Praise for

Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build, Second Edition

"Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build is a practical book covering all the essentials of MSBuild and the Team Foundation Server build system. But what makes the book extra valuable is its focus on real-life scenarios that often are hard to find a good, working solution for. In fact there is information in the book you're unlikely to find anywhere else. With the second edition of the book, the authors fill the gaps again, this time by covering the new TFS build workflow technology as well as MSBuild 4.0. It is an invaluable book that saves lots of time whenever you work with any aspect of automated builds in Visual Studio and TFS. This is a book I'll make sure to have with me all the time!"
-Mathias Olausson, ALM Consultant, QWise/Callista, Sweden

"As an ALM Consultant I come across many teams that are struggling with their build tools and processes. The second edition of Sayed and William's book is the perfect answer for these teams. Not only will it show you how to get your builds back on track, I challenge anyone not to be able to use the information in this book to improve their existing builds. It includes updated content focusing on the new Visual Studio 2010 release and is packed with practical examples you could start using straight away. You simply must include it in your technical library."
-Anthony Borton, Microsoft Visual Studio ALM MVP, Senior ALM trainer/consultant, Enhance ALM Pty Ltd, Australia

"The first edition of Inside the Microsoft Build Engine was a brilliant look at the internals of MSBuild, so it's fantastic to see Sayed and William updating it with all the new features in MSBuild 4.0 and also delving into the Team Foundation Server 2010 workflow based build process. It's also a real pleasure to see deployment with MSDeploy covered so that you can learn not only how to automate your builds, but also how to automate your deployments. A great book. Go out and get a copy now."
-Richard Banks, Visual Studio ALM MVP and Principal Consultant with Readify, Australia

"Did you know about the TaskFactory in MSBuild? If not, you're not alone - but you will know after reading this book. This book provides insights into the current technologies of the Microsoft Build Engine. Starting with background information about MSBuild, it covers also the necessary basics of Workflow Foundation which are applied during the description of advanced topics of Team Foundation Build. The level of detail is targeted to experienced build masters having a development background - even the overview is stuffed with new information, references, hints and best practices about MSBuild. Samples are provided as step-by-step guidance easy to follow inside Visual Studio. What I found astonishing is the practical focus of the samples such as web project deployment. I could have used at least half of them in my development projects! Simply put: A must read for all build experts that have to deal with MSBuild and the Team Foundation Server build engine who are not only interested in solutions but also background information!"
-Sven Hubert, AIT TeamSystemPro Team, Consultant, MVP Visual Studio ALM – www.tfsblog.de

"The reason that I only own one MSBuild/Team Build book is because there is no need for another. This book covers both topics from soup to nuts and is written in a way that allows new users to ramp up quickly. The real-world code examples used to illustrate the topics are useful in their own right. The Second Edition covers all of the changes in MSBuild 4.0 and all of the newness that is Team Build 2010. This is my 'go to' guide, and the only book on these topics that I recommend to my clients."
-Steve St Jean, Visual Studio ALM MVP, DevProcess (ALM) Consultant with Notion Solutions, an Imaginet Company

"Whether you consider yourself experienced or you are taking your first steps in the build and automation arena, this 2nd edition will prove a valuable read. Skilled MSBuild users will do well to remind themselves of the intricacies of MSBuild and learn of the new 4.0 features whilst novices are taken on a steady paced journey to quickly acquire the knowledge and confidence in developing successful solutions. This edition brings additional value to our ever changing profession in discussing MSDeploy and the new Windows Workflow 4.0 based Team Foundation Build. Regardless of your experience, I wholeheartedly recommend this book."
-Mike Fourie, Visual Studio ALM MVP and ALM Ranger, United Kingdom

"The first edition of this book had a perfect balance between a tutorial and a reference book. I say this as I used the book first to kick start my MS Build knowledge and then as reference whenever I needed information on some advanced topic. My main interest is Team Foundation Server and I learned MS Build more from necessity than an urge, hence I was very curious to see the 2nd edition. Sayed and William did not disappoint me - the four chapters on Team Build cover all points needed to customize builds. As a bonus there are three whole chapters on web deployment which is a recurrent request I hear during my consulting and presentations on TFS. If I had to summarize my opinion in a single sentence, I would just say `Buy the book, you won't regret it'."
-Tiago Pascoal, Visual Studio ALM MVP and Visual Studio ALM Ranger, Portugal

"Reliable and repeatable build processes are often the Achilles' heel of development teams. Often this is down to a lack of understanding of the underlying technologies and how they fit together. No matter which Continuous Integration (CI) tool you may be using, this book provides the fundamental information you need to establish solid build and deployment engineering practices and demystifies the various Microsoft technologies used along the way. This book is the essential reference for any team building software on the Microsoft.NET platform."
-Stuart Preston, Visual Studio ALM Ranger and Chief Technology Officer at RippleRock

"Successfully deploying application is one of the big challenges in today's modern software development. As applications become more complex to develop, they also become more complex to deploy. This well-written book provides us a deep-dive on how developers can improve their productivity and accomplish the business needs using Microsoft deployment technology: MSBuild, Web Deploy and Team Build. Microsoft provides us the right tools, and this book provides us the information we need to extract real value from these tools."
-Daniel Oliveira, MVP, Visual Studio ALM Ranger and ALM Consultant at TechResult

Microsoft

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

Inside the Microsoft® Build Engine

2
SECOND EDITION

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: 2010940848
ISBN: 978-0-7356-4524-0

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Devon Musgrave
Developmental Editor: Devon Musgrave
Project Editor: Iram Nawaz
Editorial Production: S4Carlisle Publishing Services
Technical Reviewer: Marc H. Young
Cover: Tom Draper Design

Body Part No. X17-29997

I would like to dedicate this book to my parents, Sayed A. Hashimi and Sohayla Hashimi, as well as my college advisor, Dr. Ben Lok. My parents have, over the course of the years, sacrificed a lot to give us the opportunity for us to be able to achieve our dreams. I can only hope that they are proud of the person that I have become. When I first met Ben, I wanted to get into a research program that he had going. Thankfully, he was willing to accept me. Ben helped show me how rewarding hard work can be, and he has enabled me to succeed in my career. When I look back on influences in my life, who are not relatives, he ranks at the top of my list. I am sure that I wouldn't be where I am had it not been for him.

—Sayed Ibrahim Hashimi

To my mother, Rosanna O'Sullivan, and my father, Roy Bartholomew, for their unfaltering support in all my endeavors.

—William Bartholomew

I would like to dedicate this book to my parents, Syama Mohana Rao Adharapurapu and Nalini Adharapurapu, my brother, Raghavendra Adharapurapu, my sister, Raga Sudha Vijjapurapu, and my wife, Deepti Ramakrishna.

—Pavan Adharapurapu

I dedicate this book to my wife, Samantha, and my daughters, Amelie and Madeline, as well as my parents, Leonea and Craig. Their love has no boundaries and their support has made me believe that I can accomplish anything.

—Jason Ward

Contents at a Glance

Table of Contents

Part III Advanced MSBuild Topics

Part IV MSBuild Cookbook

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Foreword

Often when people think about build, they think just about the act of compiling some source code – when I hit F5 in the IDE, it builds, right? Well yes, kind of. In a real production build system, there is so much more to it than that. There are many kinds of builds – F5, desktop, nightly, continuous, rolling, gated, buddy etc. The variety of build types is reflective of the important role build plays in the software development process and the varied ways it does so. Build is a key integration point in the process. It is where developers' work comes together; it is where developers hand off to test and where release hands off to operations. No wonder there are so many requirements on it.

As I mentioned, build is about a lot more than compiling the code. It can include making sure the right code is assembled, compiling, testing, version stamping, packaging, deployment and more. Of course, because software systems are all different and organizations are different, many of the activities need to be completely different. As a result, extensibility plays a major role. In TFS 2010, we increased the extensibility options by including a build workflow engine (based on the .NET Workflow Foundation) on top of the existing msbuild capabilities. Unfortunately, as flexibility increases, so does the amount you need to know to make sound decisions and fully automate your build workflow.

This book is a great resource to help you understand the variety of roles build plays in software development and how you can leverage msbuild and TFS. It will show you how to use "out of the box" solutions, provide guidance on when to customize, what the best customization approaches are and details on and examples of how to actually do it. I think it will be an invaluable resource to keep on your reference shelf.

Brian Harry

Technical Fellow

Team Foundation Server, Microsoft

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 build, 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 five to ten years ago. Along with that the process by which we write software is different as well. Nowadays it is not uncommon for a project to have sophisticated build processes which include such things as code generation, code analysis, unit testing, automated deployment, etc. 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 leverage it to meet their needs.

Back in 2005 Microsoft released MSBuild, which is the build engine used to build most Visual Studio projects. That release was MSBuild 2.0. Since that release Microsoft has released two major versions of MSBuild—MSBuild 3.5 and MSBuild 4.0. In MSBuild 3.5 Microsoft released such goodness as multi-processor support, multi-targeting, items and properties being defined inside of targets and a few other things which brought MSBuild to where it needed to be. In MSBuild 4.0 there were a lot of really great features delivered. The feature which stands out the most is the support for building Visual C++ projects. Starting with Visual Studio 2010 your Visual C++ project files are in MSBuild format. Modifying MSBuild to be able to support building Visual C++ projects was a big effort on Microsoft's part, but they understood that the value they were delivering to customers would be worth it. Along with support for Visual C++ there were a number of significant feature add ons, such as support for BeforeTargets/AfterTargets, inline tasks, property functions, item functions and a new object model to name a few. During that same period Team Build has undergone a number of big changes.

Team Foundation Build (or Team Build as it is more commonly known) is now in its third version. Team Build 2005 and 2008 were entirely based on MSBuild using it for both build orchestration as well as the build process itself. While this had the advantage of just needing to learn one technology MSBuild wasn't suited for tasks such as distributing builds across multiple machines and performing complex branching logic. Team Build 2010 leverages the formidable combination of Workflow Foundation (for build orchestration) and MSBuild (for build processes) to provide a powerful, enterprise-capable, build automation tool. Team Build 2010 provides a custom Workflow Foundation service host that runs on the build servers that allows the build process to be distributed across multiple machines. The Workflow Foundation based process template can perform any complex branching and custom logic that is supported by Workflow Foundation, including the ability to call MSBuild based project files.

A common companion to build is deployment. In many cases the same script which builds your application is used to deploy it. This is why in this updated book we have a section, Part VII Web Deployment Tool, in which we dedicate three chapters to the topic. MSDeploy is a tool which was first released in 2009. It can be used to deploy websites, and other applications, to local and remote servers. In this section we will show you how to leverage MSDeploy and the Web Publishing Pipeline (WPP) in order to deploy your web applications. Two chapters are devoted to the theory of both MSDeploy and the WPP. There is also a cookbook chapter which shows real world examples of how to use these new technologies. Once you've automated your build and deployment process for the first time you will wonder why you didn't do that for all of your projects.

Who This Book Is For

This book is written for anyone who uses, or is interested in using, MSBuild or Team Build. If you are using Visual Studio to your applications then you are already using MSBuild. *Inside the Microsoft Build Engine* is for all developers and build masters using Microsoft technologies. If you are interested in learning more about how your applications are being built and how you can customize this process then you need this book. If you are using Team Build, or thinking of using it tomorrow, then this book is a must read. It will save you countless hours.

This book will help the needs of enterprise teams as well as individuals. You should be familiar with creating applications using Visual Studio. You are not required to be familiar with the build process, as this book will start from the basics and build on that. Because one of the most effective methods for learning is through examples, this book contains many examples.

Assumptions

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

- You should be an familiar with Visual Studio
- You should have experience with the technologies you are interested in building
- You should have a solid grasp of XML.

Organization of This Book

Inside the Microsoft Build Engine is divided into seven parts:

Part I, "Overview," describes all the fundamentals of creating and extending MSBuild project files. Chapter 1, "MSBuild Quick Start," is a brief chapter to get you started quickly with MSBuild. If you are already familiar with MSBuild then you can skip this chapter; its content

will be covered in more detail within chapters 2 and 3. Chapter 2, "MSBuild Deep Dive, Part 1," discusses such things as static properties, static items, targets, tasks, and msbuild .exe usage. Chapter 3, "MSBuild Deep Dive, Part 2," extends on Chapter 2 with dynamic properties, dynamic items, how properties and items are evaluated, importing external files, extending the build process, property functions, and item functions.

Part II, "Customizing MSBuild," covers the two ways that MSBuild can be extended: custom tasks and custom loggers. Chapter 4, "Custom Tasks," covers all that you need to know to create your own custom MSBuild tasks. Chapter 5, "Custom Loggers," details how to create custom loggers and how to attach them to your build process.

Part III, "Advanced MSBuild Topics," discusses advanced MSBuild concepts. Chapter 6, "Batching and Incremental Builds," covers two very important topics, MSBuild batching and supporting incremental building. Batching is the process of categorizing items and processing them in batches. Incremental building enables MSBuild to detect when a target is up-to-date and can be skipped. Incremental building can drastically reduce build times for most developer builds. Chapter 7, "External Tools," provides some guidelines for integrating external tools into the build process. It also shows how NUnit and FXCop can be integrated in the build process in a reusable fashion.

Part IV, "MSBuild Cookbook," consists of two chapters that are devoted to real-world examples. Chapter 8, "Practical Applications, Part 1," contains several examples, including: setting the assembly version, customizing the build process in build labs, handling errors, and replacing values in configuration files. Chapter 9, "Practical Applications, Part 2," covers more examples, most of which are targeted toward developers who are building Web applications using .NET. It includes Web Deployment Projects, starting and stopping services, zipping output files, compressing Javascript file, and encrypting the web.config file.

Part V, "MSBuild in Visual C++ 2010" discusses how MSBuild powers various features of Visual C++ in light of Visual C++ 2010's switch to MSBuild for its build engine. Chapter 10, "MSBuild in Visual C++ 2010, Part 1" introduces the reader to the new .vcxproj file format for Visual C++ projects and illustrates the Visual C++ build process with a block diagram. Then it continues describing its features such as Build Parallelism, Property Sheets, etc. and how MSBuild enables these features. Of particular interest are the new File Tracker based Incremental Build and movement of Visual C++ Directories settings to a property sheet from the earlier Tools > Option page. Chapter 11, "MSBuild in Visual C++ 2010, Part 1" continues the theme of Chapter 10 by describing more Visual C++ features and the underlying MSBuild implementation. This includes Property Pages, Build Customizations, Platform and Platform Toolsets, project upgrade, etc. It also includes a discussion of all the default tasks, targets and property sheets that are shipped with Visual C++ 2010. Of particular interest is the section on multi-targeting which explains the exciting new feature in Visual C++ 2010 which allows building projects using older toolsets such as Visual C++ 2008 toolset. We describe both how to use this feature as well as how this feature is implemented using

MSBuild. Chapter 12, "Extending Visual C++ 2010" describes how you can extend the build system in various ways by leveraging the underlying MSBuild engine. Discussed in this chapter are authoring Build Events, Custom Build Steps, Custom Build Tool to customize Visual C++ build system in a simple way when the full power of MSBuild extensibility is not needed. This is followed by a discussion of adding a custom target and creating a Build Customization which allows you to use the full set of extensibility features offered by MSBuild. One of the important topics in this chapter deals with adding support for a new Platform or a Platform Toolset. The example of using the popular GCC toolset to build Visual C++ projects is used to drive home the point that extending platforms and platform toolsets is easy and natural in Visual C++ 2010.

Part VI, "Team Foundation Build," introduces Team Foundation Build (Team Build) in Chapter 13, "Team Build Quick Start". In this chapter we discuss the architectural components of Team Foundation Build and walkthrough the installation process and the basics of configuring it. In Chapter 14, "Team Build Deep Dive", we examine the process templates that ship with Team Build as well the Team Build API. Chapter 15, "Workflow Foundation Quick Start", introduces the basics of Workflow Foundation to enable customizing the build process. Chapter 16, "Process Template Customization", then leverages this knowledge and explains how to create customized build processes.

Part VII, "Web Deployment Tool" first introduces the Web Deployment Tool (MSDeploy) in Chapter 17 "Web Deployment Tool, Part 1". In that chapter we discuss what MSDeploy is, and how it can be used. We describe how MSDeploy can be used for "online deployment" in which you deploy your application to the target in real time and we discuss "offline deployments" in which you create a package which gets handed off to someone else for the actual deployment. In Chapter 18 "Web Deployment Tool, Part 2" we introduce the Web Publishing Pipeline (WPP). The WPP is the process which your web application follows to go from build output to being deployed on your remote server. It's all captured in a few MSBuild scripts, so it is very customizable and extensible. In that chapter we cover how you can customize and extend the WPP to suit your needs. Then in Chapter 19 "Web Deploy Practical Applications" we show many different examples of how you can use MSDeploy and WPP to deploy your packages. We cover such things as Publishing using MSBuild, parameterizing packages, deploying with Team Build, and a few others.

For Appendices A, B, and C please go to *http://aka.ms/645240/files*.

System Requirements

The following list contains the minimum hardware and software requirements to run the code samples provided with the book.

- .NET 4.0 Framework
- Visual Studio 2010 Express Edition or greater
- 50 MB of available space on the installation drive

For Team Build chapters:

- Visual Studio 2010 Professional
- Some functionality (such as Code Analysis) requires Visual Studio 2010 Premium or Visual Studio 2010 Ultimate
- Access to a server running Team Foundation Server 2010
- Access to a build machine running Team Foundation Build 2010 (Chapter 13 walks you through installing this)
- A trial Virtual PC with Microsoft Visual Studio 2010 and Team Foundation Server 2010 RTM is available from *http://www.microsoft.com/downloads/en/details.aspx?FamilyID=509c3ba1-4efc-42b5-b6d8-0232b2cbb26e*

Code Samples

Download the sample code files from this book's page online:

http://aka.ms/645240/files

Acknowledgements

The authors are happy to share the following acknowledgments.

Sayed Ibrahim Hashimi

Before I wrote my first book I thought that writing a book involved just a few people, but now having written my third book I realize how many different people it takes to successfully launch a book. Unfortunately with books most of the credit goes to the authors, but the others involved deserve much more credit than they are naturally given. As an author, the most we can do is thank them and mention their names here in the acknowledgements section. When I reflect on the writing of this book there are a lot of names, but there is one that stands out in particular, Dan Moseley. Dan is a part of the MSBuild team. He has gone way above and beyond what I could have ever imagined. I've never seen someone peer review a chapter as good, or as fast, as Dan has. Without Dan's invaluable insight the book would simply not be what it is today. In my whole career I've only encountered a few people who are as passionate about what they do as Dan. I hope that I can be as passionate about building products as he is.

Besides Dan I would like to first thank my co-authors and technical editor. William Bartholomew, who wrote the Team Build chapters, is a wonderful guy to work with. He is recognized as a Team Build expert, and I think his depth of knowledge shows in his work. Pavan Adharapurapu wrote the chapters covering Visual C++. When we first started talking about updating the book to cover MSBuild 4.0 to be honest I was a bit nervous. I was nervous because I had not written any un-managed code in more than 5 years, and because of that I knew that I could not write the content on Visual C++ and do it justice. Then we found Pavan. Pavan helped build the Visual C++ project system, and he pours his heart into everything that he does. Looking back I am confident that he was the best person to write those chapters and I am thankful that he was willing. Also I'd like to thank Jason Ward, who wrote a chapter on Workflow Foundation. Jason who has a great background in Workflow Foundation as well as Team Build was an excellent candidate to write that chapter. I started with the authors, but the technical editor, Marc Young deserves the same level of recognition. This having been my third book I was familiar with what a technical editor is responsible for doing. Their primary job is essentially to point out the fact that I don't know what I'm talking about, which Marc did very well. But Marc went beyond his responsibilities. Marc was the one who suggested that we organize all the sample code based on the chapters. At first I didn't really think it was a good idea, but he volunteered to reorganize the content and even redo a bunch of screen shots. I really don't think he knew what he was volunteering for! Now that it is over I wonder if he would volunteer again. I can honestly say that Marc was the best technical editor that I've ever worked with. His attention to detail is incredible, to the point that he was reverse engineering the code to validate some statements that I was making (and some were wrong). Before this book I knew what a technical editor was supposed to be, and now I know what a technical editor can be. Thanks to all of you guys!

As I mentioned at the beginning of this acknowledgement there are many others who came together to help complete this book besides those of us writing it. I'd like to thank Microsoft Press and everyone there who worked on it. I know there were some that were involved that I didn't even know of. I'd like to thank those that I do know of by name. Devon Musgrave, who also worked with us on the first edition, is a great guy to work with. This book really started with him. We were having dinner one night a while back and he said to me something along the lines of "what do you think of updating the book?" I knew that it would be a wonderful project and it was. Iram Nawaz who was the Project Editor of the book was just fantastic. She made sure that we stayed on schedule (sorry for the times I was late ☺) and was a great person to work with. The book wouldn't have made it on time if it was not for her. Along with these guys from Microsoft Press I would like to than the editors; Susan McClung and Nicole Schlutt for their perseverance to correct my bad writing.

There are several people who work on either the MSBuild/MSDeploy/Visual Studio product groups that I would like to thank as well. When the guys who built the technologies you are writing about help you, it brings the book to a whole new level. I would like to thank the following people for giving their valued assistance (in no particular order, and sorry if

I missed anyone); Jay Shrestha, Chris Mann, Andrew Arnott, Vishal Joshi, Bilal Aslam, Faith Allington, Ming Chen, Joe Davis and Owais Shaikh.

William Bartholomew

Firstly I'd like to thank my co-authors, Sayed, Pavan, and Jason, because without their contributions this book would not be as broad as it is. From Microsoft Press I'd like to thank Devon Musgrave, Ben Ryan, Iram Nawaz, Susan McClung, and the art team, for their efforts 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 Aaron Hallberg and Buck Hodges, for the tireless support.

Pavan Adharapurapu

A large number of people helped make this book happen. I would like to start off by thanking Dan Moseley, my manager at Microsoft who encouraged me to write the book and for providing thorough and detailed feedback for the chapters that I wrote. Brian Tyler, the architect of my team provided encouragement and great feedback. Many people from the Visual C and the project system teams here at Microsoft helped make the book a better one by providing feedback on their areas of expertise. In alphabetical order they are: Olga Arkhipova, Andrew Arnott, Ilya Biryukov, Felix Huang, Cliff Hudson, Renin John, Sara Joiner, Marian Luparu, Chris Mann, Bogdan Mihalcea, Kieran Mockford, Amit Mohindra, Li Shao. Any mistakes that remain are mine.

I would like to thank Devon Musgrave, Iram Nawaz, Susan McClung and Marc Young from Microsoft Press for their guidance and patience.

Finally, I would like to thank my wonderful wife Deepti who provided great support and understanding throughout the many weekends I spent locked up writing and revising the book. Deepti, I promise to make it up to you.

Jason Ward

First of all, I'd like to thank William Bartholomew for giving me the opportunity to contribute to this book. William displays an amazing amount of talent, passion and integrity in all his work. I'm honored to have his friendship as well as the opportunity to work with him on a daily basis.

I'd also like to thank Avi Pilosof and Rich Lowry for giving me the wonderful opportunity to work at Microsoft. From the moment I met them it was clear that moving my family half way around the world was the right thing to do. Their mentorship, passion, friendship

and overarching goal of 'doing the right thing' has only further reinforced that working at Microsoft was everything I had hoped it would be. They are the embodiment of all things good at Microsoft.

