Visual Studio
ALM Rangers

Visual Studio Build Customization

Guide

2nd Edition

2012-07-02

Visual Studio ALM Rangers

Microsoft Corporation

Please consider the environment before printing this document

Visual Studio ALM Rangers

The Visual Studio ALM Rangers, a special group with members from the Visual Studio Product Team, Microsoft Services, Microsoft Most Valued Professionals (MVPs) and Visual Studio Community Leads, created this content.

Table of Contents

TABLES

FIGURES

Authors

Contributors

André Dias, Bob Hardister, Daniel Franco Abrahão de Oliveira, David V. Corbin, Ewald Hofman, Fabio Stawinski, Giulio Vian, Jakob Ehn, Jeff Bramwell, Jens K. Süßmeyer, John Jacob (JJ), Mark Nichols, Mathias Olausson, Mike Douglas, Mike Fourie, Nico Orschel, Pete Elliott, Petr Moravek, Richard Fennell, Silfarney Wallace, Steven Lange, Sven Hubert, Tiago Pascoal, Tim Star, Tony Feissle, William Bartholomew, Willy-Peter Schaub

Reviewers

Francisco Xavier Fagas Albarracín, Ed Blankenship, Carsten Duellmann, Hassan Fadili, Jim Lamb, Jahangeer Mohammed, Nico Orschel, Leonard S. Woody III, Patricia Wagner , Martin Woodward

Introduction

Welcome

This document consolidates practical guidance and real-world experience to help you navigate the Team Foundation Server Build technology by describing how to use, customize and implement this technology through a set of common scenarios.

This guidance should be used in conjunction with documentation that accompanies the product and Microsoft Developer Network (MSDN) at http://msdn.microsoft.com

Visual Studio ALM Rangers

A Visual Studio ALM Ranger project created this content. Visual Studio ALM Rangers is a special group with members from the Visual Studio Product group, Microsoft Services, Microsoft Most Valued Professionals (MVP) and Visual Studio Community Leads. Their mission is to provide out-of-band solutions to missing features and guidance. A growing Rangers Index is available online[1].

This guide targets the Microsoft "200-300 level" users of Team Foundation Server. The target group is comprised of intermediate to advanced users of Team Foundation Server and has in-depth understanding of the product features in a real-world environment. Parts of this guide may be useful to the Team Foundation Server novices and experts, but users at these skill levels are not the focus of this content.

Codename "African Tawny Eagle" ATE

The project codename for this initiative is **African Tawny Eagle**. See Chatter … Aviation and Eagles … for those wondering where the Eagle Codenames in recent Rangers project are coming from :)[2].

"**Rosie**", the African Tawny Eagle, at home in http://birdsofprey.co.za.

What's New in the 2nd Edition

This is the 2nd major release of the ALM Rangers Build Customization Guide. In this release we have addressed many reader feedback issues, included guidance for integration with other technologies and updated the guidance posters and Hands-on Labs throughout so that they are compatible with the latest version of Team Foundation Server, Team Foundation Server 2012.

[1] http://blogs.msdn.com/b/willy-peter_schaub/archive/2010/06/18/introducing-the-visual-studio-alm-rangers-an-index-to-all-rangers-covered-on-this-blog.aspx

[2] http://blogs.msdn.com/b/willy-peter_schaub/archive/2010/09/10/chatter-aviation-and-eagles-for-those-wondering-where-the-eagle-codenames-in-recent-rangers-project-are-coming-from.aspx

To assist those who have read the first publication of this guidance, the major new sections included in this second edition are

In addition to these new sections we have provided two new Hands-on Labs

- HOL - What's new in Team Foundation Build 2012
- HOL - Team Foundation Build integration with Windows Azure

We appreciate that there is a lot to read and while you may be tempted to jump to the sections highlighted, we see value in spending time reading the guidance in full. You may encounter information you missed before or identify guidance which you feel needs clarification. We would love to hear from you!

Understanding the Personas

This guidance is based on customer types, personas, scenarios (user stories) and teams. The intention is to demonstrate, in a more realistic and convincing way, how personas perform a productive task, leveraging the technology and this guidance, to create a tangible outcome within their environment.

The following illustration summarizes the customer types, personas, and teams used in this guidance:

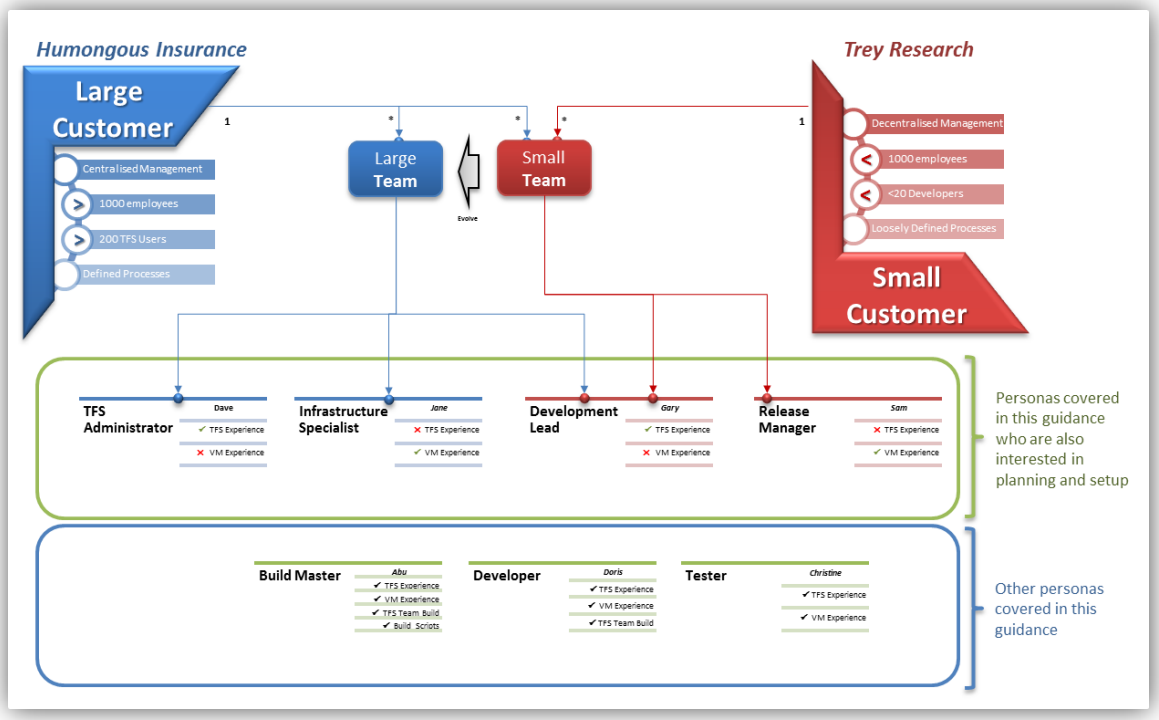


Figure 1 – Customer Types, Teams and Personas

To assist you in choosing your own adventure, a reference poster helps you decide where to focus your efforts, as shown below. For example, if you are Doris and your scenario matches "*I want practical guidance and samples for custom activities for versioning assemblies to standardize the versioning strategy*" then you should be looking at the Custom Activity section of the guidance.

A separate, high quality poster, like the example shown on the next page, is included with the guidance.

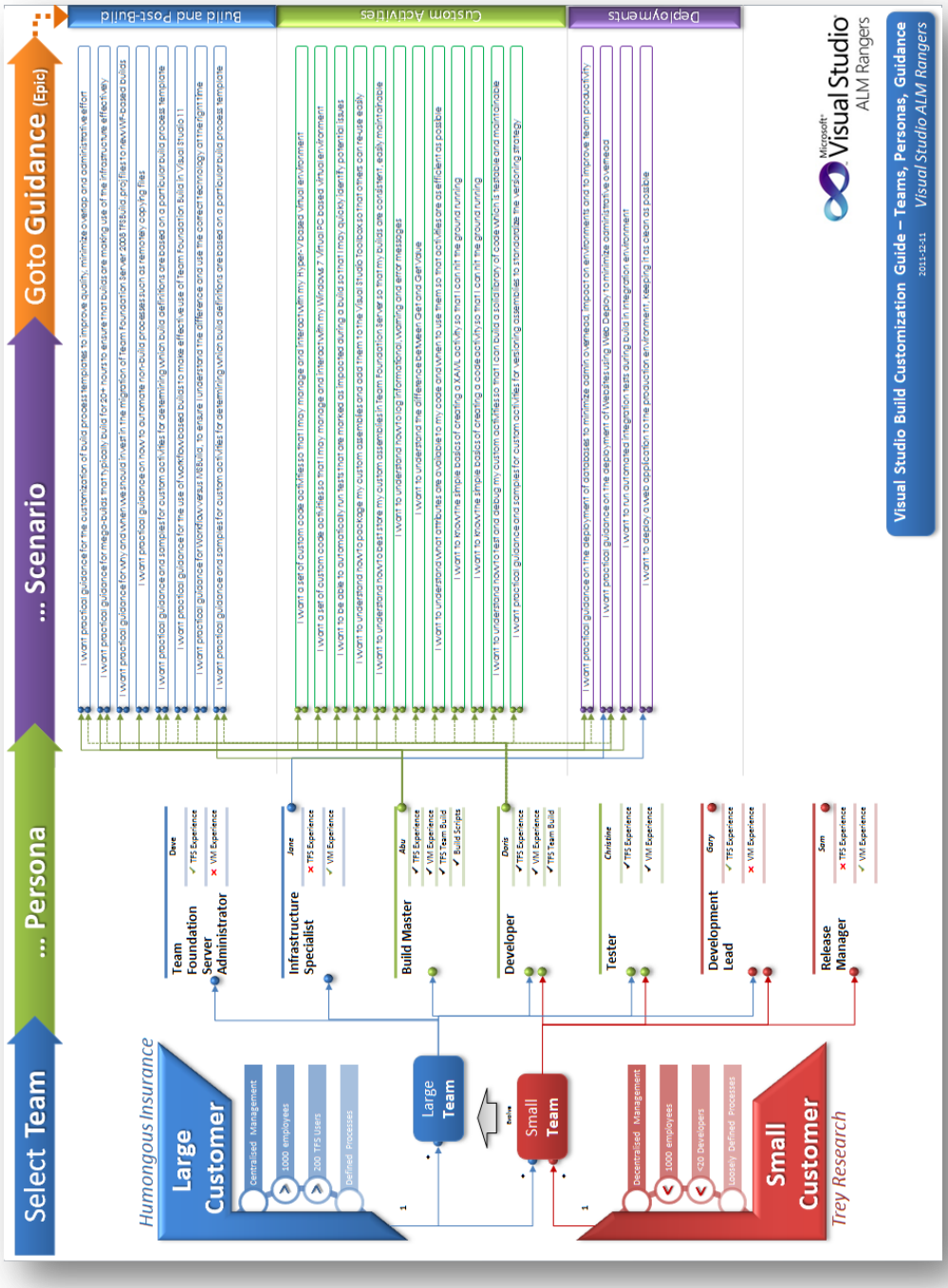


Figure 2 – Using the persona and scenario poster as a guidance map

Customer Types

Humongous Insurance – Large Enterprise Organization

Humongous Insurance is a hypothetical large investment organization whose focus is creating wealth for their clients. To allow them to monitor trading 24 hours a day, they have multiple offices all round the world. They have a centralized software development organization, but have developers within most of the offices to enable

localization activities. They have over 1000 employees and 100 - 200 of them interact or plan to interact with Team Foundation Server in one way or another.

Team Foundation Server Environment

To support their activities they have a moderate-to-complex Team Foundation Server infrastructure, which is centrally managed.

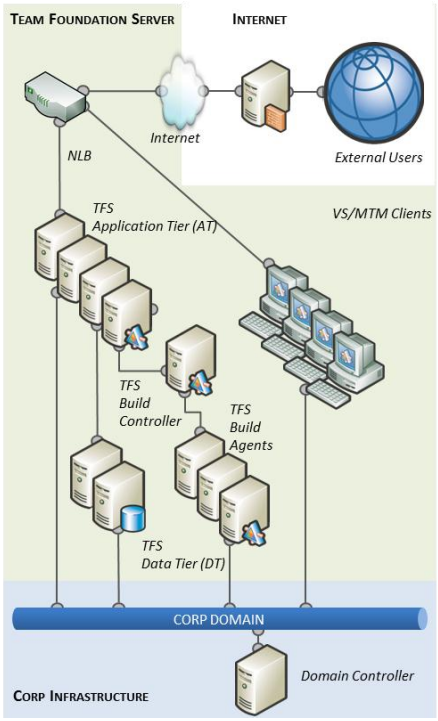


Figure 3 – Humongous Insurance Team Foundation Server Environment

Organizational Structure

The organization contains a number of teams that support their project activities. These teams include:

- **Centralized Development and Test Teams** - Consist of 20-50 people, including project managers, program directors, architects, testers.
- **Smaller Localization Development Teams** - Consist of 2-3 developers, a project manager, a business analyst, and a tester. Localization development teams are usually located in a branch office.
- **Specialized infrastructure Support Team -** Centrally manages all infrastructures for both the development team and the other areas of the business.
- **Centralized Test Team -** Performs manual system testing and acceptance testing of both internal produced and externally produced applications.
- **Centralized Architecture Team -** Performs enterprise application and infrastructure architecture.
- **Centralized Build and Deployment Team -** Performs build, packaging, and deployment services for the major applications. This team also provides consulting services to other teams, but does not necessarily perform builds.

Goals for using Microsoft Visual Studio 2010 Lab Management Feature Pack

Humongous Insurance wants to use Visual Studio Lab Management to support a richer set of features and enhance software testing automation and continuous deployment. Humongous Insurance follows a mix of formal, compliance, and agile development methodologies.

In a difficult economic climate, their goals are to reduce the costs associated with physical machines, improve the speed of environment deployment, and improve the productivity of software development projects through better cross-functional collaboration features.

The overall, driving goal of the leadership team is to increase the quality of their applications while reducing their environment management costs at the same time.

Outside of the leadership, each team within the organization has their own goals.

- **Centralized Development team** wants to increase the quality and predictability of their releases; they see Lab Management as a way to achieve this.
- **Smaller Localization Development Team** does not have access to enough environments to test their changes.
- **Specialized Infrastructure Support team** is tasked with getting Lab Management up and running within the organization.
- **Centralized Test Team** wants to have their testing environments deployed faster and have their automated tests executed as part of build deployment. They want the ability to capture environment snapshots that can be used by developers to find and verify difficult-to-diagnose bugs.
- **Centralized Architecture Team** wants to enforce the architecture of the application and sees Lab Management as a way to help assure the application architecture.
- **Build and Deployment Team** wants to reduce the cost of setting up and maintaining build definitions by easily integrating the Virtual Machine build machines into the build, deploy, and test workflow.

Trey Research - Small Organization

Trey Research is a small application development consultancy that focuses on developing solutions for a number of clients. To that end, they have to run a lean and efficient team. In addition to performing ISV work, a number of individuals from their organization travel to client sites. These individuals work on projects for the client, gather requirements, or perform integration work for solutions that they have developed. Trey Research has around 100 employees with about 20 people interacting with Team Foundation Server. After a number of projects were over-promised and under-delivered, the CEO has had enough. He wants to improve their processes, using Team Foundation Server and Lab Management.

Team Foundation Server Environment

To support their development activities, Trey Research has what could be considered a simple Team Foundation Server environment. Typically, Trey Research has adopted the Single Server Team Foundation Server deployment model on physical and/or virtualized hardware.

Refer to:

- http://msdn.microsoft.com/en-us/library/dd631899.aspx for details about the single server scenario.
- http://go.microsoft.com/fwlink/?LinkID=206576 for virtualization guidance.
- http://go.microsoft.com/fwlink/?LinkId=206575 for Team Foundation Server environment capacity planning guidance.

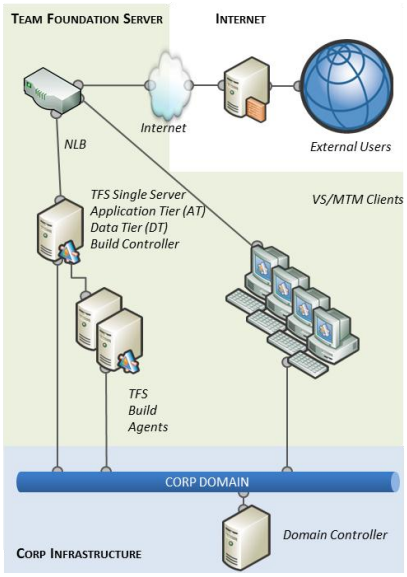


Figure 4 – Trey Research Team Foundation Server Environment

Organization Structure

Trey Research has the **New Application Development Team**, formerly known as the 'A' team within the organization. This team gets to play with all of the new technology, does all the new application development work, and uses an agile approach.

Personas

The following personas are interested in this guidance section, which focuses on automating build and non-build scenarios.

Abu the Build Master

Abu typically works with large teams and exists in small teams as a non-dedicated role.

Scenario	Refer to
Abu wants to know what changes to expect in Team Foundation Build 2012	What's new in Team Foundation Build 2012 - page **23**
Abu wants practical guidance for the customization of build process templates to improve quality, and minimize overlap and administrative effort.	Build process template customization - page **32**
Abu wants practical guidance for mega-builds that typically build for 20 or more hours to ensure that builds are using the infrastructure effectively and that long-running builds are reduced to more manageable builds.	Managing Mega-Build Environments - page **99**
Abu wants practical guidance for why and when we should invest in the migration of Visual Studio Team System 2008 Team Foundation Server TFSBuild.proj files to Visual Studio Team Foundation Server Windows Workflow Foundation-based builds, to ensure we upgrade at the right time and minimize unnecessary administrative effort and cost.	Reasons why to use the upgrade process template - page **50**
Abu wants practical guidance and samples for custom activities for determining which build definitions are based on a particular build	Tracing Build Process Template and Custom Assembly Version - page **90**

Scenario	Refer to
process template. His end goal is to improve administration in medium to large team environments that have many builds.	
Abu wants a set of custom code activities so that he can manage and interact with his Hyper-V based virtual environment.	Hyper-V - page **173**
Abu wants a set of custom code activities so that he can manage and interact with his Windows 7 VirtualPC based virtual environment.	Windows VirtualPC - page **173**
Abu wants to understand how to best store his custom assemblies in Team Foundation Server so that his builds are consistent, easily maintainable and all his teams can take advantage of them.	Storing Custom Activities in Team Foundation Server - page **153**
Abu wants to understand how to package his custom assemblies and add them to the Visual Studio Toolbox so that other developers and third parties can use them easily and conveniently.	Making a Custom Activity available in the Visual Studio Toolbox - page **155**
Abu wants practical guidance for the deployment of databases to minimize administrative overhead and the impact on environments, and to improve team productivity.	Database Deployments on page **182**
Abu wants practical guidance for the deployment of websites using Web Deploy to minimize administrative overhead and the impact on environments, and to improve team productivity.	ASP.NET Web Application in Integration and QA Environments - page **191**
Abu wants to run automated integration tests during build in Integration environment	Running automated integration tests during build in the Integration Environment - page **196**

Doris the Developer

Doris works with large or small teams and focuses on development.

Scenario	Refer to
Doris wants to know what changes to expect in Team Foundation Build 2012	What's new in Team Foundation Build 2012 - page **23**
Doris wants practical guidance for the customization of build process templates to improve quality, minimise overlap and administrative effort.	Build process template customization - page **32**
Doris wants practical guidance for the why and when we should invest in the migration of Visual Studio Team System 2008 Team Foundation Server TFSBuild.proj files to Visual Studio Team Foundation Server Windows Workflow Foundation-based builds, to ensure we upgrade at the right time and minimise unnecessary administrative effort and cost.	Reasons why to use the upgrade process template - page **50**
Doris wants practical guidance for using Windows Workflow Foundation-based builds to make effective use of Team Foundation Build in Visual Studio.	Automating build and non-build scenarios in a Microsoft World - page **22** Build Process Customization 101 page - **61**
Doris wants practical guidance for Windows Workflow Foundation versus MSBuild, to ensure that she understands the difference and uses	Guidance on when to create MSBuild Projects/Tasks versus Windows Workflow Foundation/Workflow

Scenario	Refer to
the correct technology at the right time.	Activities - page **58**
Doris wants practical guidance and samples for custom activities for determining which build definitions are based on a particular build process template to improve administration in medium to large team environments with many builds.	Tracing Build Process Template and Custom Assembly Version - page **90**
Doris wants to understand how to log informational, warning and error messages so that her build system provides sufficient information to monitor and diagnose its operation.	Logging Verbosity - page **40** Logging - page **147**
Doris wants to understand the difference between Get and GetValue so that she and her team use consistent coding practices.	GetValue vs. Get - page **144**
Doris wants to understand what attributes are available to her code and when to use them so that activities are as efficient as possible.	Attributes to minimize log noise - page **42** Attributes - page **151**
Doris wants to know the simple basics of creating a XAML activity so that she can hit the ground running.	Hands-on Lab - Developing Custom Activities
Doris wants to know the simple basics of creating a code activity so that she can hit the ground running	Hands-on Lab - Developing Custom Activities
Doris wants to understand how to test and debug her custom activities so that she can build a solid library of code that is easily testable and highly maintainable.	Testing and Debugging Custom Activities - page **158**
Doris wants practical guidance and samples for custom activities for versioning assemblies to standardize the versioning strategy.	Versioning Assemblies - page **164**
Doris wants practical guidance on the deployment of databases to minimize administrative overhead and the impact on environments, and to improve team productivity.	Database deployments - page **182**
Doris wants practical guidance on the deployment of Websites using Web Deploy to minimize administrative overhead and the impact on environments, and to improve team productivity.	ASP.NET Web Application in Integration and QA Environments - page **191**

Jane the Infrastructure Specialist

Jane works with large teams and likes a fine tuned infrastructure.

Scenario	Refer to
Jane wants to know what changes to expect in Team Foundation Build 2012	What's new in Team Foundation Build 2012 - page **23**
Jane wants practical guidance on the deployment of Websites using Web Deploy to minimize administrative overhead and the impact on environments.	ASP.NET Web Application in Integration and QA Environments - page **191**
Jane wants to deploy a web application to the production environment, keeping it as clean as possible.	Production Deployments - page **254**

Automating build and non-build scenarios

What's new in Team Foundation Build 2012

Introduction

Team Foundation Server 2012 provides a host of new build related features to improve the productivity of Build Engineers. New features have been added and existing tasks made more simple and intuitive. Team Foundation Builds are still predominantly managed from within Team Explorer; however Microsoft has overhauled Team Explorer considerably, aiming to ease the access to the most used features of Visual Studio.

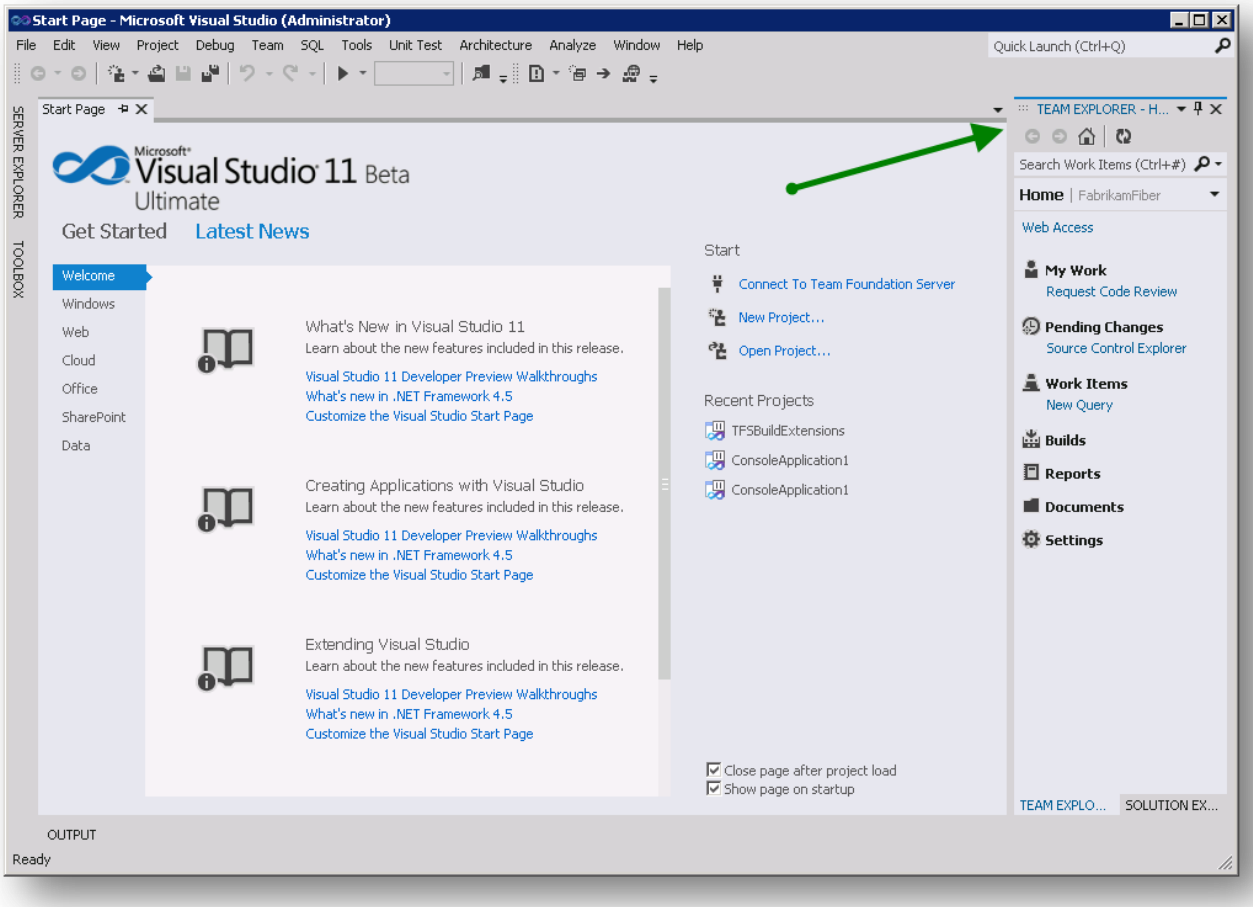


Figure 5 – New Team Explorer

Build Process Templates

The Default and LabManagement build process tempaltes are called DefaultTemplate11.xaml and LabDefaultTemplate.11.xaml in Visual Studio 2012.

