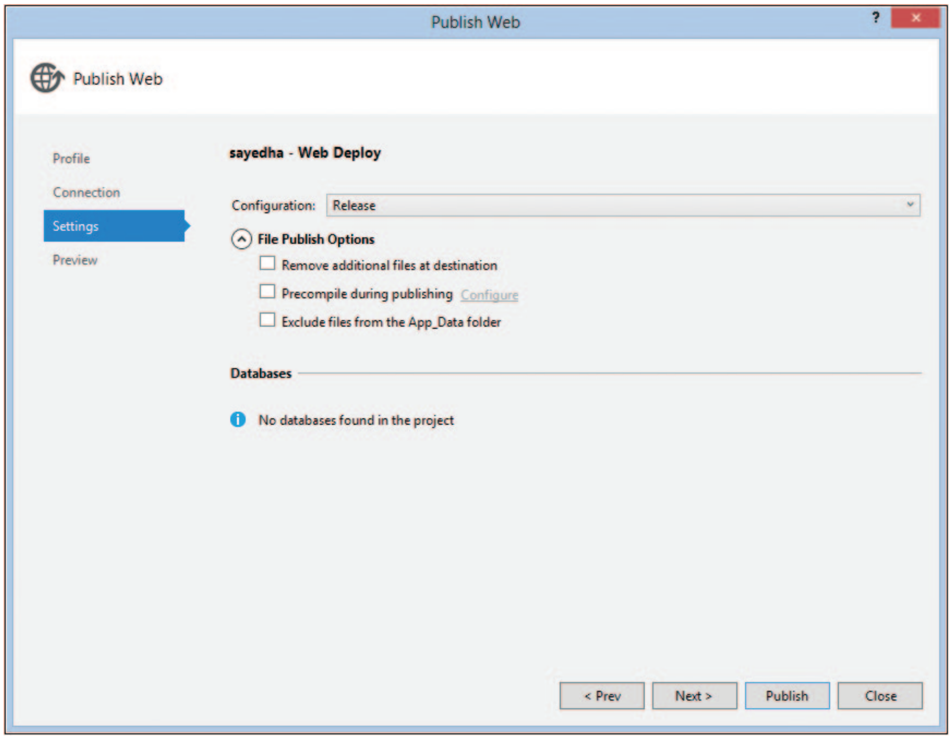
Password: ••

✓ Save password

Destination URL: http://sayedha.azurewebsites.net

FIGURE 3-3 The Connection tab of the Publish Web dialog box.

The values from the *.publishSettings* file are used to populate all the settings on the Connection tab. Depending on your hosting provider, you may need to specify the User Name and Password information here. You can also click Validate Connection to double-check that all the settings are correct. We will discuss the Connection tab in more detail when we demonstrate creating a package in the "Building web packages" section later in this chapter. The next tab in this dialog box is the Settings tab, shown in Figure 3-4.



sayedha - Web Deploy

Configuration: Release

File Publish Options

- [ ] Remove additional files at destination
- [ ] Precompile during publishing Configure
- [ ] Exclude files from the App_Data folder

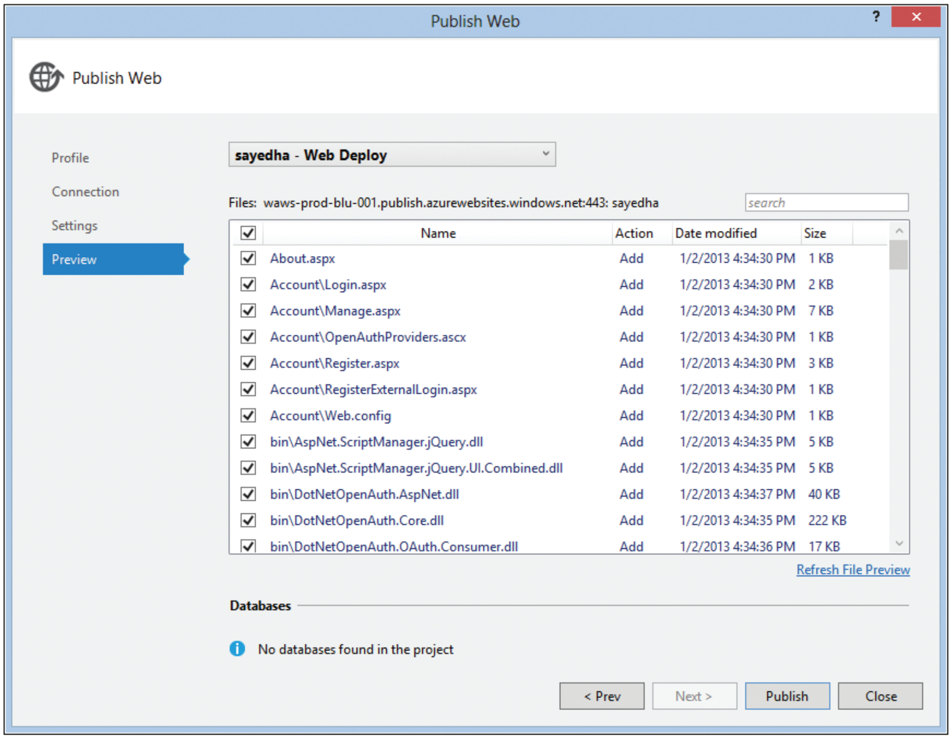
Databases

No databases found in the project

FIGURE 3-4 The Settings tab of the Publish Web dialog box.

On this tab, you can specify the build configuration that should be used during publishing by choosing an item from the Configuration drop-down list. When you configure this value, keep in mind that these values are drawn from the project build configurations, not solution build configurations. If you expect to see an additional value in the drop-down list but do not, odds are that you created a solution build configuration, but not a corresponding project build configuration. You can fix this by using the Configuration Manager in Visual Studio. One thing to be aware of with respect to this value: it's used only for the Visual Studio publishing process. For command-line scenarios, you need to specify the value for Configuration, as you would for any other build. Sayed has a good blog post with more details at *http://sedodream.com/2012/10/27/MSBuildHowToSetTheConfigurationProperty.aspx*. After you click Next, you will be taken to the Preview tab.

On the Preview tab, you can see the operations that will be performed when you publish your application. There are two areas: Files and Databases. In Figure 3-5, you can see the Preview tab populated with data from the SampleWeb project.



sayedha - Web Deploy

Files: waws-prod-blu-001.publish.azurewebsites.windows.net:443: sayedha

✓	Name	Action	Date modified	Size
✓	About.aspx	Add	1/2/2013 4:34:30 PM	1 KB
✓	Account\Login.aspx	Add	1/2/2013 4:34:30 PM	2 KB
✓	Account\Manage.aspx	Add	1/2/2013 4:34:30 PM	7 KB
✓	Account\OpenAuthProviders.ascx	Add	1/2/2013 4:34:30 PM	1 KB
✓	Account\Register.aspx	Add	1/2/2013 4:34:30 PM	3 KB
✓	Account\RegisterExternalLogin.aspx	Add	1/2/2013 4:34:30 PM	1 KB
✓	Account\Web.config	Add	1/2/2013 4:34:30 PM	1 KB
✓	bin\AspNet.ScriptManager.jQuery.dll	Add	1/2/2013 4:34:35 PM	5 KB
✓	bin\AspNet.ScriptManager.jQuery.UI.Combined.dll	Add	1/2/2013 4:34:35 PM	5 KB
✓	bin\DotNetOpenAuth.AspNet.dll	Add	1/2/2013 4:34:37 PM	40 KB
✓	bin\DotNetOpenAuth.Core.dll	Add	1/2/2013 4:34:35 PM	222 KB
✓	bin\DotNetOpenAuth.OAuth.Consumer.dll	Add	1/2/2013 4:34:36 PM	17 KB

Refresh File Preview

Databases

No databases found in the project

FIGURE 3-5 The Preview tab of the Publish Web dialog box.