Finally I'd like to thank the thousands of people working at Microsoft for producing the wonderful applications and experiences that millions of people around the world use and enjoy on a daily basis. It is truly an honor to work with you as we change the world.

Errata and Book Support

We've made every effort to ensure the accuracy of this book and its companion content. If you do find an error, please report it on our Microsoft Press site:

1. Go to *www.microsoftpressstore.com*.
2. In the Search box, enter the book's ISBN or title.
3. Select your book from the search results.
4. On your book's catalog page, find the Errata & Updates tab

You'll find additional information and services for your book on its catalog page. If you need additional support, please e-mail 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://www.microsoft.com/learning/booksurvey

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*

Part I

Overview

In this part:

Chapter 1

MSBuild Quick Start

When you are learning a new subject, it's exciting to just dive right in and get your hands dirty. The purpose of this chapter is to enable you to do just that. I'll describe all the key elements you need to know to get started using MSBuild. If you're already familiar with MSBuild, feel free to skip this chapter—all of the material presented here will be covered in later areas in the book as well, with the exception of the msbuild.exe usage details.

The topics covered in this chapter include the structure of an MSBuild file, properties, targets, items, and invoking MSBuild. Let's get started.

Project File Details

An MSBuild file—typically called an "MSBuild project file"—is just an XML file. These XML files are described by two XML Schema Definition (XSD) documents that are created by Microsoft: Microsoft.Build.Commontypes.xsd and Microsoft.Build.Core.xsd. These files are located in the %WINDIR%\Microsoft.NET\Framework*vNNNN*\MSBuild folder, where *vNNNN* is the version folder for the Microsoft .NET Framework 2.0, 3.5, or 4.0. If you have a 64-bit machine, then you will find those files in the Framework64 folder as well. (In this book, I'll assume you are using .NET Framework 4.0 unless otherwise specified. As a side note, a new version of MSBuild was not shipped with .NET Framework 3.0.) Microsoft .Build.Commontypes.xsd describes the elements commonly found in Microsoft Visual Studio-generated project files, and Microsoft.Build.Core.xsd describes all the fixed elements in an MSBuild project file. The simplest MSBuild file would contain the following:

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

This XML fragment will identify that this is an MSBuild file. All your content will be placed inside the Project element. Specifically, we will be declaring *properties*, *items*, *targets*, and a few other things directly under the Project element. When building software applications, you will always need to know two pieces of information: what is being built and what build parameters are being used. Typically, files are being built, and these would be contained in MSBuild items. Build parameters, like Configuration or OutputPath, are contained in MSBuild properties. We'll now discuss how to declare properties as well as targets, and following that we'll discuss items.

Properties and Targets

MSBuild properties are simply key-value pairs. The key for the property is the name that you will use to refer to the property. The value is its value. When you declare static properties, they are always contained in a *PropertyGroup* element, which occurs directly within the *Project* element. We will discuss dynamic properties (those declared and generated dynamically inside targets) in the next chapter. The following snippet is a simple example of declaring static properties:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <AppServer>\\sayedApp</AppServer>
    <WebServer>\\sayedWeb</WebServer>
  </PropertyGroup>
</Project>
```

As previously stated, the *PropertyGroup* element, inside the *Project* element, will contain all of our properties. The name of a property is the XML tag name of the element, and the value of the property is the value inside the element. In this example, we have declared two properties, AppServer and WebServer, with the values \\sayedApp and \\sayedWeb, respectively. You can create as many *PropertyGroup* elements under the Project tag as you want. The previous fragment could have been defined like this:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <AppServer>\\sayedApp</AppServer>
  </PropertyGroup>
  <PropertyGroup>
    <WebServer>\\sayedWeb</WebServer>
  </PropertyGroup>
</Project>
```

The MSBuild engine will process all elements sequentially within each *PropertyGroup* in the same manner. If you take a look at a project created by Visual Studio, you'll notice that many properties are declared. These properties have values that will be used throughout the build process for that project. Here is a region from a sample project that I created:

```
<Project DefaultTargets="Build"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
    <ProductVersion>8.0.50727</ProductVersion>
    <SchemaVersion>2.0</SchemaVersion>
    <ProjectGuid>{A71540FD-9949-4AC4-9927-A66B84F97769}</ProjectGuid>
    <OutputType>WinExe</OutputType>
    <AppDesignerFolder>Properties</AppDesignerFolder>
    <RootNamespace>WindowsApplication1</RootNamespace>
    <AssemblyName>WindowsApplication1</AssemblyName>
  </PropertyGroup>
```

```
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
    <DebugSymbols>true</DebugSymbols>
    <DebugType>full</DebugType>
    <Optimize>false</Optimize>
    <OutputPath>bin\Debug\</OutputPath>
    <DefineConstants>DEBUG;TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
  </PropertyGroup>
 ....
</Project>
```

You can see that values for the output type, the name of the assembly, and many others are defined in properties. Defining properties is great, but we also need to be able to utilize them, which is performed inside targets. We will move on to discuss Target declarations.

MSBuild fundamentally has two execution elements: tasks and targets. A task is the smallest unit of work in an MSBuild file, and a target is a sequential set of tasks. A task must always be contained within a target. Here's a sample that shows you the simplest MSBuild file that contains a target:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="HelloWorld">
  </Target>
</Project>
```

In this sample, we have created a new target named HelloWorld, but it doesn't perform any work at this point because it is empty. When MSBuild is installed, you are given many tasks out of the box, such as Copy, Move, Exec, ResGen, and Csc. You can find a list of these tasks at the MSBuild Task Reference (*http://msdn2.microsoft.com/en-us/library/7z253716.aspx*). We will now use the Message task. This task is used to send a message to the logger(s) that are listening to the build process. In many cases this means a message is sent to the console executing the build. When you invoke a task in an MSBuild file, you can pass its input parameters by inserting XML attributes with values. These attributes will vary from task to task depending on what inputs the task is able to accept. From the documentation of the Message task (*http://msdn2.microsoft.com/en-us/library/6yy0yx8d.aspx*) you can see that it accepts a string parameter named Text. The following snippet shows you how to use the Message task to send the classic message "Hello world!"

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="HelloWorld">
    <Message Text="Hello world!" />
  </Target>
</Project>
```

Now we will verify that this works as expected. To do this, place the previous snippet into a file named HelloWorld.proj. Now open a Visual Studio command prompt, found in the Visual Studio Tools folder in the Start menu for Visual Studio. When you open this prompt,

the path to msbuild.exe is already on the path. The command you will be invoking to start MSBuild is msbuild.exe. The basic usage for the command is as follows:

```
msbuild [INPUT_FILE] /t:[TARGETS_TO_EXECUTE]
```

So the command in our case would be

```
msbuild HelloWorld.proj /t:HelloWorld
```

This command says to execute the HelloWorld target, which is contained in the HelloWorld .proj file. The result of this invocation is shown in Figure 1-1.

```
C:\InsideMSBuild\Ch01>msbuild HelloWorld.proj /nologo
Build started 9/24/2010 5:55:31 PM.
Project "C:\InsideMSBuild\Ch01\HelloWorld.proj" on node 1 (default targets).
HelloWorld:
  Hello world!
Done Building Project "C:\InsideMSBuild\Ch01\HelloWorld.proj" (default targets).

Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 1-1 Result of HelloWorld target

Note In this example, as well as all others in the book, we specify the /nologo switch. This simply avoids printing the MSBuild version information to the console and saves space in the book. Feel free to use it or not as you see fit.

We can see that the HelloWorld target is executed and that the message "Hello world!" is displayed on the console. The Message task also accepts another parameter, Importance. The possible values for this parameter are high, normal, or low. The Importance value may affect how the loggers interpret the purpose of the message. If you want the message logged no matter the verbosity, use the *high* importance level. We're discussing properties, so let's take a look at how we can specify the text using a property. I've extended the HelloWorld.proj file to include a few new items. The contents are shown here:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="HelloWorld">
    <Message Text="Hello world!" />
  </Target>

  <PropertyGroup>
    <HelloMessage>Hello from property</HelloMessage>
  </PropertyGroup>
  <Target Name="HelloProperty">
    <Message Text="$(HelloMessage)" />
  </Target>
</Project>
```

I have added a new property, HelloMessage, with the value "Hello from property", as well as a new target, HelloProperty. The HelloProperty target passes the value of the property using

the $(*PropertyName*) syntax. This is the syntax you use to evaluate a property. We can see this in action by executing the command `msbuild HelloWorld.proj /t:HelloProperty`. The result is shown in Figure 1-2.

```
C:\InsideMSBuild\Ch01>msbuild HelloWorld.proj /t:HelloProperty /nologo
Build started 9/24/2010 5:59:26 PM.
Project "C:\InsideMSBuild\Ch01\HelloWorld.proj" on node 1 (HelloProperty target(s)).
HelloProperty:
  Hello from property
Done Building Project "C:\InsideMSBuild\Ch01\HelloWorld.proj" (HelloProperty target(s)).

Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 1-2 Result of HelloProperty target

As you can see, the value of the property was successfully passed to the Message task. Now that we have discussed targets and basic property usage, let's move on to discuss how we can declare properties whose values are derived from other properties.

To see how to declare a property by using the value of an existing property, take a look at the project file, NestedProperties.proj:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
    <DropLocation>
      \\sayedData\MSBuildExamples\Drops\$(Configuration)\$(Platform)\
    </DropLocation>
  </PropertyGroup>
  <Target Name="PrepareFilesForDrop">
    <Message Text="DropLocation : $(DropLocation)" />
  </Target>
</Project>
```

We can see here that three properties have been declared. On both the Configuration and Platform properties, a *Condition* attribute appears. We'll discuss this attribute later in this chapter. The remaining property, DropLocation, is defined using the values of the two previously declared items. The DropLocation property has three components: a constant value and two values that are derived from the Configuration and Platform properties. When the MSBuild engine sees the $(*PropertyName*) notation, it will replace that with the value of the specified property. So the evaluated value for DropLocation would be \\sayedData\MSBuildExamples\Drops\Debug\AnyCPU\. You can verify that by executing the PrepareFilesForDrop target with msbuild.exe. The reference for properties can be found at *http://msdn.microsoft.com/en-us/library/ms171458.aspx*.

When you use MSBuild, a handful of properties are available to you out of the box that cannot be modified. These are known as reserved properties. Table 1-1 contains all the reserved properties.

TABLE 1-1 Reserved Properties

Name	Description
MSBuildExtensionsPath	The full path where MSBuild extensions are located. By default, this is stored under %programfiles%\msbuild.
MSBuildExtensionsPath32	The full path where MSBuild 32-bit extensions are located. This typically is located under the Program Files folder. For 32-bit machines, this value will be the same as MSBuildExtensionsPath.
MSBuildExtensionsPath64*	The full path where MSBuild 64-bit extensions are located. This typically is under the Program Files folder. For 32-bit machines, this value will be empty.
MSBuildLastTaskResult*	This value holds the return value from the previous task. It will be *true* if the task completed successfully, and *false* otherwise.
MSBuildNodeCount	The number of nodes (processes) that are being used to build the projects. If the /m switch is not used, then this value will be 1.
MSBuildProgramFiles32*	This points to the 32-bit Program Files folder.
MSBuildProjectDefaultTargets	Contains the list of the default targets.
MSBuildProjectDirectory	The full path to the directory where the project file is located.
MSBuildProjectDirectoryNoRoot	The full path to the directory where the project file is located, excluding the root directory.
MSBuildProjectExtension	The extension of the project file, including the period.
MSBuildProjectFile	The name of the project file, including the extension.
MSBuildProjectFullPath	The full path to the project file.
MSBuildProjectName	The name of the project file, without the extension.
MSBuildStartupDirectory	The full path to the folder where the MSBuild process is invoked.
MSBuildThisFile*	The name of the file, including the extension but excluding the path, which contains the target that is currently executing.
MSBuildThisFileDirectory*	This is the full path to the directory that contains the file that is currently being executed.
MSBuildThisFileDirectoryNoRoot*	The same as MSBuildThisFileDirectory, except with the root removed.
MSBuildThisFileExtension*	The extension of the file that is currently executing.
MSBuildThisFileFullPath*	The full path to the file that is currently executing.
MSBuildThisFileName*	The name of the file, excluding the extension and path, of the currently executing file.
MSBuildToolsPath (MSBuildBinPath)	The full path to the location where the MSBuild binaries are located. For MSBuild 2.0, this property is named MSBuildBinPath; in MSBuild 3.5, it is deprecated.
MSBuildToolsVersion	The version of the tools being used to build the project. Possible values include 2.0, 3.5, and 4.0. The default value for this is 2.0.

* Denotes parameters new with MSBuild 4.0.

You would use these properties just as you would properties that you have declared in your own project file. To see an example of this, look at any Visual Studio–generated project file. When you create a new C# project, you will find the import statement `<Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />` located near the bottom. This import statement uses the MSBuildToolsPath reserved property to resolve the full path to the Microsoft.CSharp.targets file and insert its content at this location. This is the file that drives the build process for C# projects. We will discuss its content throughout the remainder of this book. In Chapter 3, "MSBuild Deep Dive, Part 2," we discuss specifically how the Import statement is processed.

Items

Building applications usually means dealing with many files. Because of this, you use a specific construct when referencing files in MSBuild: items. Items are usually file-based references, but they can be used for other purposes as well. If you create a project using Visual Studio, you may notice that you see many *ItemGroup* elements as well as *PropertyGroup* elements. The *ItemGroup* element contains all the statically defined items. Static item definitions are those declared as a direct child of the *Project* element. Dynamic items, which we discuss in the next chapter, are those defined inside a target. When you define a property, you are declaring a key-value pair, which is a one-to-one relationship. When you declare items, one item can contain a list of many values. In terms of code, a property is analogous to a variable and an item to an array. Take a look at how an item is declared in the following snippet taken from the ItemsSimple.proj file:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <SolutionFile Include="..\InsideMSBuild.sln" />
  </ItemGroup>
  <Target Name="PrintSolutionInfo">
    <Message Text="SolutionFile: @(SolutionFile)" />
  </Target>
</Project>
```

In this file, there is an *ItemGroup* that has a subelement, *SolutionFile*. *ItemGroup* is the element type that all statically declared items must be placed within. The name of the subelement, *SolutionFile* in this case, is actually the item type of the item that is created. The *SolutionFile* element has an attribute, Include. This determines what values the item contains. Relating it back to an array, *SolutionFile* is the name of the variable that references the array, and the *Include* attribute is used to populate the array's values. The *Include* attribute can contain the following types of values (or any combination thereof): one distinct value, a list of values delimited with semicolons, or a value using wildcards. In this sample, the *Include* attribute contains one value. When you need to evaluate the contents of an item, you would use the @(*ItemType*) syntax. This is similar to the $(*PropertyName*) syntax for properties. To see this in action, take a look at the PrintSolutionInfo target. This target

passes the value of the item into the Message task to be printed to the console. You can see the result of executing this target in Figure 1-3.

```
C:\InsideMSBuild\Ch01>msbuild ItemsSimple.proj /t:PrintSolutionInfo /nologo
Build started 9/24/2010 6:04:18 PM.
Project "C:\InsideMSBuild\Ch01\ItemsSimple.proj" on node 1 (PrintSolutionInfo target(s)).
PrintSolutionInfo:
  SolutionFile: ..\InsideMSBuild.sln
Done Building Project "C:\InsideMSBuild\Ch01\ItemsSimple.proj" (PrintSolutionInfo target(s)).


Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 1-3 PrintSolutionInfo result

In this case, the item *SolutionFile* contains a single value, so it doesn't seem very different from a property because the single value was simply passed to the Message task. Let's take a look at an item with more than one value. This is an extended version of the ItemsSimple .proj file shown earlier:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <SolutionFile Include="..\InsideMSBuild.sln" />
  </ItemGroup>
  <Target Name="PrintSolutionInfo">
    <Message Text="SolutionFile: @(SolutionFile)" />
  </Target>

  <ItemGroup>
    <Compile
     Include="Form1.cs;Form1.Designer.cs;Program.cs;Properties\AssemblyInfo.cs" />
  </ItemGroup>
  <Target Name="PrintCompileInfo">
    <Message Text="Compile: @(Compile)" />
  </Target>
</Project>
```

In the modified version, I have created a new item, Compile, which includes four values that are separated by semicolons. The PrintCompileInfo target passes these values to the Message task. When you invoke the PrintCompileInfo target on the MSBuild file just shown, the result will be `Compile: Form1.cs;Form1.Designer.cs;Program.cs;Properties \AssemblyInfo.cs`. It may look like the Message task simply took the value in the Include attribute and passed it to the Message task, but this is not the case. The Message task has a single input parameter, Text, as discussed earlier. This parameter is a string property. Because an item is a multivalued object, it cannot be passed directly into the Text property. It first has to be converted into a string. MSBuild does this for you by separating each value with a semicolon. In Chapter 2, I will discuss how you can customize this conversion process.

An item definition doesn't have to be defined entirely by a single element. It can span multiple elements. For example, the Compile item shown earlier could have been declared like this:

```
<ItemGroup>
    <Compile Include="Form1.cs" />
```

```
    <Compile Include="Form1.Designer.cs" />
    <Compile Include="Program.cs" />
    <Compile Include="Properties\AssemblyInfo.cs" />
  </ItemGroup>
```

In this version, each file is placed into the Compile item individually. These Compile elements could also have been contained in their own *ItemGroup* as well, as shown in the next snippet.

```
<ItemGroup>
  <Compile Include="Form1.cs" />
</ItemGroup>
<ItemGroup>
  <Compile Include="Form1.Designer.cs" />
</ItemGroup>
<ItemGroup>
  <Compile Include="Program.cs" />
</ItemGroup>
<ItemGroup>
  <Compile Include="Properties\AssemblyInfo.cs" />
</ItemGroup>
```

The end result of these declarations would all be the same. You should note that an item is an ordered list, so the order in which values are added to the item is preserved and may in some context affect behavior based on usage. When a property declaration appears after a previous one, the previous value is overwritten. Items act differently from this in that the value of the item is simply appended to instead of being overwritten. We've now discussed two of the three ways to create items. Let's look at using wildcards to create items.

Many times, items refer to existing files. If this is the case, you can use wildcards to automatically include files that meet the constraints of the wildcards. You can use three wildcard elements with MSBuild: ?, *, and **. The ? descriptor is used to denote that exactly one character can take its place. For example, the include declaration of `b?t.cs` could include values such as bat.cs, bot.cs, bet.cs, b1t.cs, and so on. The * descriptor can be replaced with zero or more characters (not including slashes), so the declaration `b*t.cs` could include values such as bat.cs, bot.cs, best.cs, bt.cs, etc. The ** descriptor tells MSBuild to search directories recursively for the pattern. In effect, "*" matches any characters except for "/" while "**" matches any characters, including "/". For example, `Include="src\**\*.cs"` would include all files under the src folder (including subfolders) with the .cs extension.

Item Metadata

Another difference between properties and items is that items can have metadata associated with them. When you create an item, each of its elements is a full-fledged .NET object, which can have a set of values (metadata) associated with it. The metadata that is available on every item, which is called *well-known metadata,* is summarized in Table 1-2.

TABLE 1-2 Well-Known Metadata

Name	Description
Identity	The value that was specified in the Include attribute of the item after it was evaluated.
FullPath	Full path of the file.
RootDir	The root directory to which the file belongs, such as C:\.
Filename	The name of the file, not including the extension.
Extension	The extension of the file, including the period.
RelativeDir	Contains the path specified in the *Include* attribute, up to the final backslash (\).
Directory	Directory of the item, without the root directory.
RecursiveDir	This is the expanded directory path starting from the first ** of the include declaration. If no ** is present, then this value is empty. If multiple ** are present, then RecursiveDir will be the expanded value starting from the first **. This may sound peculiar, but it is what makes recursive copying possible.
ModifiedTime	The last time the file was modified.
CreatedTime	The time the file was created.
AccessedTime	The last time the file was accessed.

To access metadata values, you have to use this syntax:

```
@(ItemType->'%(MetadataName)')
```

ItemType is the name of the item, and MetadataName is the name of the metadata that you are accessing. This is the most basic syntax. To examine what types of values the well-known metadata returns, take a look at the file, WellKnownMetadata.proj, shown here:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         ToolsVersion="4.0">
  <ItemGroup>
    <src Include="src\one.txt" />
  </ItemGroup>
  <Target Name="PrintWellKnownMetadata">

    <Message Text="===== Well known metadata ====="/>
    <!-- %40 = @ -->
    <!-- %25 = % -->
    <Message Text="%40(src->'%25(FullPath)'): @(src->'%(FullPath)')"/>
    <Message Text="%40(src->'%25(RootDir)'): @(src->'%(RootDir)')"/>
    <Message Text="%40(src->'%25(Filename)'): @(src->'%(Filename)')"/>
    <Message Text="%40(src->'%25(Extension)'): @(src->'%(Extension)')"/>
    <Message Text="%40(src->'%25(RelativeDir)'): @(src->'%(RelativeDir)')"/>
    <Message Text="%40(src->'%25(Directory)'): @(src->'%(Directory)')"/>
    <Message Text="%40(src->'%25(RecursiveDir)'): @(src->'%(RecursiveDir)')"/>
    <Message Text="%40(src->'%25(Identity)'): @(src->'%(Identity)')"/>
    <Message Text="%40(src->'%25(ModifiedTime)'): @(src->'%(ModifiedTime)')"/>
    <Message Text="%40(src->'%25(CreatedTime)'): @(src->'%(CreatedTime)')"/>
    <Message Text="%40(src->'%25(AccessedTime)'): @(src->'%(AccessedTime)')"/>

  </Target>
</Project>
```

Note In order to use reserved characters, such as the % and @, you have to escape them. This is accomplished by the syntax %HV, where *HV* is the hex value of the character. This is demonstrated here with %25 and %40.

Note In this example, we have specified the ToolsVersion value to be 4.0. This determines which version of the MSBuild tools will be used. Although not needed for this sample, we will be specifying this version number from this point forward. The default value is 2.0.