The DefaultTemplate11.xamltemplate is very similar to 2010 version with some minor changes:

- Logging verbosity
  - 2010: one argument for both workflow and MSBuild verbosity (Verbosity). Default value is Normal
  - 11: Separate argument for workflow (Verbosity) and for MSBuild (MSBuildVerbosity). Workflow default value is Minimal. MSBuild default verbosity level is Normal
  - As well I was told by PG that diagnostic logging is always put in a file in the drop folder
- New class for TestSpecList argument:
  - 2010: Microsoft.TeamFoundation.Build.Workflow.Activities.TestAssemblySpec

    - 11: Microsoft.TeamFoundation.Build.Workflow.Activities.AgileTestPlatformSpec – this is the default one but you can use the previous as well – what are the differences?
- User friendly display names for some activities – instead of WriteBuildWarning specific name provided
- GetWorkspace – RequestFailed (collecting failed get requests for retry?)
- Not labeling sources warning if label not required
- Exception handler for Try to compile the Project - 2 new activities:
    - GetApprovedRequests(?)
    - RetryRequests(?)
- Running test specific handling for AgileTestPlatformSpec
- Exception handler for Try run Tests - 2 new activities:
    - GetApprovedRequests(?)
    - RetryRequests (?)
- Skipping AssociateChangesetsAndWorkItems if not labeling sources

Builds Tab

Let's start by taking a look at the new Builds tab which allows you to easily manage and orchestrate your Team Foundation builds. It offers richer functionality and smoother operations in comparison to the previous version's tree view - you can set your favorite builds, search existing build definitions, check out running builds, create a new build definition, manage your build security, manage your build quality and manage your build controllers.

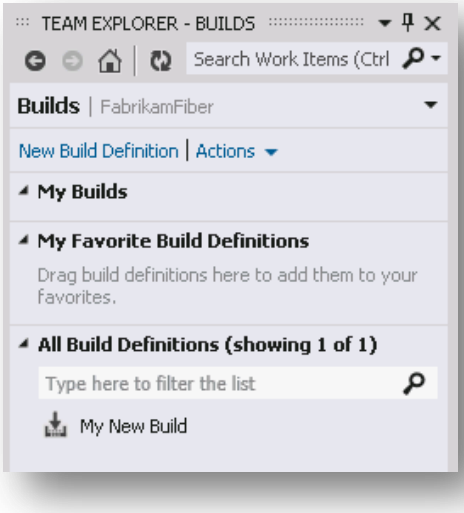


Figure 6 – The new Builds tab

My Builds

My Builds is where you can see what builds are running and which builds you have added to the build queue. Completed builds remain temporarily in this area, showing exactly which builds you have run.

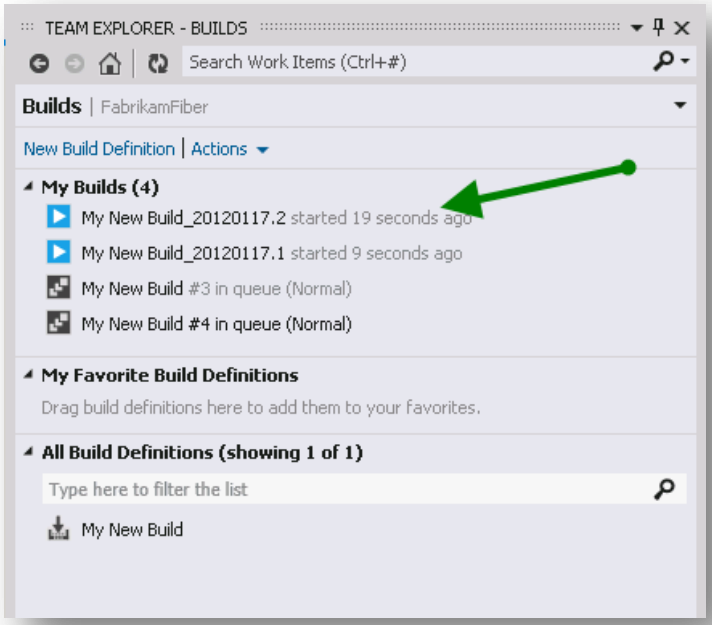


Figure 7 – Working with My Builds

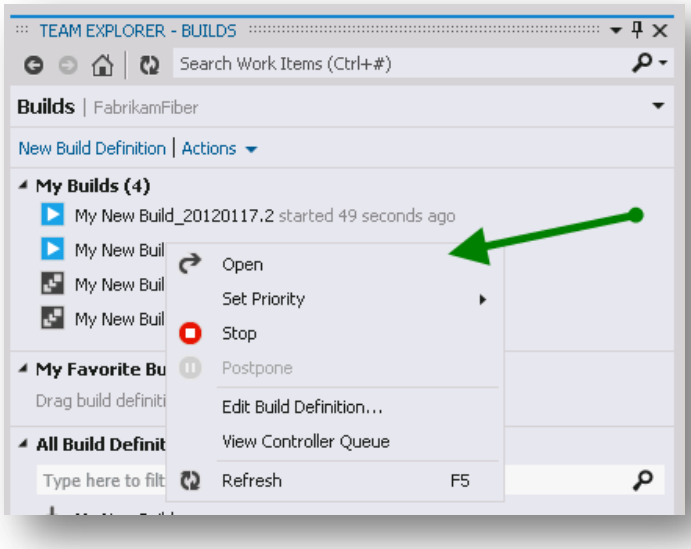


Figure 8 – Options when right clicking over a running build

Retry Builds

A great new feature is the ability to "Retry" builds. This option allows the user to queue a completed build using the same parameters. When you select the "Retry" item, nothing else is needed, the build is queue directly in contrast to the regular process of queuing a build and processing the dialogs. It is possible to "Retry" several builds from the Build Explorer window, or you can do so from the build report window

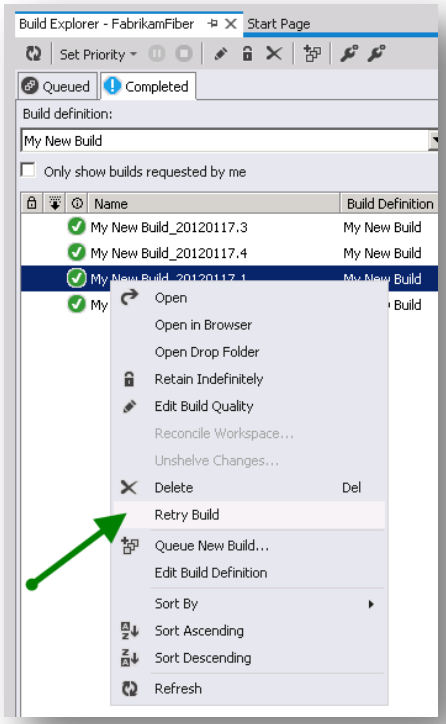


Figure 9 – Retry one or more builds from Build Explorer

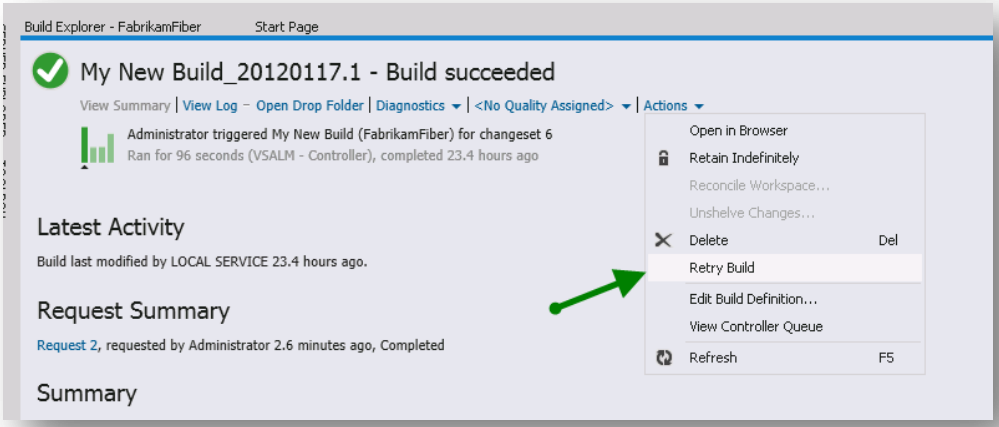


Figure 10 – Retry a build from the Build Report

The “Retry Build” option speeds up the process of re-running a build. It will build as a new build getting the latest code and building it, but using pre-defined parameters.

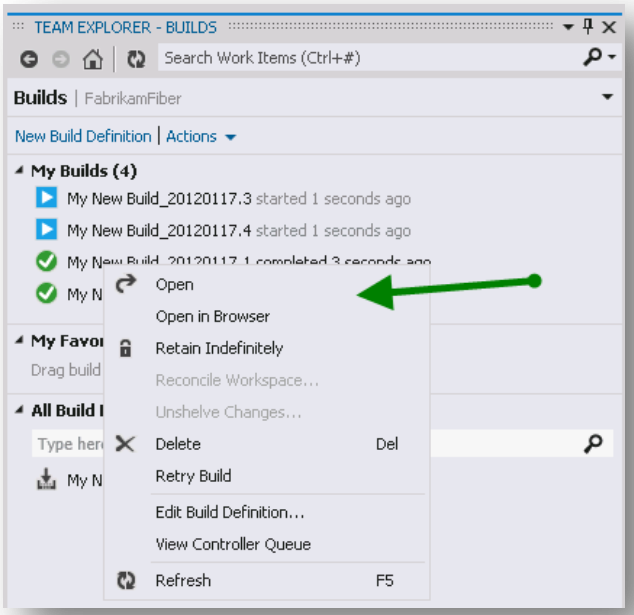


Figure 11 – Options when right click over a completed build

Favorite build definitions

It's possible to select your favorite build definitions and easily have more information about it right in your builds tab. To add a new build definition to favorites just select Add to Favorites from the build definition's context menu.

A nice feature added here is the build history, where you can see your last builds and also click on them to verify it in more details. The build time is now also calculated from when the build started, not queued, so the build performance is calculated more precisely, providing a more accurate report than in previous versions.

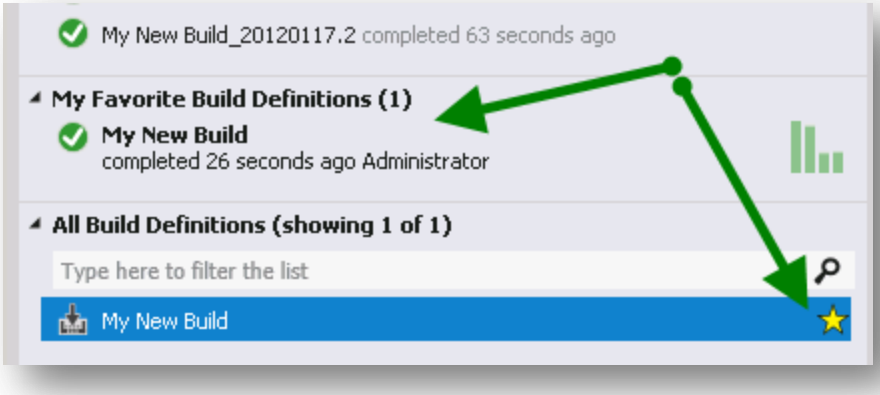


Figure 12 – Favorite build definitions

All Build Definitions

This section is where you find all build definitions of the current team project as well as a built in search box that lets you search for text matches in the titles of all build definitions (no wildcards or Regular Expressions).

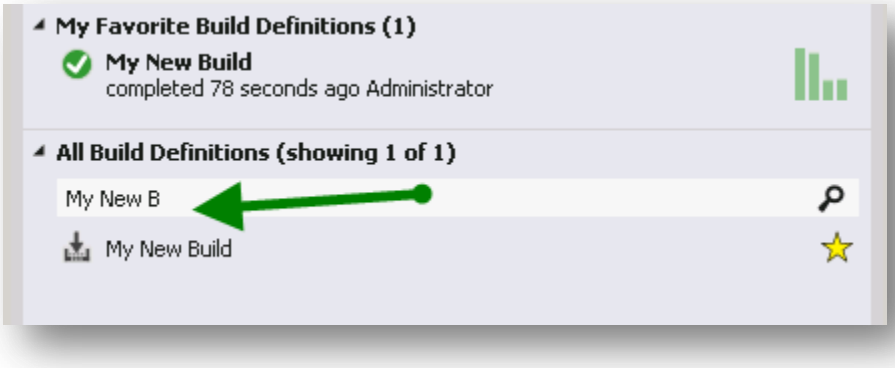


Figure 13 – All Build Definition section and its features

Gated Check-ins based on submissions

To improve the performance of gated check-ins simultaneously building submissions have been included in Team Foundation Build 2012. This means that up to a specified number of submissions will build simultaneously. For example, the picture below shows a build definition that is set up to build two check-ins simultaneously if two check-ins are waiting to be built, otherwise it will start the build with only one check-in normally.

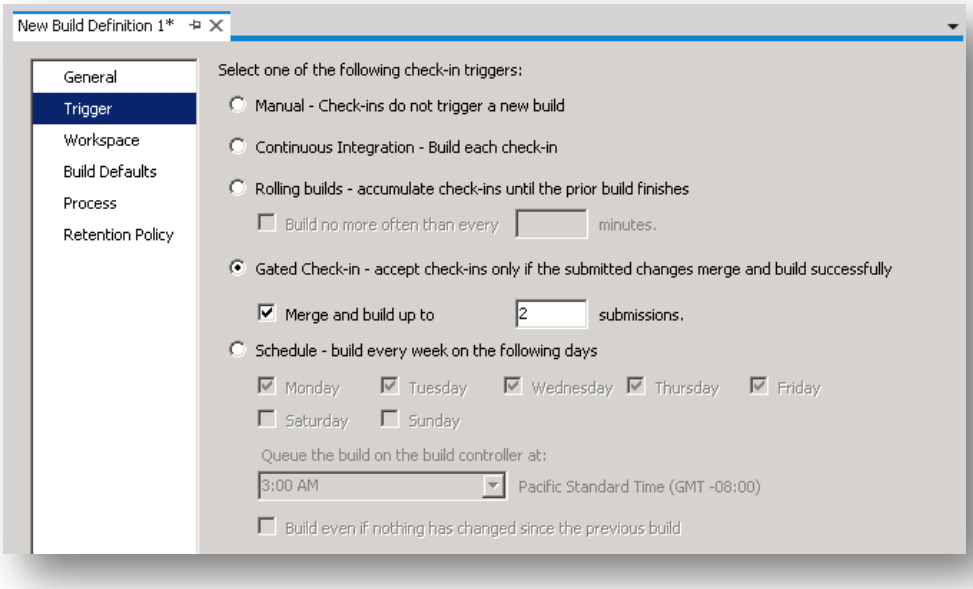


Figure 14 – Setup of a Gated Check based on submissions.

Provisional tabs

In this version, when you want to look at a build, it will open it in a provisional right aligned tab and it is up to the user to promote the provisional tabs to main tabs by clicking the open button or dragging and dropping it. If the tab is not opened then the next build viewed will re-use the tab to avoid clutter.

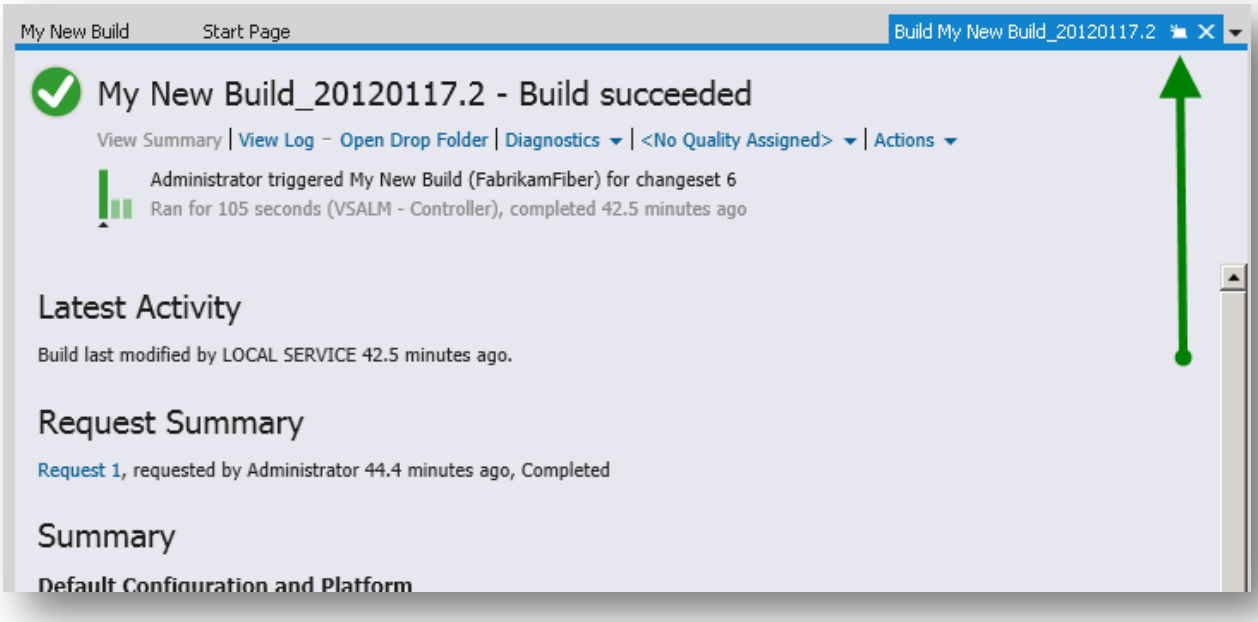


Figure 15 – Provisional Tabs

Diagnostic Logging to file

Team Foundation Server 2012 now logs all builds to file using diagnostic verbosity by default. This can be very handy in diagnosing any build issues.

If you do not configure your build to produce output (Staging location) then the Diagnostics – View Logs menu will be disabled

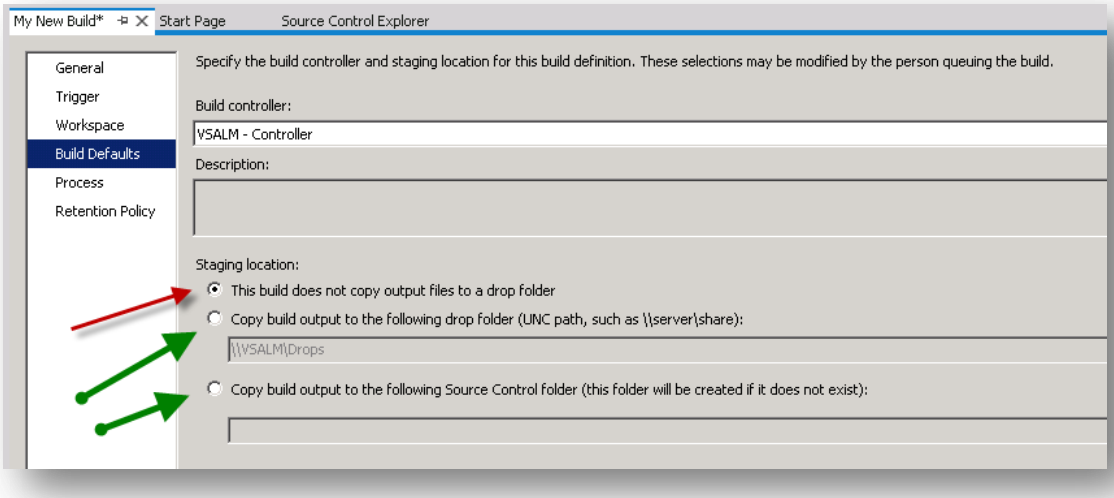


Figure 16 – Build output Staging location

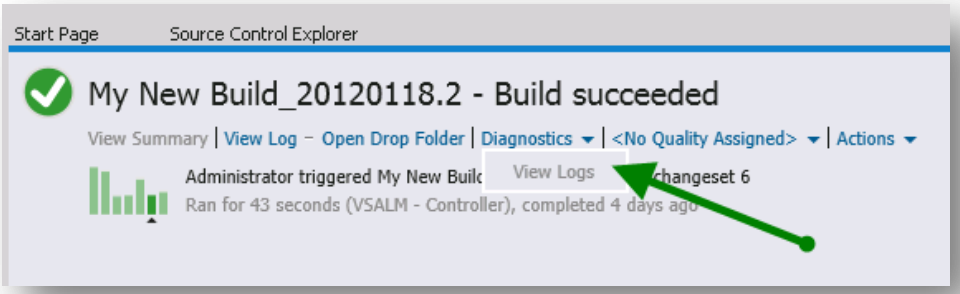


Figure 17 – Diagnostics - View Logs is disabled

Once a Staging location is configured the Diagnostics menu will provide an option to open the logs

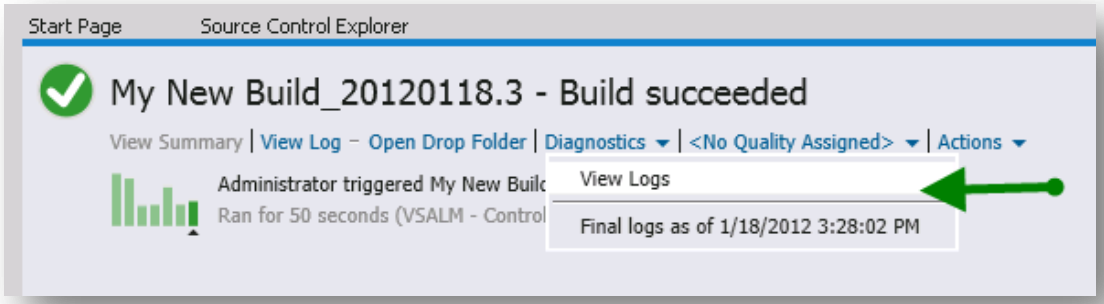


Figure 18 – Diagnostics - View Logs is configured

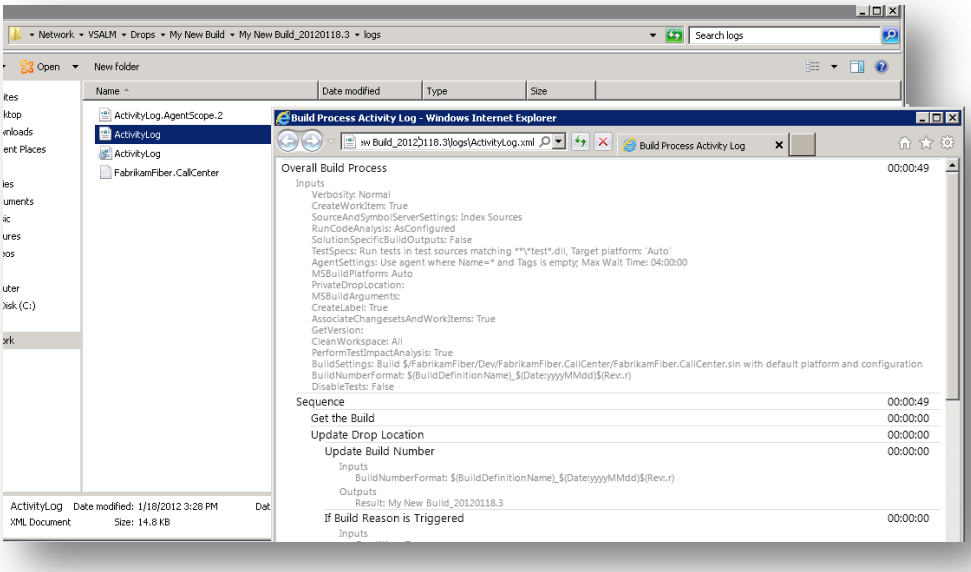


Figure 19 – Diagnostics xml and txt logs.

Drop folder in Source Control

From Team Foundation Server 2012 it will be possible to have the drop folder stored in source control. Given the new Team Foundation Service[3] on Windows Azure we have a scenario where we don't have access to UNC paths, so we must have another option for drop folder in that case.