Note You can double-click a file to see the difference between the local file and the remote file.

Because this project, SampleWeb, does not contain any databases, you only see file-related operations. When dealing with files, there are three possible Action types: *Add, Update,* and *Delete.* Because I've never published this project before, all the Action values are set to *Add.* At this point, we are ready to go, so click Publish to start the process. You can monitor the progress in the output window. After publishing your project, if a value was provided for the Destination URL on the Connection tab, that URL will be opened in a browser after a successful publish. Now that you have been introduced to the Publish Web dialog box, let's discuss how to create a web package in Visual Studio 2012.

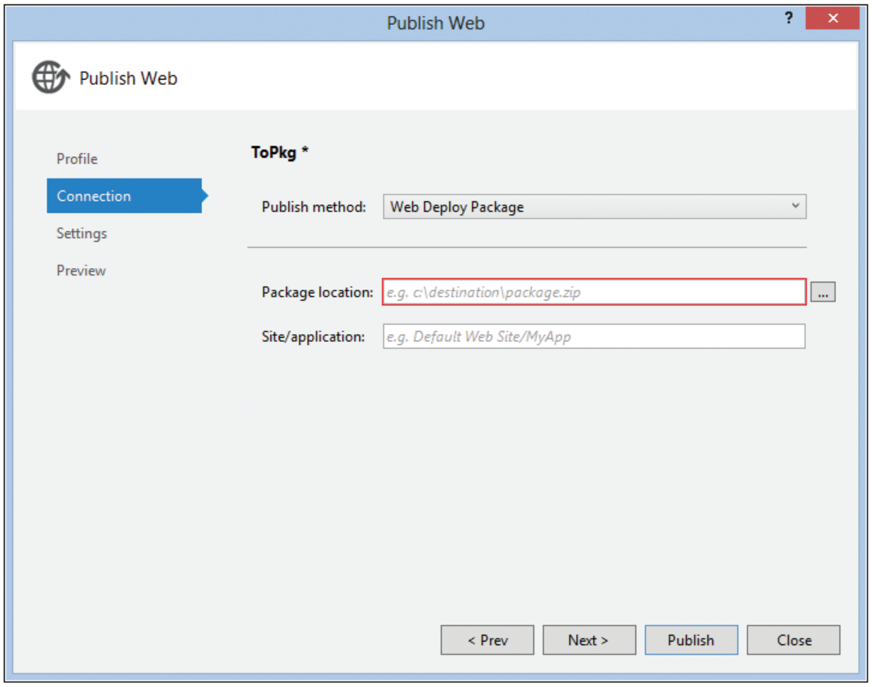
Building web packages

In Visual Studio 2012, you may have noticed that a menu option, Build Deployment Package, has disappeared. Don't worry—it's still easy to create a package. To create a web package in Visual Studio 2012, you can use the Publish Web dialog box. When you open the Publish Web dialog box, you can create a new profile on the Profile tab. To do that, use the New option in the Select Or Import A Publish Profile drop-down list. On the Connection tab, select Web Deploy Package from the Publish

Method drop-down list. There are many benefits to using a publish profile for packaging, some of which include the following:

1. Packages can include database artifacts.
2. You can customize the package process by using the *.pubxml* file.
3. You can package from the command line in the same way that you publish.

When you create the package profile in the Publish Web dialog box, the Connection tab will look like Figure 3-6.



ToPkg *

Publish method: Web Deploy Package

Package location: e.g. c:\destination\package.zip

Site/application: e.g. Default Web Site/MyApp

FIGURE 3-6 The Connection tab for the package profile.

In Figure 3-6, you can see two input fields: Package Location and Site/Application. Package Location should contain the path to the *.zip* file that you want to produce. This is a required field. The value for Site/Application is optional, but if you know the website or application that you are publishing to, you can provide the site name or application path here. When the package is published, this value will be used for the Web Deploy parameter *IIS Web Application Name*. Now let's create a package and take a look at the *.pubxml* file that was created.

Included in the samples is the PackageSample project. If you open that project, you will see that a package profile is defined. This profile, like other profiles, is stored in the PublishProfiles folder under Properties (My Project for Microsoft Visual Basic). Here are the contents of the ToPkg.pubxml file:

```
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
```

```
  <PropertyGroup>
    <WebPublishMethod>Package</WebPublishMethod>
    <SiteUrlToLaunchAfterPublish />
    <DesktopBuildPackageLocation>
        C:\InsideMSBuild\PublishOutput\PkgSample-default\PackageSample.zip
    </DesktopBuildPackageLocation>
    <PackageAsSingleFile>true</PackageAsSingleFile>
    <DeployIisAppPath />
    <PublishDatabaseSettings>
      <Objects xmlns="" />
    </PublishDatabaseSettings>
  </PropertyGroup>
</Project>
```

In this profile, you can see that *WebPublishMethod* is set to *Package,* which indicates that this is a profile that can be used to create a package. The path for the package is stored as the MSBuild property *DesktopBuildPackageLocation*. The other notable item here is the *PublishDatabaseSettings* property. Because my application did not contain any databases, this property is essentially empty. Even though it is empty, you should not remove it from the *.pubxml* file. You can easily automate the process of creating a package by following the same technique you use to automate the publishing process. Specifically, you'll create a publish profile and then use it to automate the process. Let's now take a closer look at publish profiles, including how to use them to automate packaging and publishing.

Publish profiles

When using the Publish Web dialog box after publishing or packaging, a publish profile is created. The publish profile contains all the settings entered into the Publish Web dialog box, as well as options that have not yet been seen in the dialog box. We can use these profiles from either Visual Studio or the command line. After your first publish profile is created, when you reopen the Publish Web dialog box, you are taken to the Preview tab with the most recently used profile automatically selected. On the Preview tab, you can switch profiles quickly using the drop-down list at the top of the dialog box. If you need to publish to a new destination, just go back to the Profile tab and create a new profile. You can have as many profiles defined as you like.

Publish profiles are saved in a folder named PublishProfiles under Properties (My Project for Visual Basic projects). Each profile will be saved into its own file with the extension of *.pubxml.* These files will be added to the project, and to source control, by default. Your publishing password will be saved in a *.user* file, which can only be decrypted by you, and not checked into version control, so you don't have to worry about any unauthorized publishing actions. If you want to keep a profile out of the sight of others, you can simply exclude the *.pubxml* file from the project and source control. When the Publish Web dialog box is opened, it will inspect the folder for the list of all profiles, not just profiles that are a part of the project. Now let's take a closer look at a sample *.pubxml* file.

In the following code block, you will see the contents of a Visual Studio publish profile that was created when I imported a *.publishSettings* file (these files are provided by hosting companies):

```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <WebPublishMethod>MSDeploy</WebPublishMethod>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>
    <SiteUrlToLaunchAfterPublish>http://sayedha.azurewebsites.net</SiteUrlToLaunchAfterPublish>
    <ExcludeApp_Data>False</ExcludeApp_Data>
    <MSDeployServiceURL>
          waws-prod-blu-001.publish.azurewebsites.windows.net:443</MSDeployServiceURL>
    <DeployIisAppPath>sayedha</DeployIisAppPath>
    <SkipExtraFilesOnServer>True</SkipExtraFilesOnServer>
    <MSDeployPublishMethod>WMSVC</MSDeployPublishMethod>
    <EnableMSDeployBackup>True</EnableMSDeployBackup>
    <UserName>$sayedha</UserName>
    <_SavePWD>True</_SavePWD>
  </PropertyGroup>
</Project>
```

From this code block, you can see that the *.pubxml* file is an MSBuild file. The properties declared are specific to the publishing method that is being used. Each *.pubxml* file has a single profile and contains all the values that are used by the Publish Web dialog box for this particular profile. This file is used by the Visual Studio user interface, but you can also employ this from the command line. Command-line publishing is supported only for the following publishing methods: Web Deploy, Web Deploy Package, and File System.