This MSBuild file prints the values for the well-known metadata for the src item. The result of executing the PrintWellKnownMetadata target is shown in Figure 1-4.

```
C:\InsideMSBuild\Ch01>msbuild WellKnownMetadata.proj /t:PrintWellKnownMetadata /nologo
Build started 9/24/2010 6:10:01 PM.
Project "C:\InsideMSBuild\Ch01\WellKnownMetadata.proj" on node 1 (PrintWellKnownMetadata target(s)
).
PrintWellKnownMetadata:
  ===== Well known metadata =====
  @(src->'%(FullPath)'): C:\InsideMSBuild\Ch01\src\one.txt
  @(src->'%(Rootdir)'): C:\
  @(src->'%(Filename)'): one
  @(src->'%(Extension)'): .txt
  @(src->'%(RelativeDir)'): src\
  @(src->'%(Directory)'): InsideMSBuild\Ch01\src\
  @(src->'%(RecursiveDir)'):
  @(src->'%(Identity)'): src\one.txt
  @(src->'%(ModifiedTime)'): 2010-09-08 22:15:12.4218750
  @(src->'%(CreatedTime)'): 2010-09-08 22:15:12.4218750
  @(src->'%(AccessedTime)'): 2010-09-08 22:15:12.4218750
Done Building Project "C:\InsideMSBuild\Ch01\WellKnownMetadata.proj" (PrintWellKnownMetadata targe
t(s)).

Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 1-4 PrintWellKnownMetadata result

The figure gives you a better understanding of the well-known metadata's usage. Keep in mind that this demonstrates the usage of metadata in the case where the item contains only a single value.

To see how things change when an item contains more than one value, let's examine MetadataExample01.proj:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         ToolsVersion="4.0">
  <ItemGroup>
    <Compile Include="*.cs" />
  </ItemGroup>

  <Target Name="PrintCompileInfo">
    <Message Text="Compile fullpath: @(Compile->'%(FullPath)')" />
  </Target>
</Project>
```

In this project file we simply evaluate the FullPath metadata on the Compile item. From the examples with this text, the directory containing this example contains four files: Class1.cs, Class2.cs, Class3.c, and Class4.cs. These are the files that will be contained in the Compile item. Take a look at the result of the PrintCompileInfo target in Figure 1-5.

```
C:\InsideMSBuild\Ch01>msbuild MetadataExample01.proj /t:PrintCompileInfo /nologo
Build started 9/24/2010 6:18:39 PM.
Project "C:\InsideMSBuild\Ch01\MetadataExample01.proj" on node 1 (PrintCompileInfo target(s)).
PrintCompileInfo:
  Compile fullpath: C:\InsideMSBuild\Ch01\Class1.cs;C:\InsideMSBuild\Ch01\Class2.cs;C:\InsideMSBui
  ld\Ch01\Class3.cs;C:\InsideMSBuild\Ch01\Class4.cs
Done Building Project "C:\InsideMSBuild\Ch01\MetadataExample01.proj" (PrintCompileInfo target(s)).


Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 1-5 PrintCompileInfo result

You have to look carefully at this output to decipher the result. What is happening here is that a single string is created by combining the full path of each file, separated by a semicolon. The `@(ItemType->'...%()...')` syntax is an "Item Transformation." We will cover transformations in greater detail in Chapter 2. In the next section, we'll discuss conditions. Before we do that, take a minute to look at the project file for a simple Windows application that was generated by Visual Studio. You should recognize many things.

```
<Project DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003" ToolsVersion="4.0">
  <PropertyGroup>
    <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
    <ProductVersion>8.0.50727</ProductVersion>
    <SchemaVersion>2.0</SchemaVersion>
    <ProjectGuid>{0F34CE5D-2AB0-49A9-8254-B21D1D2EFFA1}</ProjectGuid>
    <OutputType>WinExe</OutputType>
    <AppDesignerFolder>Properties</AppDesignerFolder>
    <RootNamespace>WindowsApplication1</RootNamespace>
    <AssemblyName>WindowsApplication1</AssemblyName>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
    <DebugSymbols>true</DebugSymbols>
    <DebugType>full</DebugType>
    <Optimize>false</Optimize>
    <OutputPath>bin\Debug\</OutputPath>
    <DefineConstants>DEBUG;TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">
    <DebugType>pdbonly</DebugType>
    <Optimize>true</Optimize>
    <OutputPath>bin\Release\</OutputPath>
    <DefineConstants>TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="System.Data" />
    <Reference Include="System.Deployment" />
    <Reference Include="System.Drawing" />
    <Reference Include="System.Windows.Forms" />
    <Reference Include="System.Xml" />
  </ItemGroup>
```

```
  <ItemGroup>
    <Compile Include="Form1.cs">
      <SubType>Form</SubType>
    </Compile>
    <Compile Include="Form1.Designer.cs">
      <DependentUpon>Form1.cs</DependentUpon>
    </Compile>
    <Compile Include="Program.cs" />
    <Compile Include="Properties\AssemblyInfo.cs" />
    <EmbeddedResource Include="Properties\Resources.resx">
      <Generator>ResXFileCodeGenerator</Generator>
      <LastGenOutput>Resources.Designer.cs</LastGenOutput>
      <SubType>Designer</SubType>
    </EmbeddedResource>
    <Compile Include="Properties\Resources.Designer.cs">
      <AutoGen>True</AutoGen>
      <DependentUpon>Resources.resx</DependentUpon>
    </Compile>
    <None Include="Properties\Settings.settings">
      <Generator>SettingsSingleFileGenerator</Generator>
      <LastGenOutput>Settings.Designer.cs</LastGenOutput>
    </None>
    <Compile Include="Properties\Settings.Designer.cs">
      <AutoGen>True</AutoGen>
      <DependentUpon>Settings.settings</DependentUpon>
      <DesignTimeSharedInput>True</DesignTimeSharedInput>
    </Compile>
  </ItemGroup>
  <Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
  <!-- To modify your build process, add your task
inside one of the targets below and uncomment it.
       Other similar extension points exist,
see Microsoft.Common.targets.
  <Target Name="BeforeBuild">
  </Target>
  <Target Name="AfterBuild">
  </Target>
  -->
</Project>
```

Simple Conditions

When you are building, you often have to make decisions based on conditions. MSBuild allows almost every XML element to contain a conditional statement within it. The statement would be declared in the *Condition* attribute. If this attribute evaluates to *false,* then the element and all its child elements are ignored. In the sample Visual Studio project that was shown at the end of the previous section, you will find the statement `<Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>`. In this declaration, the condition is checking to see if the property is empty. If so, then it will be defined; otherwise, the statement will be skipped. This is a method to provide a default overridable value for a property. Table 1-3 describes a few common types of conditional operators.

TABLE 1-3 **Simple Conditional Operators**

Symbol	Description
==	Checks for equality; returns *true* if both have the same value.
!=	Checks for inequality; returns *true* if both do not have the same value.
Exists	Checks for the existence of a file. Returns *true* if the provided file exists.
!Exists	Checks for the nonexistence of a file. Returns *true* if the file provided is not found.

Because you can add a conditional attribute to any MSBuild element (excluding the Otherwise element), this means that we can decide to include entries in items as necessary. For example, when building ASP.NET applications, in some scenarios, you might want to include files that will assist debugging. Take a look at the MSBuild file, ConditionExample01.proj:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         ToolsVersion="4.0">
  <PropertyGroup>
    <Configuration>Release</Configuration>
  </PropertyGroup>
  <ItemGroup>
    <Content Include="script.js"/>
    <Content Include="script.debug.js" Condition="$(Configuration)=='Debug'" />
  </ItemGroup>

  <Target Name="PrintContent">
    <Message Text="Configuration: $(Configuration)" />
    <Message Text="Content: @(Content)" />
  </Target>
</Project>
```

If we execute the command `msbuild ConditionExample01.proj /t:PrintContent`, the result would be what is shown in Figure 1-6.

```
C:\InsideMSBuild\Ch01>msbuild ConditionExample01.proj /t:PrintContent /nologo
Build started 9/24/2010 6:24:55 PM.
Project "C:\InsideMSBuild\Ch01\ConditionExample01.proj" on node 1 (PrintContent target(s)).
PrintContent:
  Configuration: Release
  Content: script.js
Done Building Project "C:\InsideMSBuild\Ch01\ConditionExample01.proj" (PrintContent target(s)).


Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 1-6 PrintContent target result

As you can see, because the Configuration value was not set to Debug, the script.debug.js file was not included in the Content item. Now we will examine the usage of the *Exists* function. To do this, take a look at the target _CheckForCompileOutputs, taken from the Microsoft .Common.targets file, a file included with MSBuild that contains most of the rules for building VB and C# projects:

```
<Target
    Name="_CheckForCompileOutputs">
```

```
    <!--Record the main compile outputs.-->
    <ItemGroup>
        <FileWrites
          Include="@(IntermediateAssembly)"
          Condition="Exists('@(IntermediateAssembly)')" />
    </ItemGroup>

    <!-- Record the .xml if one was produced. -->
    <PropertyGroup>
        <_DocumentationFileProduced
          Condition="!Exists('@(DocFileItem)')">false</_DocumentationFileProduced>
    </PropertyGroup>

    <ItemGroup>
        <FileWrites
          Include="@(DocFileItem)"
          Condition="'$(_DocumentationFileProduced)'=='true'" />
    </ItemGroup>

    <!-- Record the .pdb if one was produced. -->
    <PropertyGroup>
        <_DebugSymbolsProduced
           Condition="!Exists('@(_DebugSymbolsIntermediatePath)')">false
        </_DebugSymbolsProduced>
    </PropertyGroup>

    <ItemGroup>
        <FileWrites
          Include="@(_DebugSymbolsIntermediatePath)"
          Condition="'$(_DebugSymbolsProduced)'=='true'" />
    </ItemGroup>
</Target>
```

From the first FileWrites item definition, the condition is defined as Exists (@(IntermediateAssembly)). This will determine whether the file referenced by the IntermediateAssembly item exists on disk. If it doesn't, then the declaration task is skipped. This was a brief overview of conditional statements, but it should be enough to get you started. Let's move on to learn a bit more about targets.

Default/Initial Targets

When you create an MSBuild file, you will typically create it such that a target, or a set of targets, will be executed most of the time. In this scenario, these targets can be specified as default targets. These targets will be executed if a target is not specifically chosen to be executed. Without the declaration of a default target, the first defined target in the logical project file, after all imports have been resolved, is treated as the default target. A logical project file is one with all Import statements processed. Using default target(s) is how Visual

Studio builds your managed project. If you take a look at Visual Studio–generated project files, you will notice that the Build target is specified as the default target:

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

As mentioned previously, you can have either one target or many targets be your default target(s). If the declaration contains more than one, the target names need to be separated by a semicolon. When you use a command such as `msbuild ProjectFile.proj`, because you have not specified a target to execute, the default target(s) will be executed. It's important to note that the list of DefaultTargets will be preserved, not modified, through an Import, provided that a project previously processed hasn't had a DefaultTargets list. This is one difference between DefaultTargets and InitialTargets. Values for InitialTargets are aggregated for all imports because each file may have its own initialization checks.

These targets listed in InitialTargets will always be executed even if the project file is imported by other project files. Similar to default targets, the initial targets list is declared as an attribute on the *Project* element with the name InitialTargets. If you take a look at the Microsoft.Common.targets file, you will notice that the target _CheckForInvalidConfigurationAndPlatform is declared as the initial target. This target will perform a couple sanity checks before allowing the build to continue. I would strongly encourage the use of default targets. InitialTargets should be used to verify initial conditions before the build starts and raises an error or warning if applicable. Next, we will discuss the command-line usage of the msbuild.exe command.

MSBuild.exe Command-Line Usage

In this section, we'll discuss the most important options when invoking msbuild.exe. When you invoke the msbuild.exe executable, you can pass many parameters to customize the process. We'll first take a look at the options that are available with MSBuild 2.0, and then we'll discuss what differences exist for MSBuild 3.5 and MSBuild 4.0. Table 1-4 summarizes the parameters you can pass to msbuild.exe. Many commands include a short version that can be used; these versions are listed in the table within parentheses.

TABLE 1-4 MSBuild.exe Command-Line Switches

Switch	Description
/help (/?)	Displays the usage information for msbuild.exe.
/nologo	Suppresses the copyright and startup banner.
/version (/ver)	Displays version information.
@file	Used to pick up response file(s) for parameters.

Switch	Description
/noautoresponse (/noautoresp)	Used to suppress automatically, including msbuild.rsp as a response file.
/target (/t)	Used to specify which target(s) should be built. If specifying more than one target, they should each be separated by a semicolon. Commas are valid separators, but semicolons are the ones most commonly used.
/property:<n>=<v> (/p)	Used to specify properties. If providing more than one property, they should each be separated by a semicolon. Property values should be specified in the format: *name=value*. These values would supersede any static property definitions. Commas are valid separators, but semicolons are the ones most commonly used.
/verbosity (/v)	Sets the verbosity of the build. The options are quiet (q), minimal (m), normal (n), detailed (d), and diagnostic (diag). This is passed to each logger, and the logger is able to make its own decision about how to interpret it.
/validate (/val)	Used to ensure that the project file is in the correct format before the build is started.
/logger (/l)	Attaches the specified logger to the build. This switch can be provided multiple times to attach any number of loggers. Also, you can pass parameters to the loggers with this switch.
/consoleloggerparameters (/clp)	Used to pass parameters to the console logger.
/noconsolelogger (/noconlog)	Used to suppress the usage of the console logger, which is otherwise always attached.
/filelogger (/fl)	Attaches a file logger to the build.
/fileloggerparameters (/flp)	Passes parameters to the file logger. If you want to attach multiple file loggers, you do so by specifying additional parameters in the switches /flp1, /flp2, /flp3, and so on.
/distributedFileLogger (/dl)	Used to attach a distributed logger. This is an advanced switch that you will most likely not use and that could have been excluded altogether.
/maxcpucount (/m)	Sets the maximum number of processes that should be used by msbuild.exe to build the project.
/ignoreprojectextensions (/ignore)	Instructs MSBuild to ignore the extensions passed.
/toolsversion (/tv)	Specifies the version of the .NET Framework tools that should be used to build the project.
/nodeReuse (/nr)	Used to specify whether nodes should be reused or not. Typically, there should be no need to specify this; the default value is optimal.

Switch	Description
/preprocess (/pp)*	This will output the complete logical file to either the console or to a specified file. To have the result written out to the file, use the syntax `/pp:file`. Usually, this file will build just as if you were building the original project (there are exceptions though, such as $(MSBuildThisFile)). The real purpose of this is to help diagnose a problem with the build by avoiding the need to jump between many different files. For example, if a particular property is getting overwritten somewhere, it is much easier to search for it in the single "preprocessed" file than it is to search for it in the many imported files.
/detailedSummary (/ds)*	It displays information about how the projects were scheduled to different CPUs. You can use this to help figure out how to make the build faster. For example, you can use this to determine which project was stalling other projects.

* Denotes parameters new with MSBuild 4.0.

From Table 1-4, the most commonly used parameters are target, property, and logger. You might also be interested in using the FileLogger switch. To give you an example, I will use an MSBuild file that we discussed earlier, the ConditionExample01.proj file. Take a look at the following command that will attach the file logger to the build process: `msbuild ConditionExample01.proj /fl`. Because we didn't specify the name of the log file to be written to, the default, msbuild.log, will be used. Using this same project file, let's see how to override the Configuration value. From that file, the Configuration value would be set to Release, but we can override it from the command line with the following statement: `msbuild ConditionExample01.proj /p:Configuration=Debug /t:PrintContent`. In this command, we are using the */p* (property) switch to provide a property value to the build engine, and we are specifying to execute the PrintContent target. The result is shown in Figure 1-7.

```
C:\InsideMSBuild\Ch01>msbuild ConditionExample01.proj /p:Configuration=Debug /t:PrintContent /nolog
o
Build started 9/24/2010 6:42:28 PM.
Project "C:\InsideMSBuild\Ch01\ConditionExample01.proj" on node 1 (PrintContent target(s)).
PrintContent:
  Configuration: Debug
  Content: script.js;script.debug.js
Done Building Project "C:\InsideMSBuild\Ch01\ConditionExample01.proj" (PrintContent target(s)).


Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 1-7 Specifying a property from the command line

The messages on the console show that the value for Configuration was indeed Debug, and as expected, the debug JavaScript file was included in the Content item. Now that you know the basic usage of the msbuild.exe command, we'll move on to the last topic: extending the build process.

Extending the Build Process

With versions of Visual Studio prior to 2005, the build was mostly a black box. The process by which Visual Studio built your applications was internal to the Visual Studio product itself. The only way you could customize the process was to use execute commands for pre- and post-build events. With this, you were able to embed a series of commands to be executed. You were not able to change how Visual Studio built your applications. With the advent of MSBuild, Visual Studio has externalized the build process and you now have complete control over it. Since MSBuild is delivered with the .NET Framework, Visual Studio is not required to build applications. Because of this, we can create build servers that do not need to have Visual Studio installed. We'll examine this by showing how to augment the build process. Throughout the rest of this book, we will describe how to extend the build process in more detail.

The pre- and post-build events mentioned earlier are still available, but you now have other options. The three main ways to add a pre- or post-build action are:

- Pre- and post-build events
- Override BeforeBuild/AfterBuild target
- Extend the BuildDependsOn list

The pre- and post-build events are the same as described previously. This is a good approach for backward compatibility and ease of use. Configuring this using Visual Studio doesn't require knowledge of MSBuild. Figure 1-8 shows the Build Events tab on the ProjectProperties page.

Here, you can see the two locations for the pre- and post-build events toward the center of the image. The dialog that is displayed is the post-build event command editor. This helps you construct the command. You define the command here, and MSBuild executes it for you at the appropriate time using the Exec task (*http://msdn2.microsoft.com/en-us/library/x8zx72cd.aspx*). Typically, these events are used to copy or move files around before or after the build.

Using the pre- and post-build event works fairly well if you want to execute a set of commands. If you need more control over what is occurring, you will want to manually modify the project file itself. When you create a new project using Visual Studio, the project file generated is an MSBuild file, which is an XML file. You can use any editor you choose, but if you use Visual Studio, you will have IntelliSense when you are editing it! With your solution loaded in Visual Studio, you can right-click the project, select Unload Project, right-click the project again, and select Edit. If you take a look at the project file, you will notice this statement toward the bottom of the file.

```
<!-- To modify your build process, add your task inside one
       of the targets below and uncomment it.
       Other similar extension points exist, see Microsoft.Common.targets.
  <Target Name="BeforeBuild">
  </Target>
  <Target Name="AfterBuild">
  </Target>
  -->
```

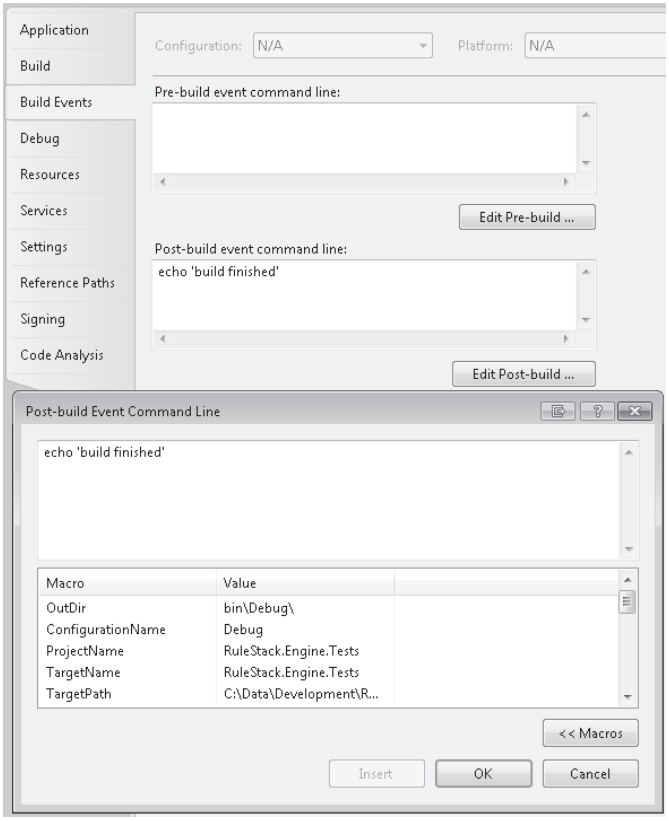


FIGURE 1-8 Build Events tab

From the previous snippet, we can see that there are predefined targets designed to handle these types of customizations. We can simply follow the directions from the project file, by defining the BeforeBuild or AfterBuild target. You will want to make sure that these definitions are **after** the *Import* element for the Microsoft.*.targets file, where * represents the language of the project you are editing. For example, you could insert the following AfterBuild target:

```
<Target Name="AfterBuild">
  <Message Text="Build has completed!" />
</Target>
```

When the build has finished, this target will be executed and the message '`Build has completed`!' will be passed to the loggers. We will cover the third option, extending the BuildDependsOn list, in Chapter 3.

In this chapter, we have covered many features of MSBuild, including properties, items, targets, and tasks. Now you should have all that you need to get started customizing your build process. From this point on, the remainder of the book will work on filling in the details that were left out here so that you can become an MSBuild expert!

Chapter 13

Team Build Quick Start

MSBuild is a build engine rather than a build automation tool, which is where Team Foundation Build (which we will refer to as *Team Build* for short) comes into the picture. Team Build is a component of Microsoft Visual Studio Application Lifecycle Management. Team Build provides build automation that integrates tightly with the other Visual Studio Application Lifecycle Management components, such as version control, work-item tracking, testing, and reporting.

Why discuss Team Build in a book about MSBuild? Apart from the fact that both are build tools, the good news is that Team Build uses MSBuild to build solutions and projects, so the MSBuild knowledge that you've gained in the previous chapters will be put to good use.

Team Build changed significantly between Visual Studio Team System 2008 and Visual Studio 2010 by moving the build process orchestration from being MSBuild-based to Workflow Foundation–based. This change enables scenarios that were difficult to implement using MSBuild (such as distributing builds across multiple machines), provides a graphical build process designer, and provides a customizable user interface for queuing builds and editing build definitions.

Introduction to Team Build

This section discusses the features and architecture of Team Build to familiarize you with its key components and how they relate to each other. These features and components are covered in more depth in later sections.

Team Build Features

Team Build 2010 has a comprehensive set of features that should meet the needs of almost all build automation requirements, and even if it doesn't, it is highly configurable and extensible.

Some of the key features in Team Build 2010 are as follows:

- Provides a default build process suitable for building most Microsoft .NET Framework applications
- Build process is based on Workflow Foundation and is highly configurable and extensible

- Supports the queuing of builds and multiple build machines
- Supports manual, scheduled, continuous integration, and gated check-in builds
- Private builds (also known as *buddy builds*)
- Retention policies for removing old builds
- Integrates with reporting, testing, version control, and work item–tracking components of Visual Studio Application Lifecycle Management
- Includes an API for automating, extending, and integrating with Team Build

High-Level Architecture

A high-level diagram of Team Build's architecture is shown in Figure 13-1.

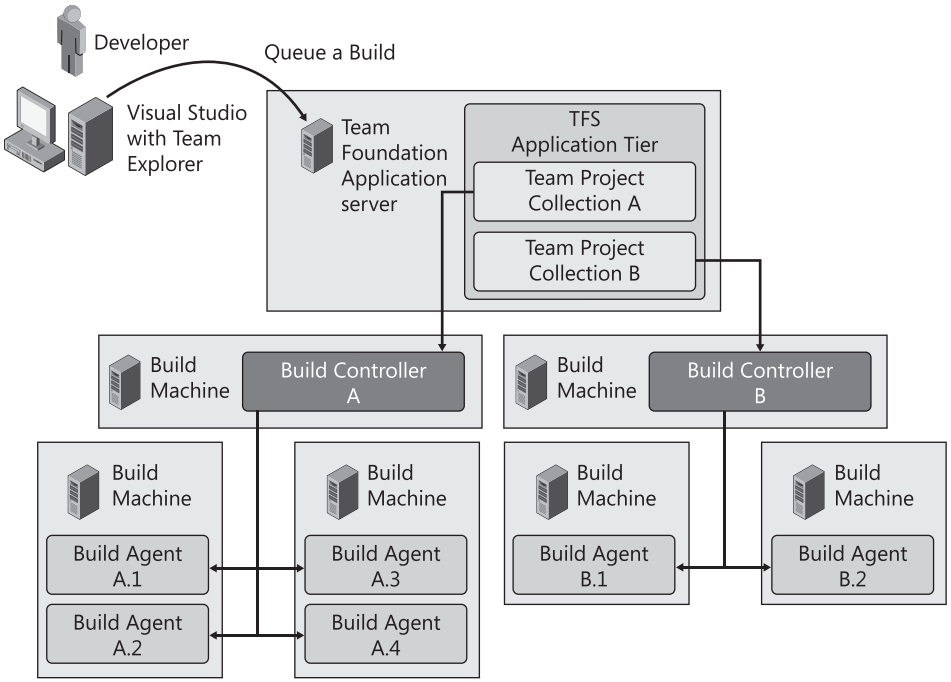


FIGURE 13-1 High-level architecture

The Team Build architecture includes:

- **Team Build client** Visual Studio provides a number of built-in clients for Team Build, including Team Explorer, which is an add-in for Visual Studio; TfsBuild.exe, which is a command-line client for Team Build (and is described in detail in the section entitled "Working with Build Queues and History," later in this chapter); and Team Foundation

Server Web Access, which is a Web interface for Team Build (and other components of the Visual Studio Application Lifecycle Management). Team Build also has an API that can be used to develop your own clients for Team Build, and that will be discussed in Chapter 14, "Team Build Deep Dive."