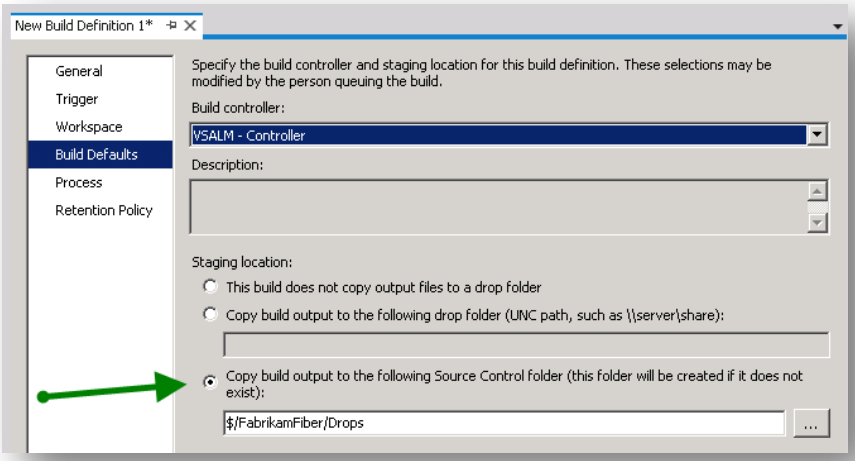


Figure 20 – Store build output in Source Control

Queue processing

Team Foundation Server 2012 comes with a new status for queue processing: Paused. This status allows the user to queue the build but it is kept as paused, until some administrator starts the build. It can be used if, for example, if the build machine needs to be fixed, in that case the queue will be kept and can be started once the build machine is ready, or perhaps in some admin controlled process for initiating and controlling builds.

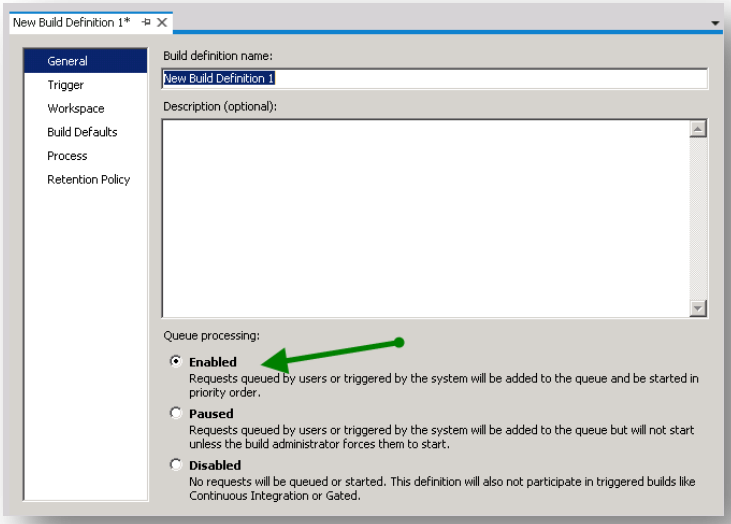


Figure 21 – Queuing processing options

[3] http://go.microsoft.com/fwlink/?LinkID=240175

Running Unit Tests within the Build Process

Prior to Visual Studio 2012 the only unit testing framework that integrated 'out the box' with the IDE was MSTest. To integrate any other testing frameworks with the IDE required the use of third party add-in test runners such as TestDiven.NET or JetBrains ReSharper etc.

In Visual Studio 2012 a new Unit Testing Framework was added that provides an extensible model allowing for different unit testing frameworks to be used, whilst retaining the full range of IDE features. The range of test frameworks support is growing, at the release of the Beta the list of supported frameworks could be found on Peter Provost's blog, "Visual Studio 11 Beta - Unit Testing Plugins List"[4].

By default the Visual Studio 2012 test runner will only run the MStest test, to get it to support other testing frameworks such as XUnit and nUnit you must install adaptors e.g. xUnit.net runner for Visual Studio 11 Beta[5] and NUnit Test Adapter (Beta)[6] either from the Visual Studio gallery or via the Tools –> Extension Manager.

Once these adaptors are installed the Test Runner will be able to discover and run tests written using different frameworks.

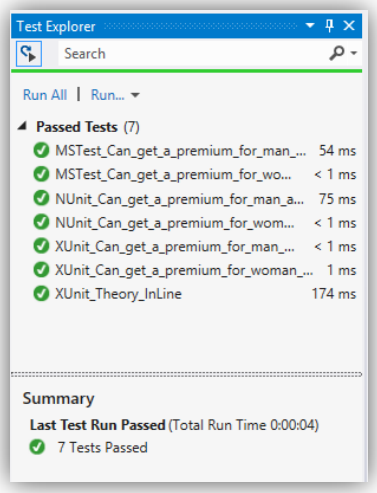


Figure 22 - Test Runner running MSTest, nUnit and xUnit tests.

The Test Runner shows all the unit tests discovered within the open solution. It is possible to set it so that every time you compile a solution the Test Runner triggers (Unit Testing –> Unit Test Settings –> Run Test After Build[7]) and runs the unit test.

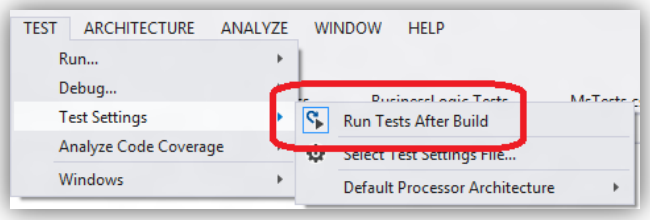


Figure 23 - Enabling auto running of tests

[4] http://www.peterprovost.org/blog/post/Visual-Studio-11-Beta-Unit-Testing-Plugins-List.aspx

[5] http://visualstudiogallery.msdn.microsoft.com/463c5987-f82b-46c8-a97e-b1cde42b9099

[6] http://visualstudiogallery.msdn.microsoft.com/6ab922d0-21c0-4f06-ab5f-4ecd1fe7175d

[7] You can also set this option using the button to the left of the search box at the top of the Test Explorer window.

Running the tests in an automated build

Team Foundation Build 2012 will also only support MSTest tests by default. To support other testing frameworks it too must have the extension adaptors installed. This can be done in [three ways](8)

1. Drop them in the "Extensions" folder
2. Install the VSIX package on the build server
3. Check the extension into version control

If you are using the new TFSPreview hosted build services, as announced at VS Live[9], only the third method is open to you as you do not have access to the Virtual Machine running the build to upload files other than by source control. For consistency using the third method for all build boxes, whether on premises or in the cloud, is perhaps best practice.

Configuring unit test extensions for build boxes

1. Download the required unit test extension packages e.g. xUnit.net runner for Visual Studio 11 Beta or NUnit Test Adapter (Beta) .VSIX packages from Visual Studio Gallery.
2. Rename the downloaded files as a .ZIP file and unpack them
3. In your Team Foundation Server's' source control create a folder **CustomActivities** under the **BuildProcessTemplates** for your team project. The same folder can be used for custom build extensions; hence the same name.
4. Copy the .DLLs from the unpacked renamed .VSIX files into this folder and check the files into TFS. If using NUnit and XUnit you would have a list as below

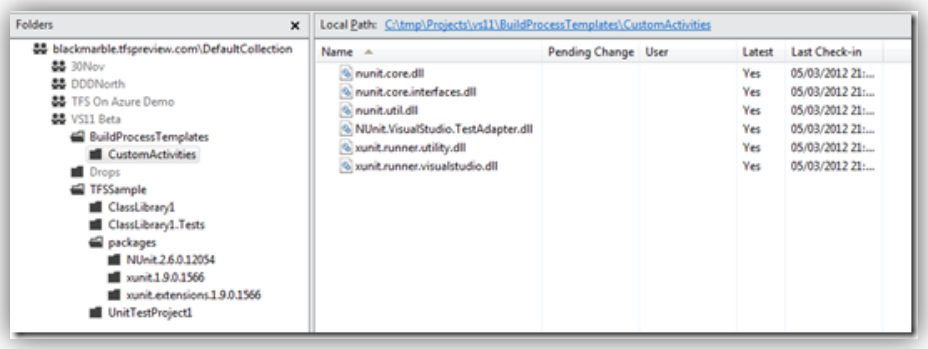


Figure 24 - DLLs from VSIX packages checked into source control

5. In Team Explorer –> Build Hub, select the Actions menu option –> Manage Build Controllers. Set the **Version control path for custom assemblies** to the new folder.

[8] http://blogs.msdn.com/b/aseemb/archive/2012/03/03/how-to-make-your-discoverer-executor-extension-visible-to-ute.aspx

[9] http://blogs.msdn.com/b/bharry/archive/2012/03/27/announcing-a-build-service-for-team-foundation-service.aspx

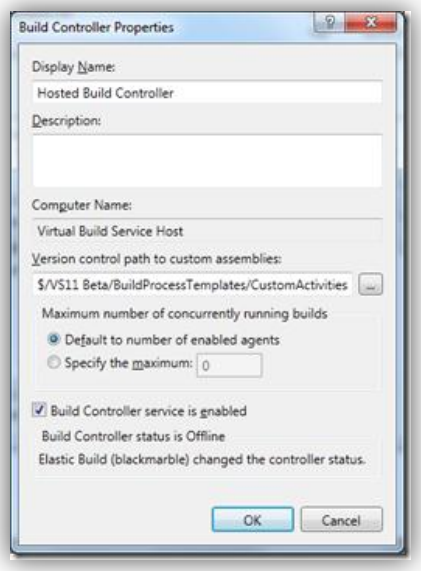


Figure 25 - Setting the custom activities path

6. You do not need to add any extra files to enable xUnit or nUnit tests as long as you checked in the runtime xUnit and nUnit assemblies from the Nuget package at the solution level.
7. You can now queue a build and you should see all the tests are run.

Using TFSBuild.exe

A command-line executable called tfsbuild.exe[10] is provided by Team Foundation Server and being a command-line tool this provides scripting funcationality to assist administrators with managing Team Foundation Server. You can use the TFSBuild.exe command-line tool to start or stop a build or to delete or destroy a build definition. This tool is located in Drive:\Program Files\Microsoft Visual Studio 10.0\Common7\IDE.

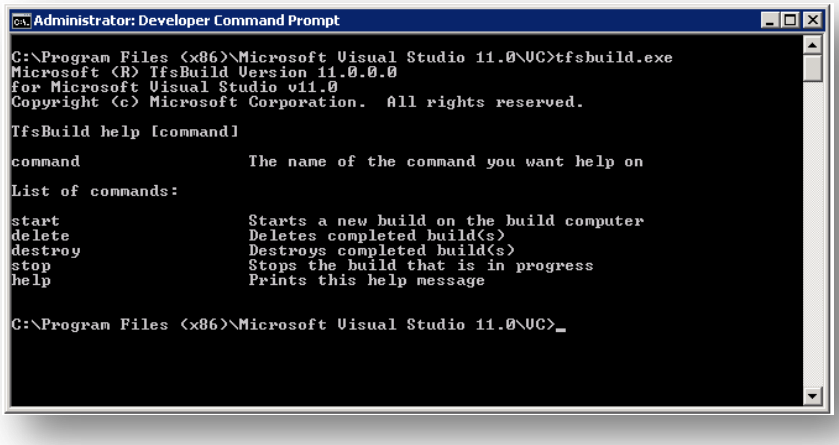


Figure 26 – TFSBuild.exe command-line tool

[10] http://msdn.microsoft.com/en-us/library/bb558974.aspx

Build process template customization

Team Foundation Build is based on build templates, which are used to create build processes that are initiated manually, per schedule or triggered. The following illustration shows three common ways that the build process templates are created, customized, and used to automate end-to-end build, deploy and test processes.

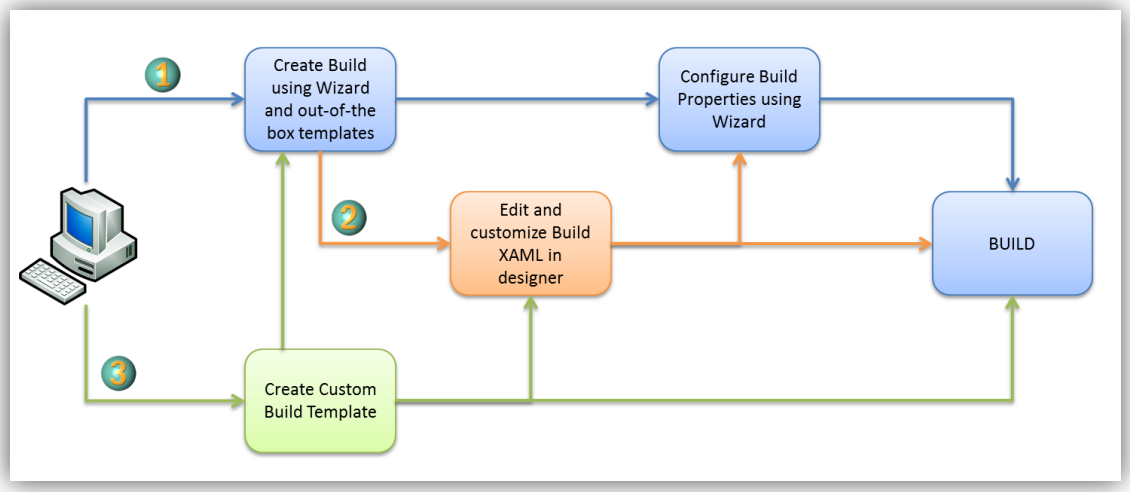


Figure 27 – Build Creation Processes

1. Typically, you create a build using the default build process template that ships with Team Foundation Build and then use the Build Process Wizard to create and configure the build. We will explore the minimal working process template in this section.
2. The second option is to follow the usual way, as in the first option, but to open and edit the process template XAML files in the XAML designer to customize the standard process and incorporate activities that are included with the Team Foundation Build and Windows Workflow Foundation products. Refer to **Build Process Template Customization 101** on page 25of this document, the book "*Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build*[11]", and MSDN: Customizing Process Templates[12] for more information about this approach. Be aware that using this option has an impact on all existing build definitions that use this build process template, so you may prefer to save it under different name.
3. The third and advanced route is to create a build process template from scratch to automate your specialized end-to-end build automation process. Refer to **Build Process Template Customization**, on page 58, **Empowering developers and build engineers with build activities** on page 138, and "*Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build*[13]" and MSDN: Customizing Process Templates[14] for more information about this approach.

Understanding the Team Foundation Build Topology

In this section, we will briefly describe the Team Foundation Build Topology and specifically, how it all fits together with the build definition and its underlying process template.

Each Team Project Collection[15] can queue builds via its build controller[16](s). The build controller[17] can select the build agent[18] from the pool of agents that meets the criteria specified by the build definition[19]. The build definition,

[11] http://go.microsoft.com/fwlink/?LinkID=206999&clcid=0x409
[12] http://msdn.microsoft.com/en-us/library/ms243782.aspx
[13] http://go.microsoft.com/fwlink/?LinkID=206999&clcid=0x409
[14] http://msdn.microsoft.com/en-us/library/ms243782.aspx
[15] http://msdn.microsoft.com/en-us/library/dd236915.aspx
[16] http://msdn.microsoft.com/en-us/library/ms181711.aspx
[17] http://msdn.microsoft.com/en-us/library/ee330987.aspx
[18] http://msdn.microsoft.com/en-us/library/bb399135.aspx

in turn, defines the various attributes of a build, based on the build process template[20] that it derives from. The build agent executes the build process defined by the build definition and typically performs the activities in the build process template; at the very rudimentary level, those activities could include:

1. Getting sources from the Team Foundation Server Version Control
2. Compiling the sources
3. Packaging the compiled binaries
4. Propagating the compiled binaries to the drop server
5. Publishing debug symbol information to the symbol server

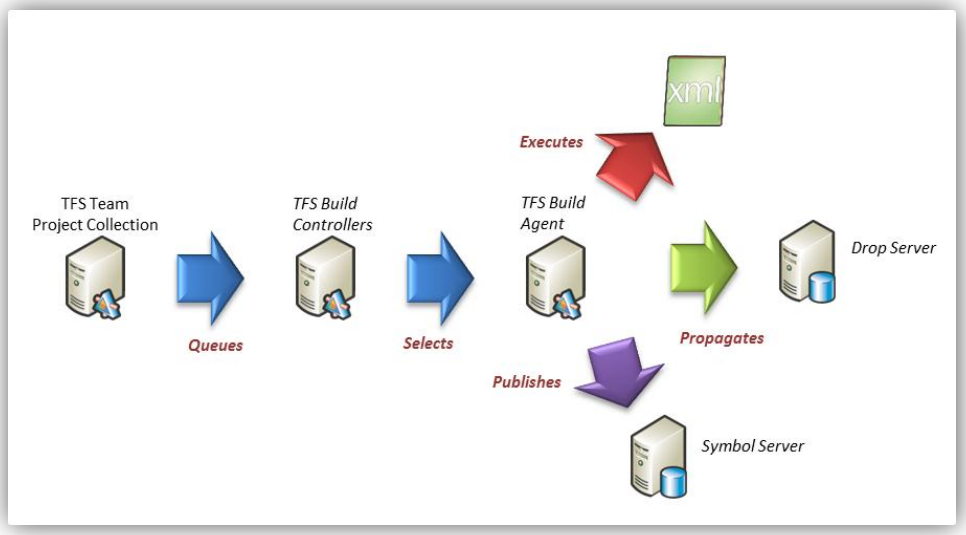


Figure 28 – Build Process

Understand the Key Components of a Team Foundation Build

Build Machine

The build machine is the physical hardware that hosts the build service, which in turn can host a build controller service and/or build agents. The build service is the means by which the Team Foundation Build communicates between the Team Projects and the build machine. Each build machine is dedicated to a given team project collection and may be shared only between team projects that are contained within the given team project collection. A build machine may host both a build controller and build agent, or just the build agent.

Build Controller

The build controller is a Windows service that orchestrates the end-to-end build process; in this role, it handles initialization of the build, allocates build agents, delegates build activities to one or more agents from the agent pool, and finalizes the builds. A build machine can have only a single build controller installed on it and, as such, that build machine and controller belong exclusively to the Team Project Collection they are associated with. Multiple build controllers and build machines running them can be associated with a single Team Project Collection.

Build Agent

The build agent is a Windows service that executes the processor-intensive and disk-intensive work associated with tasks such as getting files from and checking files into the version control system, provisioning the workspace

[19] http://msdn.microsoft.com/en-us/library/ms181715.aspx

[20] http://msdn.microsoft.com/en-us/library/dd647547.aspx

for the builds, compiling the source code and running tests. Each build agent is dedicated to and controlled by a single build controller. Several build agents may be associated with a build controller and, as such, you can increase the build throughput of a given build machine by installing multiple build agents on a machine. While several build agents may be installed on a single build machine, it is not necessary for the build server that is running the build agents to have the build controller installed on it. Build agents on one build server can be associated with a build controller that is running on another server.

Build Definition

A build definition contains the information and the logic needed to execute the build, such as:

- The name of the build – example: *Nightly_Main_Build, CI_Dev_Build, Rolling_Iteration1_Build, Gated_Main_Build*. A build name is usually a mnemonic indicating the type of build, its frequency, and the source code branch for which it was initiated.
- The trigger for the build to decide how the build is initiated – whether manually, or based on a check-in event, or rolling several check-ins within a repeating time interval (once every 15 minutes, every hour).
- The workspace scope for the source code to be compiled – example *$/Tailspin Toys/Main* and the local build agent folder where the source code is to be mapped (for compilation, checking in).
- The build defaults for associating the build definition with the available build controllers and where the final compiled and packaged binaries will be propagated– via the drop folder setting.
- The retention policy for how long the builds on the drop folder should be retained, based on criteria such as build outcome (how the build was triggered, and their pass/fail/incomplete status), how many recent builds need to be retained, and specific artifacts to delete (details, drops, test results, labels). Note that various artefacts will remain in Team Foundation Server; you should look at the Destroy[21] command provided by TFSBuild.exe to completely remove old build records an potentially save space in the database.
- The Build Process Template that determines the build workflow for all the activities the given build definition must accomplish. The next section describes build process template in further detail.

Build Process Template

A build process template is a workflow of build activities and their logical sequencing of dependencies based on Windows Workflow Foundation XAML. Build definitions are created, based on any one of the following templates:

- Default Template (DefaultTemplate.xaml in Visual Studio 2010, DefaultTemplate.11.xaml in Visual Studio 2012) – the template for the most standard build process.
- Upgrade Template (UpgradeTemplate.xaml) – the template for build definitions created in earlier versions (Visual Studio 2005 and Visual Studio 2008).
- LabDefaultTemplate.xaml (LabDefaultTemplate.11.xaml in Visual Studio 2012) – the template for builds running in the **Microsoft Visual Studio 2010 Lab Management Feature Pack** environment.
- Any customized template that can be uploaded to a given Team Project.

Process templates are stored under version control at $/<TeamProjectName>/BuildProcessTemplates and may be checked out, edited, and checked in like any other source control system artifact.

Build Process Parameters

Each process template definition can provide a set of build process parameters available under the process settings from the build definition dialog. Below is a sample of those provided by the Default Template.

[21] http://msdn.microsoft.com/en-us/library/ee794689.aspx

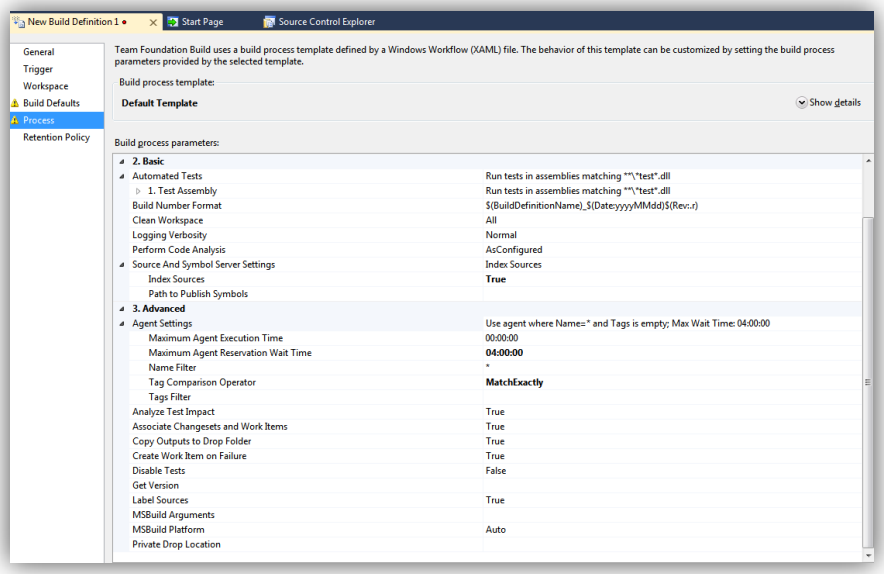


Figure 29 – Default Build Tempalte parameters

In the Default Template parameters are classified under the following criteria:

- **Required**: Parameters that define the items to build such as the solution file in version control, the configuration (debug, retail) and the platforms to build (.NET, x64, x86, CPU etc.).
- **Basic**: Parameters for defining tests to run (via wildcard matching of test binaries such as ***test*.dll), build number formatting, denoting clean or incremental builds (via deletion of workspace artifacts), logging verbosity, performing code analysis, and indexing of symbols and sources.
- **Advanced**: Parameters for selection of agents based on tags, analysis of test impact, association of changesets and work items in a build report, propagating build outputs to the drop location, and specifying the version of source code to build, among other criteria.

Built-in Build Process Templates

Team Foundation Build ships with three build process templates:

- **Default Template** is the default build process template for .NET Framework applications. See **Default Template** on page 39 for more information.
- **Upgrade Template** is a stopgap solution for upgrading from Team Foundation Build in Visual Studio 2008 build processes. See **Upgrade Template** on page 49 for more information.
- **Lab Default Template** provides integration between Team Foundation Build and Visual Studio Lab Management. This template is out of scope for this guidance and is discussed in the Rangers Lab Management Guide[22].

NOTE

The BRD Lite Reference Build Process Template is based on the default template and presents a reference and ready-to-use build process template that may be further customized. See Reference build template embracing the guidance (BRD Lite) on page 258 for details.

[22] http://go.microsoft.com/fwlink/?LinkID=206935&clcid=0x409

Default Template

Before we start delving into the customization of existing Team Foundation Build process templates or the creation of templates from scratch, it is useful to know the default build process template flow and features.