Web publish profiles are designed to allow you to extend the build and publishing process for a given publishing operation. When a publish profile is used to publish your application, the publish profile will be imported into the project itself. Because the *.pubxml* file is imported into the project file, you have full access to all MSBuild properties and items defined in the project. Because of this, from the *.pubxml* file, you can customize the build process and the publishing process. From the second edition, you may remember that you could customize the publishing process by editing the *.wpp.targets* file. Let's look at how to use this profile to publish the project from the command line.

Automating web publishing using a publish profile

Publishing from the command line is much easier than it used to be. If you remember from Chapter 19, "Web deployment tool practical applications," when publishing from the command prompt, you were required to pass in about 10 properties. Let's take a look at how simple the new command can be. This command can be used to publish SampleWeb using the profile *named to-prod:*

```
msbuild SampleWeb.sln /p:DeployOnBuild=true /p:PublishProfile=to-prod /p:Password=<insert-
password>
```

Note Depending on the web host that you are publishing to, you may need to also add `/p:AllowUntrustedCertificate=true`.

Tip If you are building a Visual Studio project file instead of the solution file, you should also specify the value for `/p:VisualStudioVersion=11.0.` Without this, the default value of `10.0` will be used.

With this command, the solution file will be built and published. When the *DeployOnBuild* property is set to *True,* the build process will be extended to publish the project as well. The name of the publish profile is passed in as the *PublishProfile* property. When specifying the value for *PublishProfile,* you have two options. You can pass in the name of the profile, in which case the build will use the named profile from the default location, or you can pass in the file path to the *.pubxml* file. Now let's look at how to use this same approach to create packages.

Automating web packaging using a publish profile

You can use this same technique when creating a package from the command line. The PackageSample project has a profile named ToPkg, which will create the web deploy package at C:\InsideMSBuild\PublishOutput\PkgSample-default\PackageSample.zip. In order to create this from the command line, we can execute the following command when building the project:

```
msbuild PackageSample.csproj /p:DeployOnBuild=true /p:PublishProfile=ToPkg
/p:VisualStudioVersion=11.0
```

With this command, you can override specific properties as well. For example, we showed previously that the package location is stored in the *.pubxml* file as an MSBuild property, *DesktopBuildPackageLocation.* If you would like to override the location where the package is created, pass the property as a command-line argument. For example, if I wanted to publish the package to C:\Temp\AltDest\Mypackage.zip, you can use the following command (which shows the value for *DesktopBuildPackageLocation* in bold).

```
msbuild PackageSample.csproj /p:DeployOnBuild=true /p:PublishProfile=ToPkg
/p:VisualStudioVersion=11.0 /p:DesktopBuildPackageLocation=c:\temp\altDest\mypackage.zip
```

Because the *DesktopBuildPackageLocation* property is specified as a command-line parameter, it overrides the value in the *.pubxml* file. Following the execution of this command, the package is written to the location provided. Now that we've shown how you can publish and package from the command line using a publish profile, we will discuss how the *.pubxml* file relates to the *.wpp.targets* file.

Relationship between publish profiles and *.wpp.targets*

In the second edition of this book, we showed how to customize the publishing process by creating a file named *{ProjectName}*.wpp.targets in the root of your project folder. The support for importing the *.wpp.targets* file has been in place since Visual Studio 2010. When a web project is built, it will look for a file in the same folder as the project file, with the naming pattern *{ProjectName}*.wpp.targets. If the file exists, it will be imported using the MSBuild *Import* element. This is very similar to how publish profiles work. There is one significant difference though. The *.wpp.targets* file will be imported for *every build,* not just for publishing. Because of this, you have to be a bit more careful, as it may affect other scenarios besides publishing.

For publishing customizations, it's recommended that you place those customizations inside the publish profile instead of a *.wpp.targets* file. This is because the modifications will only affect that particular publish profile, and it is much easier for others to diagnose any issues with publishing. Not many users would think to check for a *.wpp.targets* file if there are publishing issues.

You should use a *.wpp.targets* file if one of the following conditions exists:

- You want to extend the build process.
- You want to extend the publishing process for all publish profiles.

If you have existing projects with a *.wpp.targets* file, you do not need to modify them. They will continue to work. For new projects, you should place publish customizations in the publish profile. Let's move on to discuss the database support that exists in the web publish support.

Database publishing support

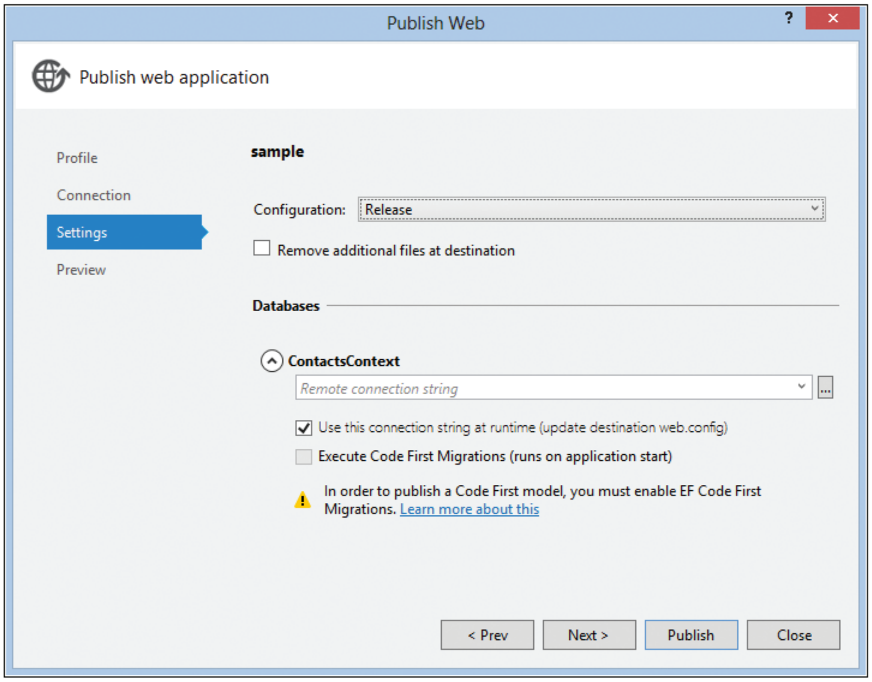
With the release of Visual Studio 2012, you now have the ability to publish database artifacts incrementally. Visual Studio has support for publishing databases in two ways: using Entity Framework (EF) Code First migrations and using a data-tier application package (DACPAC). We will first discuss EF Code First support and then discuss DACPAC support.

EF Code First migrations

If you have a web application that uses EF Code First, the recommended method to publish its database artifacts is to use EF Code First migrations. As you change your web application's database model, those changes are captured in code called *migrations*. When executed, migrations make the necessary changes to the database, thus keeping your database model and database in sync. You can easily move from one version of your database model to another by executing those migrations. Let's take a quick look at how EF Code First migrations work and then we will describe the support offered by the Publish Web dialog box.

When using EF Code First, you will create a context class, which you will use to access your database. After creating your context class, you will create one or more migration classes using the Package Manager Console. There are two ways to execute these migrations against a database: by using the Package Manager Console and by executing them at run time. The Publish Web dialog box involves the latter mechanism. You can configure your application to execute migrations at run time in two ways: by adding some code to your project to invoke the migrations, or by adding some elements to the Web.config file. The Publish Web dialog box uses the second approach to enable the migrations. Let's take a closer look at that.