- **Build controllers** This Windows Service orchestrates the overall build process and is responsible for initializing the build, reserving build agents, delegating parts of the build process to one or more build agents, and finalizing the build. A Team Project Collection can have one or more build controllers associated with it, but each build controller can be associated with only a single Team Project Collection and a machine can have only a single build controller installed on it.
- **Build agents** This Windows Service is responsible for executing the bulk of the build process. A build controller can have multiple build agents associated with it, but each build agent can be associated with only a single build controller. Unlike build controllers, a machine can have multiple build agents installed on it. Because builds are CPU- and I/O-intensive, this is generally not recommended, but if you have sufficiently powerful hardware or your build process isn't resource-intensive, you may be able to increase build throughput by running multiple build agents on each physical build machine.
- **Team Project Collection** Team Project Collections are a new concept in Team Foundation Server 2010, and as you might expect, they are collections of Team Projects. The Team Projects in a Team Project Collection share a database on the database tier and can be backed up, restored, and managed as a single entity. Each Team Project Collection is completely independent, and this is the reason that a build controller can be associated with only a single Team Project Collection.
- **Team Foundation Server application tier** Any Team Build client that wants to communicate with a build controller does so through the Team Foundation Server's application tier. The application tier is implemented as a number of web services hosted using IIS. Communication from the application tier to build agents is always done via the controller.
- **Team Foundation Server data tier** The data tier for Team Foundation Server is hosted as a configuration database (TFS_Configuration), a warehouse database (TFS_Warehouse), and a database for each Team Project Collection (for example, TFS_DefaultCollection) in Microsoft SQL Server.
- **Team Project Collection database** This database stores operational build data such as the list of build controllers and agents, build definitions, build queues, build history, and so on.
- **TFS_Warehouse database** This database stores historical build data for reporting even after it has been purged from the Team Project Collection database.

- **Cube** This multidimensional online analytic processing (OLAP) cube is implemented in SQL Server Analysis Services and is populated regularly from the TFS_Warehouse database for high-performance reporting.
- **Drop folder** When a build completes the build logs, build outputs (if the build is successful or partially successful) and test results are copied to a shared network folder. Public and private builds for the same build definition can be dropped to separate root drop folders.

Preparing for Team Build

In this section, we're going to look at the preparations that you'll need to make to set up the necessary infrastructure before you start automating your build processes using Team Build. Assuming that you've already set up your Team Foundation Server, the first step is to set up at least one build controller and agent to execute your builds. A build controller or build agent is simply a machine that has the Team Build service installed on it and is configured as a build controller, one or more build agents, or both.

Team Build Deployment Topologies

The ability to have multiple build controllers per Team Project Collection and multiple build agents per build controller provides a lot of flexibility, but it also raises questions about when and why you'd want to do this.

Reasons for wanting to have multiple build controllers include:

- **Build agent pooling** Build controllers are a grouping of build agents so that you can use multiple build controllers to segregate your build agents into pools. You may want to do this to dedicate certain agents for certain types of builds [for example, release builds or continuous integration (CI) builds] or to group build agents by physical location for performance.
- **Using different custom workflow activities or extensions** Build controllers specify a version control path from where custom workflow activities and extensions are downloaded. Having multiple controllers allows you to have a controller use a different set of custom workflow activities or extensions. For example, you might have a controller dedicated to testing new versions of custom workflow activities or extensions before you roll them out for production builds.

Reasons for wanting to have multiple build agents include:

- **Redundancy** Having more than one build agent will allow developers to continue to process builds in the event of a build agent failure.
- **Ability to scale out** Multiple build agents will allow builds to be processed concurrently.

- **Distributed builds** By customizing the build process template (which is discussed in Chapters 15 and 16), you could enable a single build to be distributed across multiple build agents to reduce build time.
- **Mutually exclusive dependencies** Different versions of the software that you're building may have dependencies on different versions of third-party software that can't be installed side by side on your build agents. Having multiple build agents enables you to have different versions installed on different build agents. Later in this chapter, we discuss agent tags, which can be used to identify which agents have which dependencies installed.

The other topological consideration is whether you should install build controllers and build agents on the same machine. This is a very valid topology and is especially useful in smaller environments (for example, the build controller has only a single agent) because it requires only one machine. If your build controller is going to manage multiple build agents, then it is recommended to be on its own machine.

What Makes a Good Build Machine?

You should take the following factors into account when selecting and configuring hardware to run Team Build (these factors apply to both build controllers and agents):

- Build machines should be kept as simple as possible. Even minor changes on a build machine can affect the outcome of a build, and if the configuration of a build machine is complex, then it increases the chance of discrepancies if a build agent needs to be rebuilt, when adding additional build machines, or when reproducing an old build.
- Builds usually have to read a large amount of data (the source files) from the Team Foundation Server and write a large amount of data (the build outputs) to the drop folder. Because of this, the build agent should have fast network access to both of these locations. In Chapter 14, we look at how to configure Team Build to use the Team Foundation Proxy to improve performance when the build agent has limited bandwidth to the Team Foundation Server.
- Builds are typically I/O-bound rather than CPU-bound (although there can be exceptions to this), so investing in fast disk and network infrastructure will have a large impact on the performance of your builds.
- Build machines should only be build machines—nothing else. Running other services on the build machine results in Team Build having to compete with them for resources. In particular, avoid disk-intensive services such as the Indexing Service and antivirus software. Many corporate environments require antivirus software; in this case, you should disable scanning for the build agent's working folders to improve performance and reduce the chance that locking issues will cause spurious build failures.

- The build agent needs sufficient disk space to store a copy of the source code and build outputs for each build definition. You should also allow additional disk space for any temporary files produced during the build process.
- The TEMP directory should be located on the same logical drive as the Team Build working directory. The get process is more efficient in this configuration because it can perform move rather than copy operations.
- Team Build 2008 and later have the ability to take advantage of the parallel build functionality introduced in MSBuild 3.5 so multiple processors can improve the performance of your builds.

There might be circumstances where Team Build needs to be installed on developers' workstations. This can be particularly useful when developing, testing, and debugging build customizations or to allow developers to run full end-to-end builds on their local machines.

Installing Team Build on the Team Foundation Server

Although it's technically possible to install a build controller, a build agent, or both on the same machine as the Team Foundation Server, this is not recommended for a number of reasons:

- Compiling software is particularly resource-intensive, and this could be detrimental to the performance of the Team Foundation Server.
- Build scripts and unit tests might be written by people who aren't Team Foundation Server administrators, and having these running on the Team Foundation Server could compromise its security, integrity, and stability.
- Build scripts and the projects being compiled often require third-party software or libraries to be installed on the build agent, and installing these on the Team Foundation Server could also compromise its security, integrity, and stability.

Tip The only time you should consider installing a build controller, a build agent, or both on the same machine as Team Foundation Server is when building a virtual machine for demonstration or testing purposes where it is not practical to have a separate virtual machine acting as the build controller and agent.

Setting Up a Build Controller

The Team Build installation process is quite simple, but it is recommended that you document the process that you use to set up your first build controller and agent so that the process can be repeated if you add additional build controllers or agents to your environment in the future.

Note When installing any Team Foundation Server component, you should download and refer to the latest version of the Team Foundation Installation Guide for Visual Studio 2010 from *http://go.microsoft.com/fwlink/?LinkId=127730*.

Installing Prerequisites

Before installing a build controller, you will need a domain account for the Team Build service to run if you choose not to use the NT AUTHORITY\NETWORK SERVICE account. This account doesn't need to be, and shouldn't be, that of an administrator on either the build server or the Team Foundation Server, but it does need to be added to the Project Collection Build Service Accounts group of the Team Project Collection for which it will execute builds. See the section entitled "Team Build Security," later in this chapter, for more information about securing Team Build.

Installing a Build Controller

The installation process for build controllers is as follows:

1. Insert the installation media.
2. Run setup.exe from either the TFS-x86 or TFS-x64 directory (for 32-bit or 64-bit machines, respectively).
3. Click Next on the Welcome To The Microsoft Team Foundation Server 2010 Installation Wizard page.
4. Accept the license terms and click Next.
5. Select Team Foundation Build Service on the Select Features To Install page and click Install.
6. Make sure that the Launch Team Foundation Server Configuration Tool check box is selected on the last page of the wizard, and then click Configure.
7. Select the Configure Team Foundation Build Service wizard and click Start Wizard.
8. Click Next on the Welcome To The Build Service Configuration Wizard page.
9. Select the Team Project Collection to which you want to connect the build controller and click Next.
10. On the Build Services page, choose how many build agents that you want to run on the build controller machine (this can be none if it's a dedicated controller machine), choose the Create New Build Controller option, and click Next.
11. On the Settings page, enter the account details for your Team Build service account and click Next.
12. On the Review page, review the settings that you've entered, and then click Next.

13. On the Readiness Checks page, resolve any errors and then click Configure.
14. On the Complete page, click Finish.

Configuring a Build Controller After Installation.

Once a build controller has been installed, you can configure it either from Visual Studio on any computer (as described here) or from the Team Foundation Server Administration Console on the build controller itself.

1. Open Visual Studio 2010.
2. Open Team Explorer.
3. Expand a Team Project.
4. Right-click Builds, and click Manage Build Controllers. This will open the Manage Build Controllers dialog shown in Figure 13-2.

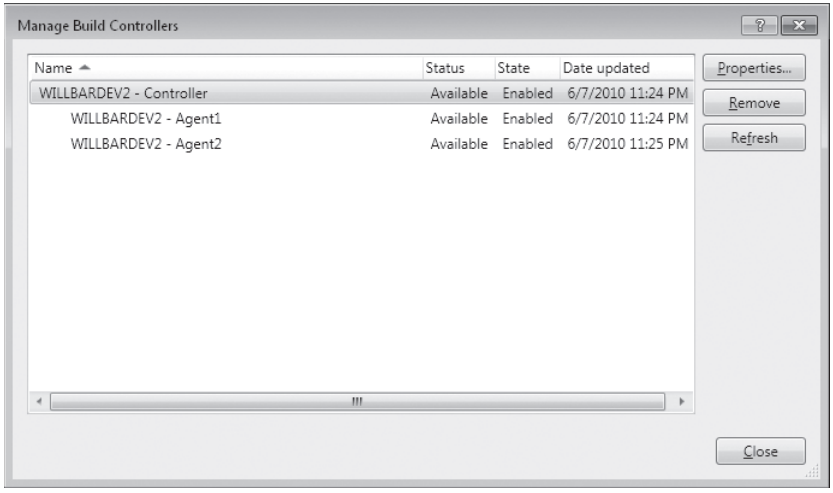


FIGURE 13-2 Manage Build Controllers dialog

5. Select the build controller that you want to configure and click Properties to open the Build Controller Properties dialog shown in Figure 13-3.

The Display Name and Description fields are used to describe the build controller.

The Computer Name field is the host name of the build controller. This will be used by Team Build to communicate with the build controller so the Computer Name should be resolvable from the Team Foundation Server.

The Version Control Path To Custom Assemblies is a server path to a folder containing any custom workflow activities or extensions. The build controller and its agents will download any custom assemblies from the location as required. Creating custom activities is discussed in detail in Chapters 15 and 16.

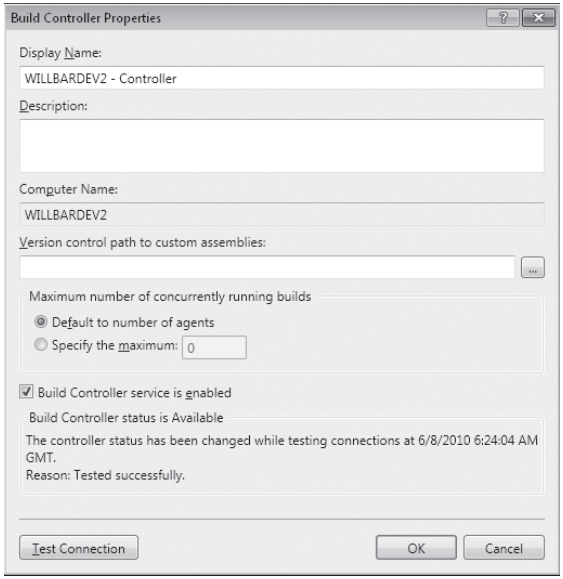


FIGURE 13-3 Build Controller Properties dialog

Tip To make it easier to test changes to your custom workflow activities and extensions, consider having two separate version control folders for custom workflow activities and extensions (one for production and one for testing), and then set up a dedicated controller for testing that uses the testing version control folder.

Setting Up a Build Agent

The build agent installation process is quite similar to the build controller installation process, but because the majority of the build process is run on the build agent, the prerequisites are more complex.

Installing Prerequisites

Before installing a build agent, the following prerequisites need to be met:

- You will need a domain account for the Team Build service to run if you choose not to use the NT AUTHORITY\NETWORK SERVICE account. This account can, and usually is, the same account used to run the build controller.
- You will need any other software or libraries required by your build process or the software you're building. This would include any utilities or MSBuild tasks called by your build process (such as the MSBuild Extension Pack), as well as any global assembly cache (GAC) references required by the projects you're building (such as the Microsoft Office primary interop assemblies).

- You will need the appropriate version of Visual Studio to use any of the features listed in Table 13-1 as part of your build process.

TABLE 13-1 Team Build Prerequisites

Feature	Required Software
Code Analysis	Visual Studio Premium
Code Coverage	Visual Studio Premium
Coded UI Tests	Visual Studio Premium
Database Projects	Visual Studio Premium
Lab Management	Visual Studio Lab Management
Layer Diagram and Dependency Validation	Visual Studio Ultimate
Load Testing	Visual Studio Ultimate
MSBuild Project Types	.NET Framework SDK
Non-MSBuild Project Types (for example, Deployment Projects)	Any edition of Visual Studio able to build the specific project type
Test Impact Analysis	Visual Studio Premium
Third-Party Build Dependencies	The corresponding third-party software
Third-Party GAC References	The corresponding third-party software
Unit Testing	Visual Studio Professional
Visual C++ Projects	Visual Studio Professional
Web Testing	Visual Studio Ultimate

Installing a Build Agent

The installation process for a build agent is as follows:

1. Insert the installation media.
2. Run setup.exe from either the TFS-x86 or TFS-x64 directory (for 32-bit or 64-bit machines, respectively).
3. Click Next on the Welcome To The Microsoft Team Foundation Server 2010 Installation Wizard page.
4. Accept the license terms and click Next.
5. Select Team Foundation Build Service on the Select Features To Install page and click Install.
6. Make sure that the Launch Team Foundation Server Configuration Tool check box is selected on the last page of the wizard, and then click Configure.
7. Select the Configure Team Foundation Build Service wizard and click Start Wizard.
8. Click Next on the Welcome To The Build Service Configuration Wizard page.

9. Select the Team Project Collection to which you want to connect the build controller and click Next.
10. On the Build Services page, choose how many build agents you want to run on the build agent machine, choose the build controller to which you want to attach them, and click Next.
11. On the Settings page, enter the account details for your Team Build service account and click Next.
12. On the Review page, review the settings that you've entered and then click Next.
13. On the Readiness Checks page, resolve any errors and then click Configure.
14. On the Complete page, click Finish.

Configuring a Build Agent After Installation

A build agent can also be configured either from Visual Studio on any computer (as described here) or from the Team Foundation Server Administration Console on the build agent itself, as follows:

1. Open Visual Studio 2010.
2. Open Team Explorer.
3. Expand a Team Project.
4. Right-click Builds, and then click Manage Build Controllers.
5. Select the build agent that you want to configure and click Properties to open the Build Agent Properties dialog box shown in Figure 13-4.

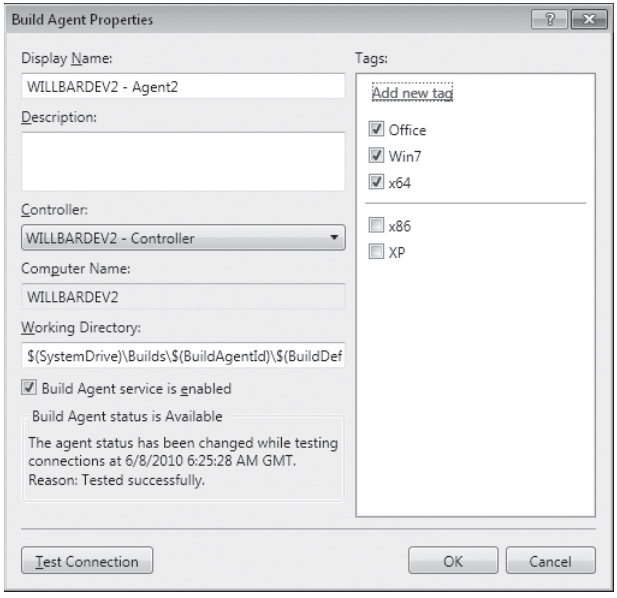


FIGURE 13-4 Build Agent Properties dialog

The Display Name and Description fields are used to describe the build agent.

The Tags allow you to apply arbitrary strings to the agent that can be used to select agents meeting certain criteria. Build definitions can define the tags that they require their agents to have, and then Team Build will automatically select the appropriate agent. Common uses for tags include specifying what operating system and other software the build agent has installed on it, as well as the bit-ness of the build agent. Chapter 14 discusses how you can configure build definitions to require agents with certain tags.

The Controller field allows you to select the build controller that the build agent is associated with.

The Computer Name field is the host name of the build agent. This will be used by Team Build to communicate with the build agent, so the Computer Name should be resolvable from the build controller.

The Working Directory field allows you to specify which directory on the build agent will be used as the working directory during the build. This default working directory is $(SystemDrive)\Builds\$(BuildAgentId)\$(BuildDefinitionPath). For example, if you have a Team Project called Contoso with a build definition called HelloWorldManual running on build agent 12, then the working directory would be C:\Builds\12\Contoso\HelloWorldManual.

You might want to modify the working directory in these scenarios:

- If your build agent has multiple disk partitions, you might want to change the working directory to use one of the additional disk partitions—for example, E:\$(BuildAgentId)\$(BuildDefinitionPath).
- If the source code or build outputs have a particularly deep directory structure or particularly long file names, you may want to use a shorter path—for example, E:\$(BuildAgentId)\$(BuildDefinitionId). This is particularly important when building database projects whose naming conventions result in very long file names.

You should usually include $(BuildAgentId) or $(BuildAgentName) and $(BuildDefinitionPath) or $(BuildDefinitionId) in your working directory so that multiple build agents and definitions can exist side by side in the build agent's working directory. The variables available in the *Working Directory* field are listed in Table 13-2.

TABLE 13-2 Working Directory Variables

Variable Name	Description
BuildAgentId	Contains the integer identifier for the Build Agent in the Team Build database.
BuildAgentName	Contains the Build Agent name.
BuildDefinitionId	Contains the integer identifier for the Build Definition in the Team Build database.

Variable Name	Description
BuildDefinitionPath	Contains the Team Project Name and the Build Definition Name; for example, Contoso\HelloWorldManual.
Environment Variables	Each environment variable on the build agent is available as a property. For example, $(Temp) expands to C:\Documents and Settings\TFSBUILD\Local Settings\Temp\ if the Team Build service account is TFSBUILD.

You can toggle whether or not the build agent is enabled using the Build Agent Service Is Enabled check box. When the agent is disabled, builds can still be queued on it, but they won't be processed until it's enabled.

Clicking Test Connection will verify connectivity from the Team Foundation Server to the build controller and from the build controller to the build agent. If the build controller detects that the build agent is offline, then it will automatically disable the build agent. Team Build will automatically enable the agent when it comes back online, but you can force this to occur earlier by clicking Test Connection.

Note Chapter 14 discusses the advanced configuration options that are available for build controllers and build agents.

Drop Folders

The final piece of infrastructure that needs to be in place before you create a build definition is a drop folder, where the build agent puts the build logs and outputs.

Because a Team Build environment may have multiple build agents, drop folders are typically located on a separate network share that all the build agents use. This means that developers, testers, and other users can access drop folders from a single central location.

The drop folder is typically a share on a file server of some description, but it could just as easily be a Network Attached Storage device or some other shared storage device. There are only a few requirements for the drop folder:

- It must be accessible via a UNC path from all of the build agents.
- The Team Build service account must have Full Control permission to it. This is required for the build agent to be able to drop the build logs and outputs.
- It must have sufficient space available to store the number of builds retained by the retention policies that you define.

Tip There is nothing worse than builds failing simply because there is not enough space available in the drop location, especially because you don't find this out until the very end of the build process. It is recommended that you set up monitoring of the available space in the drop location so that you are alerted if it falls below a threshold.

Creating a Build Definition

Now that the necessary infrastructure is in place, you can create your first build definition. Build definitions define the information required to execute a build, such as what should be built, what triggers a build, and how long these builds should be retained.

To create a new build definition, perform the following steps:

1. Open Visual Studio 2010.
2. Open Team Explorer.
3. Expand a Team Project.
4. Right-click Builds, and click New Build Definition.
5. Enter the desired information on each of the tabs, as described in the remainder of this section.
6. Click Save.

General

The General tab shown in Figure 13-5 allows you to name the build definition and optionally describe it. The description is displayed when a developer queues the build, so this can be useful to communicate additional information about what the build definition is for.

You can also temporarily disable the build definition from here as well, which can be used to prevent developers from queuing builds for obsolete or archived build definitions without having to delete the build definition. If using gated check-ins (as discussed in the section entitled "Gated Check-in," later in this chapter) and if the build definition is disabled, then developers will be able to check in without running a validation build.

Tip Be aware that the build definition name is often used from the command line and as a part of the build agent's working directory path, so you should minimize the length of the name (to avoid exceeding maximum path lengths) and avoid unnecessary special characters, including spaces.