The following illustration, which is one of the quick reference posters included in this guidance to help you visualize the various build process templates, summarizes the process of the default process template:

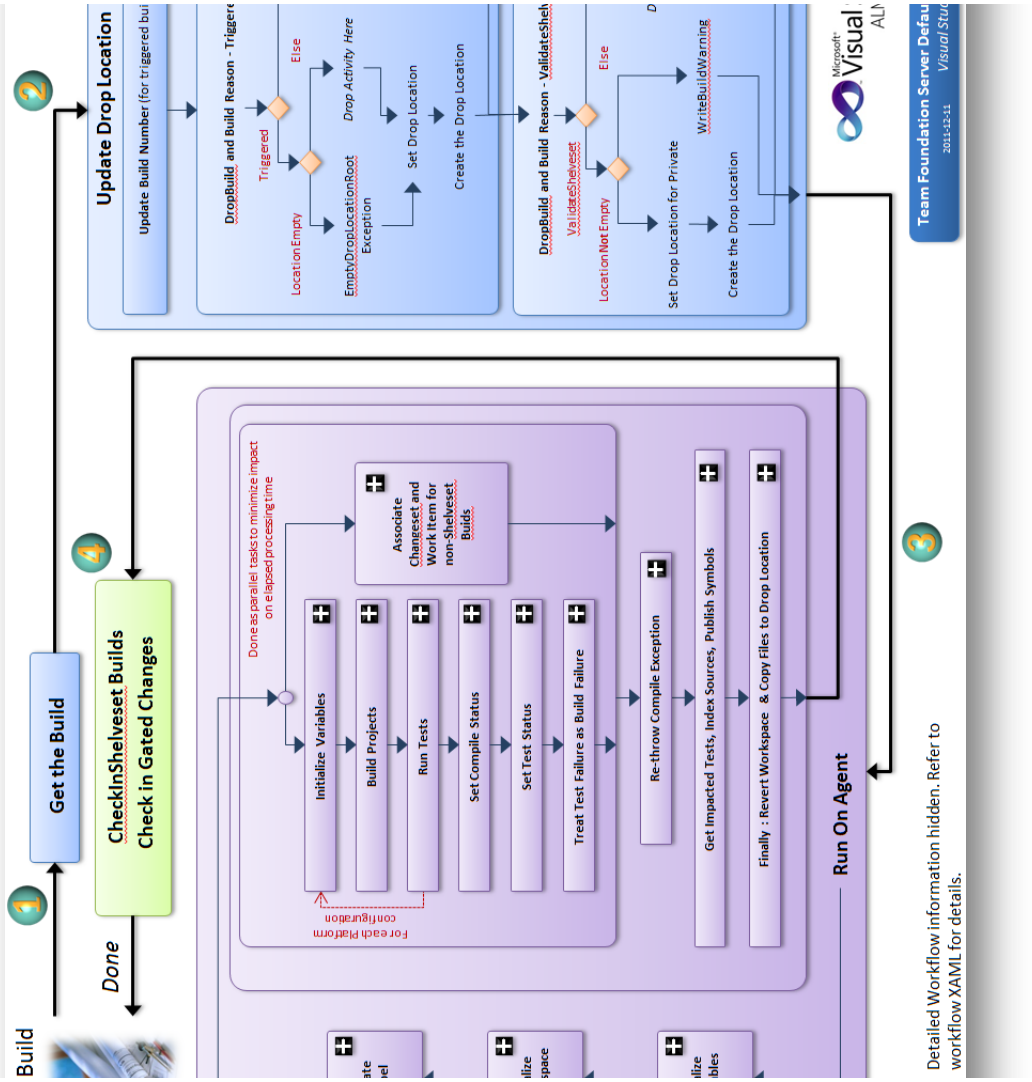


Figure 30 – Default Build Process

Default template is the main process template that should be used when you create Build Definitions in Team Foundation Server. It provides a workflow that covers all standard areas of a build:

- Initializes the build – build number, drop folder
- Cleans workspace and downloads sources
- Compiles projects (using MSBuild) and creates work item on a failure
- Runs automated tests
- Associates work items and changesets with the build
- Performs test impact analysis
- Indexes Sources and Publishes Symbols
- Copies outputs to a drop folder
- Checks in gated changes

Some of the steps can be skipped, depending on the type of build. Some steps can be excluded (enabled/disabled) by some build definition parameters.

This template will often be used as a base for customization, simply because it already contains all the standard steps and although some steps can be customized, others will be easily reused.

When should we use the default build process template?

The default build process template is feature-rich and suited for applications based on the .NET Framework and suited to the majority of Visual Studio based development environments and applications.

If your application is not based on the .NET Framework, has feature requirements that are not covered by the large number of features packed into the default build process template, or requires a custom and specialized build process, then you are entering the realm where using the default build process template might not be suitable.

In essence, the reasons for using the default template include:

- **Wider feature set** – See the feature matrix on page 51.
- **Future support** – Although MSBuild remains the core compilation engine, Windows Workflow Foundation is now the main platform of the Team Foundation Build workflow. It will be enhanced in the future and we can expect strong growth of third party custom activity development
- **Easier configuration of the build definition** – In the upgrade template, many parameters must be provided in the MSBuild script and cannot be easily changed per build run.
- Better **logging** and **traceability**.
- Better **readability**.
- Easier **customization**.

RECOMMENDATION

We recommend that you use or at least start with the default build process template, to ensure that you start with a feature rich, extensible, and most importantly, a tested base.

What are some of the Team Foundation Build activities that we can tweak in the default build process template?

Let us briefly explore some of the key features exposed in the default build process template. You can find more information about these features and other features in the book "*Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build*[23]", MSDN: Customizing Process Templates[24] and Define a Build Using the Default Template[25]

Logging verbosity

Probably one of the most important attributes is the one that controls the level of logging verbosity, which controls the level of logging detail of activity executions and MSBuild within the build process. Even though you probably prefer minimal logging noise in a perfect world, you need verbose logging when you need to explore and diagnose build failures, as shown in the following illustration:

[23] http://go.microsoft.com/fwlink/?LinkID=206999&clcid=0x409

[24] http://msdn.microsoft.com/en-us/library/ms243782.aspx

[25] http://msdn.microsoft.com/en-us/library/dd647547.aspx

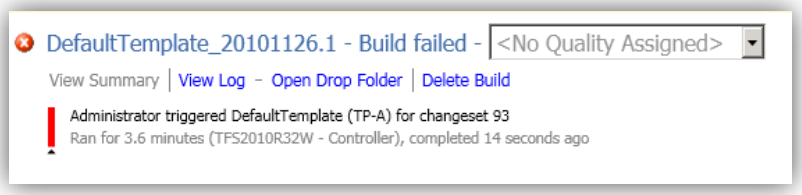


Figure 31 – Build Failure Example … what now?

Modify the logging verbosity by editing the build definition in Team Explorer:

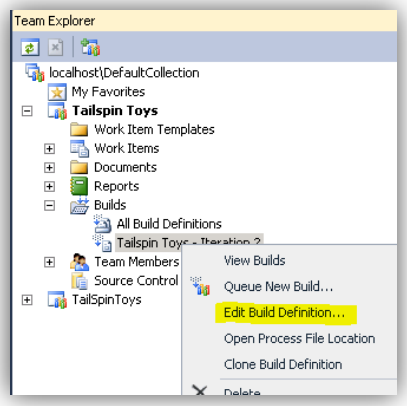


Figure 32 – Editing a Build Definition

Change the Logging Verbosity in the process definition, within the basic category, as shown in the following illustration:

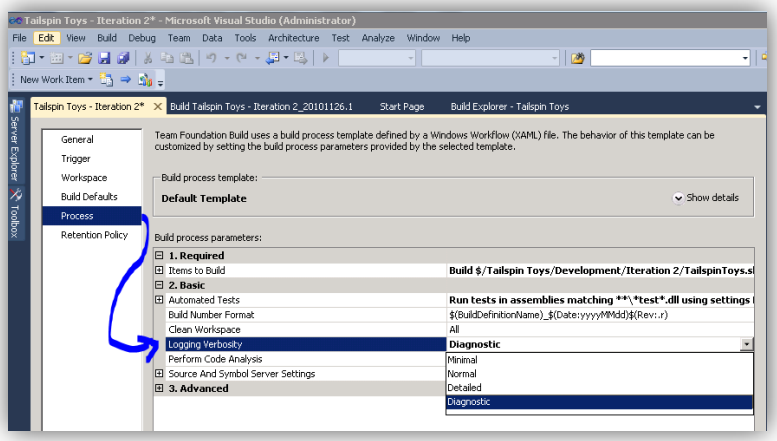


Figure 33 – Changing the Logging Verbosity

The logging verbosity can also be specified when you queue a build. This affects what is logged to the UI and text log file. The actual build process is always logged with verbose diagnositcs to the Staging location.

The following two illustrations demonstrate the difference in information displayed when you select normal versus diagnostic logging levels.

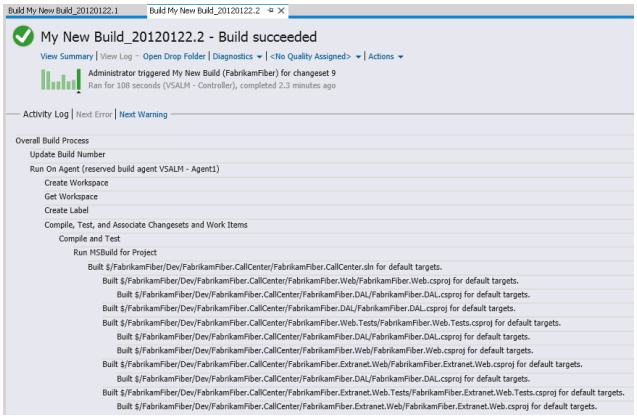


Figure 34 – Example Build Logging using Normal

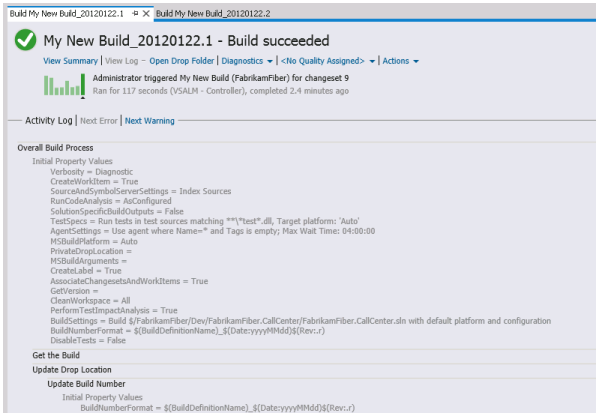


Figure 35 – Example Build Logging using Diagnostic

While it is tempting to select diagnostic, it is important to realize that the more information you capture and log, the slower the overall build process and the higher the resource utilization. As shown in the preceding illustrations, the sample build increased in duration from three minutes and 27 seconds to five minutes and 48 seconds, just by increasing the logging verbosity. More importantly, the storage resources increased from 45KB for the minimal logging build to 3003KB for the diagnostic build, which amounts to a staggering growth of 6573% for this simple sample build.

RECOMMENDATION

Less is definitely better, unless you need to diagnose a build problem.

NOTE

Team Foundation Server 2012 now logs all builds to file using diagnostic verbosity by default. This can be very handy in diagnosing any build issues.

For more information and examples, see “
Logging” on page 147.

Attributes to minimize log noise

Although logging is quite often an effective means of troubleshooting failed builds, it can also be cumbersome. Excessive logging can create “noise” in the build report, making it more difficult to find the most pertinent information. Users can minimize this log noise in a few ways.

Define the overall verbosity of a build’s logging using the Verbosity argument in the build process template. Most commonly, it is set in the build definition on the **Process** tab by specifying **Logging Verbosity**, as shown in Figure 33 on page 41.

Each of the available values (Minimal, Normal, Detailed, Diagnostic) provide gradual levels of logging in the build log. The following table lists these values and their corresponding impact on the build log verbosity.

	Minimal	Normal	Detailed	Diagnostic
Build Errors	✓	✓	✓	✓
Build Warnings		✓	✓	✓
High-Importance Build Messages		✓	✓	✓
Normal-Importance Build Messages			✓	✓
Low-Importance Build Messages				✓
Workflow Activity Properties (inputs/outputs)				✓

Table 1 – Gradual Levels of Logging

If the build definition is using MSBuild, the selected Verbosity value is also passed to MSBuild. Logging Verbosity defaults to Normal.

Another property, which affects the overall log detail, is BuildTrackingParticipant.Importance. Define this property in the XAML for the build process template on each activity. This property effectively specifies the importance of logging verbosity for a particular activity. For example:

```
<mtbwa:DeleteDirectory Directory="[BinariesDirectory]" DisplayName="Delete Binaries Directory"
                       mtbwt:BuildTrackingParticipant.Importance="Normal" />
```

There are four values for BuildTrackingParticipant.Importance: None, Low, Normal, and High. These values are used in conjunction with the defined verbosity level to determine when to log an activity.

	BuildTrackingParticipant.Importance			
Verbosity	**None**	**Low**	**Normal**	**High**
Minimal				✓
Normal			✓	✓
Detailed		✓	✓	✓
Diagnostic		✓	✓	✓

Table 2 – BuildTrackingParticipant.Importance

The BuildTrackingParticipant.Importance property can be set for each workflow activity. If not specified it defaults to Normal. To streamline logging verbosity, consider the following recommendations for setting Importance based on the activity type.

Activity Type	Suggested Importance
Sequence	None – unless the activity groups relevant activities in the build log
Assign	Low
If	Low
ForEach	Low
TryCatch	Low

Table 3 – Activity Type and Logging Verbosity

Three logging activities are available within the build process template as part of the toolbox:

- WriteBuildMessage
- WriteBuildWarning
- WriteBuildError

The WriteBuildMessage activity takes a Microsoft.TeamFoundation.Build.Workflow.BuildVerbosity parameter that defines in which verbosity level(s) the message will be logged. Setting this parameter appropriately helps to keep the build log concise so that it shows only the most pertinent messages.

NOTE

The WriteBuildWarning and WriteBuildError activities will always log output.

SupportedReasons

SupportedReasons is a property in the build process template that defines what types of triggers are supported in the template (BuildReason is a [Flags] enum, meaning multiple selections are permitted). In the default template, all reasons are supported.

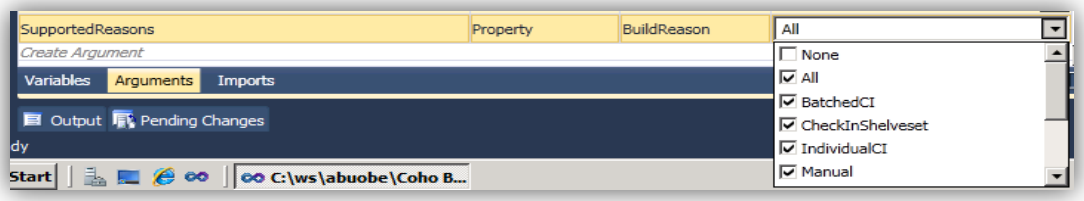


Figure 36 – Supported Reasons in default template

Most of these reasons have a natural alignment with a given option on the **Trigger** tab in the Build Definition dialog.

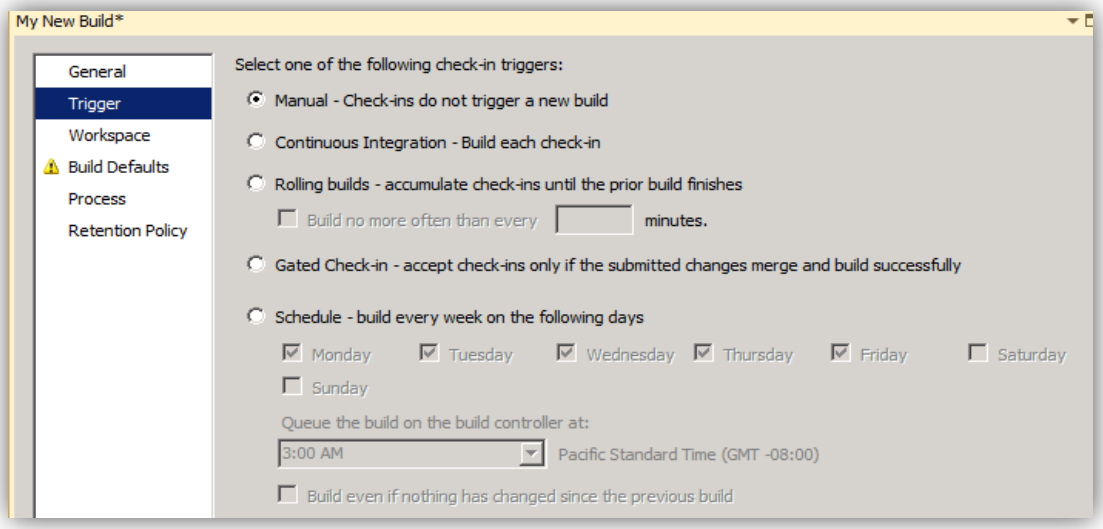


Figure 37 – Build Definition Dialog and Supported Reasons

These alignments are as follows:

Trigger Option	Build Reason	Build Reason Description
Manual	Manual	Build started manually.
Continuous Integration	IndividualCI	Build started due to individual check-in.
Rolling Builds	BatchedCI	Build was started due to batched check-in.
Gated Check-in	CheckInShelveset	Build was started to check in a shelveset.
Schedule	Schedule	Build was started at a scheduled time, only if changes were made.
Schedule (with the "Build even if nothing has changed since the previous build" option selected)	ScheduleForced	Build was started at a scheduled time, even if no changes were made.

Table 4 – Supported Reasons Alignment

There are a few additional BuildReason values that do not directly map to a listed build trigger:

Reason	Description
UserCreated	Build started due to user-defined reason. This value is primarily reserved for triggers from the Microsoft Visual Studio Team System 2005, Team Foundation Server 2005 API.
ValidateShelveset	Build started to validate a shelveset. This is usually the result of a private build[26], also known as a "buddy build". This value is applied when the "Check-in changes when the build completes successfully" box is not selected.

Table 5 – Not Directly Mapped Build Reasons

Finally, there is an "All" value, which allows for any and all reasons to be supported.

Defining "Supporting Reasons" lets you narrow the scope of a given build process template to a more specific purpose. For example, by setting "Supported Reasons" to "Schedule" only, users can only use the template for a scheduled build. If someone tries to use it against a different trigger, the user will see a message like the following:

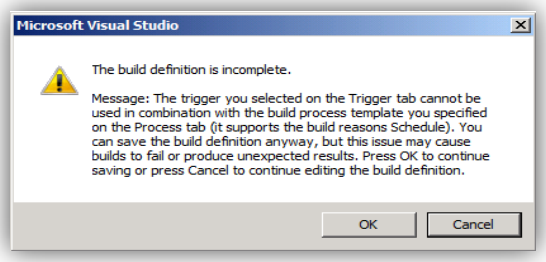


Figure 38 – Build Definition Error Dialog showing reason enforcement

Being aware that the Supported Reasons argument in the build process template can help you define a more concise workflow, and reduce the chance that a build process template could be used for a purpose other than intended. Remember that when you define a minimal template, it might be useful for fast, light builds but it might

[26] http://msdn.microsoft.com/en-us/library/ms181722.aspx

not be appropriate for larger integration or for nightly builds where heavier testing, logging verbosity, and code analysis may be required.

Parameter metadata

Parameter metadata provides the mechanism that enables your build template to display extra details, and configure parameters at the time of build queuing for your custom parameters. A process parameter can have the following metadata:

- **Parameter Name** – A unique name by which it is defined in the Arguments tab.
- **Display Name** – A user-friendly name for the parameter.
- **Category** – A user specified category under which it is grouped and displayed. Default Categories are: Required, Basic, Advanced. Defaults to Misc if not specified.
- **Description** – Provides a meaningful description of what the parameter does.
- **Editor** – Specifies the custom editor to be used.
- **Required** – Specifies if the user must define the parameter on the build definition.
- **BrowseableWhen** – Allows the user to specify when the parameter is viewable. By default, it is viewable in the process tab of the build definition. If it is set to Always, the parameter is displayed on the Queue Build Dialog.

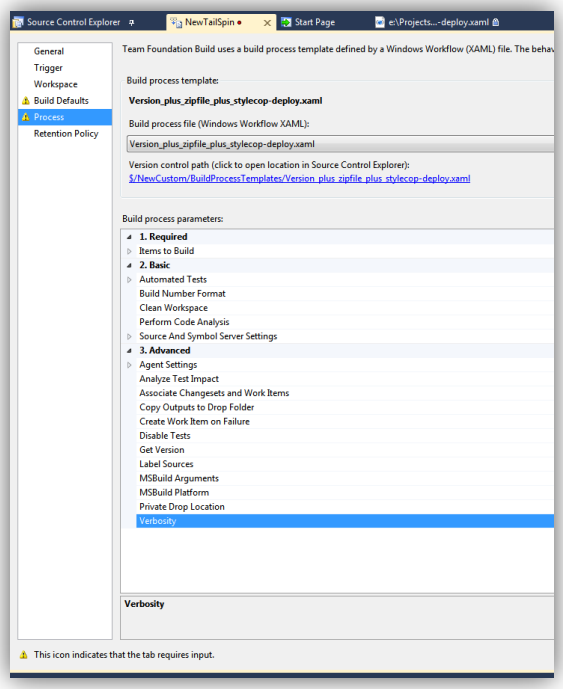


Figure 39 – Build Definition Parameter Metadata

Process parameter metadata is configured by clicking the ellipsis (….) on the metadata row from the **Arguments** tab:

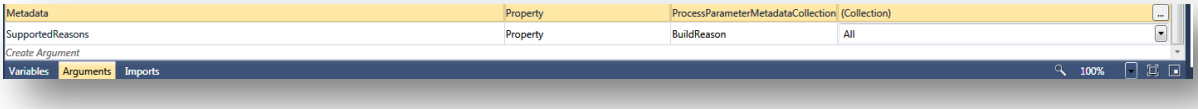


Figure 40 – Build Definition Parameter Metadata Configuration

If you want the Verbosity parameter to always display, type **Advanced** in the **Category** field and then click **Always show the parameter**, as show in the following illustration:

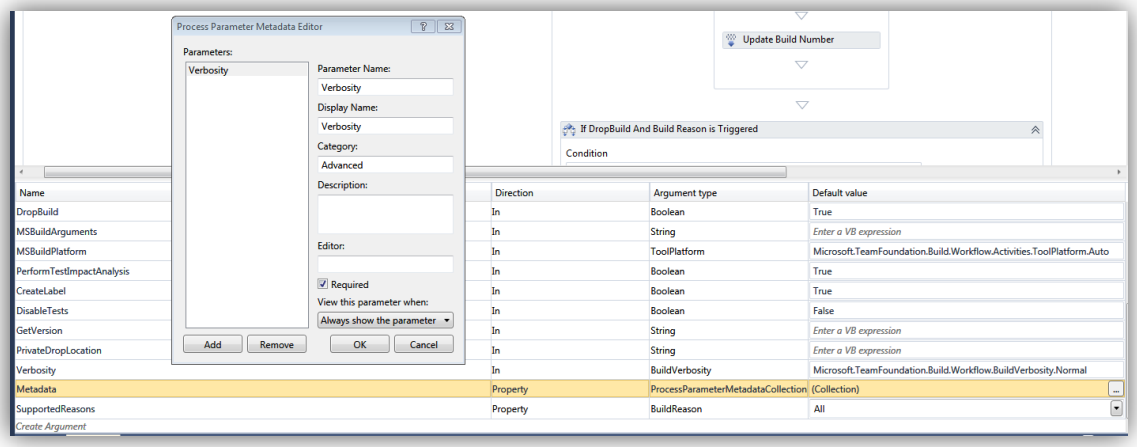


Figure 41 – Build Definition Parameter Metadata Example: Verbosity

When queuing a new build based on the customization shown in the previous illustration, the user must choose from the following options for the Verbosity parameter:

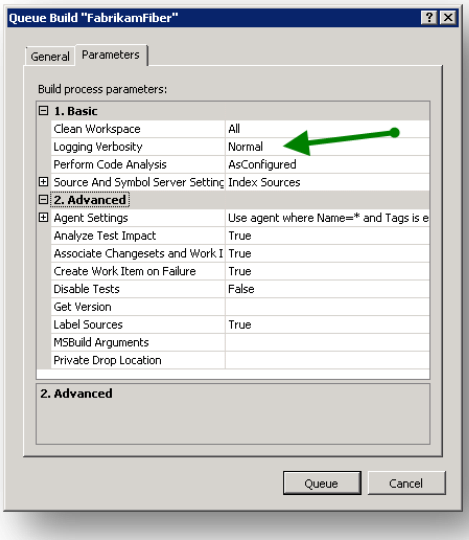


Figure 42 – Build Definition Parameter Metadata Parameters when queuing build

Delegate work to a specified build agent

The majority of the work done by the Default Template runs on the build agent rather than the controller.

By default, the Team Foundation Build system build controller will select one of the build agents that are idle. If you want to control the agent selection process, for example, to redirect specialized builds to specialized build agents, you can tweak the build agent reservation properties of the default build process template.