If you have a web project with EF Code First contexts (classes extending DbContext), when you open the Publish Web dialog box, you will see those contexts on the Settings tab. Figure 3-7 shows what the dialog box looks like when you have an EF Code First context in your project but there are no migrations associated with it.



sample

Configuration: Release

☐ Remove additional files at destination

Databases

ContactsContext

Remote connection string

☑ Use this connection string at runtime (update destination web.config)

☐ Execute Code First Migrations (runs on application start)

In order to publish a Code First model, you must enable EF Code First Migrations. Learn more about this

FIGURE 3-7 No mitigations are associated with your project.

In Figure 3-7, you can see the *ContactsContext* class on the Settings tab of the Publish Web dialog box. In this case, there is a message indicating that you will need to create EF Code First migrations to publish the database associated with the context. Once you add migrations for the context, then the Execute Code First Migrations check box will be enabled. After adding a migration, when you re-enter the Publish Web dialog box, you can enter a destination connection string and enable the migrations to be executed. The connection string provided will be used for executing both the migrations and the run-time connection string.

Tip If you have a project with an EF Code First context and do not see it in the Publish Web dialog box, close the dialog box, rebuild the project, and then reopen the dialog box.

When you publish or package your web project, the final Web.config file will have the elements required to invoke the migrations. The migrations will be executed the first time that the EF Code First context is accessed. If your Web.config file does not have a connection string entry for the EF Code First context, then one will be added automatically to the published Web.config file. Now that we have discussed EF Code First contexts, let's move on to discuss the DACPAC support that is built in.

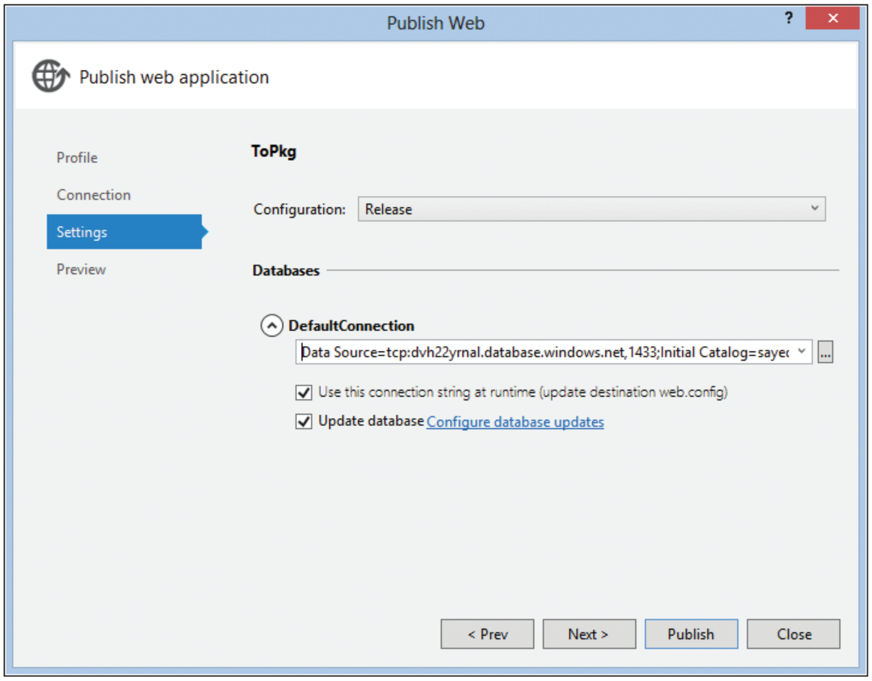
Incremental database publishing with DACPACs

If you need to publish the schema for a database incrementally, you can use a DAC package, also known as a *DACPAC*. A DACPAC is defined as follows in the MSDN library:

> *A DAC is a self-contained unit of SQL Server database deployment that enables data-tier developers and database administrators to package SQL Server objects into a portable artifact called a DAC package, also known as a DACPAC.*

In other words, a DACPAC contains all the schema artifacts that the database consists of. The significance of the words *portable artifact* should be highlighted here. The aspect that makes a DACPAC portable is the incremental publish support that is built on top of it. When using a DACPAC during publish time, the schema captured in the DACPAC is compared to that of the target database. The publish process will compute the difference between the DACPAC and the target database and then execute the difference against the target database. If the two are equal, then a no-op will be performed. Let's see how this works during the Publish Web workflow.

When you open the Publish Web dialog box, if you have any connection strings in the Web.config file that are not associated with an EF Code First context, then you will see those on the Settings tab. For example, in Figure 3-8, you can see the Settings tab for the ContactsSample project.



ToPkg

Configuration: Release

Databases

DefaultConnection

Data Source=tcp:dvh22yrnal.database.windows.net,1433;Initial Catalog=sayec

[✓] Use this connection string at runtime (update destination web.config)

[✓] Update database Configure database updates

FIGURE 3-8 The Publish Web dialog box with a database selected for publishing.

When you check the Update Database check box on the Settings tab when your web project is published or packaged, a DACPAC is created from the source connection string. This DACPAC is then transferred to the remote server to publish the database-related artifacts. This is facilitated by the new dbDacFx Web Deploy provider. This process is depicted in Figure 3-9.

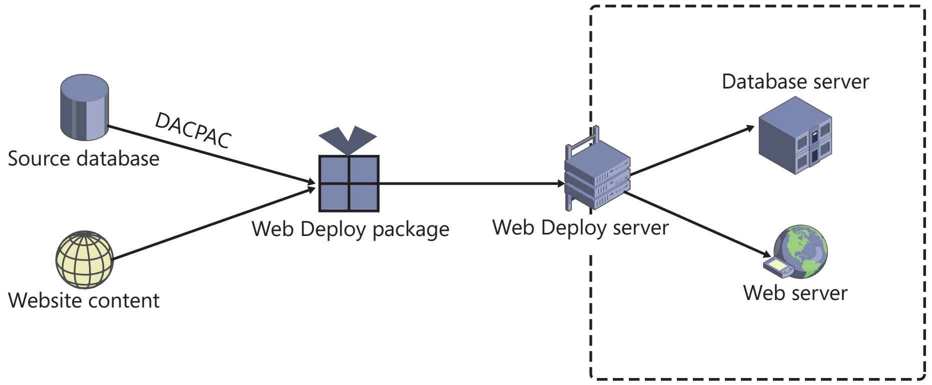


FIGURE 3-9 A Web and DACPAC publishing diagram.

In Figure 3-9, you can see that a DACPAC is created from the source database and placed in a Web Deploy package (or a folder for the direct publish case), and the web content is also placed there. The database schema will be published first, followed by any web updates. Both of these processes will be incremental; that is, only the changes will be applied, not a full publish. In Figure 3-9, the dotted line represents a firewall that may be in place. When publishing, if you do not have direct access to the remote database (which is common for many cloud hosting providers by default), that is OK so long as the Web Deploy server has access to it. When creating a Web Deploy package, the DACPAC will be placed inside the package and Web Deploy parameters will be created so that you can update the connection string during publishing. We will now discuss how to create a Web Deploy package with a DACPAC.

In the samples, you will find the ContactsSample project, which is a basic web application that stores contacts in a Microsoft SQL Server database. When creating a package for this on the Settings tab, I've chosen to package the database and provide a default connection string as well. This was shown previously in Figure 3-8. The resulting package will have the DACPAC for the source database in the root of the package. Let's see what happens when you import this package using the Microsoft IIS Manager user interface. Using IIS Manager, you can right-click a site and then select Import Application under the Deploy menu to import a Web Deploy package (see Figure 3-10).