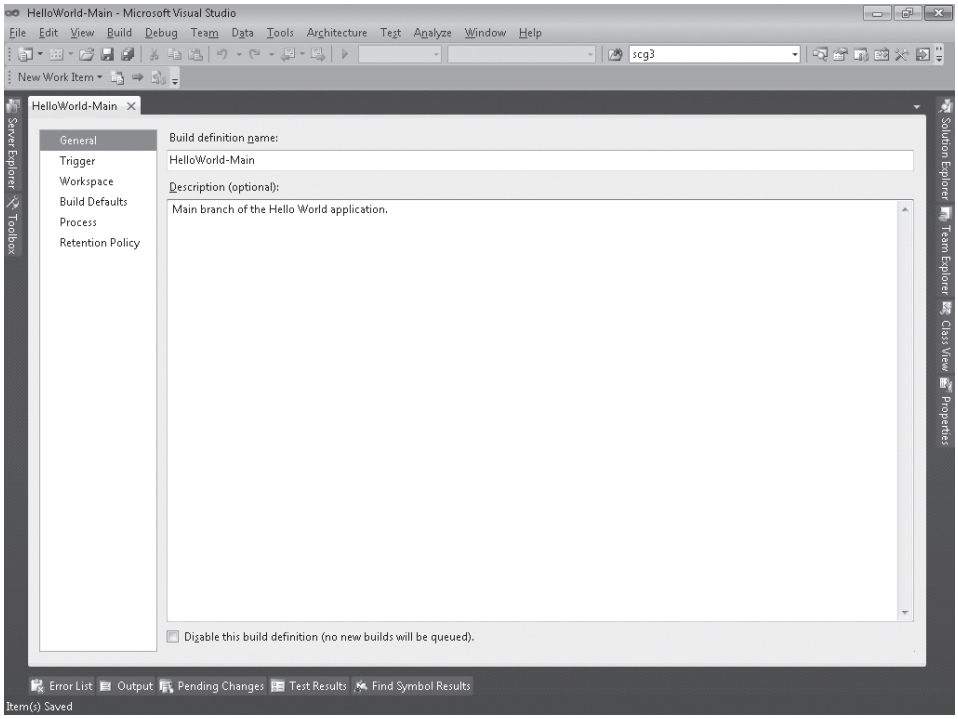


FIGURE 13-5 Build Definition: General

Trigger

Team Build 2005 only provided the ability for builds to be triggered manually, either from within Team Explorer, using the *TfsBuild.exe start* command, from Team Foundation Server Web Access, or using the Team Build API. These methods of starting builds provided build administrators and developers with a large amount of flexibility in how they started builds, but common requirements, such as scheduled builds and continuous integration, required additional programming, scripting, or third-party solutions to implement.

These are now implemented in Team Build 2010 by allowing build administrators to specify what triggers a build in the build definition. The triggers implemented are:

- Manual
- Continuous integration
- Rolling builds
- Gated check-in
- Scheduled

These triggers are configured on the Trigger tab of the Build Definition window, shown in Figure 13-6.

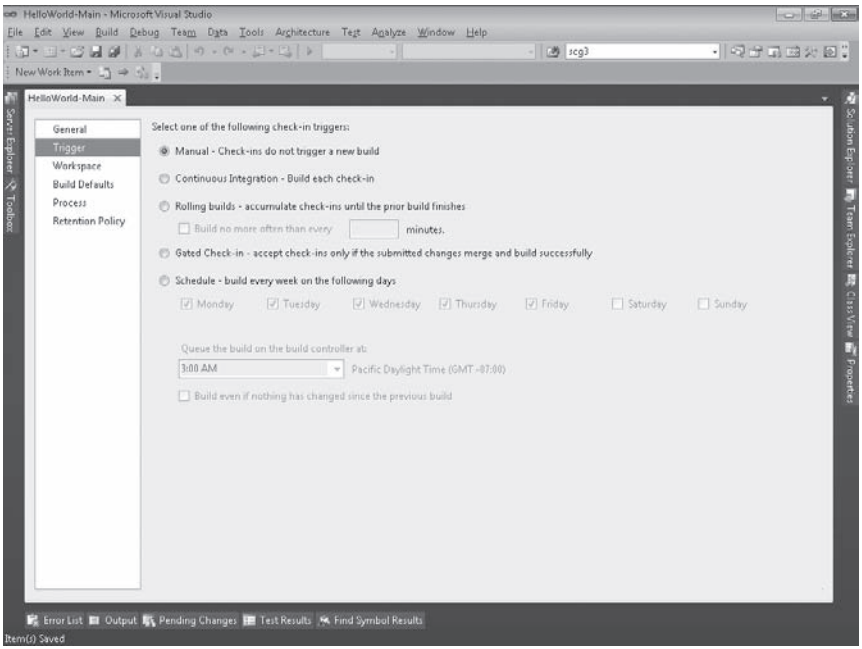


FIGURE 13-6 Build Definition: Trigger

Manual

The simplest (and default) trigger is that builds need to be started manually. This trigger provides exactly the same experience that was available in Team Build 2005, with the exception that in Team Build 2008 and later, builds can be queued rather than failing if a build is already in progress.

Continuous Integration

Continuous integration (CI) is a set of practices from the agile community that provides early warning of bugs and broken code. By building and testing each changeset that has been checked in, any issues can be identified and resolved quickly, minimizing the disruption caused to other developers.

When Team Build 2005 was released, many saw the lack of a CI capability as a huge oversight, especially given its popularity at the time. Microsoft rectified this oversight in Team Build 2008 by adding a CI trigger that removes the need to rely on third-party CI solutions.

The CI trigger causes each check-in to the build definition's workspace to queue a new build, as shown in Figure 13-7.

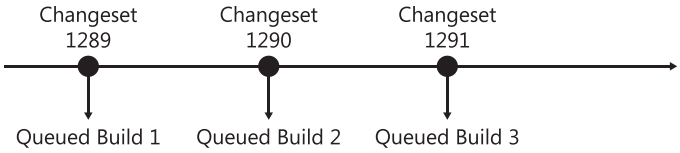


FIGURE 13-7 Changeset to queued build mapping for CI rolling builds

For long-running builds or workspaces that have a large number of check-ins, the CI trigger may result in unacceptably long build queues. The Rolling Builds trigger minimizes this issue by accumulating any check-ins to the build definition's workspace until the currently running build completes; once the build completes, a single build will be queued to build the changesets.

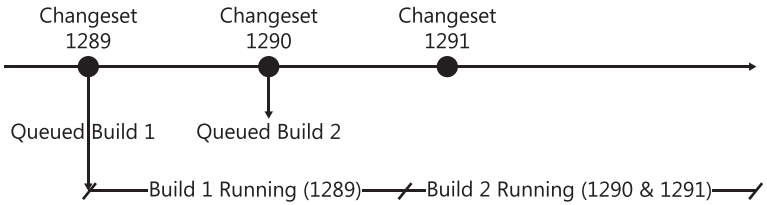


FIGURE 13-8 Changeset to queued build mapping for rolling builds

Even this trigger may result in build queues being dominated by a few build definitions. To add a lag between the builds to allow builds from other build definitions to be executed, you can enable the Build No More Than Every *X* Minutes option of this trigger, shown in Figure 13-9, to ensure that the builds are not executed back to back.

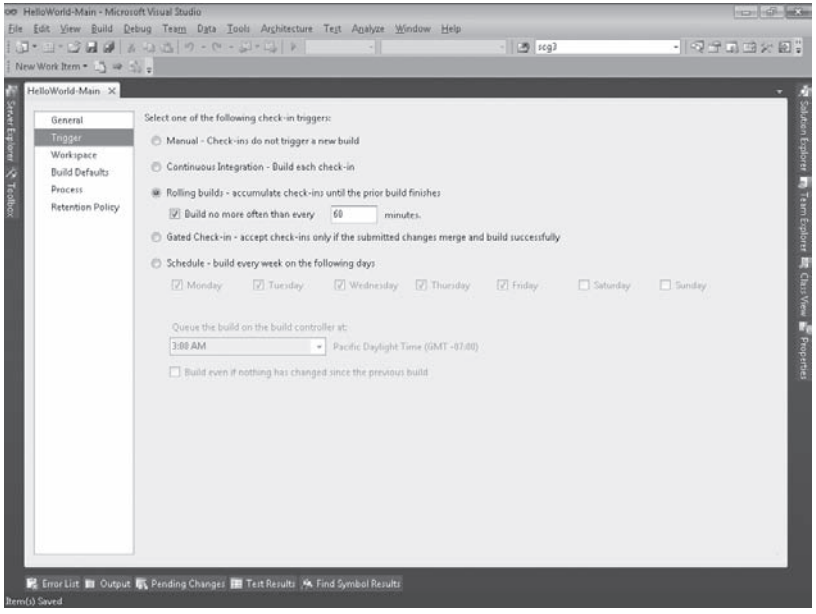


FIGURE 13-9 Build Definition: Trigger (with lag)

Gated Check-in

Team Build 2010 introduces a new trigger called *Gated Check-in*. This trigger behaves similarly to the CI trigger, except that it intercepts the developer's changes before they're checked into version control, builds them, and then, if they build successfully, checks them in on the developer's behalf.

Tip If you think of CI as something that detects bad changes that have made it into version control, then think of Gated Check-in as a mechanism to stop them getting in there in the first place.

Whenever a developer checks changes into a file or folder that is part of the workspace of a build definition that uses the gated check-in trigger, they will be presented with the dialog shown in Figure 13-10.

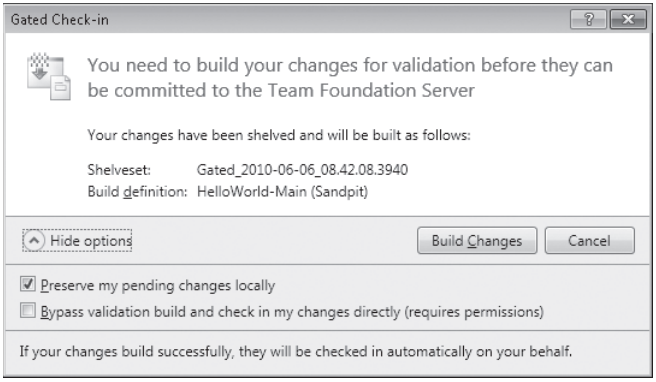


FIGURE 13-10 Gated Check-in dialog

This dialog informs the developer that their changes need to pass a validation build before they're checked in. At this point, the developer's changes have been automatically shelved, and they can choose whether they want to preserve their changes locally or not.

If they've been granted the Override Check-in Validation By Build permission, they also have the option of bypassing the validation build and checking their changes in directly. See the section entitled "Team Build Security," later in this chapter, for more information about this and other Team Build permissions.

Once a gated check-in build completes, the developer will be alerted via the Build Notifications tray to either reconcile their workspace (if the build succeeds) or unshelve their changes (if the build fails). You can also explicitly perform these actions when the build completes by right-clicking the build in the Build Explorer or from the build's Build Details window.

If you did not keep pending changes, then reconciling your workspace is unnecessary, although you should perform a get to bring your workspace up to date. If you did keep your

pending changes, then the Reconcile Workspace dialog (shown in Figure 13-11) can be used to undo any redundant pending changes and bring these files up to date with the changeset that was checked in.

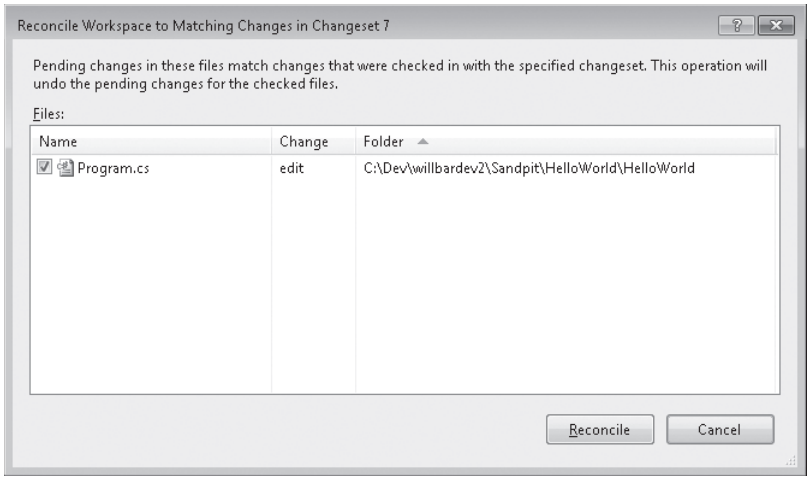


FIGURE 13-11 Reconcile Workspace dialog

Schedule

The Schedule trigger allows builds to be scheduled to run on specific days at a certain time rather than having to use third-party scheduling applications. By default, scheduled builds will be skipped if no changes have been checked in since the previous build. However, this behavior can be overridden by selecting the Build Even If Nothing Has Changed Since The Previous Build check box.

Note One limitation of the scheduling functionality is that you can't schedule a build to be run multiple times a day. If you need this capability, you can either create a new build definition for each time you'd like the build to be run or use a scheduler (such as the built-in Windows Scheduler) to call the TfsBuild.exe command-line client to queue builds.

Workspace

The Workspace tab shown in Figure 13-12 allows you to define which version control folders Team Build will get to execute the build. You can specify multiple folders to get by adding additional working folder mappings with a status of Active, or you can prevent Team Build from getting a folder by changing the status of the mapping from Active to Cloak, as demonstrated in Figure 13-13, which shows that the HelloWorld folder will download but not the HelloWorld/HelloWorld.Tests folder.

Tip If you create a build definition while you have a solution open, then the build definition's workspace mappings will default to the workspace mappings for the workspace containing the solution.

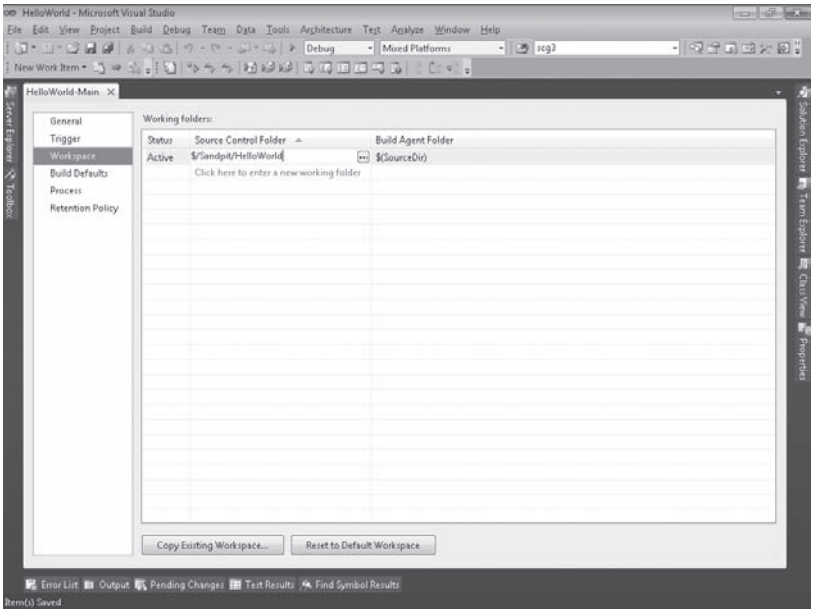


FIGURE 13-12 Build Definition: Workspace tab

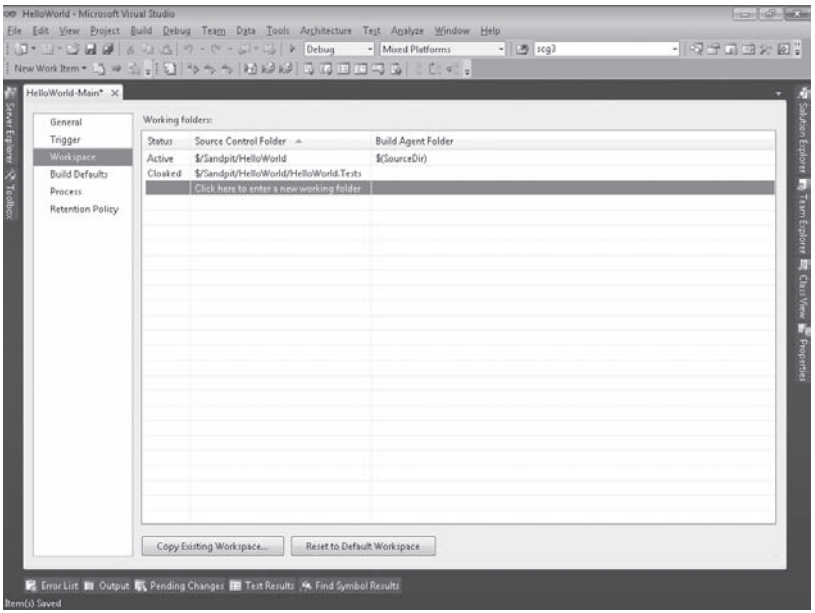


FIGURE 13-13 Build Definition: Workspace tab (multiple working folders)

By default, any other mapping that you add will be mapped to a local folder with the same name as the source control folder. You can override the default by changing the value in the Build Agent Folder column.

If one of the developers already has a workspace that contains the necessary working folder mappings, you can click Copy Existing Workspace to copy the mappings from that workspace into the build definition.

Tip The default working folder mapping on the Workspace tab will download all of the files in the Team Project (or, if you have a solution open when you create the build definition, the workspace containing that solution). If these contain a large number of files and folders that aren't needed by a build definition, you can significantly improve its performance by mapping only the required folders or by cloaking folders that aren't required.

Build Defaults

The Build Defaults tab, shown in Figure 13-14, allows you to specify the default build controller that the build will be queued on and, optionally, where the build outputs will be dropped when the build completes. These are defaults and can be overridden by the developer when they queue the build.

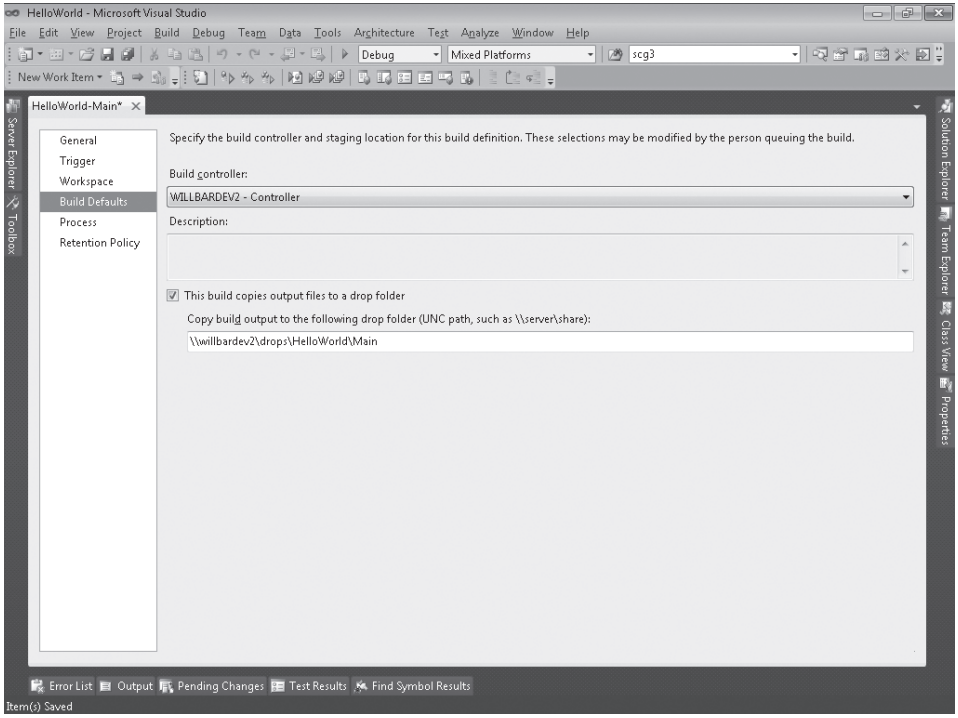


FIGURE 13-14 Build Definition: Build Defaults tab

Process

Build definitions are linked to a Build Process Template that defines the build workflow that will be used. In fact these Build Process Templates are implemented using Workflow Foundation workflows. Chapters 15 and 16 discuss in detail how to customize existing Build Process Templates, as well as how to create your own.

A default Build Process Template will be selected when you create your build definition, but by clicking Show Details, you can select a different Build Process Template, as shown in Figure 13-15.

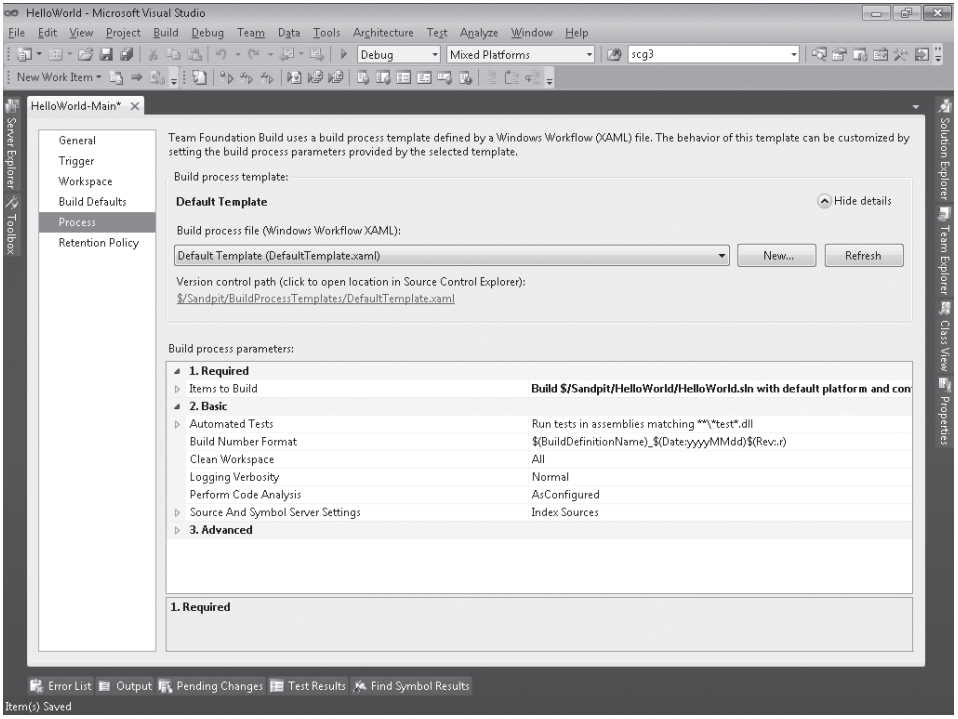


FIGURE 13-15 Build Definition: Process tab

In addition to selecting the Build Process Template, this tab is where you specify the Build Process Parameters. Each Build Process Template defines its own Build Process Parameters, so if you select a different Build Process Template, then you will see different Build Process Parameters selected.

In this section, we'll cover the minimum Build Process Parameters for the Default Template that are needed to get your new build definition working. Chapter 14 will cover all of the Build Process Parameters for the Default Template and the Upgrade Template; and Chapter 16, "Process Template Customization," will cover how to customize Build Process Templates and define your own Build Process Parameters.

The only Build Process Parameter that we need to provide to get our first build definition working is Projects To Build. To provide this parameter, select Items To Build and click the ellipsis to open the Items To Build dialog. Now click Add, browse to the solution or project that you want to build, and then repeat this for each additional solution or project that you want to build. If the solutions or projects have a build order dependency, then you can use the Move Up and Move Down buttons to arrange them in the order they need to be built.

Tip When you create a new build definition, if you have a solution open that's in a version-controlled folder, then the path to that solution will be automatically placed into the Projects To Build build process parameter.