The following diagram represents a hypothetical build environment with one Team Foundation Build Controller ❶ and four Team Foundation Build Agents ❷, each with a unique name and a set of tags, as shown in the following illustration:

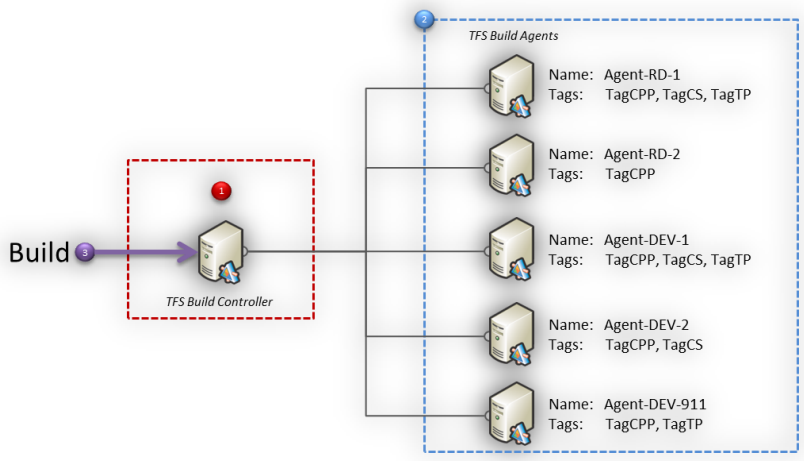


Figure 43 – Influencing the agent reservation process

When we schedule a build, we can define an agent name filter, a set of tags and a tag matching criteria. This controls the way the Team Foundation Build Controller selects and reserves a Team Foundation Build Agent to perform our build. The following table lists a few sample build definitions and shows which Agent would be selected and why:

No	Agent Name Filter	Agent Tags	Tag Match	Agent Selected	Why?
1	Default (*)	Default (None)	Default (MatchExactly)	None - FAIL	It seems that no reservation criteria are defined. However, all agents have tags and the tag match is asking for any agent that has no tags.
2	Default (*)	Default (None)	MatchAtLeast	Any of the agents	No reservation criteria defined. Controller will select the next agent that is idle for the build.
3	*DEV*	Default (None)	MatchAtLeast	Agent-DEV-1 or Agent-DEV-2 or Agent-DEV-911	The agent name filter restricts the selection of agents to any agent that has the word DEV in its name, using the zero or more wildcard (*).
4	Agent-DEV-?	Default (None)	MatchAtLeast	Agent-DEV-1 or Agent-DEV-2	The agent name filter restricts the selection of agents to any agent that has the word Agent-DEV- followed by one character wildcard (?).
5	Default (*)	TagTP	MatchAtLeast	Agent-RD-1 Agent-DEV-1 Agent-DEV-911	The agent tag and the tag match criteria are asking for any agent that has at least the TagTP tag, which three of the agents have.
6	Default (*)	TagCPP, TagCS	MatchExactly	Agent-DEV-2	The agent tag and the tag match criteria are asking for any agent that has an exact match of the TagCPP and TagCS tags.
7	Agent-DEV-*??	TagCS	MatchAtLeast	None - FAIL	The agent tag and the tag match criteria would resolve to three agents, however, the name filter is asking for an agent with the name Agent-DEV-, followed by at least two digits. Agent-DEV-911 would qualify, but fails the tag test.

Table 6 – Agent Reservation Properties Example

Define the agent name and the tags using the **Team Foundation Administration Console** and edit the agent properties as shown in the following illustration. Note that you can also do this from inside Visual Studio by right-clicking the **Builds** node in Team Explorer, choosing **Manage Build Controllers**, and double-clicking a particular agent:

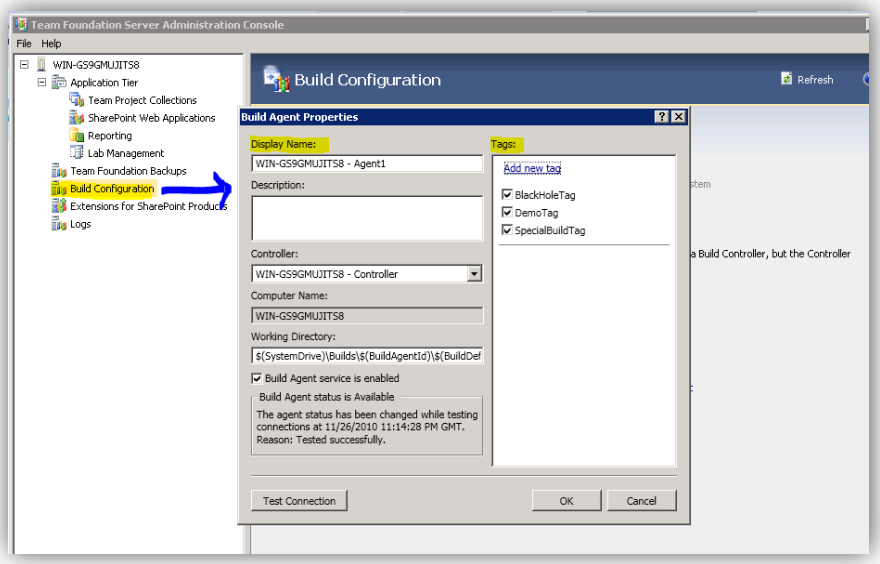


Figure 44 – Changing the Agent Settings for the Build Agent

Change the Agent Settings in the process definition, within the advanced category, as shown in the following illustration:

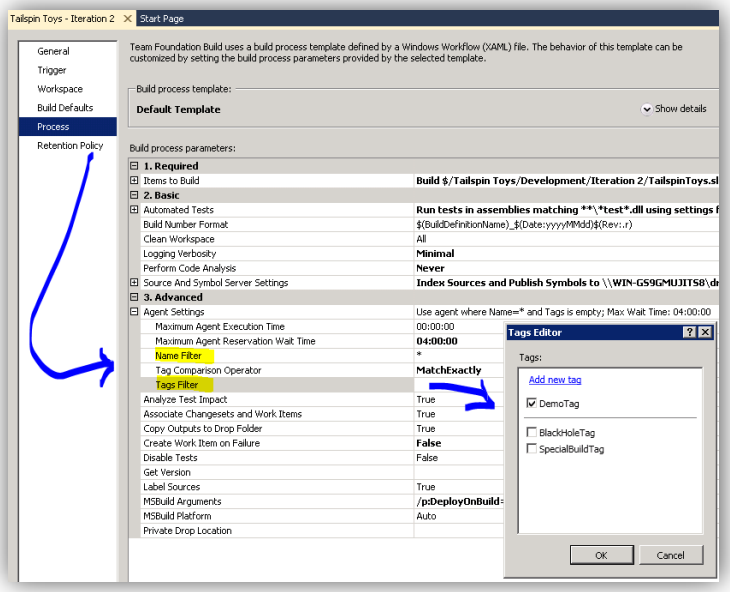


Figure 45 – Changing the Agent Settings for the Build Definition

Upgrade Template

The following illustration summarizes the process of the upgrade process template. It is also one of the quick reference posters that help you visualize the various build process templates and it is included in this guidance..

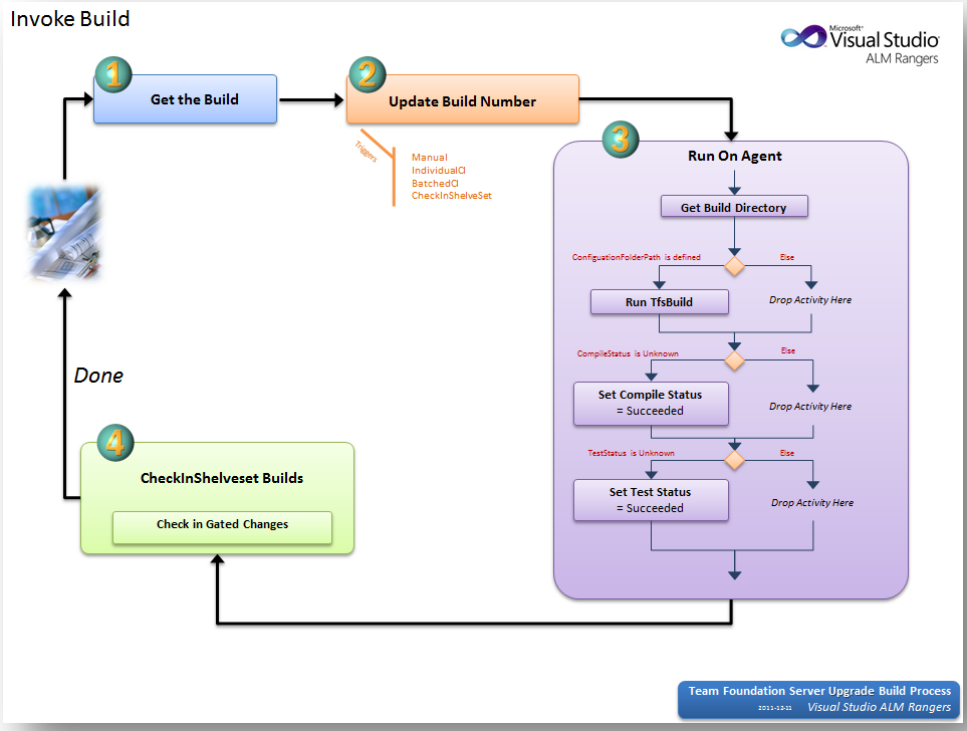


Figure 46 – Upgrade Build Process

The upgrade template is very simple and serves for seamless upgrades of legacy build definitions that are based on MSBuild scripts. Specifically, the upgrade template is for Team Foundation Server 2005 and Team Foundation Server 2008 to upgrade to Team Foundation Server 2010 / Team Foundation Server 2012 definitions that use Windows Workflow Foundation. The upgrade process will encapsulate the original TFSBuild.proj by this workflow in all existing build definitions. The existing build definitions will continue to work with the new build and no manual modifications will be required.

Even after you upgrade, you can still create or modify any TFSBuild.proj script and create a new build definition, using the upgrade template. In addition, a build definition created by legacy Visual Studio versions will automatically use this template. For more details, see http://msdn.microsoft.com/en-us/library/dd647548.aspx. If you have a heavy investment in a complex TFSBuild.proj file, you will probably want to leave it as is. Consisder the Default Template for new build definitions.

The template Workflow contains only a few activities such as Get the Build, Update Build Number, Run TfsBuild for Configuration folder, and Check In Gated Changes.

TfsBuild activity is responsible for most of the build steps and performs them simply by running the original TFSBuild.proj script with MSBuild. These steps cover many of those performed by different workflow activities in the default template, such as getting sources, compilation, running tests, etc.

Reasons to use the upgrade process template

1. Upgrading from Team Foundation Build in Visual Studio Team System 2005 / 2008 – either automatically or reusing TFSBuild.proj manually.
2. Reusing your previous MSBuild scripts investments - MSBuild script infrastructure, custom targets, and tasks.

Default and Upgrade Template Feature Matrix

The feature matrix is based mostly on the configuration properties of the build definition, which allows either change or activation of a build behavior.

Feature	Default Template	Upgrade Template
Projects to build	(definition)	(script)
Configurations to Build	(definition)	(script)
Logging Verbosity	(both)	(both)
Agent Settings	(both)	(both)
Binaries, Sources and Test Results subdirectories configuration	(Workflow)	(definition)
MSBuild Arguments	(both)	(both)
MSBuild Platform	(definition)	(definition)
Automated Tests	(definition)	(script)
Build Number Format	(definition)	(Workflow)
Clean Workspace	(both)	(script)
Perform Code Analysis	(both)	(script)
Source and Symbol Server setting	(both)	
Analyze Test Impact	(both)	
Associate Changesets and Work Items	(both)	(script)
Copy Outputs to Drop Folder	(both)	
Create Work Item on Failure	(both)	(script)
Disable Tests	(both)	
Get Version	(both)	
Label Sources	(both)	(script)
Private Drop Location	(both)	
Private Build	(queue)	(queue)

Table 7 – Default and Upgrade Template Feature Matrix

- Feature is available and easily configurable in UI (text specifies if in build definition or when queuing the build).
- Feature is available through a property that can be changed only in a script or Workflow template that is saved in Version Control.
- Feature is not available without complex customization, such as custom MSBuild targets and tasks, etc.

Migrating build definitions based on Upgrade Template to Default Template

One of the questions after you upgrade your environment from Team Foundation Server 2008 to Team Foundation Server 2010 or Team Foundation Server 2012 might be how you can convert the legacy build definitions to use the Default template. The complexity of this task depends on the amount of custom modifications you have made to the Team Foundation Build script - TFSBuild.proj.

If your build definitions use the default version of the script, it is relatively easy to convert the legacy build definitions. You just have to perform the following steps in the build definition editor:

- Change the build process template from Upgrade to Default.
- For Build Process Parameters in Items to Build, select the same configurations and solutions your script is building. These can be identified as SolutionsToBuild and ConfigurationsToBuild items in TFSBuild.proj.

If the only things you have customized are the values of build properties or items, you can usually change corresponding Build Process Properties in the build process editor. In the table, you can see examples of some mapping between old Team Foundation Build properties to Default Template properties:

Team Foundation Server 2008 Build Property	Team Foundation Server 2010 / 2012 Default Template Process Property
SolutionsToBuild	ItemsToBuild - ProjectsToBuild
ConfigurationsToBuild	ItemsToBuild - ConfigurationsToBuild
RunCodeAnalysis	Perform Code Analysis
RunTests, TestContainer	Automated Tests
IncrementalGet, IncrementalBuild	Clean Workspace

Table 7 – Build Property vs.Default Template Process Properties

If you have used custom MSBuild targets and tasks, you will probably have to customize the Default Template, unless it provides the feature out of the box. The customization chapter provides guidance for customizing the Workflow template, creating custom activities, etc. Some of the common scenarios you could encounter are:

- For BuildNumberOverrideTarget customizations you may use Build Number Format process parameter in some cases.
- For file copy tasks you may use CopyDirectory Workflow activity or xcopy from InvokeProcess Workflow activity.
- For executing other processes (exec task) you may use InvokeProcess Workflow activity.
- You may need to rewrite your custom MSBuild task as custom Workflow activities.

Rewriting a custom MSBuild task as a custom activity can be quite straightforward. Once you setup a Workflow activity project, as described in Build Process Template Customization 101, on page 58, you can quickly rewrite your custom task with the following steps:

- Add you custom task class file to the activity library project.
- Change the base class from Microsoft.Build.Utilities.Task to System.Activities.CodeActivity.
- Apply Microsoft.TeamFoundation.Build.Client.BuildActivity attribute on the class.
- You may want to change public properties to System.Activities.InArgument<T> or OutArgument<T> where T is the original type, and change the MSBuild attributes on the properties to Workflow attributes.
- Add System.Activities.CodeActivityContext parameter to your Execute method.
- In the Execute method extract the parameters from the context to local variables and use these in the method code instead of the original properties.
- Replace other MSBuild specific code by Windows Workflow Foundation specific code (logging, etc.).

Working Effectively with Build Triggers

Choosing a Trigger

One of the fundamental settings to configure for each build is the Trigger which defines when your build will execute. There is no definitive rule for choosing one build trigger over another, instead there are several factors to take into consideration; these include

- Build estate capacity (number of agents available)
- Build duration
- Target Audience
- Check-in velocity

Taking all these factors into consideration, the end goal is the same – ***Maximize the velocity of the development team***.

The longer it takes to find out about a build break, the more significant the impact to the team. It is also worth keeping in mind that the trigger you choose may need to change further down the development process. If your builds grow in number and duration and you see development velocity slowing, it is time to review your triggers and if possible the build estate capacity. Remember that you can only choose one trigger per build definition.

NOTE

If you would like a build to run on multiple schedules or perhaps run a build which has a non-scheduled trigger at a specific date and time, you can use the Windows Scheduler to trigger the builds. This is preferred to duplicating the build definition as it typically involves less maintenance.

The table below offers some *general guidance* assuming that your build estate (budget) is limited. If you have the budget to keep adding build servers, then you are fortunate and can be more adventurous with your choices.

Trigger	Description	Typical Usage Scenario / Build Content	Dev. Velocity Implications	Typical Audience / Consumer
Manual	This is the default setting. Check-ins will not trigger the build.	Builds which are not often required. Unless protected by CI builds, these would most likely run all quality checks and package the code.	If your main build is manual, be careful as if it is broken you may not find out for some time and this may impact your delivery.	Test Team
Continuous Integration	Every Check-in will trigger a build.	Builds that are short, typically designed to build and test areas of high code churn. You may configure these to run only impacted tests[27] for even quicker builds	High check-in velocity and long Continuous Integration builds will cause your build agents to start queuing. You should look to dedicate parts of your estate to Continuous Integration if this is the case or consider Rolling Builds.	Dev Team
Rolling Builds	Check-ins will accumulate until the prior build finishes.	You might choose this option if your build takes	This is a friendly trigger as you can manipulate the time	Dev Team

[27] http://scrumdod.blogspot.com/2011/03/tfs-2010-build-only-run-impacted-tests.html

Trigger	Description	Typical Usage Scenario / Build Content	Dev. Velocity Implications	Typical Audience / Consumer
	Note that you can specify to build no more than every X specified minutes so that a minimum time between builds is provided.	a long time and you have a large number of check-ins that would result in an unacceptable number of builds in the build queue.	between builds to ensure that you have a build running at a sufficient interval to identify breaks early enough to reduce the impact on the development team whilst effectively managing your build estate resource. Note that check-ins will be accumulated and it may not be immediately apparent who broke the build.	
Gated Check-in	Check-ins are placed in a shelveset and are only checked in if the build is successful.	This should be used for short builds which validate the quality of what is being checked in. These could be designed to protect longer running Continuous Integration builds.	This can severely affect velocity if the builds are long and there are a lot of check-ins. A new feature in Team Foundation Build 2012 allows you to simultaneously build a selected number of check-ins, however builds should remain as short as possible so as not to block the sharing of code between colleagues. Also keep in mind that file types which automatically Lock on checkout should be avoided if using Gated Check-ins as the builds will block developers from checking the files out.	Dev Team
Schedule	Check-ins do not trigger a build but instead accumulate until a specified time and date, when they are all built together. If no check-ins have occurred at the given schedule, you may force a built to be taken anyway. If you have the capacity you should force the build as it tests your build infrastructure, or perhaps your build depends on more than just version control content.	Builds of the entire product(s) targeted at the Test Team or some other adopter.	Scheduled builds should be 'protected' by Continuous Integration, Rolling and Gated Check-In builds so that failures are highly unlikely. A failure in a scheduled build will likely impact daily deployments of code. You could think of the other build types, particularly Continuous Integration, as the heartbeat of the development process and the scheduled build as the 'health-check'. Typically these builds will run out of hours and may perform longer running tests.	Test Team or other internal / external group

Table 8 – General build trigger guidance

Build Queue Processing

If a build is disabled then all triggers will be ignored and no builds will be queued. If a build is paused, a feature provided by Team Foundation Build 2012, the builds will be queued until the build is re-enabled or the administrator forces them to start.

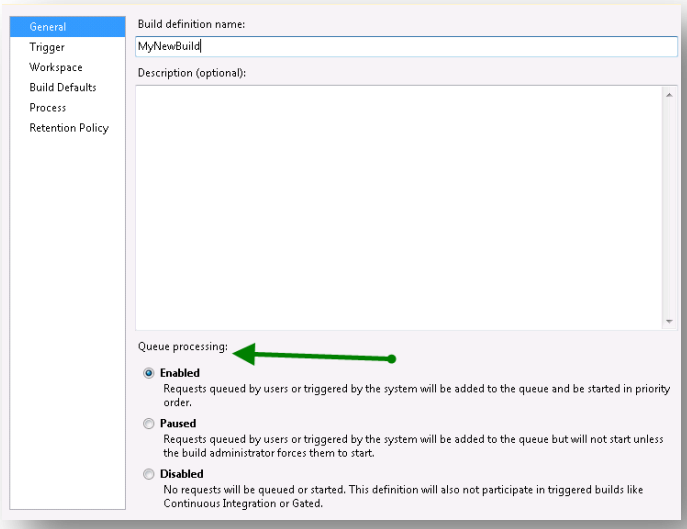


Figure 47 – Build Queue Processing

The reality of Gated Check-ins

It sounds like the perfect solution. You write code, you send it to the build server and if it is good enough the server will accept it and check it in. This is not the most productive way of using this trigger though. Introducing Gated check-ins to a development team needs to happen with care and guidance as the additional dialogs and reconciliation process can confuse those submitting code. The build estate is also a valuable and limited resource. Code should be developed with care and confidently submitted rather than relying on the build server to catch issues. It is an important balance of effort to consider.

Developers and Build Engineers must understand that Gated Check-ins use shelvesets which may lock resources and are not easily consumed by other team members. Gated Check-in builds should be short and ideally lead to Continuous Integration builds which further test the quality in anticipation of a full scheduled build. Note that one scenario where Gated Check-in builds do work well is globally distributed teams. If one team checks-in and goes home they could break the build for a team just starting their day on the other side of the world.

Something worth pointing out is that Gated Check-ins are useful for integration branches. The idea is to use a simple, fast continuous integration build for your feature branch, and use a gated check-in build to validate merges into the integration branch so it stays “clean.”

NOTE

Team Foundation Build 2012 introduces a new option to Gated Check-in which allows you to specify the number of check-ins to batch together in a Gated Check-in build. If the build succeeds then the build will check in each changeset individually. This helps to reduce build queuing.

Trigger Co-operation[28]

You may have multiple builds with different triggers mapping the same folders in their workspace. What happens if you check into a folder which is mapped to a Gated and a CI build? What happens if you check into a folder which is mapped to multiple CI builds? It is worth taking a look at this in a little more detail.

Gated Check-In vs. Continuous Integration Builds

If you have Gated Check-in and Continuous Integration builds mapped to the same Source Control folders, a check-in to those folders will, by default, only trigger the Gated Check-in build. When the Gated Check-in build completes successfully, it will check in the changes with the special ***NO_CI*** comment to prevent any Continuous Integration builds being triggered. Note, if a user decides to override a Gated Check-in build, the check-in will trigger the Continuous Integration build(s) unless they enter ***NO_CI*** somewhere in the check-in comment.

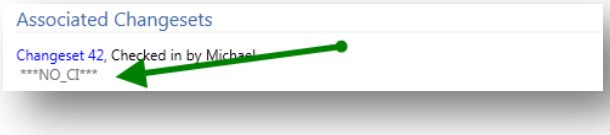


Figure 48 – Gated Check-in changeset comment prevents Continuous Integration build

NOTE

What if you want to use the power of gated check-in to protect your source code repository, but then run a Continuous Integration builds as a result? For example, you may configure a fast Gated Check-in build to perform an incremental get and run a subset of tests to validate the check-in. If the Gated Check-in build completes successfully you then want to move to the Continuous Integration level of builds. Fortunately, there is a way to do this. You need to edit the SyncWorkspace activity in the build template and set NoCIOption="false" on it. Once that is set, your Gated Check-in build will check-in without the ***NO_CI*** comment and your Continuous Integration builds will trigger after the Gated Check-in build completes successfully. If you are working with the UpgradeTemplate, then you can set the `GetNoCIOption` property to false to remove the ***NO_CI*** which is appended to gated check-ins.

Continuous Integration Builds vs. Continuous Integration Builds

What happens when a folder is mapped to multiple Continuous Integration builds? In this case multiple builds are triggered. This is by design as it is perfectly valid for these builds to be performing different operations on the code base. These builds will be queued and handled by your controller / agent configuration independently.

Gated Check-in Builds vs. Gated Check-in Builds

Finally, if you have multiple Gated Check-in builds configured to the same folder(s), you will be presented with an option of choosing which Gated Check-in build to trigger. This isn't a recommended configuration. Making the developer choose which build definition to build in this scenario is rarely desirable as it will cause confusion.

[28] http://mikefourie.wordpress.com/2011/08/30/build-trigger-management-in-team-foundation-build-2010/

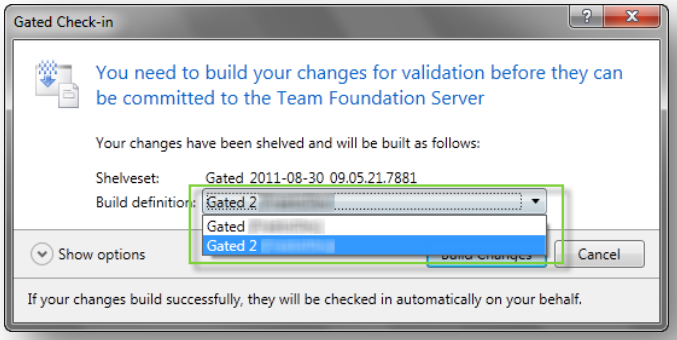


Figure 49 – Gated Check-in dialog

Determining the changeset to build

It's worth understanding how the builds you have configured determine which changeset they will build so you have confidence when developers query whether their work is being built.

Trigger	Changeset Logic
Manual	The latest changeset is used when the user queues the build.
Continuous Integration	The changeset which triggered the build is used.
Rolling Builds	If no Rolling Build is in progress • the changeset the user creates to trigger the rolling build will be used. If a Rolling Build is in progress • a new build will be started as soon as the current build finishes. The new build will be against the latest changeset at the time the build is queued. This may be a later changeset than would be the case if a build was not running.
Gated Check-in	The latest changeset available at the time the build starts (not queue time) is taken and the content of the users' shelveset is added.
Schedule	The build uses the latest changeset available at the time the build is queued.

Table 9 – Determining the changeset to build

Conclusion

Your build servers should be treated as the workhorses of your development process. Managing how they are triggered and what they build are key to maintaining development velocity. There is no point in requiring a large team of developers to test their code to the n^{th} degree and run code analysis and any other long running code quality checks if the code still compiles (i.e. is consumable by other team members). Provide them with a suitable check-in bar to meet and put the effort onto the build servers to report any issues.