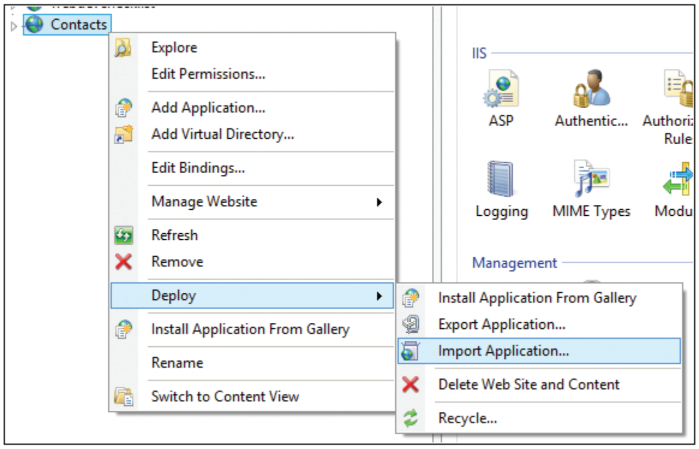
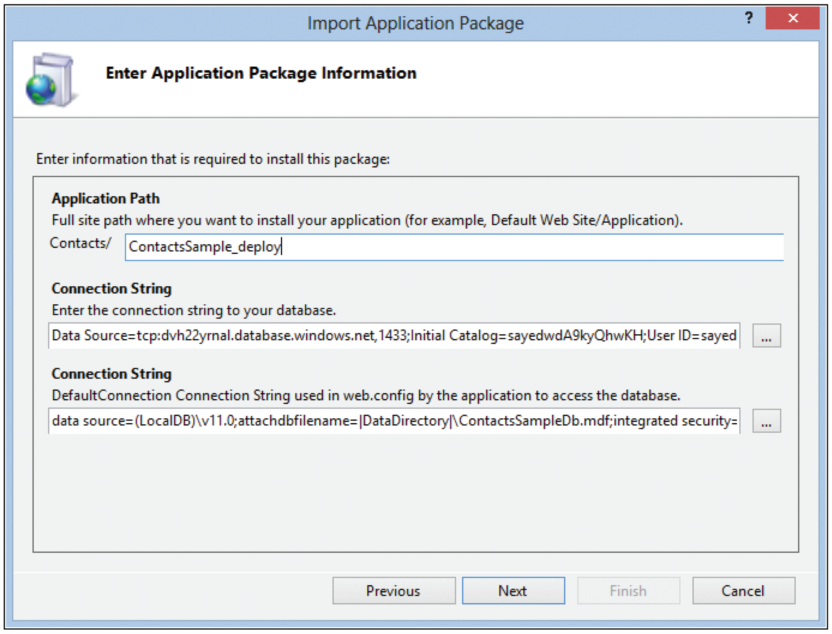


FIGURE 3-10 The Import Application option is the IIS Manager.

Tip If you do not see the Import Application option, you need to install Web Deploy with the IIS Manager Extensions option checked.

After selecting the package to be imported, you will be prompted to fill in the values for the Web Deploy parameters (as shown in Figure 3-11).


Import Application Package
Enter Application Package Information

Enter information that is required to install this package:

Application Path
Full site path where you want to install your application (for example, Default Web Site/Application).
Contacts/ ContactsSample_deploy

Connection String
Enter the connection string to your database.
Data Source=tcp:dvh22yrnal.database.windows.net,1433;Initial Catalog=sayedwdA9kyQhwKH;User ID=sayed

Connection String
DefaultConnection Connection String used in web.config by the application to access the database.
data source=(LocalDB)\v11.0;attachdbfilename=|DataDirectory|\ContactsSampleDb.mdf;integrated security=

FIGURE 3-11 Parameter prompts in IIS Manager for the ContactsSample package.

In Figure 3-11, you can see three parameters. The first parameter will define the IIS App path where your application will be installed. The next two parameters are connection strings for the DACPAC. The first is for the connection string used to publish the database related artifacts, and the final one is for the run-time connection string that goes in the Web.config file. If you want to use a lower-privileged connection string at run time, you can do so. After clicking Next, the database publish operations will be performed, followed by an update of the site itself. Now that we've discussed database publishing with DACPACs, let discuss the updates that are available for Web.config transforms.

Profile-specific Web.config transforms

In Visual Studio 2010, Web.config transforms were introduced, which you can use to update the Web.config file during the publish/package operation. If you are rusty on the basics of transforms, take a look back at Chapter 18, "Web deployment tool, part 2," in the second edition. In Visual Studio 2010, these transforms were tied to the build configuration. If you published using the Release build configuration, then the Web.config file would be transformed using Web.release.config. In Visual Studio 2012, you can now also have profile-specific transforms. A convention is used to associate a profile with a specific transform. To have a Web.config transform for a given profile, create a file with the pattern web.*{ProfileName}*.config next to Web.config. When the file is detected, it will be applied after the build configuration transform. You can see the Web.config transforms in Figure 3-12.

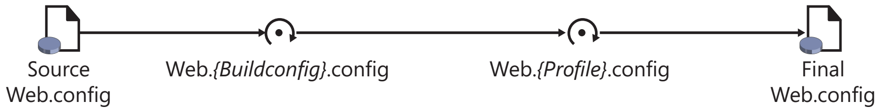


FIGURE 3-12 A Web.config transformation illustration.

When the Web.config file is being transformed, if either the build configuration transform or the profile-specific transform does not exist, that particular transform will simply be skipped. Let's take a look at how this works.

When Visual Studio 2012 was initially released, the underlying support to invoke these transforms existed in the web MSBuild targets, but there was no way to create these transforms easily. You had to create the transforms manually. In the ASP.NET 2012.2 update for Visual Studio 2012, a new context menu was added to help you create these transforms. With this update, you can create a profile-specific transform easily by right-clicking the *.pubxml* file and selecting Add Config Transform. You can see this new menu option in Figure 3-13.

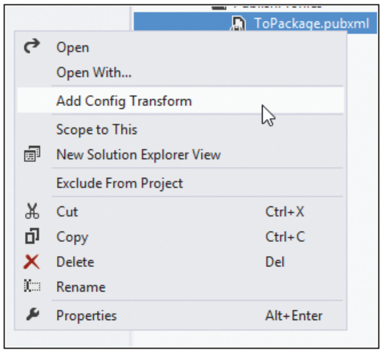


FIGURE 3-13 The Add Config Transform menu option for publish profiles.

When you invoke the Add Config Transform command, it will create the Web.config transform in the root of the project with the correct name and open it automatically. In the samples, you will find a project, TransformSample, that contains the ToPackage.pubxml publish profile. This publish profile is used when creating a web deploy package for this project. In this project, we have created the following transforms:

- Web.debug.config
- Web.release.config
- Web.ToPackage.config

Along with the Web.config file, the contents of these transforms are shown next. We will leave off the Web.debug.config file because it is not used in this demo.

Web.config file

```
<configuration>

  <appSettings>
    <add key="default" value="default"/>
  </appSettings>

    <system.web>
      <compilation debug="true" targetFramework="4.0" />
    </system.web>

</configuration>
```

Web.release.config

```
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
  <appSettings>
    <add key="release" value="from-release" xdt:Transform="Insert"/>
  </appSettings>

  <system.web>
    <compilation xdt:Transform="RemoveAttributes(debug)" />
  </system.web>
</configuration>
```

Web.ToPackage.config

```
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">

  <appSettings>
    <add key="to-package" value="from-ToPackage-transform" xdt:Transform="Insert"/>
  </appSettings>

</configuration>
```

The Web.release.config transform adds a new *appSettings* entry named *release* and removes the *debug* attribute from the compilation element. The Web.ToPackage.config transform adds a new *appSettings* entry named *to-package*. Packaging the application using the ToPackage profile produces the following Web.config file:

Final web.config after transforms

```
<configuration>

  <appSettings>
    <add key="default" value="default"/>
    <add key="release" value="from-release"/>
    <add key="to-package" value="from-ToPackage-transform"/>
  </appSettings>

    <system.web>
      <compilation targetFramework="4.0" />
    </system.web>

</configuration>
```

In this file, you can see that the release transform inserted the release app setting and removed the *debug* attribute from the *compilation* element. You can also see that the Web.ToPackage.config transform was invoked. Another subtle thing to notice here is the order in which the app settings were inserted. The release setting was inserted before the *to-package* element. This indicates that the Web.release.config transform was invoked before Web.ToPackage.config.