If you don't specify any configurations, then each solution's default configuration will be built, the Configurations tab shown in Figure 13-16 allows you to specify configurations and platforms to be built for the selected solutions. If you specify multiple entries, then the solutions will be built multiple times (once per entry) and the build outputs placed in separate subfolders of the drop folder. In this example, the solution will be built four times, and the build outputs will be placed in the subfolders Release, Debug, Release\x86, and Debug\x86.

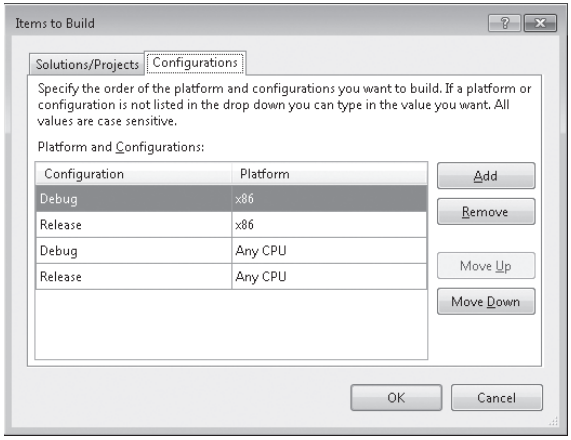


FIGURE 13-16 Configurations tab

Tip If the configuration or platform that you would like to build isn't listed, you can type the name of it into the appropriate combo box.

Retention Policy

In Team Build 2005, build administrators often ran out of disk space in their drop folder. The reason for this is that Team Build 2005 did not provide a solution to automatically remove builds that were no longer required.

Enterprising build administrators worked around this by either scripting the `TfsBuild.exe delete` command or by using third-party solutions (such as the Build Clean-up service, written by Mitch Denny).

Team Build 2008 and later solve this problem by introducing retention policies that allow you to specify which builds should be retained based on criteria in the build definition. The current version of this functionality is limited to retaining builds based on the type of build (Manual And Triggered or Private), the outcome of the build (that is, successful, partially succeeded, stopped, and failed) and the number of builds (for example, retain the last two successful builds). If your requirements are more complex, such as wanting to retain builds based on number of days or on build quality, then you will still need to implement your own solution.

The Retention Policy tab, shown in Figure 13-17, allows you to configure how many builds will be retained for each build outcome.

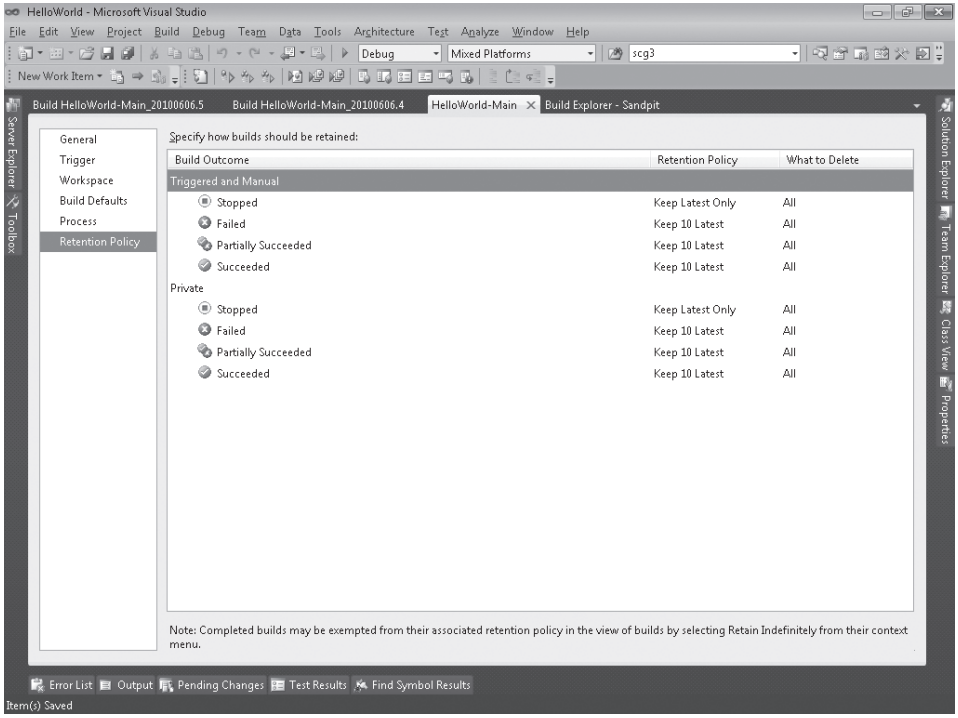


FIGURE 13-17 Build Definition: Retention Policy tab

Tip It's easy to think that you wouldn't want to retain any failed builds, but when builds are removed by the retention policy, everything associated with them, including the build log, is removed. If you don't retain at least one failed build, it might be very difficult to determine the cause of a build failure so that it can be resolved.

When a build is removed by the retention policy, the following items are also removed by default:

- Build details
- Drop folder, including the build logs and binaries
- Test results
- Version control label
- Symbols

Note Although the build details are removed, they are still available for reporting in the TFSWarehouse database and OLAP cube if the warehouse was updated between when the build completed and when it was deleted.

In the What To Delete column, you can override this default for a particular build type and outcome using the Build Delete Options dialog shown in Figure 13-18.

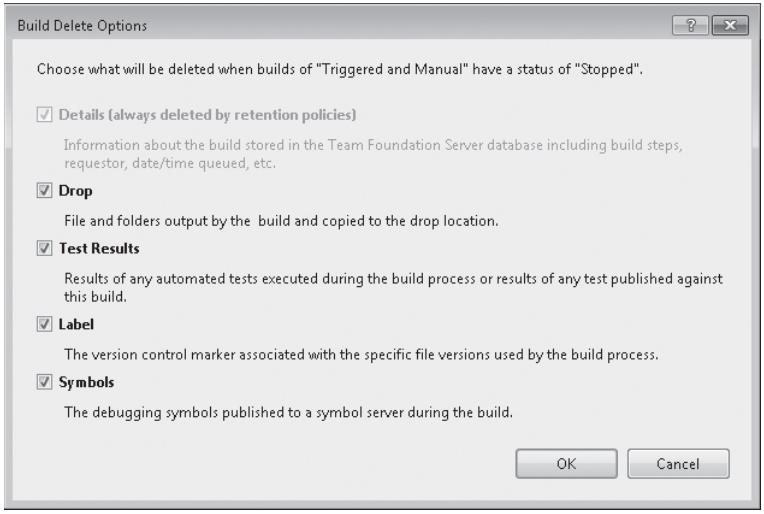


FIGURE 13-18 Build Delete Options dialog

Even if retention policies are enabled for a build definition, individual builds can still be explicitly retained or deleted as discussed in the next section.

Working with Build Queues and History

Congratulations—you've now created your first build definition. Once you have a build definition, you can use Team Build clients such as Visual Studio or the TfsBuild.exe command line to queue builds and work with the build queues and history.

Visual Studio

Developers spend the majority of their time in Visual Studio, so it is logical to be able to work with builds from there. Team Explorer is the entry point to Team Foundation Server functionality within Visual Studio, and Team Build is no exception to this. The Builds node within a Team Project allows build administrators and developers to queue builds and view and manage build queues and individual builds.

Queuing a Build

To queue a build, you right-click the Builds node in Team Explorer and choose Queue New Build to open the Queue Build dialog shown in Figure 13-19. Alternatively, you can right-click a specific build definition and choose Queue New Build, which opens the same dialog but will automatically select that build definition.

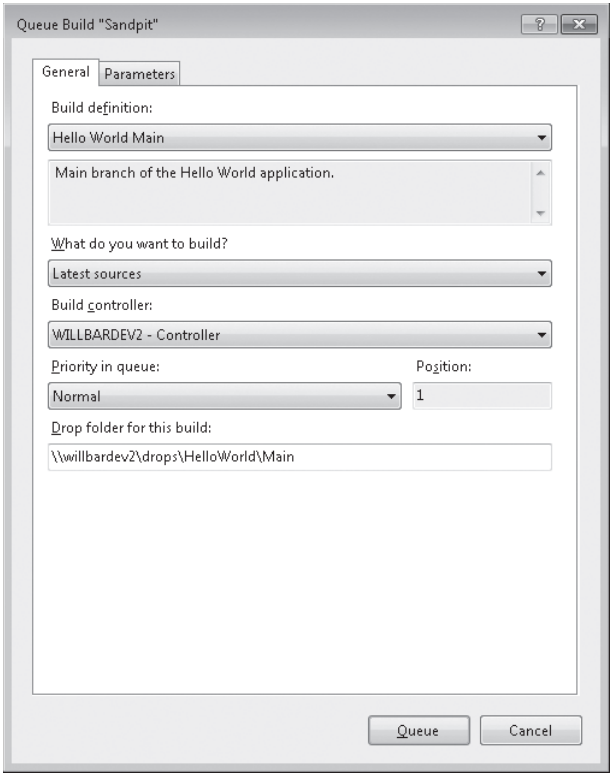


FIGURE 13-19 Queue Build dialog: General tab

The What Do You Want To Build? drop-down list will default to Latest Sources, but developers can change this to Latest Sources With Shelveset to queue a private build against a shelveset containing the changes they'd like to validate. This is discussed in more detail in the section entitled "Queuing a Private Build," later in this chapter.

The Build Controller and Drop Folder For This Build will default to the values selected when you created the new build definition, but developers can override these if desired.

The Position setting indicates where this build will be in the queue if queued on the selected build controller. This is refreshed whenever a different build controller is selected, but there can be a small delay while the position is calculated. You can also change the priority that the build is queued with. As you might expect, the higher the priority, the higher in the queue it will be placed.

On the Parameters tab, shown in Figure 13-20, the developer can override the parameters specified in the build definition for this build process template. Chapter 14 discusses the parameters available for the templates that ship with Team Build, and Chapter 16 discusses how you can define parameters and custom parameter user interfaces for your custom build process templates.

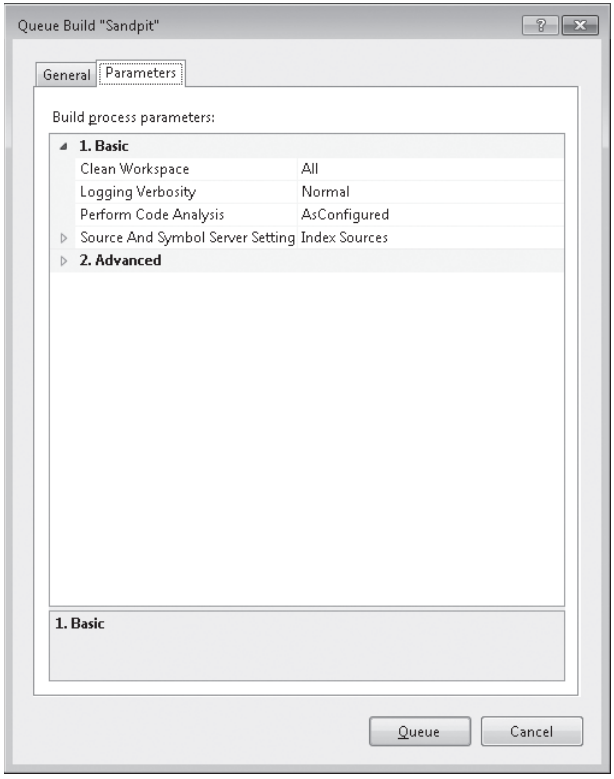


FIGURE 13-20 Queue Build dialog: Parameters tab

If developers always have to override certain parameters, they could create specific build definitions specifying these parameters so they can just queue these build definitions instead.

Clicking Queue will then queue the build on the selected build controller and open the Build Explorer window so you can monitor the progress of your build.

Queuing a Private Build

Private builds (also known as *buddy builds*) allow developers to run a build based on the contents of a shelveset and, optionally, check in the shelveset after a successful build. This can be used to detect compilation errors and test failures before changes are checked in and can affect other developers.

In Team Build 2008, private builds were done by running MSBuild on the TFSBuild.proj in the developer's local workspace. This approach was simple, but it suffered from a number of drawbacks:

- Private builds could be done only from the command prompt.
- The developer's workspace could be out of date, and as such, the build and test results would be inconsistent with the results of building and testing against the latest source code.
- Developers' workstations needed all the prerequisites of the end-to-end build process installed on them.
- Configuration differences between the developer's workstation and the build machines would reduce confidence in the changes actually building successfully when checked in.
- The desktop build process and the end-to-end build process had significant differences that would further reduce confidence in the changes building successfully.
- The build outputs weren't dropped in the same way as the end-to-end build process and couldn't be easily shared with others.

Team Build 2010 takes a different approach and allows developers to shelve their changes and queue an end-to-end build against this shelveset and optionally check the changes in automatically if the build completes successfully.

Note The only shipping template that supports private builds is the Default Template.

Private builds are queued against a build controller, just like triggered and manual builds are, and as such, they use the same hardware, software, configuration, and build process as a triggered or manual build. This increases a developer's confidence that the changes will build and test successfully when checked in.

In some circumstances, it can be seen as a negative that private builds no longer support building on the developer's workstation, but this can be enabled by installing a Team Build controller and agent and choosing that controller when queuing the build. You should be aware of the drawbacks discussed previously of using a developer's workstation for validating changes before check-in.

To enable a build definition to drop the build outputs for private builds, you must configure a Private Drop Location. If you do not do this, then the build will still validate that the shelveset compiles and passes tests, but the build outputs will not be dropped.

Tip You should drop private builds to a separate location from your triggered and manual builds so they aren't accidentally shipped or used as production builds. Private builds contain changes that aren't checked into version control, are based on non-versioned and auditable shelvesets, and as such, they are not reproducible.

To set the Private Drop Location, edit the build definition, and in the Advanced category of the Process tab, enter a UNC path in the Private Drop Location parameter, as shown in Figure 13-21.

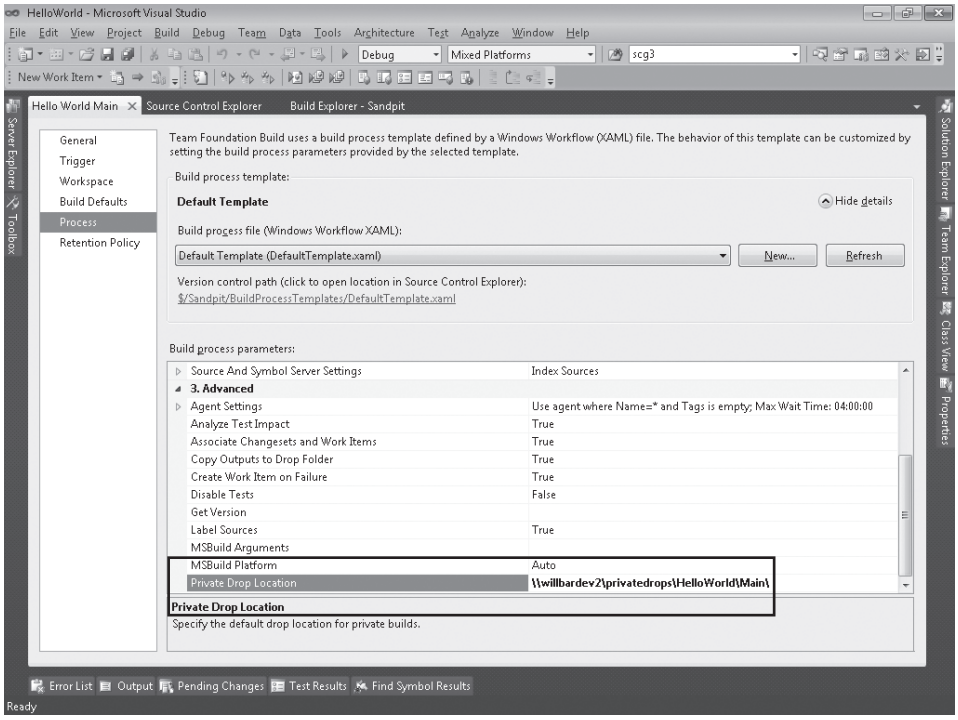


FIGURE 13-21 Private Drop Location parameter

The developer can queue a private build by performing the following steps:

1. Right-click the build definition in Team Explorer and choose Queue New Build.
2. In the What Do You Want To Build? drop-down list, select Latest Sources With Shelveset.
3. Click the ellipsis button and choose the shelveset containing the changes they want to validate. Alternatively, you can create a shelveset based on the pending changes in the workspace by clicking Create.

4. Choose the Check In Changes After Successful Build check box if you want your changes checked into version control if the build completes successfully.
5. Click Queue.

Figure 13-22 shows the Queue Build dialog when queuing a private build of Hello World Main for the shelveset Increase Exclamation.

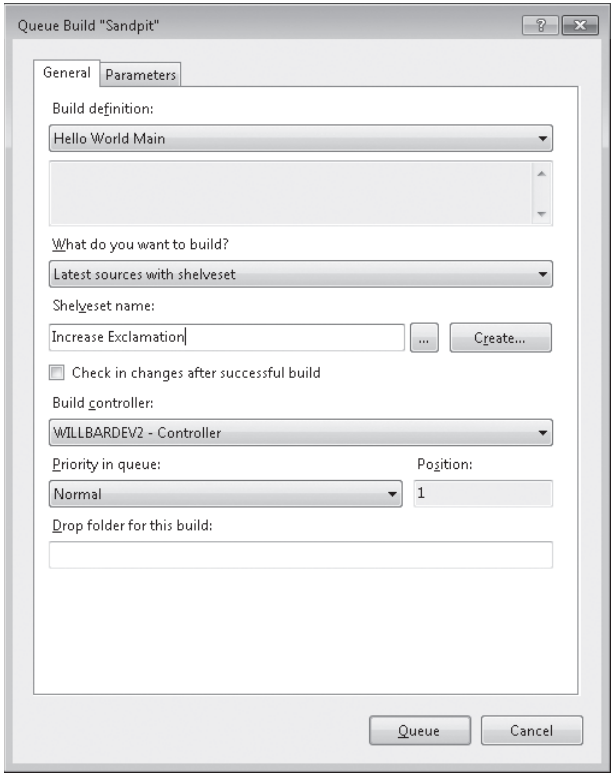


FIGURE 13-22 Queue Private Build dialog

Note In the Team Build 2010 RTM, there is a bug such that the What Do You Want To Build drop-down list sometimes becomes disabled and you won't be able to select Latest Sources With Shelveset. Restarting Visual Studio will usually resolve this.

Private builds need to strike the right balance between speed and completeness to ensure that developers can validate their changes in a reasonable amount of time and still have a high level of confidence that a successful private build will typically mean a successful triggered or manual build.

If private builds take too long or have too much friction, then developers will bypass them and check in without validating their changes (although this can be prevented with the

gated check-in trigger discussed in the section entitled "Trigger," earlier in this chapter). For this reason, it can be beneficial to have a dedicated build definition for private builds that is configured to reduce build times (such as doing incremental gets and builds, running a smaller set of tests, and so on). Chapter 14 discusses the different properties that can be set to modify the default build process provided by Team Build.

Build Explorer

The Build Explorer window, shown in Figure 13-23, is the main way to manage build queues and view the build history. The Build Explorer can be opened by right-clicking the Builds node in Team Explorer and choosing View Builds. You can also double-click a build definition, which will open the Build Explorer and automatically filter it to builds of that build definition.

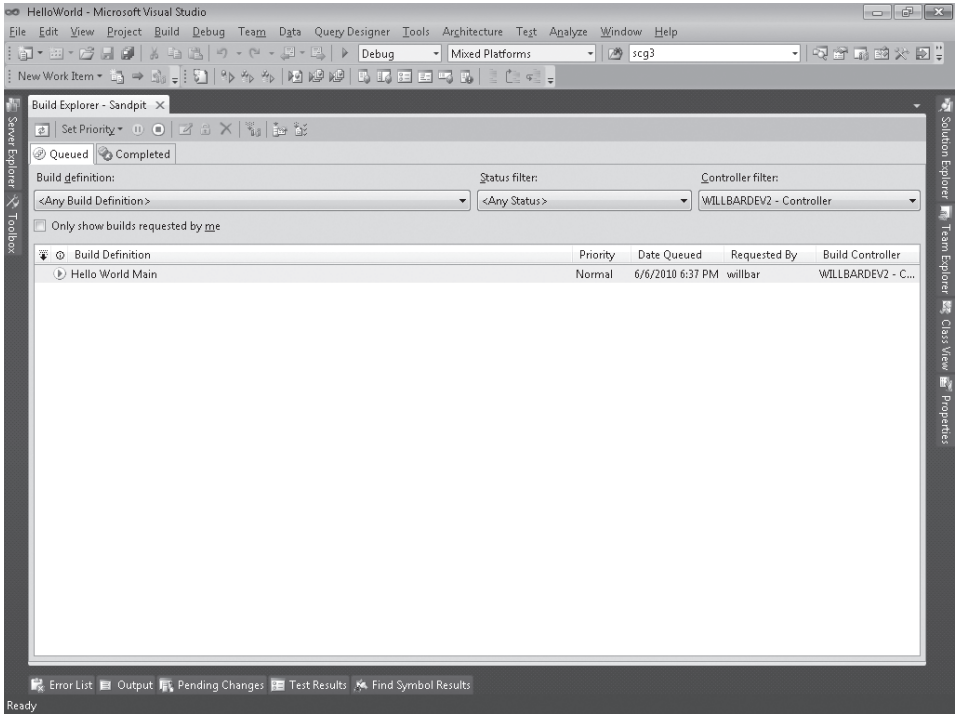


FIGURE 13-23 Build Explorer window

When first opened, the Build Explorer window will show only queued builds, which can be confusing if you expect to see the completed builds as well (as was the case in Team Build 2005). To see completed builds, you need to click the Completed tab at the top of the window.

Note Queued builds will remain on the Queued tab for up to five minutes after they complete.

The Queued build list can be filtered by selecting the filter criteria from the Build Definition, Status Filter, and Controller Filter lists at the top of the window. The Completed build list can be filtered as well, but by Build Definition, Quality, Date, and to builds requested by you.

Cancelling, Stopping, Postponing, and Reprioritizing Builds

If a build is queued but isn't running yet, you can right-click it and choose Cancel to remove it from the queue. Similarly, if a build is currently running, you can stop it by right-clicking the build in the Queued tab of the Build Explorer and choosing Stop.

More Info The actions described in this section are significantly easier to do than they were in Team Build 2005, which required builds to be stopped using the TfsBuild.exe command-line client (which is still possible, as described in the section entitled "Working with Builds from the Command Line," later in this chapter).

Rather than cancelling a queued build, you can postpone it by right-clicking it and choosing Postpone. This places the build on hold, and it won't be built until you right-click the build again and clear the Postpone option.

Builds can be reprioritized to change their position in the queue by right-clicking the build, choosing Set Priority, and then choosing the new priority; the queue will then be refreshed to display the new queue order.

Important The ability to manage the build queue can be restricted via permissions. See the section entitled "Team Build Security," later in this chapter, for details.

Viewing Build Details

Double-clicking a running or completed build in the Build Explorer will open the Build Details window. Note that you can't open the Build Details window for a queued build.