When to create MSBuild Tasks versus Windows Workflow Activities

Introduction

For people who have experience with previous versions of Team Foundation Build, the first surprise in Team Foundation Server 2010 is that MSBuild TFSBuild.proj files are no longer used to orchestrate the build process. Instead, each build definition is based on a Build Process Template, which is actually a Windows Workflow Foundation template that is saved as a. xaml file in Team Foundation Server Version Control.

Now the people confused by this change ask questions like, “Is MSBuild dead?” “Does Workflow replace this compilation engine?” “Alternatively, if MSBuild and Workflow coexist, how should we decide what we should customize to achieve our goals?” This chapter will try to answer these questions and provide some guidance on customization using one of the two approaches.

Roles clarification

First, Windows Workflow Foundation does not replace MSBuild. MSBuild remains as the core build /compilation engine. In fact, when you examine the main build template, DefaultTemplate.xaml, you find out that some of the steps such as projects clean up or compilation still call MSBuild using MSBuild activity.

Windows Workflow Foundation provides a higher-level orchestration layer on top of the core build engine, which is MSBuild in the build process templates we include in the box. It makes it possible to do things like distribute a process across multiple machines and to tie the process into other WF-based processes. Windows Workflow Foundation is usually more straightforward and simpler to use for designing many build processes than MSBuild, which can become very complex and confusing with all the imports, target dependences, properties, items, etc. Windows Workflow Foundation covers steps such as build number definition, workspace initialization, compilation, automated tests, copying outputs to drop folder, etc.

Decision making

As a general rule, what happens in Visual Studio regarding your projects configuration and customization relates to MSBuild, whereas what happens specifically on a build machine and not on a developer desktop relates to the workflow. Remember that Windows Workflow Foundation is a template that can be used by many different build definitions for different Visual Studio solutions and projects. The best practice is not to create tight dependencies on a specific Visual Studio project.

As mentioned in Jim Lambs blog http://blogs.msdn.com/b/jimlamb/archive/2010/06/09/windows-Workflow-vs-msbuild-in-tfs-2010.aspx these are the main decision points:

- If the task requires knowledge of specific build inputs or outputs, use MSBuild.
- If the task is something you need to happen when you build in Visual Studio, use MSBuild.
- If the task is something you only need to happen when you build on the build server, use Windows Workflow Foundation unless the task requires knowledge of specific build inputs/outputs.

Legacy builds, targets and tasks

A specific and quite extensive area is using the upgraded build definitions and legacy build targets and tasks. Often, the reasons to migrate these definitions, build targets, and tasks to Windows Workflow Foundation are not strong enough. First, it is not always the best idea to modify something that works pretty well. Second, the amount of work needed to migrate can be quite large, depending on how extensive the MSBuild customizations are.

A related aspect is the expertise of the team. Sometimes, accomplishing the required task can be much faster for the team if they use MSBuild instead of Windows Workflow Foundation simply because the team has been using MSBuild customizations for the last few years.

Typical situations in such cases are the following:

- If your client tool is Visual Studio 2005 or Visual Studio 2008, you need to keep using the upgrade template and customize your TFSBuild.proj file.
- If it is hard to migrate your upgraded builds to Windows Workflow Foundation, the new customizations of the existing builds will usually be done the old way in TFSBuild.proj file.

- If you want to use some custom or third-party build target or task and there is no Workflow Activity alternative yet available, you may use MSBuild activity to run the task or include the task in a project customization.

Examples of Windows Workflow Foundation customizations

Windows Workflow Foundation should be used for customization that relates the orchestration tasks and the properties of the whole Team Foundation Build process. If you see a tight coupling of your Workflow template with a specific project or items in your Visual Studio solution, consider that some modifications are done more easily in the MSBuild script of the project.

The Workflow template customizations can be done on different levels. First, you can modify just parameters of the template in a build definition Process tab. This has an advantage of simplicity and reusability of the template but it is limited by options provided by the template.

Second, you can open the build process template in a Workflow editor and change properties of activities. You can also remove or add standard Workflow activities and activities provided by Team Foundation Build. The changes apply to all build definitions using the template.

The last and most advanced way of customizing the build workflow is to create custom Workflow activities and use them in the template. This usually requires writing some code. It has some deployment consideration as well.

Some typical custom modifications of the build process template are the following:

- Build number – Usually done to include the version. Simple customizations can be done through process parameter.
- Versioning – This may require some kind of project coupling to apply version on all AssemblyInfo.cs files, for example.
- Drop location outputs – If you need different folder structure than the default one.
- Deployment – Any custom deployments of databases, binaries, web application, etc.

Examples of MSBuild customizations

We already mentioned that MSBuild customizations should be focused on a specific project and, unless skipped by a condition, they should apply both in Visual Studio client and in Team Foundation Build.

If you find yourself in the following situation, you are probably not on the right path. You create an MSBuild project file to manipulate other projects, overall outputs, etc., and you never open them in Visual Studio. Additionally, you use MSBuild activity in a Workflow template to call this project file in Team Foundation Build. If this describes what you do, you should consider using custom activities.

There are two main ways to manipulate and customize MSBuild scripts. The first one is using the Visual Studio UI when you define items and properties of your projects. The second is direct modification of the project file xml code in an editor of your choice. For example, you could use unload Project and Edit *.proj in Visual Studio Solution Explorer.

Some of the common customizations are the following:

- Defining project items, properties, references, configurations, etc. This is usually done through the Visual Studio UI.
- Modifying default items properties and metadata, such as default compilation names, etc.
- Adding custom steps before or after standard project targets such as build, compile, generate resources, etc. For example:
    - Calling external tools and custom tasks.
    - Defining the project output structure.
- Using imports to reuse and share your customizations across projects.
- Legacy build scripts maintenance. For example:
    - TFSBuild.proj for upgraded definitions.
    - MSBuild activity for custom and third-party tasks.

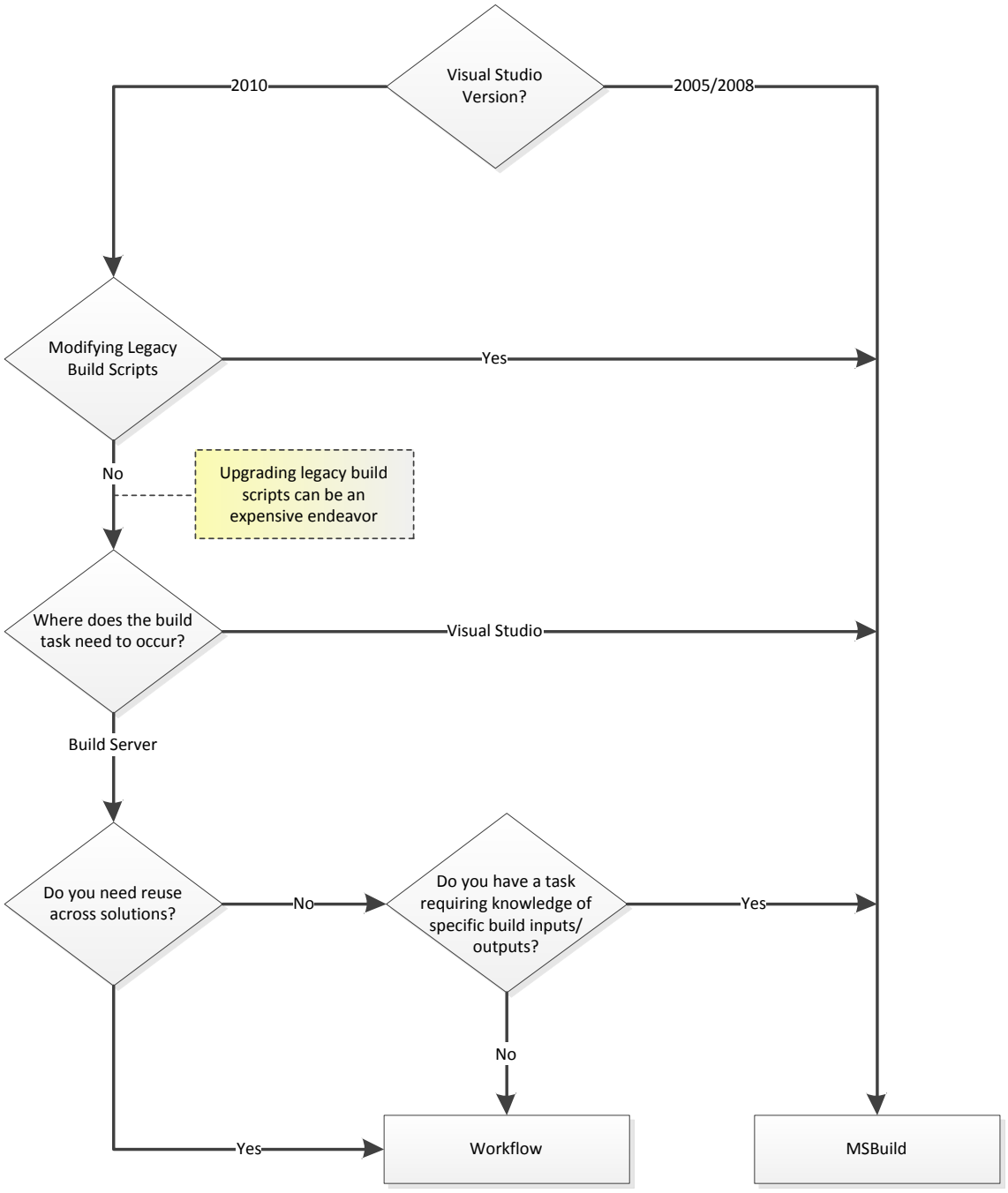


Figure 50 – MSBuild Projects/Tasks versus Workflows/Workflow Activities Decision Chart

Build Process Template Customization 101 Checklist

Checklist

The following checklist guides you through the typical template customization steps:

Step		Description	Page
1	☐	Understand the Team Foundation Build Topology	35
2	☐	Understand the Key Components of a Team Foundation Build	36
3	☐	Create a process template library	62
4	☐	Create a custom activity library	68
5	☐	Define process parameters	69
6	☐	Define Team Foundation Build activities	72
7	☐	Define custom activities	73
8	☐	Define Logging attributes	73
9	☐	Deploy the custom template	74

Table 10 – Template Customization 101 Checklist

Build Process Customization 101

The out-of-the-box process templates, using DefaultTemplate.xaml as an example, contain the most rudimentary build activities orchestrated in the following sequence:

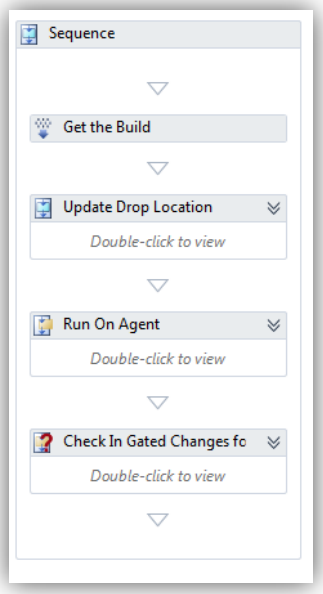


Figure 51 – Build Process Workflow Sequence

These activities and sequence might not meet all the needs of your end-to-end build automation, which might require additional tasks that are specific to your software lifecycle, such as the need for kick-starting test automation by deploying a new build on test machines, or code-signing your binaries, or designating a default drop server. You might find you have to add additional activities to existing templates or create a new process template

from scratch to suit the workflow of your build lifecycle, both of which are easily accomplished by creating a custom build process template.

The **MSDN** article: Create and Work with a Custom Build Process Template[29] describes a very basic approach to customizing a build process template XAML file from existing templates. This approach limits you to activities available in the Team Foundation Build and Windows Workflow Foundation. The following sections will describe an advanced approach for custom templates with custom activities using the development experience in the Visual Studio IDE. The advance approach uses the following key tasks:

1. Creating a process template library
2. Creating a custom activity library
3. Defining process parameters
4. Defining Team Foundation Build activities
5. Defining custom activities
6. Defining logging attributes
7. Deploying the customized template

NOTE

If you want to download a finished solution template with the setup that is described here, you can do that at the Community TFS Build Extensions CodePlex site that is located at: http://tfsbuildextensions.codeplex.com/

The solution template is located at *$/teambuild2010contrib/CustomActivities/MAIN/Templates/CustomProcesses*

Creating a process template library

- Create a Workflow activity library called CustomProcesses in a solution:
  - Click **File**, then click **New Project**.
  - Under **Installed Templates**, choose the language **Type** Visual C#.
  - Choose Workflow under Installed Templates
  - Choose **Activity Library** as the project type
  - In the **Name** field, type **CustomProcesses** for the name of the solution. Click the folder **Location** and click OK.
  - The CustomProcesses solution contains our Workflow library, which is named CustomProcesses.

[29] http://msdn.microsoft.com/en-us/library/dd647551.aspx#1

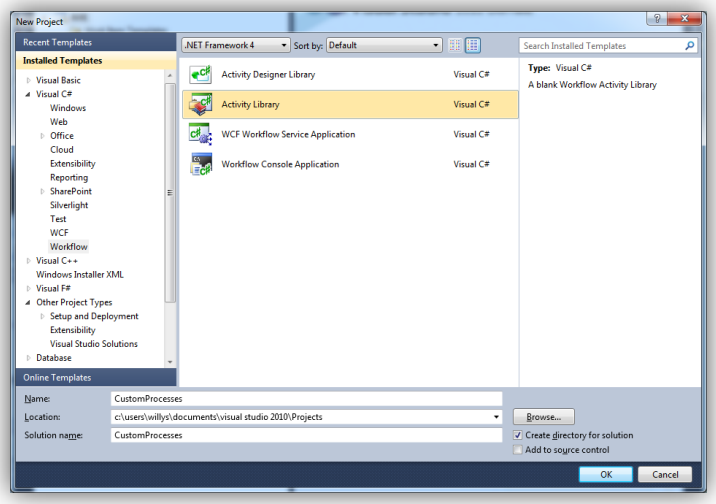


Figure 52 – Create a new CustomProcess solution

- Change the Workflow activity libraries to the .NET Framework Full Profile from the default Client Profile:
  - In Solution Explorer, right-click CustomProcesses project and select **Properties**.
  - Click **Application** and then click the **Target framework** drop-down arrow. Click .**NET Framework 4**.
  - Click **Yes** on the **Target Framework Change** dialog.

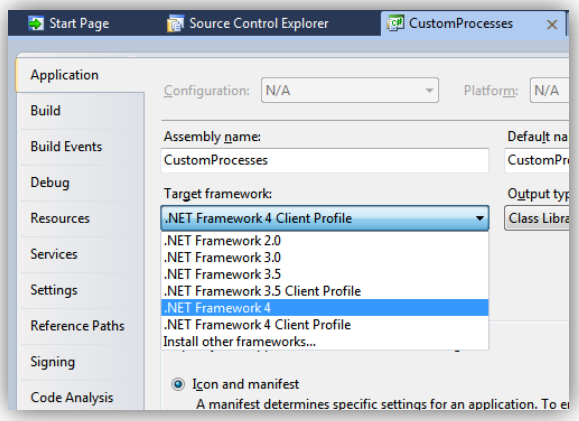


Figure 53 – Change the Target Framework

- Add reference to assemblies to build the process template library:

Assembly Name	Location[30]
Visual Studio 2010	
Microsoft.TeamFoundation.Build.Client	%ProgramFiles(x86)%\Microsoft Visual Studio 10.0\Common7\IDE\ReferenceAssemblies\v2.0\Microsoft.TeamFoundation.Build.Client.dll
Microsoft.TeamFoundation.Build.Workflow	%ProgramFiles(x86)%\Microsoft Visual Studio 10.0\Common7\IDE\PrivateAssemblies

[30] Path is for 64 bit systems. On 32 bit systems it is %ProgramFiles%.

Assembly Name	Location[30]
Visual Studio 2010	
Microsoft.TeamFoundation.TestImpact.BuildIntegration	(%ProgramFiles(x86)%\Microsoft Visual Studio 10.0\Common7\IDE\PrivateAssemblies
Microsoft.TeamFoundation.TestImpact.Client	%windir%\assembly\GAC_MSIL\Microsoft.TeamFoundation.TestImpact.Client\10.0.0.0__b03f5f7f11d50a3a\Microsoft.TeamFoundation.TestImpact.Client.dll
Microsoft.TeamFoundation.VersionControl.Client	%ProgramFiles(x86)%\Microsoft Visual Studio10.0\Common7\IDE\ReferenceAssemblies\v2.0\Microsoft.TeamFoundation.VersionControl.Client.dll
Microsoft.TeamFoundation.WorkItemTracking.Client	%ProgramFiles(x86)%\Microsoft Visual Studio 10.0\Common7\IDE\ReferenceAssemblies\v2.0\Microsoft.TeamFoundation.VersionControl.Client.dll
Visual Studio 2012	
Microsoft.TeamFoundation.Build.Client	%ProgramFiles(x86)%\Microsoft Visual Studio 11.0\Common7\IDE\ReferenceAssemblies\v2.0\Microsoft.TeamFoundation.Build.Client.dll
Microsoft.TeamFoundation.Build.Workflow	%ProgramFiles(x86)%\Microsoft Visual Studio 11.0\Common7\IDE\PrivateAssemblies
Microsoft.TeamFoundation.TestImpact.BuildIntegration	(%ProgramFiles(x86)%\Microsoft Visual Studio 11.0\Common7\IDE\PrivateAssemblies
Microsoft.TeamFoundation.TestImpact.Client	%windir%\assembly\GAC_MSIL\Microsoft.TeamFoundation.TestImpact.Client\11.0.0.0__b03f5f7f11d50a3a\Microsoft.TeamFoundation.TestImpact.Client.dll
Microsoft.TeamFoundation.VersionControl.Client	%ProgramFiles(x86)%\Microsoft Visual Studio11.0\Common7\IDE\ReferenceAssemblies\v2.0\Microsoft.TeamFoundation.VersionControl.Client.dll
Microsoft.TeamFoundation.WorkItemTracking.Client	%ProgramFiles(x86)%\Microsoft Visual Studio 11.0\Common7\IDE\ReferenceAssemblies\v2.0\Microsoft.TeamFoundation.VersionControl.Client.dll
Common	
System.Drawing	%ProgramFiles(x86)%\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.0\System.Drawing.dll

Table 11 – Reference Assemblies

- Set the properties of each of these references to *"Copy Local=False"* to prevent them from being propagated to the output folder.

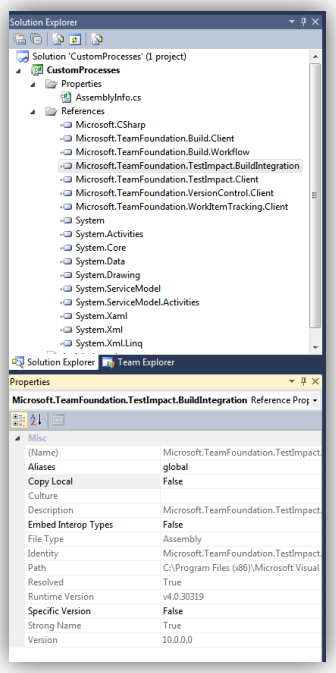


Figure 54 – Solution Explorer (top) and Properties (bottom)

- If you are customizing the default process template (DefaultTemplate.xaml), get the latest copy from the version control folder and add the DefaultTemplate.xaml to the project.

NOTE

When you add more than one custom template based on the default template, you need to change the workflow class name to make it unique:<Activity mc:Ignorable="**sap" x:Class="TfsBuild.Process**". Otherwise, you will get a lot of errors similar to this one when you compile the project: **The type 'TfsBuild.Process' already contains a definition for '_AgentSettings'**.

  - Get the latest copy from $/Tailspin Toys/BuildProcessTemplates/DefaultTemplate.xaml, and copy it to the **CustomProcesses** folder.
  - Click **CustomProcesses** and then right-click to select the context menu.
  - Click **Add** and then click **Existing Item**.

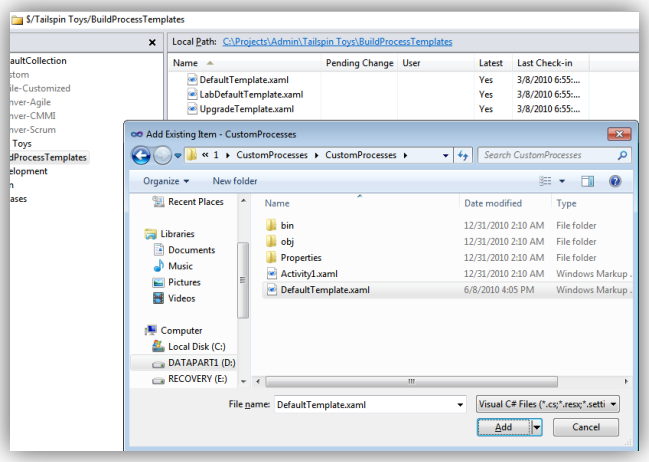


Figure 55 – Add DefaultTemplate to source control

- If you're creating a brand new process template from scratch, add a new activity item to the project:
  - Select and right-click the project and then click **Add**, and then click **New Item**.

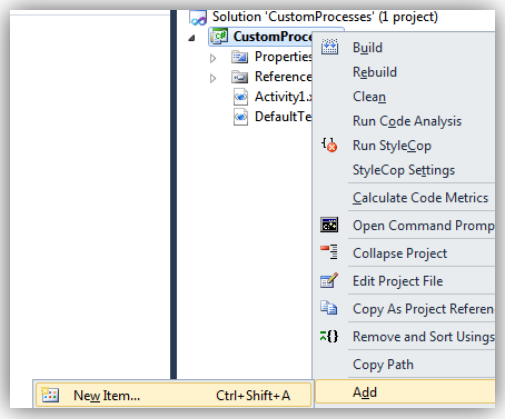


Figure 56 – Add New Item to source control

- Click **Add** and then click **New Item.** Use the selections that are shown in the following illustration. Name it **CustomProcesses** and click OK:

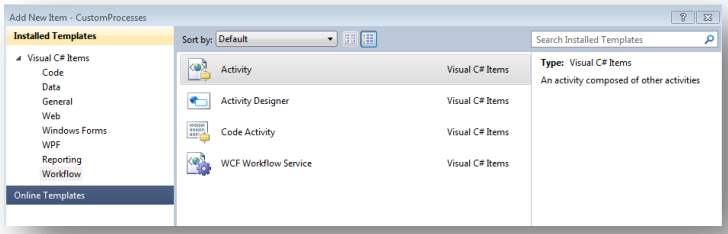


Figure 57 – Add Activity to CustomProcesses

- If you open the DefaultTemplate.xaml, you'll find that it loads the following activities in your toolbox:
  - Team Foundation Build Activities

- Team Foundation Lab Management Activities

- These activities are absent in the new process template because it contains no Team Foundation Build Workflow activity. To add a Workflow activity:
  - Open the new template **CustomProcesses** in the Workflow Designer.

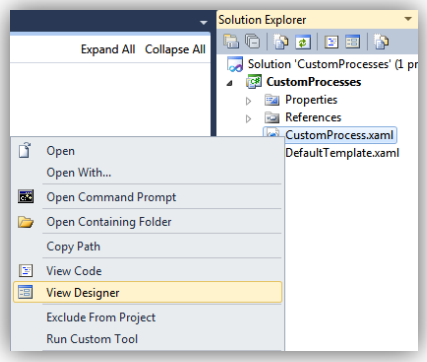


Figure 58 – Open CustomProcesses template in Workflow Designer

  - From the designer surface, select the **Imports** tab at the bottom, and type **Microsoft.TeamFoundation.Build.Workflow.Activities** to add the namespace to existing imports. Save the template.

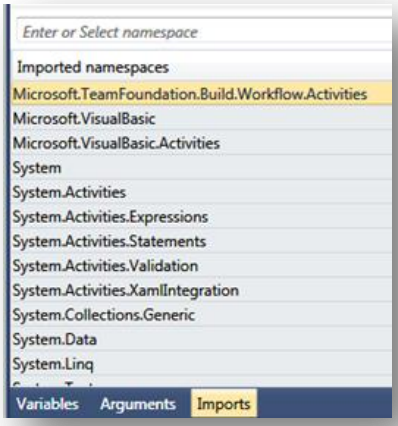


Figure 59 – Add Workflow Activities Import

- The solution we created compiles, but it outputs a DLL. Because we need to output XAML files that must be deployed as build process templates, we will configure the logic for this:
    - Right-click **Solution** and then click **Properties**.
    - Make the following changes to the Post-build event command line in the Build Events tab:

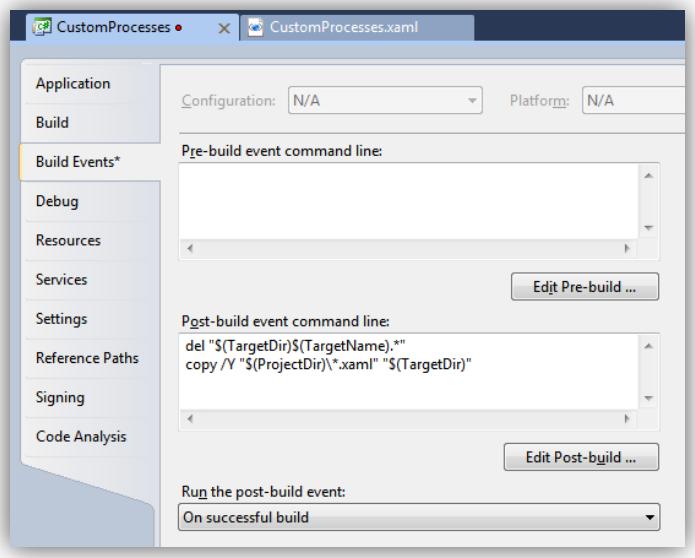


Figure 60 – Post-build event command line changes

    - A rebuild of the solution now correctly outputs the files we need.

```
PostBuildEvent:
  del "d:\rangers\101\1\CustomProcesses\CustomProcesses\bin\Debug\CustomProcesses.*"
  copy /Y "d:\rangers\101\1\CustomProcesses\CustomProcesses\\*.xaml"
          "d:\rangers\101\1\CustomProcesses\CustomProcesses\bin\Debug\"
  d:\rangers\101\1\CustomProcesses\CustomProcesses\\CustomProcesses.xaml
  d:\rangers\101\1\CustomProcesses\CustomProcesses\\DefaultTemplate.xaml
          2 file(s) copied.
```

We have now created a process template library that can be compiled and outputs the XAML files we need to deploy. These steps will allow us to create customized process templates using existing Workflow activities.