Another feature released with Visual Studio 2012 is the ability to preview these transforms. In Visual Studio 2010, if you wanted to see the resulting Web.config transform, you would have to either publish or package your project, which made developing these transforms much more difficult than it should have been. In Visual Studio 2012, however, you can now preview Web.config transforms easily. The preview functionality works for build configuration transforms as well as profile-specific ones. You can right-click and select Preview Transform on any of the transforms. This new option is shown in Figure 3-14.

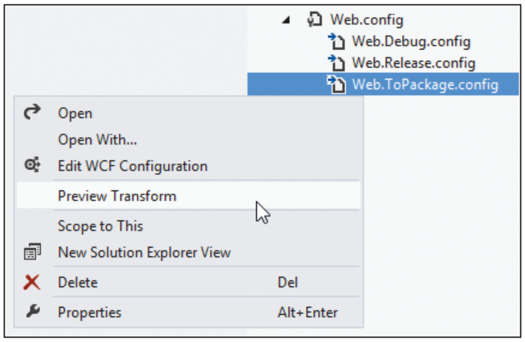
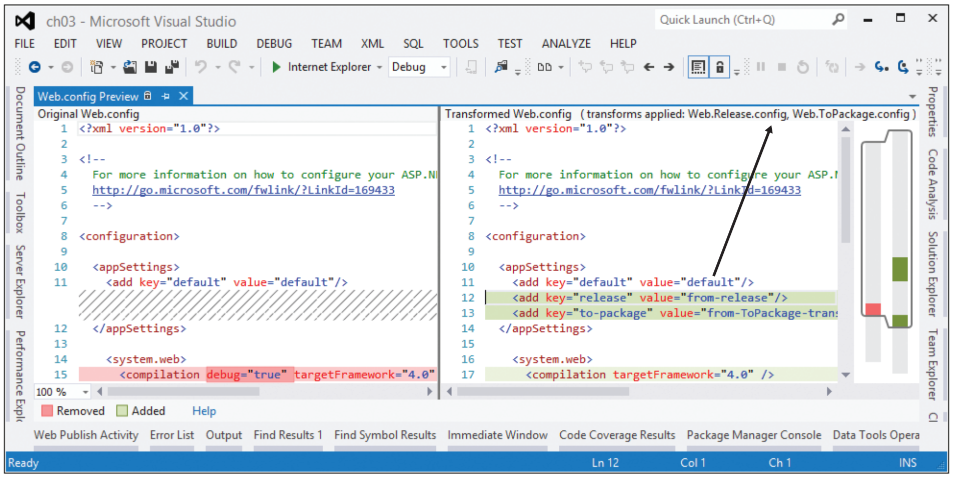


FIGURE 3-14 The Preview Transform menu option.

Once you invoke this preview, you will be able to see the final Web.config transform. When you preview a profile-specific transform, it will invoke the correct build configuration transform before applying the profile-specific one. It mimics the behavior that it will show when publishing. When you are viewing the preview results, you can see which transforms have been applied in the upper-right corner (see Figure 3-15).



Original Web.config

```
<?xml version="1.0"?>

<!--
  For more information on how to configure your ASP.N
  http://go.microsoft.com/fwlink/?LinkId=169433
  -->

<configuration>

  <appSettings>
    <add key="default" value="default"/>

  </appSettings>

    <system.web>
      <compilation debug="true" targetFramework="4.0"
```

Transformed Web.config (transforms applied: Web.Release.config, Web.ToPackage.config)

```
<?xml version="1.0"?>

<!--
  For more information on how to configure your ASP.N
  http://go.microsoft.com/fwlink/?LinkId=169433
  -->

<configuration>

  <appSettings>
    <add key="default" value="default"/>
    <add key="release" value="from-release"/>
    <add key="to-package" value="from-ToPackage-tran
  </appSettings>

    <system.web>
      <compilation targetFramework="4.0" />
```

FIGURE 3-15 A Web.config transform preview.

With these updates for Web.config transforms, it's much easier to create and use Web.config transforms. This concludes the Web.config transform content, as well as the section covering the new features. We will now move on to look at some real-world examples.

How to publish a package to multiple destinations

One of the most common questions that I'm asked is, "How can I create a Web Deploy package that can be used to publish to multiple servers?" Any Web Deploy package can be published to any destination, but you may need to tweak the content for the destination. This is particularly true for Web.config. One of the challenges when creating a package from Visual Studio that can target multiple destinations is the handling of Web.config. When you create a package in Visual Studio or from the command line, the Web.config that is placed in the resulting package is already transformed. This makes these packages difficult to publish to multiple different locations by default. There is an extension created by Sayed that can help here, called PackageWeb.

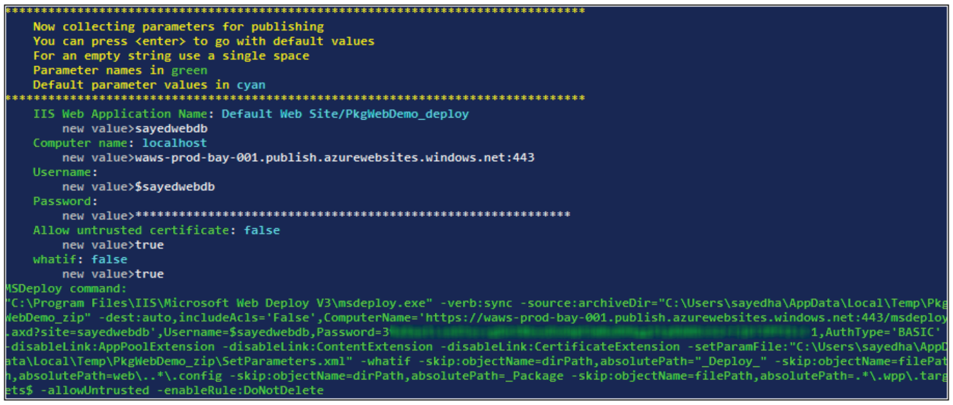
PackageWeb can be used to help create portable packages that can be published easily to multiple destinations. PackageWeb is a *NuGet* package that can be installed into web projects. Let's see how PackageWeb works. To use PackageWeb, you will need to add the package to your web project. You can do this either from the Manage NuGet Packages dialog box or from the Package Manager Console. From the Package Manager Console, you can execute the following command:

```
Install-Package PackageWeb
```

Once the package has been installed in your project, it will extend the package process. When you create a package after installing PackageWeb, you will see a new file, Publish-Interactive.ps1, in the output location. This is a Windows PowerShell script that can be used to publish this package. From a PowerShell prompt, you can invoke this script to start the publish process. Once you invoke this script, you will be prompted for the following set of values:

- Web.config transform to execute
- Web Deploy publish settings
- Web Deploy parameter values

After providing these values, the Web.config file will be transformed with the given transform and the publish operation will be invoked. Let's see this in action. In the samples, the project PkgWebDemo already has PackageWeb installed. After creating the package, you can invoke Publish-Interactive.ps1 to start the publish process. Figure 3-16 shows PackageWeb prompts for the Web Deploy publish settings. Because there are no Web Deploy parameters created for the sample, you are not prompted for those.



```
********************************************************************************
    Now collecting parameters for publishing
    You can press <enter> to go with default values
    For an empty string use a single space
    Parameter names in green
    Default parameter values in cyan
********************************************************************************
    IIS Web Application Name: Default Web Site/PkgWebDemo_deploy
        new value>sayedwebdb
    Computer name: localhost
        new value>waws-prod-bay-001.publish.azurewebsites.windows.net:443
    Username:
        new value>$sayedwebdb
    Password:
        new value>************************************************************
    Allow untrusted certificate: false
        new value>true
    whatif: false
        new value>true
MSDeploy command:
"C:\Program Files\IIS\Microsoft Web Deploy V3\msdeploy.exe" -verb:sync -source:archiveDir="C:\Users\sayedha\AppData\Local\Temp\Pkg
WebDemo_zip" -dest:auto,includeAcls='False',ComputerName='https://waws-prod-bay-001.publish.azurewebsites.windows.net:443/msdeploy
.axd?site=sayedwebdb',Username=$sayedwebdb,Password=3[redacted]1,AuthType='BASIC'
-disableLink:AppPoolExtension -disableLink:ContentExtension -disableLink:CertificateExtension -setParamFile:"C:\Users\sayedha\AppD
ata\Local\Temp\PkgWebDemo_zip\SetParameters.xml" -whatif -skip:objectName=dirPath,absolutePath="_Deploy_" -skip:objectName=filePat
h,absolutePath=web\..*\.config -skip:objectName=dirPath,absolutePath=_Package -skip:objectName=filePath,absolutePath=.*\.wpp\.targ
ets$ -allowUntrusted -enableRule:DoNotDelete
```

FIGURE 3-16 PackageWeb prompts for the Web Deploy settings.