This window has two main views: the Activity Log view, which shows an activity hierarchy for the build; and the Summary view, which summarizes the build results. As shown in Figure 13-24, both views show the build number, latest result, build quality, build history graph, information about how the build was triggered and by whom, how long the build ran, on which controller it ran, and when it completed. You can also change the build quality, open the build's drop folder, toggle retain indefinitely, and delete the build.

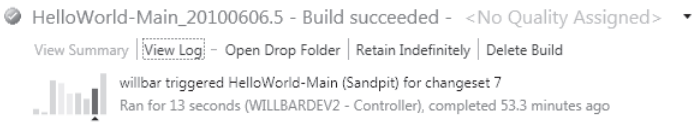


FIGURE 13-24 Build Details header

The build history graph provides an "at a glance" view of the build definition's history. The current build is indicated with a small triangle, the relative height of the bars indicates how long the build ran, and the color indicates the build's outcome (green for successful, orange for partially succeeded, and red for failed). Clicking a bar will take you to the build details for that particular build.

While the build is running, you can only see the Activity Log view (and it will automatically refresh until the build completed) but once the build has completed, you will be shown the Summary view by default. You can toggle between the views using the View Summary and View Log hyperlinks at the top of the window.

The Activity Log view (shown in Figure 13-25) shows a tree of the activities being executed and how long the activity took, which provides an easy way of monitoring the progress of the build and allows you to quickly see what step caused the build to fail.

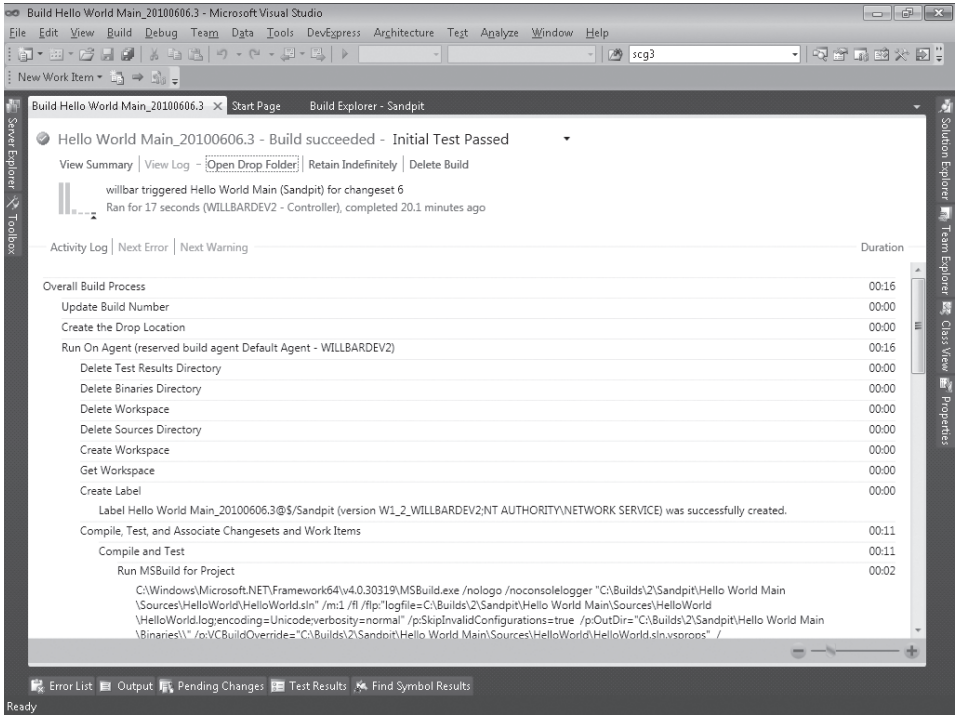


FIGURE 13-25 Build Details window: Activity Log

In Figure 13-26, you can see that the activities preceding compilation succeeded but the compilation itself failed, and you can see exactly what project or configuration caused the build failure. In addition, you can click that project's MSBuild log file to open it.

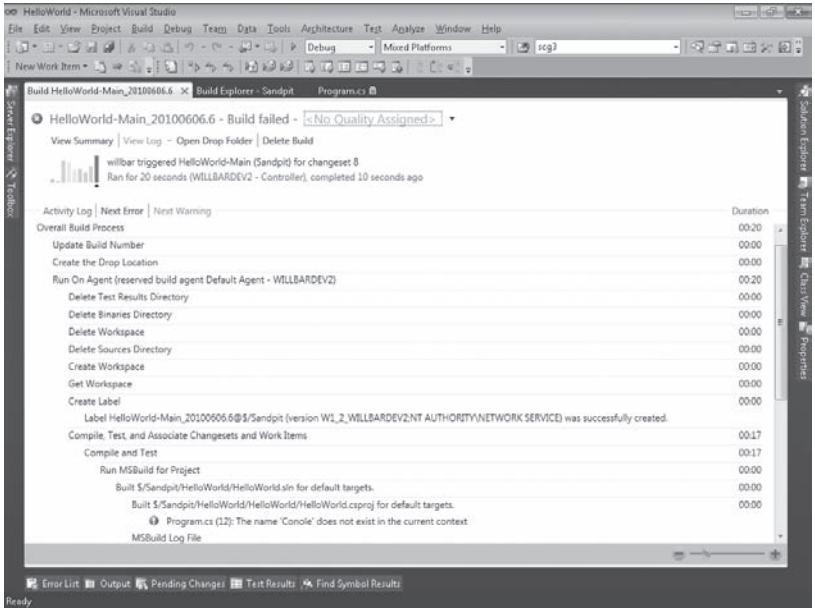


FIGURE 13-26 Build details for a failed build

The Summary view, shown in Figure 13-27, shows the latest activity on the build, a summary of the build results for each configuration and platform (including compilation warnings and errors, test results, and code coverage data), associated changesets and work items, and impacted tests. If the build fails, the Latest Activity section will link to the build failure work item that is created automatically and show its current status, as well as to whom it's assigned.

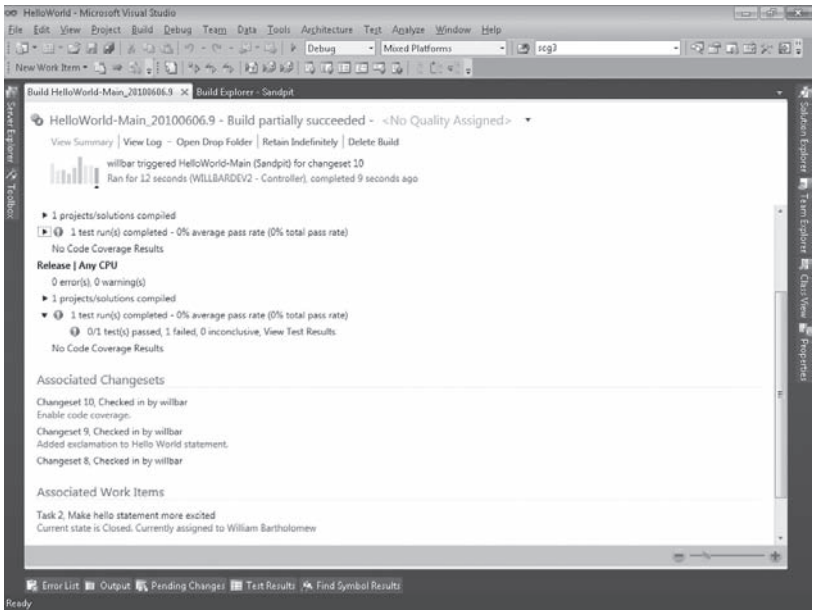


FIGURE 13-27 Build Details window: Summary

The Associated Changesets and Associated Work Items sections list the changesets and work items that are associated with this build, but not earlier builds of the same build definition. This information is extremely useful for providing traceability and in identifying what change caused a build failure or to guide the testing of specific builds. Clicking the changeset number opens the changeset in the standard Changeset dialog, and clicking the work item number opens the work item in the standard Work Item window.

Changing Build Qualities

Once a build has completed, it often goes through a number of other processes before it is released. For example, a build might be installed in a testing environment, pass testing, and then be released.

To provide the ability to track the status of a build, Team Build allows you to flag builds with a build quality. The first step is to define the list of build qualities with which you'd like to be able to flag builds. You can open the Edit Build Qualities dialog, shown in Figure 13-28, by right-clicking the Builds node of Team Explorer and choosing Manage Build Qualities. Figure 13-28 shows the default list of build qualities provided with Team Build, but these can be customized to meet your requirements.

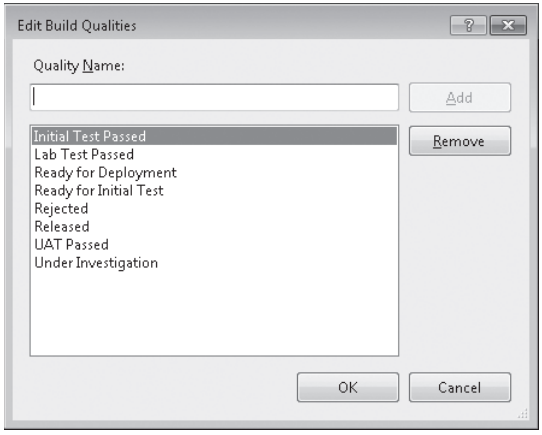


FIGURE 13-28 Edit Build Qualities dialog box

Once the list of build qualities has been defined, you can assign a build quality to a build by opening the build's Build Detail window and changing the drop-down list at the top, as shown in Figure 13-29. You can also change the build quality from the Build Explorer by right-clicking the build and choosing Edit Build Quality. Assigning or changing a build's build quality requires the user to be assigned the Edit Build Quality permission.

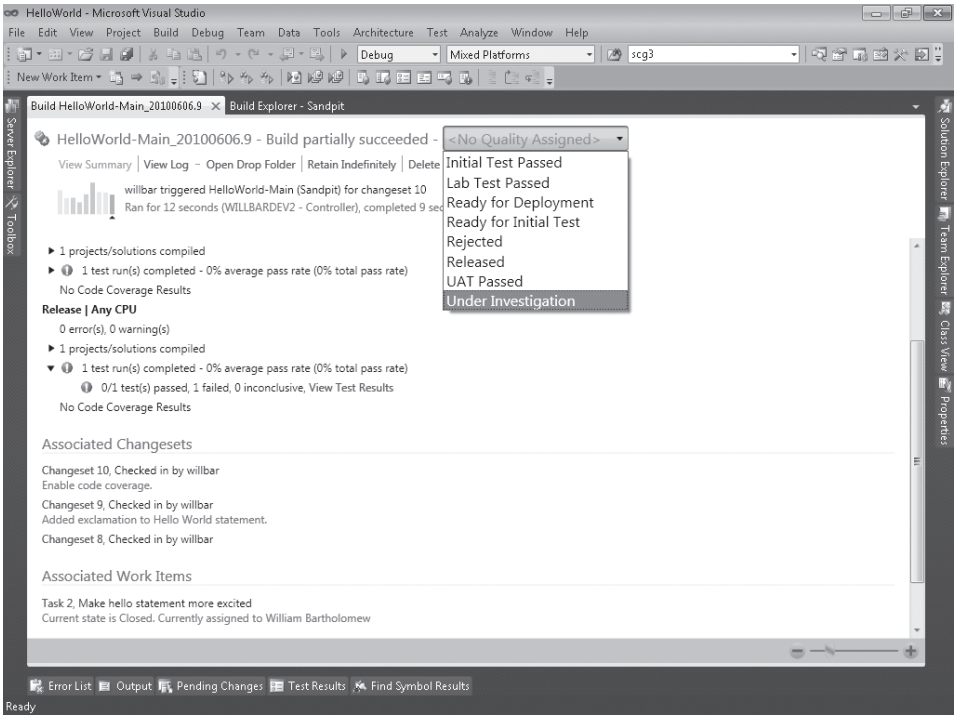


FIGURE 13-29 Changing a build's quality

Retaining Builds

There are situations where you may want to retain builds that otherwise would be removed by the build definition's retention policy, such as builds that you are in the process of testing or that you have released to customers.

You can flag a build to be retained indefinitely by opening the build's Build Details window and clicking Retain Indefinitely at the top. In addition, you can turn this flag on by right-clicking the build in the Completed tab of the Build Explorer window and choosing Retain Indefinitely. If in the future you decide that you no longer want to retain the build, you can repeat this process to turn off the Retain Indefinitely flag.

Deleting Builds

Sometimes you might want to explicitly remove a build even though retention policies haven't been enabled for the build definition or before the retention policy would have removed the build automatically. One reason you might want to do this could be to recover disk space or to remove extraneous builds from the build history.

You can explicitly remove a build by opening the build's Build Details window and clicking Delete Build at the top. You can also delete the build by right-clicking the build on the

Completed tab of the Build Explorer window and choosing Delete. You will be prompted to choose which build artifacts you want to delete, as shown in Figure 13-30.

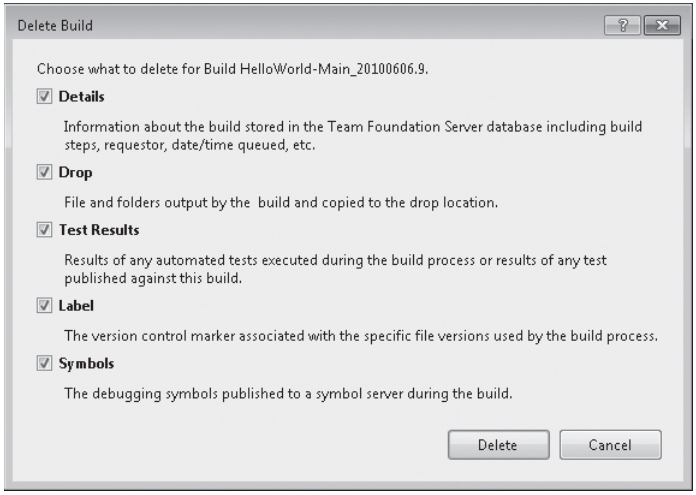


FIGURE 13-30 Delete build options

Working with Builds from the Command Line

Build administrators (and most developers) are command-line fans at heart, and Team Build provides a command-line client for queuing, stopping, and deleting builds. Even if you're not overly fond of using the command line, it also provides a simple way to script Team Build commands as part of a larger process.

The command-line client is called TfsBuild.exe and is installed in the %ProgramFiles%\Microsoft Visual Studio 10.0\Common7\IDE directory as part of the Team Foundation Client. The easiest way to run it is from the Visual Studio 2010 command prompt, which includes this directory in its default path.

The first parameter to TfsBuild.exe is the command to execute. The available commands are listed in Table 13-3.

TABLE 13-3 TfsBuild.exe Commands

Command	Description
Help	Prints general help for the TfsBuild.exe command-line client as well as command-specific help
Start	Starts a new build either synchronously or asynchronously
Stop	Stops one or more running builds
Delete	Deletes one or more completed builds and their artifacts
Destroy	Destroys (purges) previously deleted builds permanently

To print general help and a list of available commands, run TfsBuild.exe help.

To print help for a specific command, run the following code:

```
TfsBuild.exe help <command>
```

where <*command*> is the command in question (for example, `TfsBuild.exe help start`).

Note Any arguments containing a space should be enclosed in double-quotation marks.

Queuing a Build

The TfsBuild.exe command line provides two variations of the start command. The first has the following syntax, and its parameters are described in Table 13-4:

```
TfsBuild start /collection:<teamProjectCollectionUrl> /buildDefinition:<definitionSpec>
               [/dropLocation:dl] [/getOption:go] [/priority:p]
               [/customGetVersion:versionSpec] [/requestedFor:userName]
               [/msBuildArguments:args] [/queue] [/shelveset:name [/checkin]] [/silent]
```

TABLE 13-4 TfsBuild.exe Start Parameters

Parameter	Description
/collection:<teamProjectCollectionUrl>	The full URL of the Team Project Collection (for example, *http://TFSRTM10:8080/tfs/defaultcollection*).
/buildDefinition:<definitionSpec>	The full path of the build definition in the format \<*Team Project*>\<*BuildDefinitionName*> (for example, \Contoso\HelloWorldManual).
/dropLocation:<dl>	If specified, overrides the drop location in the build definition.
/getOption:<go>	If specified, states what version of the source code Team Build will get. Table 13-5 lists the available *get* options.
/priority:<p>	Set to either *Low, BelowNormal, Normal, AboveNormal,* or *High*. This parameter will default to *Normal* if not provided.
/customGetVersion:<versionSpec>	If /getOption:Custom is specified, this parameter must be supplied and specifies the version of the source code that Team Build should get. The available *versionspec* options are listed in Table 13-6.
/requestedFor:<userName>	By default, the build will be requested for the user that runs the TfsBuild.exe command line, or if you wish, you can pass this parameter to request a build on behalf of another user if you have sufficient permissions.
/msBuildArguments:<args>	Quoted arguments to be passed to MSBuild when executing TFSBuild.proj. For example, to enable optimizations and increase the logging verbosity to diagnostic, you would specify `/msBuildArguments:"/p:Optimize=true /v:diag"`.

Parameter	Description
/queue	By default, the TfsBuild.exe command line will return an error immediately if the build won't be processed immediately by a build controller (that is, if it needs to be queued). If the build is processed immediately by a build controller, TfsBuild.exe won't return until the build has completed. If this parameter is used, TfsBuild.exe will return as soon as the build has been queued on the build controller.
/shelveset:name	Includes a shelveset in the build by unshelving it after the get has completed.
/checkin	Specifies that the shelveset should be checked in if the build completes successfully.
/silent	If specified, suppresses any output from the TfsBuild.exe command line other than the logo information.

TABLE 13-5 Get Options

Option	Description
LatestOnQueue	Builds the latest version of the source code at the time the build is queued.
LatestOnBuild	Builds the latest version of the source code at the time the build starts (this is the default).
Custom	Builds the version specified by the /customGetVersion parameter.

TABLE 13-6 Versionspec Options

Name	Prefix	Example	Description
Date/Time	D	D07/22/2010 or D07/22/2010T18:00	Builds the source code at a specific date and time. Any string that can be parsed into a System.DateTime structure by the .NET Framework is supported.
Changeset Version	C	C1133	Builds the source code at a specific changeset number.
Label	L	Lcheckpoint2label	Builds the source code at the version specified by the label.
Latest Version	T	T	Builds the latest version of the source code.
Workspace Version	W	Wmyworkspace; my-username	Builds the version of the source code currently in the specified workspace.

The second variation of the start command provides the same functionality as the first but mimics the syntax of the start command in Team Build 2005:

```
TfsBuild start <teamProjectCollectionUrl> <teamProject> <definitionName>
               [/dropLocation:dl] [/getOption:go] [/priority:p]
               [/customGetVersion:versionSpec] [/requestedFor:userName]
               [/msBuildArguments:args] [/queue]
               [/shelveset:name [/checkin]] [/silent]
```

Stopping a Build

You can also stop a running build from the TfsBuild.exe command line by using the stop command.

There are three variations of the stop command, and their parameters are described in Table 13-7:

```
TfsBuild stop [/noPrompt] [/silent] /collection:<teamProjectCollectionUrl>
              /buildDefinition:<definitionSpec> <buildNumbers> ...

TfsBuild stop [/noPrompt] [/silent] /collection:<teamProjectCollectionUrl>
              <buildUris> ...

TfsBuild stop [/noPrompt] [/silent] <teamProjectCollectionUrl> <teamProject>
              <buildNumbers> ...
```

TABLE 13-7 TfsBuild.exe Stop Parameters

Parameter	Description
/noPrompt	If specified, suppresses TfsBuild.exe confirming you want to stop the build
/silent	If specified, suppresses any output from the TfsBuild.exe command line other than the logo information
/collection:<teamProjectCollectionUrl>	The full URL of the Team Project Collection (for example, *http://TFSRTM10:8080/tfs/defaultcollection*)
/buildDefinition:<definitionSpec>	The full path of the build definition in the format \<*Team Project*>\<*BuildDefinitionName*> (for example, \Contoso\HelloWorldManual)
buildNumbers	Space-separated list of build numbers to be stopped
buildUris	Space-separated list of build Uniform Resource Identifiers (URIs) to be stopped

Deleting a Build

You can also delete a build from the TfsBuild.exe command line by using the delete command.

There are five variations of the delete command, and their parameters are described in Table 13-8:

```
TfsBuild delete [/noPrompt] [/silent] [/preview] [deleteOptions:do]
                /collection:<teamProjectCollectionUrl> /buildDefinition:<definitionSpec>
                <buildNumbers> ...

TfsBuild delete [/noPrompt] [/silent] [/preview] [deleteOptions:do]
                /collection:<teamProjectCollectionUrl> <buildUris> ...
```

```
TfsBuild delete [/noPrompt] [/silent] [/preview] [deleteOptions:do]
                <teamProjectCollectionUrl> <teamProject> <buildNumbers> ...

TfsBuild delete [/noPrompt] [/silent] [/preview] [deleteOptions:do]
                /collection:<teamProjectCollectionUrl>
                /buildDefinition:<definitionSpec>
                /dateRange:<fromDate>~<toDate>

TfsBuild delete [/noPrompt] [/silent] [/preview] [deleteOptions:do]
                /collection:<teamProjectCollectionUrl>
                /dateRange:<fromDate>~<toDate> <teamProject>
```

TABLE 13-8 TfsBuild.exe Delete Parameters

Parameter	Description
/noPrompt	If specified, suppresses TfsBuild.exe confirming that you want to delete the build.
/silent	If specified, suppresses any output from the TfsBuild.exe command line other than the logo information.
/preview	Outputs a list of the artifacts that would be deleted without actually deleting them.
/collection:<teamProjectCollectionUrl>	The full URL of the Team Foundation Server (for example, *http://TFSRTM10:8080/tfs/defaultcollection*).
/buildDefinition:<definitionSpec>	The full path of the build definition in the format \<*Team Project*>\<*BuildDefinitionName*> (for example, \Contoso\HelloWorldManual).
/deleteOptions:<do>	If specified, specifies which build artifacts should be deleted. Table 13-9 lists the available delete options. Multiple delete options can be comma-separated (for example, /deleteOptions:Details,DropLocation). The delete command can be run multiple times on the same builds if different delete options are specified.
/dateRange:<fromDate>~<toDate>	The date range of builds that should be deleted. Dates can be specified in any .NET-parsable date format.
buildNumbers	Space-separated list of build numbers to be deleted.
buildUris	Space-separated list of build URIs to be deleted.

TABLE 13-9 Delete Options

Option	Description
All	Deletes all the build artifacts listed in this table.
Details	Marks the build as deleted so that it is hidden in the Team Foundation Client. The build will be permanently deleted only if purged.
DropLocation	Deletes the build outputs from the build's drop location.
Label	Deletes the build's version control label.
TestResults	Deletes the build's test results.
Symbols	Deletes the build's symbols from the symbol store.

Team Build Security

Securing Team Build is a critical part of configuring Team Foundation Server and installing new build agents. Even if your Team Foundation Server environment is safely contained within your corporate firewall, this is still important to prevent inadvertent changes to your build agents and the builds that they produce.

Service Accounts

The first consideration when installing Team Build is to decide under what account to run the Team Build service. There are two options:

- **NT AUTHORITY\NETWORK SERVICE** This built-in Windows account is a limited-privilege account that can access network resources using the computer account's credentials. The account does not have a password and cannot be used to log on to the computer interactively or remotely. For more information about the NETWORK SERVICE account, refer to *http://www.microsoft.com/technet/security/guidance/serversecurity/serviceaccount/sspgch02.mspx#EBH*.
- **Domain Account** Team Build can also run as an arbitrary domain account. Using a domain account allows you to log on to the build machine using this account to install or configure applications that use per-user settings (which you can't do with the NETWORK SERVICE account because you can't log on interactively with it). This can also be useful to debug build problems related to permissions on the build machine or other network resources.