Creating a custom activity library

The next step in customizing our build process templates is to develop custom activities by creating a Custom Activity Library.

Although it is possible to add our custom activities to the Process Template Library we just created, this will not work. This is because the Workflow Designer will interpret these custom activities as unqualified references because those custom activities, which are effectively compiled binaries, are not part of the project and will not be resolved because they cannot be found.

To create a Custom Activity Library, repeat the steps provided in the Process Template Library section, except for the steps to define the post-build copy steps. The result is a solution that builds and outputs custom assemblies. You can then reference the custom assemblies from your Process Template Library project by following these steps:

- In Solution Explorer, right-click the **Process Template Library** and click **Add Reference.**
- Click the **Projects** tab in the **Add Reference** dialog.
- Select your Custom Activity Library and click OK.

You can now use any custom activities that you create in your custom process templates.

Defining process parameters

On page 37 we described how process parameters could be configured for each build definition. It is possible to extend the same parameters to apply at the process template level, so that all build definitions based on the given process template could reuse the parameters.

It is important to emphasize the differences between Variables and Arguments – both of which are presented on the designer view when you design your build workflow.

Variables are scoped to the activities they are defined in, and serve to hold values for processing within a sequence.

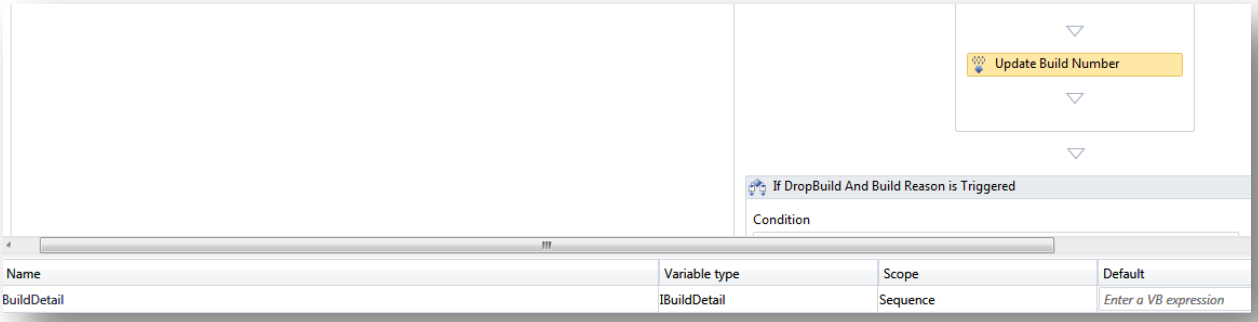


Figure 61 – Variable scoping to activities

Arguments, on the contrary, are much like input and out parameters that are exchanged between activities, and are therefore marked with properties such as In (for input parameters), Out (for output parameters), or In/Out.

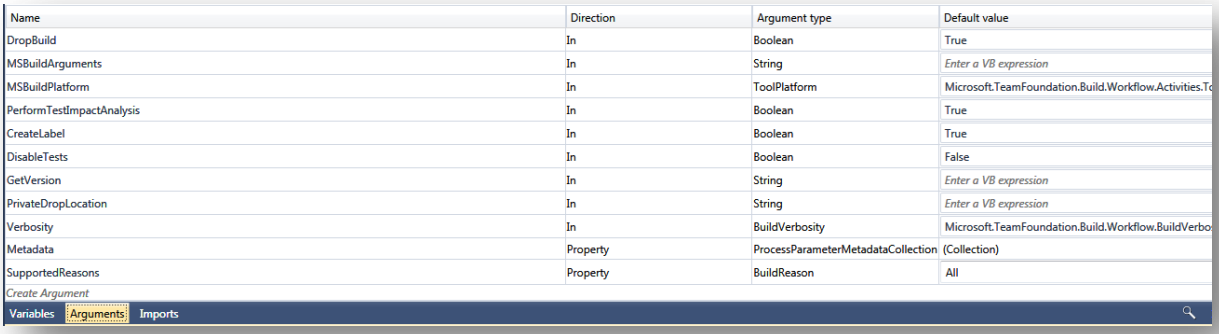


Figure 62 – Activity Arguments

To define a new process parameter to a process template, follow these steps:

- Open the process template solution, right-click **XAML** and then click **View Designer**.

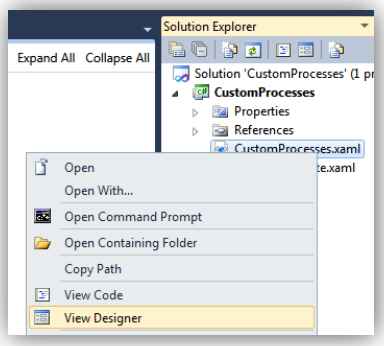


Figure 63 – View XAML in designer

- Click the **Arguments** tab:

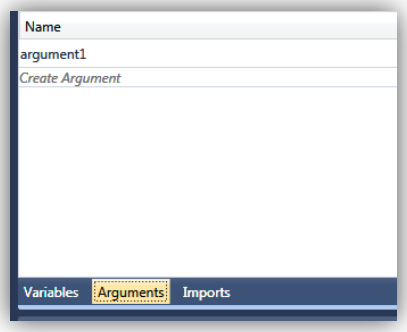


Figure 64 – Select Arguments Tab

- Click **Create Argument**. This will create a new row for you to add list of arguments, define their direction (In, Out, In / Out, Property), Argument Type (Boolean, Int32, String, Object, Array [T], etc.), and a default value.

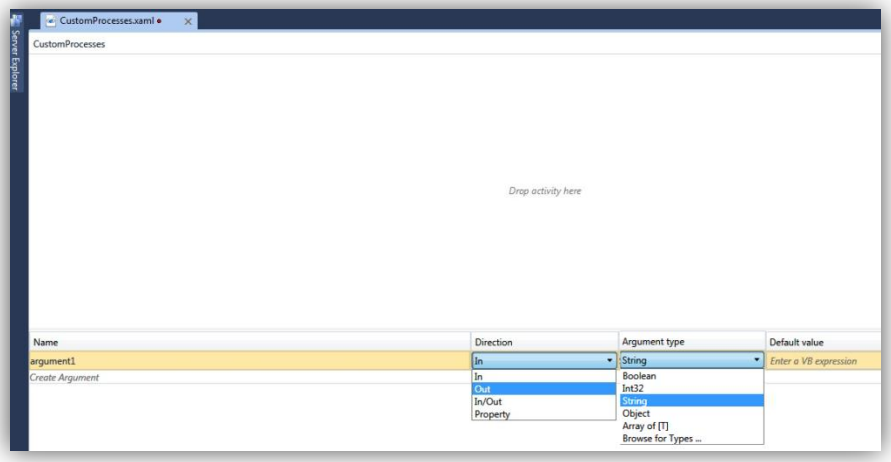


Figure 65 – Definition of arguments

DefaultTemplate.xaml Parameters

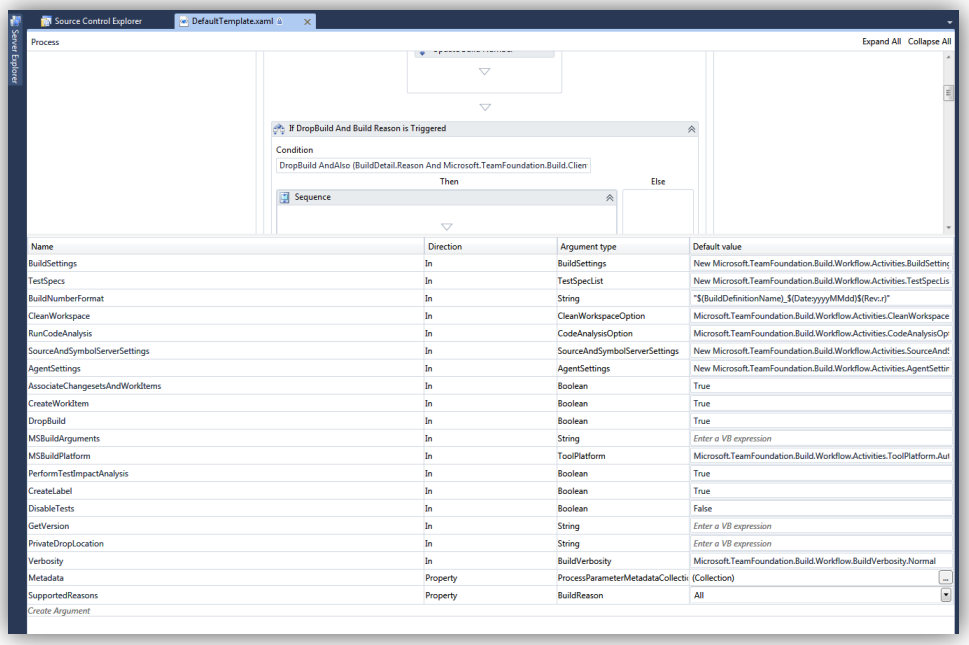


Figure 66 – Default Template Parameters

Metadata

In addition to process parameters, you can define additional metadata to arguments. You can customize the category of a parameter and customize its name and description. You can also define whether that parameter is available when you define or queue a build, whether it is required or not, and which editor to use to edit the values and expressions.

The Metadata argument is created, just like any other process parameter argument, through *Create Argument.* It is edited from an existing process template, as shown in the following illustration:

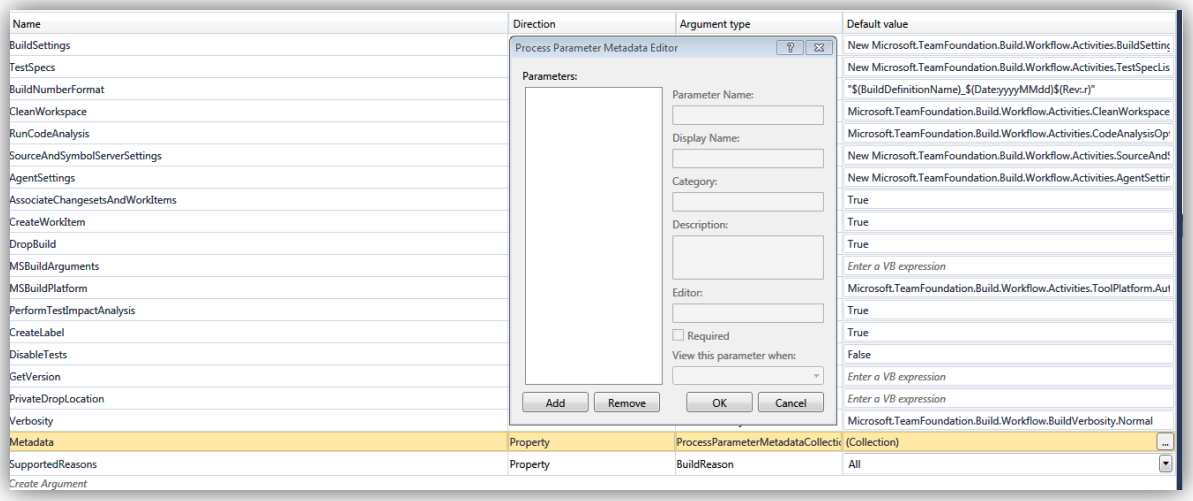


Figure 67 – Metadata

The Process Parameter Metadata Editor becomes available after you click the ellipsis for its default value.

User Interface:

Defining a user interface provides a Windows Form editor to edit the custom types. This Windows Form editor is a lot like the default editors provided by Team Foundation Build for built-in types such strings, integers, Booleans, enumerations, and array types. After you create it, the custom editor is associated with the process parameter using the Process Parameter Metadata Editor.

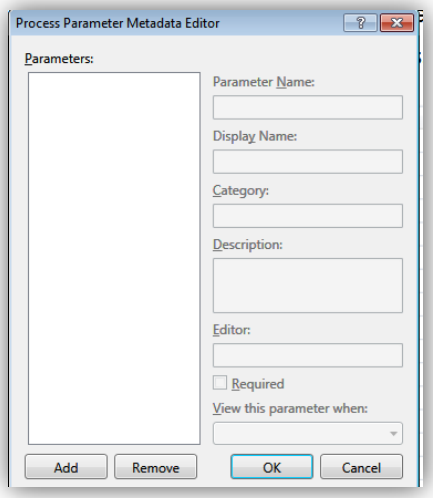


Figure 68 – Process Parameter Metadata Editor

Supported Reasons:

Supported reasons define the triggers that are supported by the process template. The triggers are defined like the rest of the process parameters, as the following example from the DefaultTemplate.xaml demonstrates:

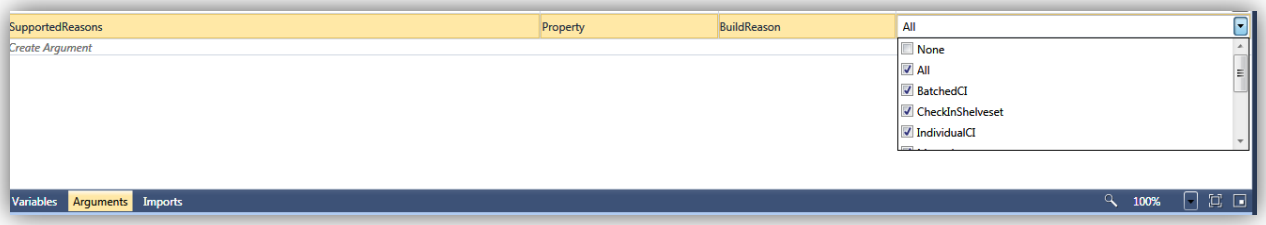


Figure 69 – Supported Reasons

In the following example, clearing the Manual check box disables Manual and Triggered builds, and enables all the others:

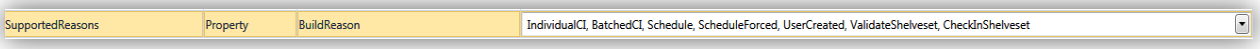


Figure 70 – Custom Supported Reasons

Defining Team Foundation Build activities

Team Foundation Build provides out-of-the-box support for typical build activities. A complete list of these build activities is available at [Team Foundation Build Activities](http://msdn.microsoft.com/en-us/library/gg265783.aspx)[31].

[31] http://msdn.microsoft.com/en-us/library/gg265783.aspx

Defining custom activities

Team Foundation Build custom activities allow you to fill the specific gaps and needs in your build activities that are not addressed by the out- of- the-box activities previously described. Team Foundation Build supports any of the activity base classes supported by Windows Workflow Foundation, including:

- Activity - declarative XAML activities
- CodeActivity
- AsyncCodeActivity
- NativeActivity

Extensions

Another way to customize is to use Workflow extensions to make certain objects available to custom activities without their having to be passed as arguments throughout the Workflow. A custom activity *Execute* method can gain access to the context object through the *GetExtension* method on the context object.

Team Foundation Build extensions are described in detail at: Extending Team Foundation: Build[32]

Type	Available on Controller	Available on Agent
Microsoft.TeamFoundation.Build.Client.IBuildAgent	No	Yes
Microsoft.TeamFoundation.Build.Client.IBuildDetail	Yes	Yes
Microsoft.TeamFoundation.Build.Workflow.BuildEnvironment	Yes	Yes
Microsoft.TeamFoundation.Build.Workflow.Tracking.BuildTrackingParticipant	Yes	Yes
Microsoft.TeamFoundation.Client.TfsTeamProjectCollection	Yes	Yes

Table 12 – Build Extensions

Defining logging attributes

A critical step in customizing build process templates is to define how builds based on it will log their output. This is critical for easy discoverability of issues and to process bugs and failures. Several logging mechanisms are available to make your build logging richer and meaningful.

Verbosity

Logging can be fine-tuned, based on different levels: detailed, diagnostic, minimal, and normal. All these levels can be customized or standardized for your build lifecycle in the process template by changing the Verbosity argument. To add or modify the Verbosity argument, add or edit the Verbosity argument by editing the template in the Workflow designer as follows:

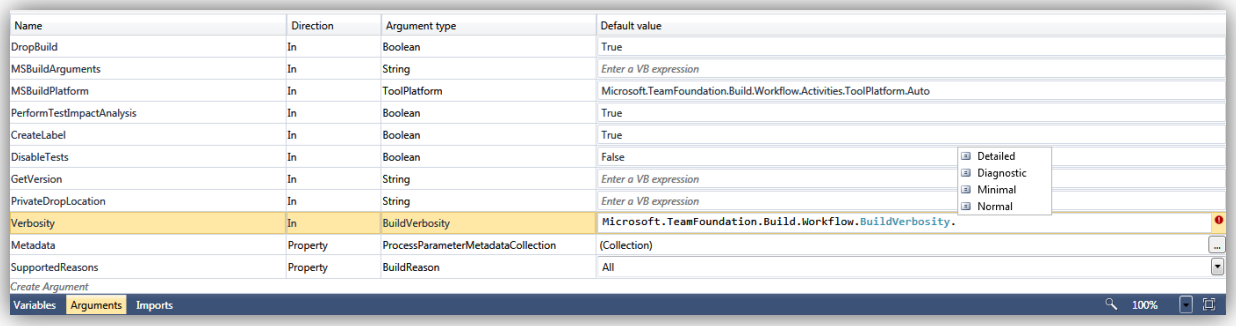


Figure 71 – Modify Logging Verbosity

[32] http://msdn.microsoft.com/en-us/library/bb130146.aspx

Logging Activities

Process templates can be customized further to log additional information via three activities:

- WriteBuildMessage
- WriteBuildWarning
- WriteBuildError

Team Foundation Build ships three activities for logging from within the process template. They can be found in the Team Foundation Build Activities tab in the Toolbox. They are as follows:

- WriteBuildMessage
- WriteBuildWarning
- WriteBuildError

They are available from the Team Foundation Build Activities toolbox:

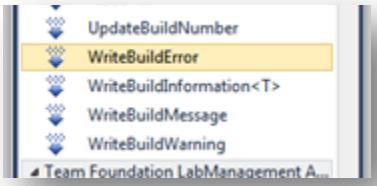


Figure 72 – Logging Build Activities

Programmatic Logging

Additionally, you can programmatically log build activity from custom activities. To log from a custom activity that inherits from CodeActivity or CodeActivity<T>, you can use the extension methods in the *Microsoft.TeamFoundation.Build.Workflow.Activities* namespace.

See the book "*Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build*[33]" or **Empowering developers and build engineers with build activities** on page 138 for more information.

Additional Logging Mechanisms

Additionally, you can customize your process to add hyperlinks, track attributes, and log exceptions.

See the book "*Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build*" for more information.

Deploying the customized template

Discovery of templates

The final step after customizing build templates is to deploy them into team projects where they can be used to create build definitions. Deploying build templates involves two steps:

1. Checking in to version control
2. Creating build definitions based on checked-in templates

$/<TeamProject>/BuildProcessTemplates is the default location for maintaining build process templates under version control, so it may make sense to continue the practice by checking into the default location. As an extension of this practice, you might consider adding a folder structure that reflects your branching strategy for source code, especially given that build processes change as code progresses through your development lifecycle into production. For example, you might have less stringent exception handling, or you might require additional logging and verbosity that you would like to enforce in the development stages through a process template.

[33] http://go.microsoft.com/fwlink/?LinkID=206999&clcid=0x409

Conversely, you might want to enforce more stringent compilation switches and enable code-signing processes as defaults for production builds. Again, these may require a customized process template.

When you have decided on a folder/branching structure for storing your process templates version control, it is easy to create build definitions based on them by making them aware of their location in version control. The following steps describe how to do this:

- Create or edit a build definition.
- On the **Process** tab, click the arrow next to **Hide Details** to toggle to **Show Details**. Click **New**.

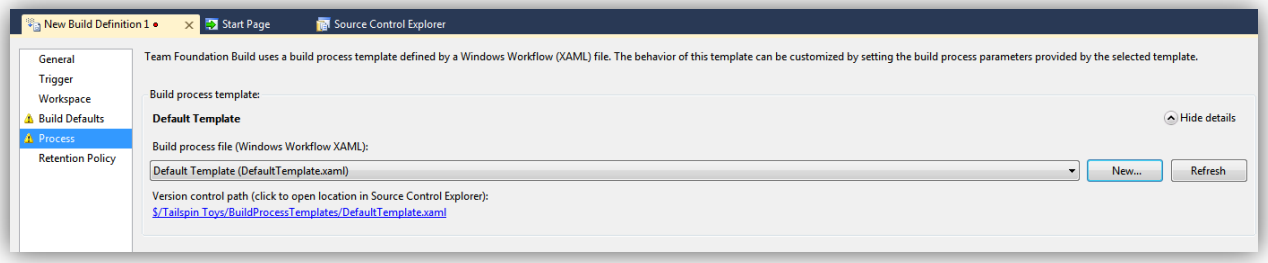


Figure 73 – New Build Process Template dialog

The New Build Process Template dialog provides the options for creating a new template from an existing one (Copy) or selecting a checked-in XAML file.

- Choose the option to **Select an existing XAML file** and browse to the version control location where your templates are checked in.

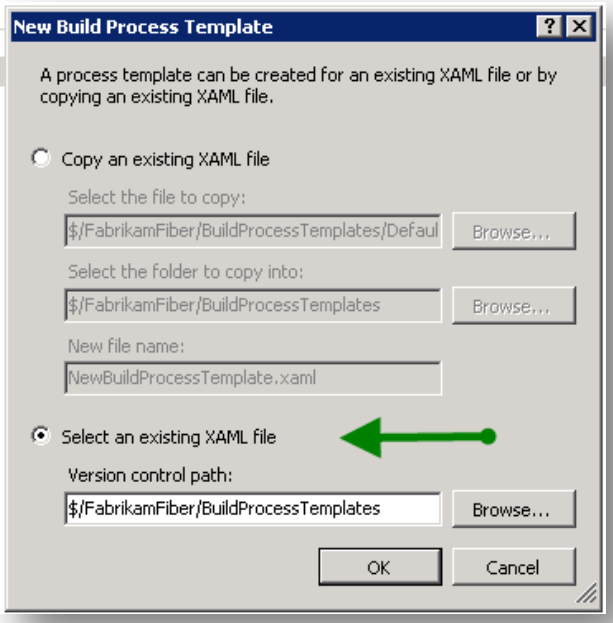


Figure 74 – Choose a Build Process Template

Discovery of Custom Activities (Assemblies)

Because custom activities are closely associated with process templates, it makes sense to follow the same practices for custom assemblies as we would for the process templates, namely:

- Checking custom assemblies into version control.
- Configuring Build controllers for a download location from the source control location.

Refer to Branching Process Templates on page 78 for a discussion on folder structure and branching.

After you have decided on the folder/branching structure and checked in the custom assemblies alongside their corresponding templates, the next step is to configure build controllers. Configuring the build controllers makes them aware of where the custom assemblies must be downloaded from to kick off the build processes on their agents. This is done through the **Manage Build Controllers** option in Team Explorer.

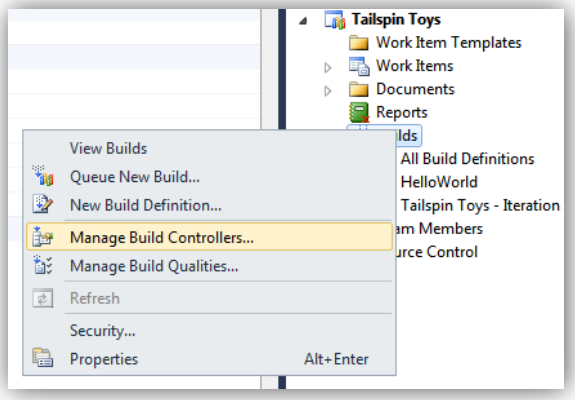


Figure 75 – Choosing to Manage Build Controllers

By selecting **Manage Build Controllers**, you can use the following dialogs to set the required options for configuring assemblies. We expand **Build Controller Properties**, and locate the specific version control folder that contains our custom assemblies.