After completing the prompts, Msdeploy.exe is called to start the publish process. In Figure 3-16, you can see the call to Msdeploy.exe that is invoked. The password value in this image is blurred for security reasons. Once you fill in the prompts and publish your package, a new file, PublishConfiguration .ps1.readme, will be created in the same folder. This file contains all the values for the prompts that were entered. The only value not persisted is the password; you will need to update this value manually. In order for PakageWeb to pick up this file automatically, just remove the *.readme* extension.

When PackageWeb is invoked, it will look for a file named PublishConfiguration.ps1 in the same folder as the Publish-Interactive.ps1 file. If that file exists, it will be imported, and you will not be prompted for values. (This is a very brief discussion of PackageWeb; for more details visit *http://msbuildbook.com/packageweb*.) Let's move on to discuss a neat trick that you can use during the package process.

Customizing the folder structure inside the package

When you create a package in Visual Studio by default, the full source structure is replicated inside the package. For example, when packaging the PackageSample project using the ToPkg profile, you can see how the contents are structured in the resulting *.zip* file in Figure 3-17.

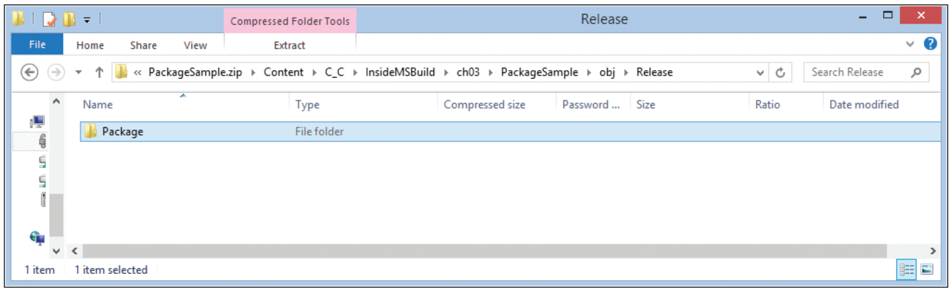


FIGURE 3-17 The default folder structure for the PackageSample project.

In Figure 3-17, you can see that the source folder structure is replicated inside the generated PackageSample.zip file. This behavior is annoying, but it can go beyond that and cause real difficulties if you need to expand this package. When publishing with Web Deploy, the depth of these folders does not matter, but if you expand them on disk and manipulate the files, you may exceed the maximum path length. To avoid this, it would be better to create a Web Deploy package that did not have these unnecessary folders. Let's see what it would take to simplify the folder structure here.

When creating a package using web projects, the following basic steps are followed:

1. Build a project.
2. Gather all files in a Temp directory.
3. Create the package by calling Web Deploy.

Step 3 is executed by creating an XML file that describes how to create the package. This is referred to as a *Source Manifest file*. You can find this file in the same folder in which the package is created. If you inspect the file generated when packaging the PackageSample project, you will find the contents to be as shown in the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<sitemanifest>
  <IisApp path="C:\InsideMSBuild\ch03\PackageSample\obj\Release\Package\PackageTmp"
                    managedRuntimeVersion="v4.0" />
  <setAcl path="C:\InsideMSBuild\ch03\PackageSample\obj\Release\Package\PackageTmp"
                    setAclResourceType="Directory" />
  <setAcl path="C:\InsideMSBuild\ch03\PackageSample\obj\Release\Package\PackageTmp"
                    setAclUser="anonymousAuthenticationUser" setAclResourceType="Directory" />
</sitemanifest>
```

In this manifest, you can see that three Web Deploy providers will be called when the package is created. Each of these providers references the full path to the temporary package folder. These are the values shown in bold in this code, and this is what we want to update during the package creation process. You can see the goal in Figure 3-18.

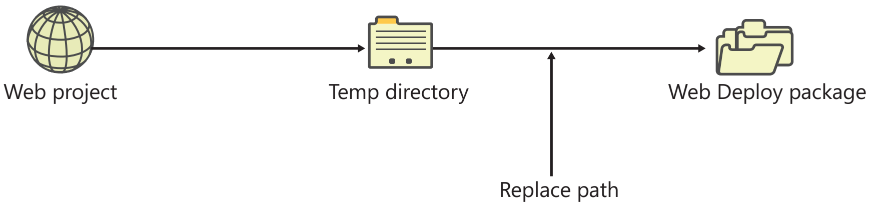


FIGURE 3-18 The process to update file paths for the generated package.

As you can see, we will replace the path *during* the package operation. We will do this with a Web Deploy replace rule. Let's see how to do that with some customizations to the *.pubxml* file.

When creating a package using Web Projects, you have the ability to replace values as the *.zip* file is being created. This is facilitated by using a replace rule. The replace rule that we want to create should match the package path and replace it with a much simpler value. To add a Web Deploy replace rule, you need to populate the MSDeployReplaceRules item list in the *.pubxml* file before the package is created. The next code fragment needs to be added to the *.pubxml* file to simplify these paths. The entire profile can be found in the PackagePath.pubxml file in the PackageSample project.

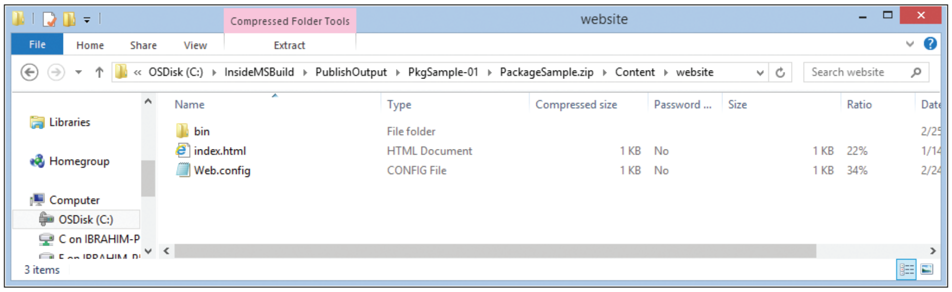
```
<PropertyGroup>
  <PackagePath Condition=" '$(PackagePath)'=='' ">website</PackagePath>
  <PackageDependsOn>
    $(PackageDependsOn);
    AddReplaceRuleForAppPath;
  </PackageDependsOn>
</PropertyGroup>

<Target Name="AddReplaceRuleForAppPath">
  <PropertyGroup>
    <_PkgPathFull Condition=" '$(WPPAllFilesInSingleFolder)'!='' ">
$([System.IO.Path]::GetFullPath($(WPPAllFilesInSingleFolder)))</_PkgPathFull>
    <!-- $(WPPAllFilesInSingleFolder) is not available on VS2010 so fall back to
$(_PackageTempDir) -->
    <_PkgPathFull Condition=" '$(_PkgPathFull)' == '' ">
$([System.IO.Path]::GetFullPath($(_PackageTempDir)))</_PkgPathFull>
  </PropertyGroup>