To change the service account used by a build agent or build controller, you should use the Team Foundation Server Administration Console rather than the Services MMC snap-in because it will correctly configure the permissions required by Team Build. The steps are as follows:

1. Log on to the build agent or controller for which you want to change the service account.
2. Open the Team Foundation Server Administration Console (shown in Figure 13-31).
3. Click Stop at the top of the console to stop the build service.
4. Click Properties (shown in Figure 13-32).
5. Enter new credentials for the build service.
6. Click Start.

Note The Team Build service account should not need to be a member of the build machine's Administrators security group. The account should be granted the specific permissions needed by your build processes rather than granting it administrator access to the build machine. This is to minimize the damage of malicious or badly written build scripts.

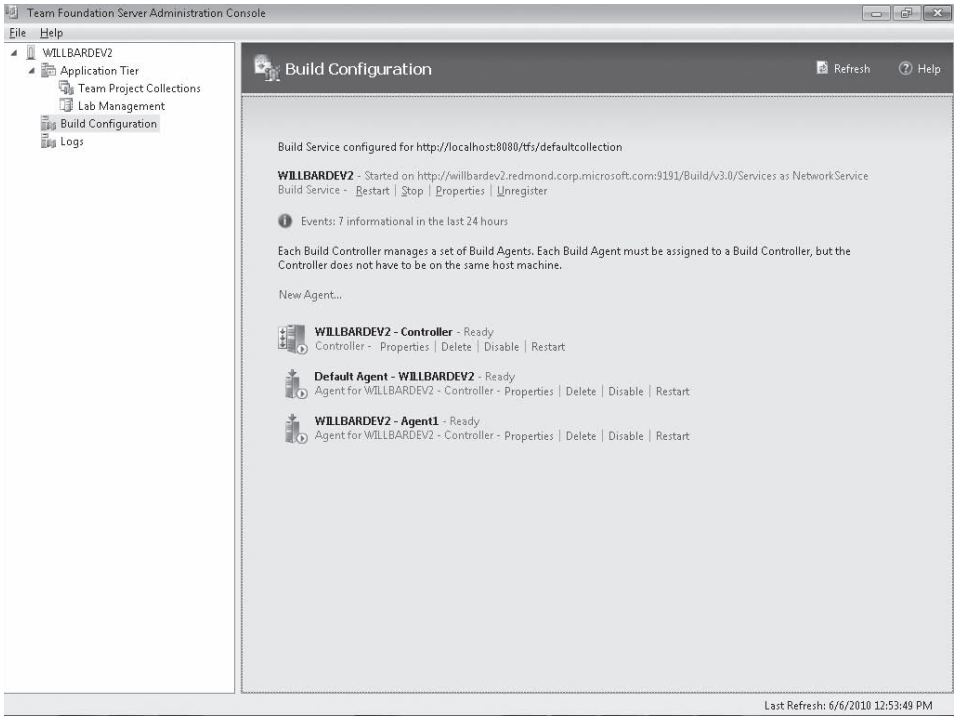


FIGURE 13-31 Team Foundation Server Administration Console

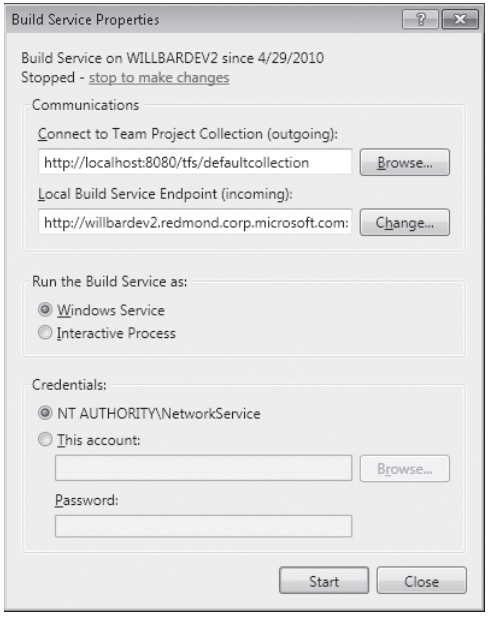


FIGURE 13-32 Configure Team Build service account

The account also needs to be added to the Project Collection Build Service Accounts group for the Team Project Collection for which it will execute builds, as shown in Figure 13-33. This group grants Team Build access to the source, as well as the Team Project Collection permissions required to execute builds. To do this, perform the following steps:

1. Open Visual Studio 2010.
2. Open Team Explorer.
3. Right-click the Team Project Collection.
4. Click Team Project Collection Settings.
5. Click Group Membership.
6. Select the Project Collection Build Service Accounts security group.
7. Click Properties.
8. Click Windows User Or Group.
9. Click Add.
10. Select the domain account that the Team Build service is running as, or the build machine's computer account if it is running as NT AUTHORITY\NETWORK SERVICE.
11. Click OK.
12. Click OK.
13. Click Close.

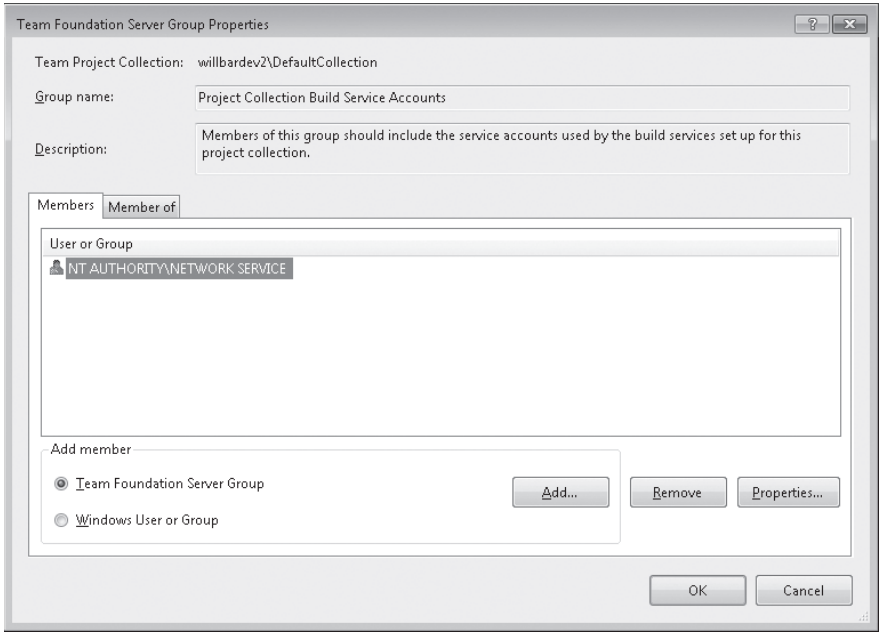


FIGURE 13-33 Build Services Security Group Properties dialog

Note The Team Build service account should not be the Team Foundation Server service account or a member of the Project Collection Administrators, Project Collection Service Accounts, or [Team Project]\Project Administrators security groups. If the Team Build service account is a member of any of these groups, then malicious or badly written build scripts could cause irreparable damage to the Team Foundation Server.

The Team Build service account also requires Full Control file system permission to the drop location.

Permissions

Permissions to both Team Foundation Server or Windows users and groups can be allowed or denied (or left unset). When there is a conflict between allow and deny permissions for a user, deny will take precedence. For more information about how permissions are granted and evaluated in Team Foundation Server, refer to *http://msdn.microsoft.com/en-us/library/ms252587.aspx*.

Team Build provides a number of Team Project Collection–level permissions for controlling access to Team Build functionality. These permissions are detailed in Table 13-10.

TABLE 13-10 Team Project Collection–Level Permissions

Permission	Description	Granted by Default To
Manage Build Resources	Permits the user to manage the build controllers and build agents associated with the Team Project Collection, as well as managing the Use Build Resources and View Build Resources permissions.	Project Collection Administrators; Project Collection Build Administrators; Project Collection Build Service Accounts
Use Build Resources	Permits the user to reserve and allocate build agents. This permission should be granted only to build service accounts.	Project Collection Administrators; Project Collection Build Service Accounts
View Build Resources	Permits the user to see the build controllers and build agents associated with the Team Project Collection.	Project Collection Administrators; Project Collection Build Administrators; Project Collection Build Service Accounts; Project Collection Valid Users

The permissions in Table 13-11 can be managed at either the Team Project level (by right-clicking Builds in Team Explorer and clicking Security) or at the build definition level (by right-clicking the build definition in Team Explorer and clicking Security). Permissions that haven't been overridden at the build definition level will inherit the Team Project level permissions.

Certain Team Build operations (such as creating build definitions and modifying permissions) are limited to users that have the Destroy Builds, Manage Build Queue, and Delete Build Definition permissions.

TABLE 13-11 Team Project– and Build Definition–Level Permissions

Permission	Description	Granted by Default To
Delete Build Definition	Permits the user to delete build definitions.	Project Collection Administrators; [Team Project]\Builders; [Team Project]\Project Administrators
Delete Builds	Permits the user to delete completed builds.	Project Collection Administrators; [Team Project]\Builders; [Team Project]\Project Administrators
Destroy Builds	Permits the user to permanently delete completed builds.	Project Collection Administrators; [Team Project]\Builders; [Team Project]\Project Administrators
Edit Build Definition	Permits the user to create new build definitions (only if applied at the Team Project level) or to edit existing build definitions.	Project Collection Administrators; [Team Project]\Builders; [Team Project]\Project Administrators
Edit Build Quality	Permits the user to set or change the build quality for an individual build.	Project Collection Administrators; Project Collection Build Service Accounts; [Team Project]\Builders; [Team Project]\Contributors; [Team Project]\Project Administrators
Manage Build Qualities	Permits the user to maintain the list of build qualities.	Project Collection Administrators; [Team Project]\Builders; [Team Project]\Project Administrators
Manage Build Queue	Permits the user to cancel, postpone, or change the priority of queued builds. Users without this permission can still cancel their own builds, but they won't be able to postpone or change the priority of any builds, including their own.	Project Collection Administrators; [Team Project]\Builders; [Team Project]\Project Administrators

Permission	Description	Granted by Default To
Override Check-In Validation By Build	Permits the user to bypass gated check-in by checking changes in directly without running a gated check-in build.	Project Collection Administrators; Project Collection Build Service Accounts
Queue Builds	Permits the user to queue a new build.	Project Collection Administrators; Project Collection Build Service Accounts; [Team Project]\Builders; [Team Project]\Contributors; [Team Project]\Project Administrators
Retain Indefinitely	Permits the user to exclude builds from the retention policy.	Project Collection Administrators; [Team Project]\Builders; [Team Project]\Project Administrators
Stop Builds	Permits the user to stop a build that's in progress. Users without this permission can still stop their own builds.	Project Collection Administrators; [Team Project]\Builders; [Team Project]\Project Administrators
Update Build Information	Permits the user to add arbitrary information to the build. This permission should be granted only to build service accounts.	Project Collection Build Service Accounts
View Build Definition	Permits the user to view the details of a build definition.	Project Collection Administrators; Project Collection Build Service Accounts; Project Collection Test Service Accounts; [Team Project]\Builders; [Team Project]\Contributors; [Team Project]\Project Administrators; [Team Project]\Readers
View Builds	Permits the user to view queued and completed builds.	Project Collection Administrators; Project Collection Build Service Accounts; Project Collection Test Service Accounts; [Team Project]\Builders; [Team Project]\Contributors; [Team Project]\Project Administrators; [Team Project]\Readers

The Team Project–level permissions in Table 13-12 are not specific to Team Build but are granted to build service accounts by default.

TABLE 13-12 Other Build-Related Permissions

Permission	Description	Granted By Default To
Create Test Runs	Permits the user to publish test results against any build. Also permits the user to modify test runs or remove test results from any build. Note that this permission can be set only at the Team Project level.	Project Collection Administrators; Project Collection Build Service Accounts; Project Collection Test Service Accounts; [Team Project]\Builders; [Team Project]\Contributors; [Team Project]\Project Administrators
View Project-Level Information	Permits the user to view Team Project–level group membership and permissions.	Project Collection Administrators; Project Collection Build Service Accounts; Project Collection Test Service Accounts; [Team Project]\Builders; [Team Project]\Contributors; [Team Project]\Project Administrators; [Team Project]\Readers;
View Test Runs	Permits the user to view test runs for the Team Project.	Project Collection Administrators; Project Collection Build Service Accounts; Project Collection Test Service Accounts; [Team Project]\Builders; [Team Project]\Contributors; [Team Project]\Project Administrators; [Team Project]\Readers

Index

Symbols and Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

About the Author

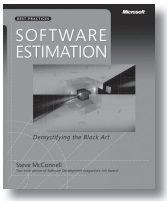
Sayed Ibrahim Hashimi has a computer engineering degree from the University of Florida. He is currently working at Microsoft as a program manager, creating better web development tools. Previously, he was a Microsoft Visual C# MVP. Along with this book he is also a coauthor of *Deploying .NET Application: Learning MSBuild and Click Once* (Apress, 2006), and has written several publications for magazines such as the *MSDN Magazine*. He has previously worked as a developer and independent consultant for companies ranging from Fortune 500 to startups. He is an expert in the financial, education, and collection industries.

William Bartholomew is a software development engineer at Microsoft Corporation in Redmond, Washington. He is a member of the Developer Division Engineering Systems group, which includes the build lab responsible for building and shipping Microsoft Visual Studio.

Pavan Adharapurapu is a software developer at Microsoft. He was part of the team that was responsible for migrating Microsoft Visual C++ over to MSBuild in Visual Studio 2010. He is currently working in the Cloud Computing space and is part of the Azure AppFabric Services team.

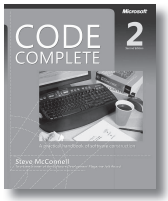
Jason Ward is a development manager at Microsoft. He has more than two decades of experience as a software developer, having worked in Australia and the United Kingdom before moving to Redmond, Washington, where he currently lives with his wife and two daughters.

Best Practices for Software Engineering



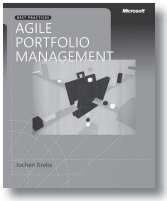
Software Estimation: Demystifying the Black Art
Steve McConnell
ISBN 9780735605350

Amazon.com's pick for "Best Computer Book of 2006"! Generating accurate software estimates is fairly straightforward—once you understand the art of creating them. Acclaimed author Steve McConnell demystifies the process—illuminating the practical procedures, formulas, and heuristics you can apply right away.



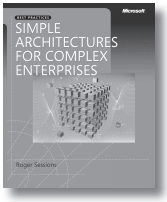
Code Complete, Second Edition
Steve McConnell
ISBN 9780735619678

Widely considered one of the best practical guides to programming—fully updated. Drawing from research, academia, and everyday commercial practice, McConnell synthesizes must-know principles and techniques into clear, pragmatic guidance. Rethink your approach—and deliver the highest quality code.



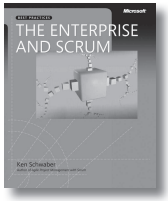
Agile Portfolio Management
Jochen Krebs
ISBN 9780735625679

Agile processes foster better collaboration, innovation, and results. So why limit their use to software projects—when you can transform your entire business? This book illuminates the opportunities—and rewards—of applying agile processes to your overall IT portfolio, with best practices for optimizing results.



Simple Architectures for Complex Enterprises
Roger Sessions
ISBN 9780735625785

Why do so many IT projects fail? Enterprise consultant Roger Sessions believes complex problems require simple solutions. And in this book, he shows how to make simplicity a core architectural requirement—as critical as performance, reliability, or security—to achieve better, more reliable results for your organization.



The Enterprise and Scrum
Ken Schwaber
ISBN 9780735623378

Extend Scrum's benefits—greater agility, higher-quality products, and lower costs—beyond individual teams to the entire enterprise. Scrum cofounder Ken Schwaber describes proven practices for adopting Scrum principles across your organization, including that all-critical component—managing change.

ALSO SEE

Software Requirements, Second Edition
Karl E. Wiegers
ISBN 9780735618794

More About Software Requirements: Thorny Issues and Practical Advice
Karl E. Wiegers
ISBN 9780735622678

Software Requirement Patterns
Stephen Withall
ISBN 9780735623989

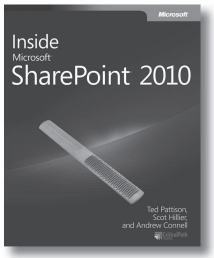
Agile Project Management with Scrum
Ken Schwaber
ISBN 9780735619937

Solid Code
Donis Marshall, John Bruno
ISBN 9780735625921

microsoft.com/mspress

Microsoft® Press

Collaborative Technologies—Resources for Developers

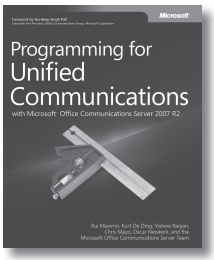


Inside Microsoft® SharePoint® 2010

Ted Pattison, Andrew Connell, and Scot Hillier

ISBN 9780735627468

Get the in-depth architectural insights, task-oriented guidance, and extensive code samples you need to build robust, enterprise content-management solutions.

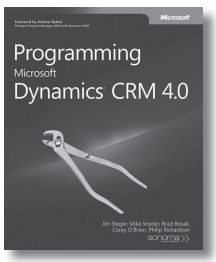


Programming for Unified Communications with Microsoft Office Communications Server 2007 R2

Rui Maximo, Kurt De Ding, Vishwa Ranjan, Chris Mayo, Oscar Newkerk, and the Microsoft OCS Team

ISBN 9780735626232

Direct from the Microsoft Office Communications Server product team, get the hands-on guidance you need to streamline your organization's real-time, remote communication and collaboration solutions across the enterprise and across time zones.

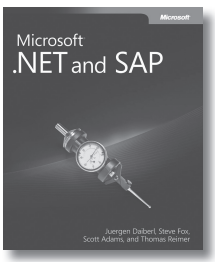


Programming Microsoft Dynamics® CRM 4.0

Jim Steger, Mike Snyder, Brad Bosak, Corey O'Brien, and Philip Richardson

ISBN 9780735625945

Apply the design and coding practices that leading CRM consultants use to customize, integrate, and extend Microsoft Dynamics CRM 4.0 for specific business needs.



Microsoft .NET and SAP

Juergen Daiberl, Steve Fox, Scott Adams, and Thomas Reimer

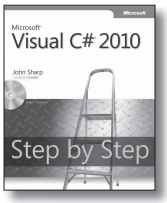
ISBN 9780735625686

Develop integrated, .NET-SAP solutions—and deliver better connectivity, collaboration, and business intelligence.

microsoft.com/mspress

Microsoft® Press

For C# Developers

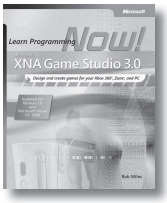


Microsoft® Visual C#® 2010 Step by Step

John Sharp

ISBN 9780735626706

Teach yourself Visual C# 2010—one step at a time. Ideal for developers with fundamental programming skills, this practical tutorial delivers hands-on guidance for creating C# components and Windows–based applications. CD features practice exercises, code samples, and a fully searchable eBook.

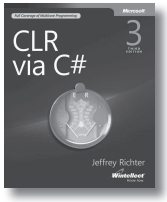


Microsoft XNA® Game Studio 3.0: Learn Programming Now!

Rob Miles

ISBN 9780735626584

Now you can create your own games for Xbox 360® and Windows—as you learn the underlying skills and concepts for computer programming. Dive right into your first project, adding new tools and tricks to your arsenal as you go. Master the fundamentals of XNA Game Studio and Visual C#—no experience required!

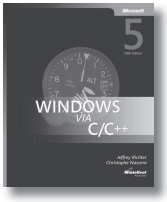


CLR via C#, Third Edition

Jeffrey Richter

ISBN 9780735627048

Dig deep and master the intricacies of the common language runtime (CLR) and the .NET Framework. Written by programming expert Jeffrey Richter, this guide is ideal for developers building any kind of application—ASP.NET, Windows Forms, Microsoft SQL Server®, Web services, console apps—and features extensive C# code samples.

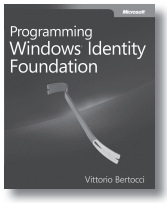


Windows via C/C++, Fifth Edition

Jeffrey Richter, Christophe Nasarre

ISBN 9780735624245

Get the classic book for programming Windows at the API level in Microsoft Visual C++®—now in its fifth edition and covering Windows Vista®.

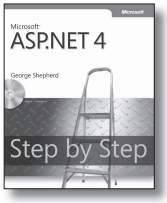


Programming Windows® Identity Foundation

Vittorio Bertocci

ISBN 9780735627185

Get practical, hands-on guidance for using WIF to solve authentication, authorization, and customization issues in Web applications and services.



Microsoft® ASP.NET 4 Step by Step

George Shepherd

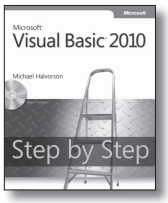
ISBN 9780735627017

Ideal for developers with fundamental programming skills—but new to ASP.NET—who want hands-on guidance for developing Web applications in the Microsoft Visual Studio® 2010 environment.

microsoft.com/mspress

Microsoft® Press

For Visual Basic Developers

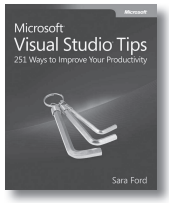


Microsoft® Visual Basic® 2010 Step by Step

Michael Halvorson

ISBN 9780735626690

Teach yourself the essential tools and techniques for Visual Basic 2010—one step at a time. No matter what your skill level, you'll find the practical guidance and examples you need to start building applications for Windows and the Web.



Microsoft Visual Studio® Tips 251 Ways to Improve Your Productivity

Sara Ford

ISBN 9780735626409

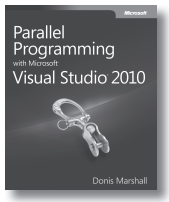
This book packs proven tips that any developer, regardless of skill or preferred development language, can use to help shave hours off everyday development activities with Visual Studio.

Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build, Second Edition

Sayed Ibrahim Hashimi, William Bartholomew

ISBN 9780735645240

Your practical guide to using, customizing, and extending the build engine in Visual Studio 2010.

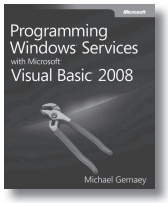


Parallel Programming with Microsoft Visual Studio 2010

Donis Marshall

ISBN 9780735640603

The roadmap for developers wanting to maximize their applications for multicore architecture using Visual Studio 2010.



Programming Windows® Services with Microsoft Visual Basic 2008

Michael Gernaey

ISBN 9780735624337

The essential guide for developing powerful, customized Windows services with Visual Basic 2008. Whether you're looking to perform network monitoring or design a complex enterprise solution, you'll find the expert advice and practical examples to accelerate your productivity.

microsoft.com/mspress

Microsoft® Press

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft® Press

Stay in touch!

To subscribe to the *Microsoft Press® Book Connection Newsletter*—for news on upcoming books, events, and special offers—please visit:

microsoft.com/learning/books/newsletter