  <!-- escape the text into a regex -->
  <EscapeTextForRegularExpressions Text="$(_PkgPathFull)">
    <Output TaskParameter="Result" PropertyName="_PkgPathRegex" />
  </EscapeTextForRegularExpressions>

  <!-- add the replace rule to update the path -->
  <ItemGroup>
    <MsDeployReplaceRules Include="replaceFullPath">
      <Match>$(_PkgPathRegex)</Match>
      <Replace>$(PackagePath)</Replace>
    </MsDeployReplaceRules>
  </ItemGroup>
</Target>
```

In this fragment, you can see the *AddReplaceRuleForAppPath* target. This target is injected into the package process by appending it to the *PackageDependsOn* property. When this target is invoked, it will determine the full path to the temporary package folder. This path is converted to a regular-expression format by using the EscapeTextForRegularExpressions task. Then the value is appended to the MSDeployReplaceRules item list. As a result, when the package is created, the complex folder structure will be replaced with a folder named Website, defined in the *PackagePath* property. When you create the package after these changes, you can see the new structure of the created *.zip* file in Figure 3-19.



Name	Type	Compressed size	Password ...	Size	Ratio	Date
bin	File folder					2/25
index.html	HTML Document	1 KB	No	1 KB	22%	1/14
Web.config	CONFIG File	1 KB	No	1 KB	34%	2/24

FIGURE 3-19 A simplified view of the package structure.

In Figure 3-19, you can see that the complex folder structure has been replaced with the Website folder, in which all the web content that will be published resides. Now that we've shown how to create better web packages, we will move on to the next sample.

How to publish a folder with Web Deploy

There are many scenarios in which it would be helpful to simply publish the contents of a local folder to a remote IIS server. For example, your web project may serve some binary content created by another group, and that content is not in source control. In this case, you can directly use Msdeploy.exe to publish that content. We will show how to publish a folder to a remote site using Web Deploy. The exact command that is required may vary based on how the IIS host is running Web Deploy. For this sample, we will be demonstrating this while publishing to Windows Azure Web Sites. Windows Azure Web Sites host MSDeploy using the Web Management Service (WMSvc), which is the common method for most third-party IIS hosting companies. Let's see how to accomplish this.

The Web Deploy provider that has the information on how to publish a folder is the contentPath provider, and this is what we will be using. When invoking MSDeploy to sync a folder, the basic command structure is as follows:

```
msdeploy.exe
    -verb:sync
    -source:contentPath="<source-path>"
    -dest:contentPath="<dest-path>"
```

Because we are attempting to synchronize two folders, the *sync* verb is used and we use *contentPath* for both the source and destination. The source folder that we want to publish is C:\InsideMSBuild\Ch03\FolderPublish\ToPublish, and we would like to publish it to the Media folder under the FolderPub site. Let's make a first attempt to figure out what the final command might look like:

```
msdeploy.exe
    -verb:sync
    -source:contentPath="C:\InsideMSBuild\ch03\FolderPublish\ToPublish"
    -dest:contentPath="FolderPub/Media"
```

This command would work great if the site you want to publish to was running on the local box. Because it is not, we will need to start adding some information to the destination to indicate the server against which this command should execute. We will need to add the following parameters to the command:

- **ComputerName** The URL, or computer name, that will handle the publish operation.
- **Username** The user name for the publish operation.
- **Password** The password for the publish operation.
- **AuthType** Describes what authentication mechanism is used. The options here are either Basic or NTLM. Typically, you will use Basic when Web Deploy is running under WMSvc and NTLM for when it is hosted using the Remote Agent Service.

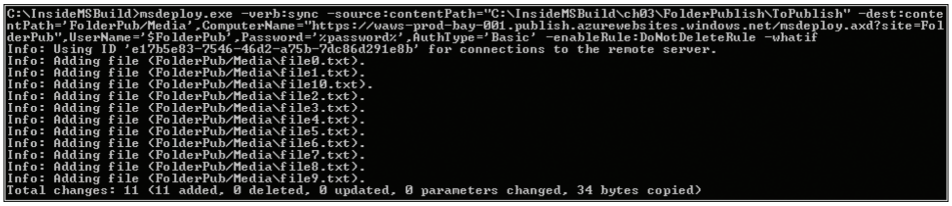
In this case, the values that we will use for these are

- **ComputerName** *https://waws-prod-bay-001.publish.azurewebsites.windows.net/msdeploy .axd?site=FolderPub*
- **Username** $FolderPub
- **Password** <*Publishing password*>, where *Publishing password* is whatever password you have chosen
- **AuthType** Basic

For Windows Azure Web Sites, you can find these values in the publish profile, which you can download from the Azure portal. Let's add these values to the command:

```
msdeploy.exe
    -verb:sync -source:contentPath="C:\InsideMSBuild\ch03\FolderPublish\ToPublish"
    -dest:contentPath='FolderPub/Media'
      ,ComputerName="https://waws-prod-bay-001.publish.azurewebsites.windows.net/msdeploy.axd?
          site=FolderPub"
      ,UserName='$FolderPub'
      ,Password='%password%'
      ,AuthType='Basic'
    -enableRule:DoNotDeleteRule
    -whatif
```

In this command, we've added the destination values, as well as two additional options: *-enableRule:DoNotDeleteRule* and *-whatif*. We pass the *DoNotDeleteRule* to ensure that any files in the folder that are on the server but not the client remain on the server. For now, we are also passing *-whatif*, which displays the command's operations without actually performing them, but we will remove that when we are ready to publish the folder. You can find the result of this command in Figure 3-20.



```
C:\InsideMSBuild>msdeploy.exe -verb:sync -source:contentPath="C:\InsideMSBuild\ch03\FolderPublish\ToPublish" -dest:conte
ntPath='FolderPub/Media',ComputerName="https://waws-prod-bay-001.publish.azurewebsites.windows.net/msdeploy.axd?site=Fol
derPub",UserName='$FolderPub',Password='%password%',AuthType='Basic' -enableRule:DoNotDeleteRule -whatif
Info: Using ID 'e17b5e83-7546-46d2-a75b-7dc86d291e8b' for connections to the remote server.
Info: Adding file (FolderPub/Media\file0.txt).
Info: Adding file (FolderPub/Media\file1.txt).
Info: Adding file (FolderPub/Media\file10.txt).
Info: Adding file (FolderPub/Media\file2.txt).
Info: Adding file (FolderPub/Media\file3.txt).
Info: Adding file (FolderPub/Media\file4.txt).
Info: Adding file (FolderPub/Media\file5.txt).
Info: Adding file (FolderPub/Media\file6.txt).
Info: Adding file (FolderPub/Media\file7.txt).
Info: Adding file (FolderPub/Media\file8.txt).
Info: Adding file (FolderPub/Media\file9.txt).
Total changes: 11 (11 added, 0 deleted, 0 updated, 0 parameters changed, 34 bytes copied)
```

FIGURE 3-20 The result of invoking the Msdeploy.exe command.

At this point, we are ready to execute this command and publish the folder. You can find this command in the samples for Chapter 3 in the file FolderPublish\publishFolder-standard.cmd. There is another cmd file in that same folder, called PublishFolder-auto.cmd. This file shows how you can use this same technique with the -dest:auto provider. We won't cover that here, but it is in the samples for you to reference.

In this chapter, we have covered a lot of new material, including the Publish Web dialog box, updates to website project publishing, packaging, publish profiles, and more. That's a lot of material to discuss in just a few pages, and we didn't even cover all the new features. This chapter should serve as a solid starting point for your journey in web publishing. From here, the best thing to do is practice. If you get stuck, try StackOverflow.com (and you can typically find Sayed hanging around there as well—if you see him, say hello).

Index