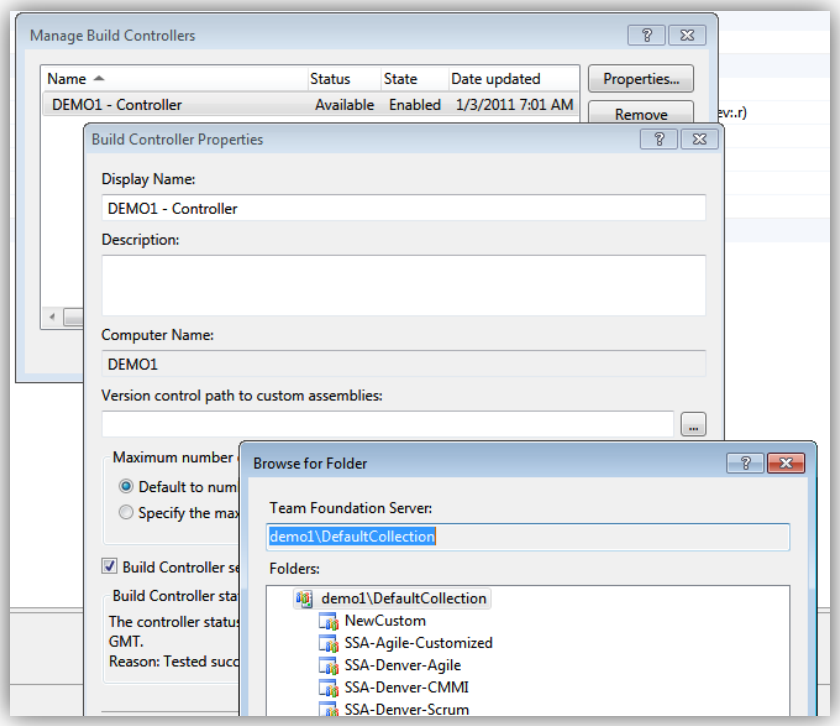


Figure 76 – Configuring the Custom Assemblies path

Specifying Download and Loading of Assemblies

Only assemblies that contain types marked with either the BuildActivity or BuildExtension attribute will be loaded into the process. For the assemblies that you depend upon and want to download, it is recommended that you add these dependencies into a file. This file must be checked in alongside your custom assemblies and must list the required assemblies. – For example, here is a file namedCustomActivitiesAndExtensions.xml with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<Assemblies>
  <Assembly FileName="MyDependency.dll">
    <Extensions>
      <Extension FullName="MyDependency.CustomException" />
    </Extensions>
  </Assembly>
  <Assembly FileName="MyNewDependency.dll">
    <Extensions>
      <Extension FullName="MyNewDependency.CustomEventArgs" />
    </Extensions>
  </Assembly>
</Assemblies>
```


Branching Process Templates

This section primarily focuses on customizing *Build Process Templates, and the value of branching in that context.*

IMPORTANT

As a prerequisite, please read the Visual Studio Team Foundation Server Branching Guide [34] and verify that this topic has not been covered in the latest Branching Guidance, in which case this section is obsolete.

In this Build Customization Guide, branching scenarios will be based primarily on the "*Basic Branch Plan"* from the Rangers branching guide. The Basic Branch Plan provides us with a flexible versioning approach for Build Process Templates. This flexible approach has several advantages. It allows us to maintain the version of the Build Process Template that is currently in production and provide support for fixing bugs while we continue to enhance these templates.

NOTE

Generally, Build Process Template customization only needs a simple versioning structure. If you require a more complex branching configuration, feel free to select a different Branch Plan to suit your needs.

Branch Structure for Deploying

NOTE

The recommendation is typically (and there is always an exception) that process templates and custom assemblies should be kept together. That is, store custom assemblies in a sub-directory under the process templates to avoid controller restarts when you are simply updating templates. We also recommend that process templates and custom assemblies should be branched as if they are their own product. Branching process templates and custom assemblies as part of the source requires one build controller per branch and increases maintenance, etc. Typically, but not always, a build controller per branch is excessive.

After you have finished customizing your Build Process Templates and custom assemblies, this section describes how to deploy them to version control so Team Foundation Build can consume them.

The following examples are based on the Basic Branch Plan from the Ranger's Branching Guidance:

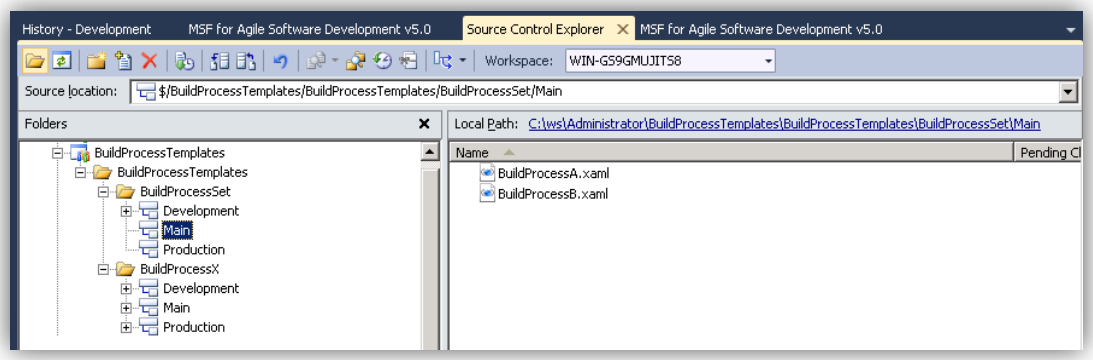


Figure 77 – Versioning multiple branch process templates together

[34] http://vsarbranchingguide.codeplex.com

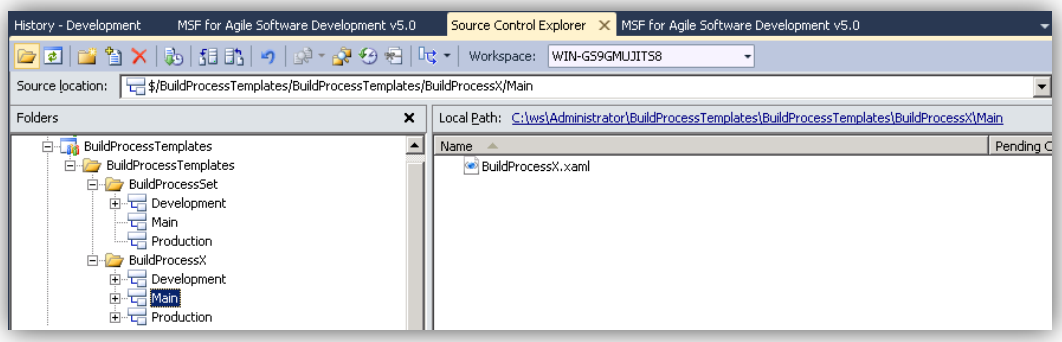


Figure 78 – Versioning a single branch process template

In the BuildProcessesSet / division, we show a branch structure with a set of build templates being versioned together. When there is coupling between the branch process templates, and no need to version them independently, keep them together in the same branch.

In the BuildProcessX / division, we have a single build template being versioned alone.

In our examples, each build process template folder (BuildProcessesSet/ and BuildProcessX/) can have its own versioning lifecycle. This allows the teams to spend time working in one structure of process templates at a time or even work in more than one in parallel, for example.

Follow the best practices from the Rangers Branching Guidance for customizing code or templates in the development branch, labeling, and merging these changes to Main, and then branching them to Production when they are ready to be released.

NOTE

It is also important to use the same structure with build templates and their related custom assemblies when they are all being versioned together.

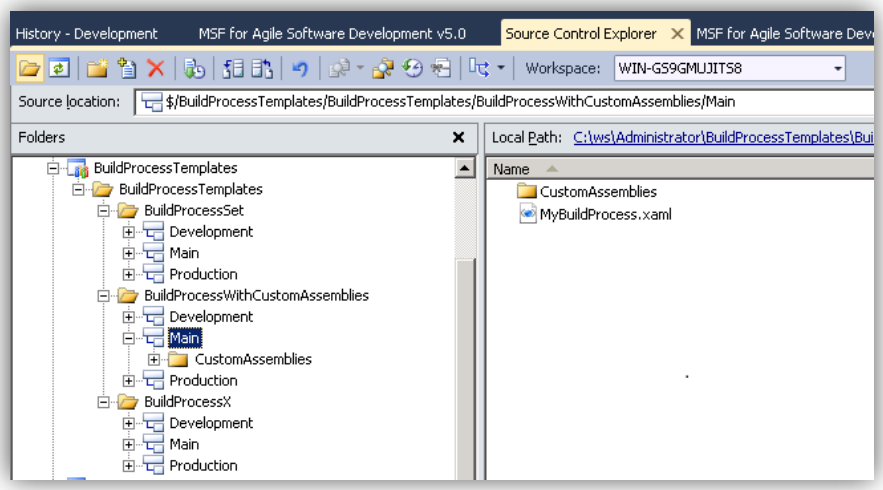


Figure 79 – Adding a folder for Custom Assemblies

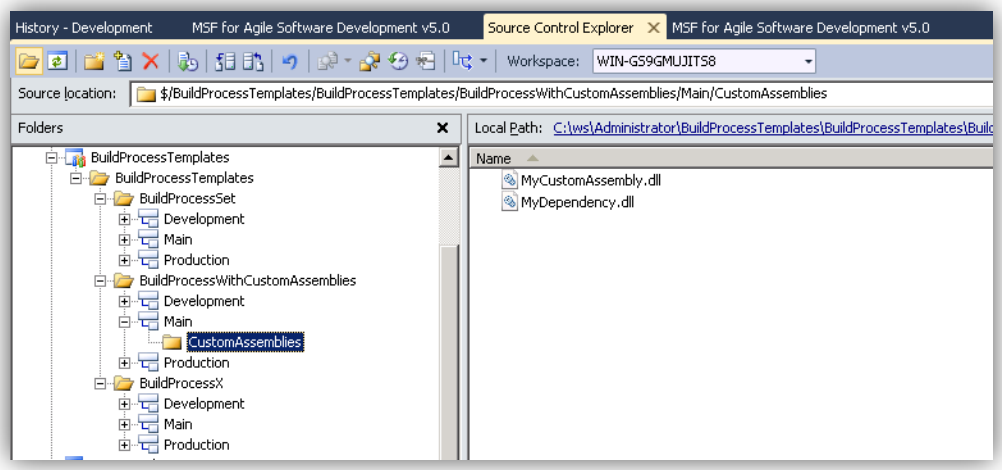


Figure 80 – Adding Custom DLLs to be versioned with the build process template

In the preceding examples, there are three branches – Development, Main and Production/Release – as recommended by Team Foundation Server Branching Guide. Each branch has its own set of files and folders versioned.

While we have a build template released (branched to Production), the development team can be working on the next version of the build process templates and custom assemblies in the Development branch.

NOTE

For more details on deploying custom build process templates and custom activities, see Empowering developers and build engineers with build activities on page 138.

Managing Default Build Process Templates

A common best practice is to maintain a set of ”golden” build process templates stored in a central location in your project collection. These templates typically implement different kinds of builds, be it release builds, QA builds or any kind of customized builds that you want to reuse across several team projects. Ideally the build templates should be generic enough to allow for reuse across different projects, but of course the templates will sometimes need to be customized for a particular project.

As discussed previously, build process templates are checked into source control as XAML files. However, the templates will not appear automatically in the list of available build process templates when creating build definitions.

To make a template appear in the build process template dropdown, you must create (at least) one build definition using that template. This is done by using the “Select an existing XAML file” option on the Process tab, and then browse to the source control path of the build process template. After this is done, your customized template will appear as a choice for all new build definitions *in that team project.* When creating a build definition in another team project, you will have to go through this process again.

When the “golden” build process templates have been checked in to a common location (for example in **$/Common/BuildProcessTemplates/)**, they should be added to all team projects that could benefit from using them. This can be done as described above, by manually creating a new build definition in each team project and for each build definition browse to the XAML file, but this is of course not very practical.

There is no functionality in Team Explorer for adding a build process template to another team project, but it can be done by using the Team Foundation Server API. Also, the reference application for this guidance (Community TFS Build Manager) contains functionality for adding and removing build process template to one/several/all team projects in a project collection.

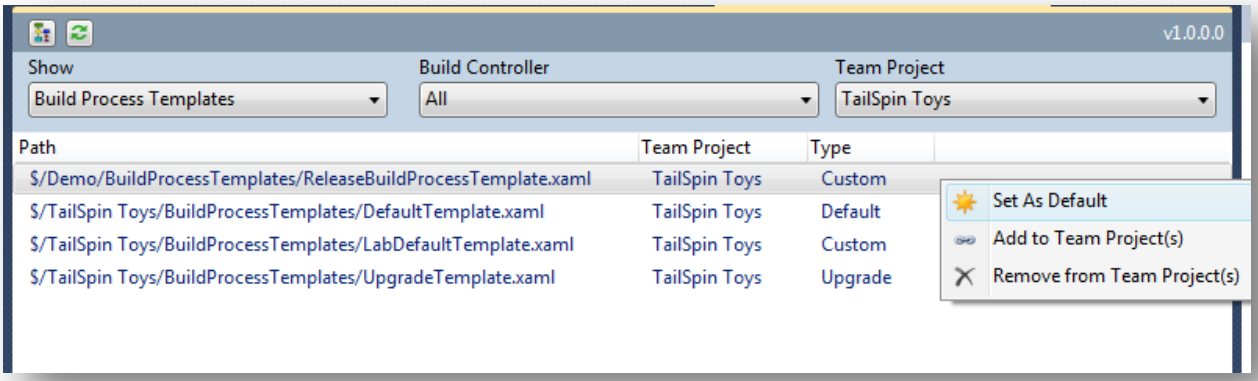


Figure 81 - Managing Build Process Templates using Community TFS Build Manager

In the figure above you can see that in addition to the three default templates (DefaultTemplate.xaml, LabDefaultTemplate.xaml and UpgradeTemplate.xaml), there is one more build template that is located in another team project called Demo. By using the menu, templates can be removed from and added to other team projects. It is also possible to set a build process template as default for a team project, which means that this build template will automatically be selected when creating a new build definition in that team project.

Note that when adding a build process template to a team project, you are still referencing the original source control path of the build process template XAML, so any change to that template will affect all build definitions that are using the template. Often this is the desired behavior, since it allows a build manager to add features and fix bugs in one loction and it will apply to all build definitions. If this is not desired, the build process template should be branched as discussed in the *Branching Process Templates section.*

The following code snippet shows how a build process template can be added to a specific team project.

```
public void AddTemplateToTeamProject(Uri tfsUri, string template, string teamProject)
{
    var tfsCollection = new TfsTeamProjectCollection(tfsUri);
    var bs = (IBuildServer)tfsCollection.GetService(typeof(IBuildServer));

    //Make sure that the build process template does not already exist
    // in this team project
    if (!bs.QueryProcessTemplates(teamProject).Any(pt => pt.ServerPath == template))
    {
        //Create/Register the template in this team project and save it
        var t = bs.CreateProcessTemplate(teamProject, template);
        t.Save();
    }
}
```

For more information about using the API for managing build process templates, see Jason Prikett's blog post TFS 2010 – Managing Build Process Templates (what are those?)[35].

[35] http://blogs.msdn.com/b/jpricket/archive/2010/04/08/tfs-2010-managing-build-process-templates-what-are-those.aspx

Map Team Foundation Build in Visual Studio Team System 2008 Extensibility Targets to Team Foundation Build in Visual Studio 2010 / Visual Studio 2012

The following extract from the community Team Foundation Server 2008 Build Process poster, which can be obtained from http://www.saarchitect.net/index.php?q=filebrowser/download/1712 , shows the major areas and extension points in Team Foundation Server 2008.

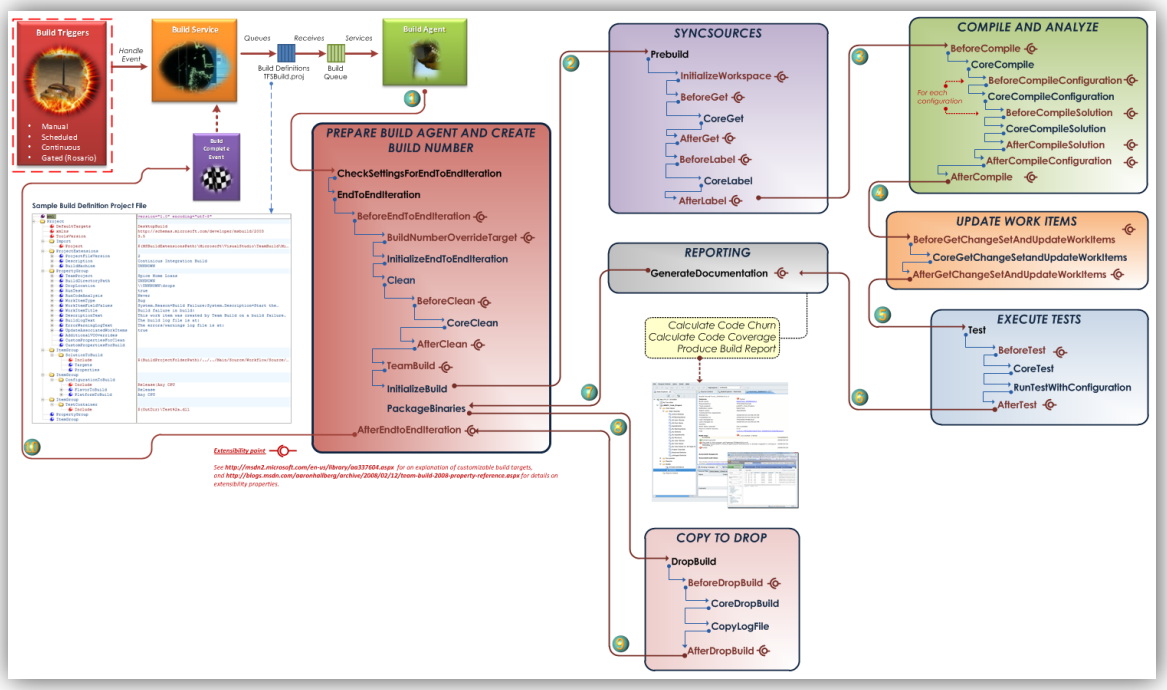


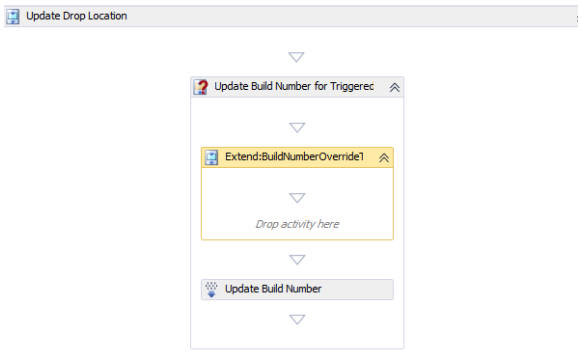
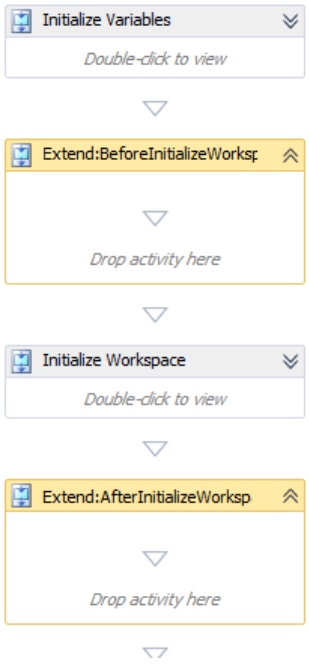
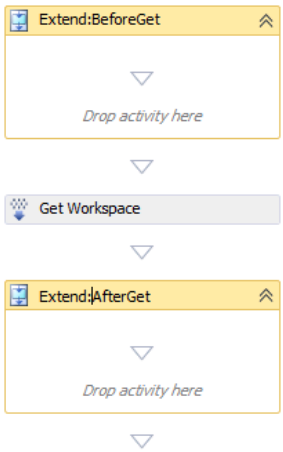
Figure 82 – Team Foundation Server 2008 Build Process Community Poster Extract

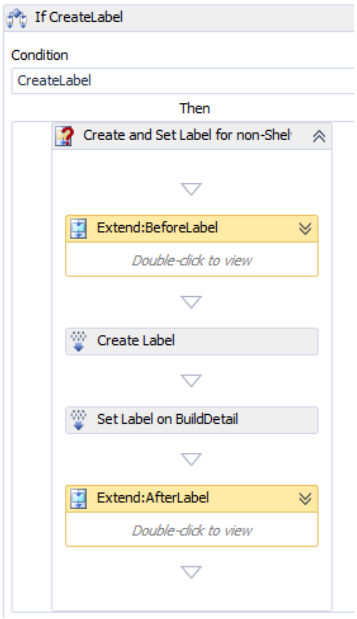
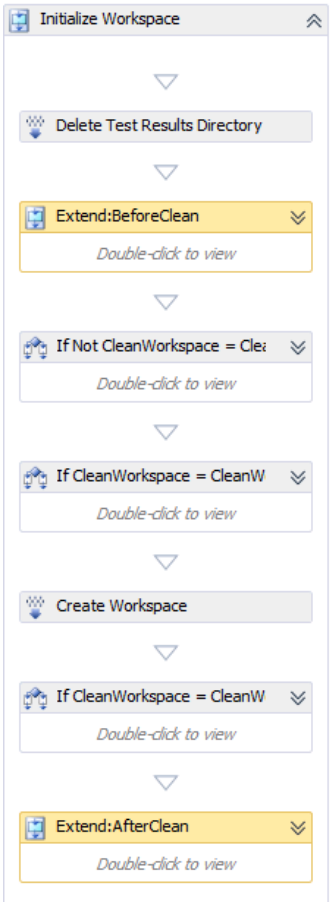
This section presents empty sequences to recommend where to start customizing the Team Foundation Server 2010 build workflow, without reusing the old MSBuild scripts. If you have MSBuild scripts, you should consider the Upgrade Template instead. For more information, see page 49,

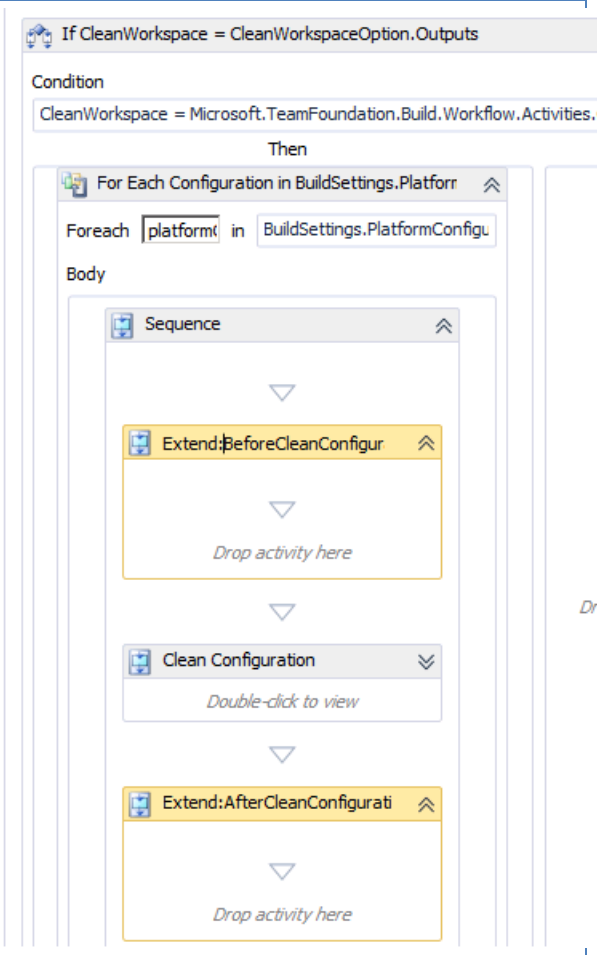
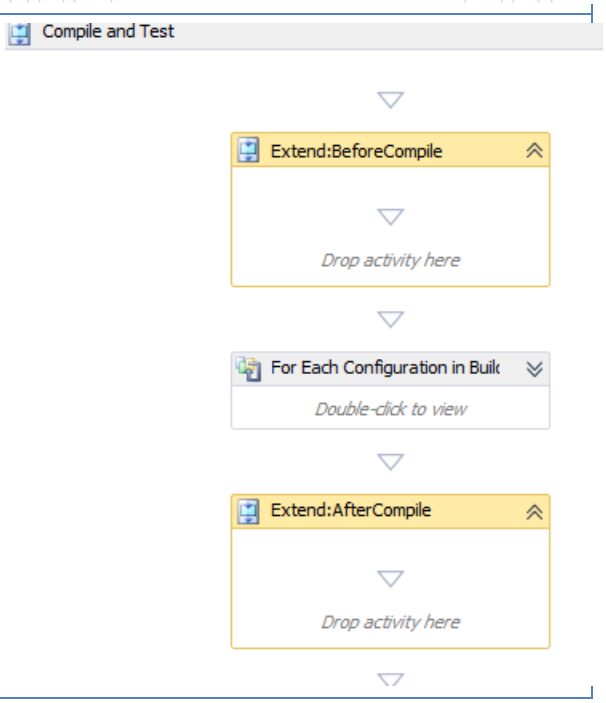
IMPORTANT

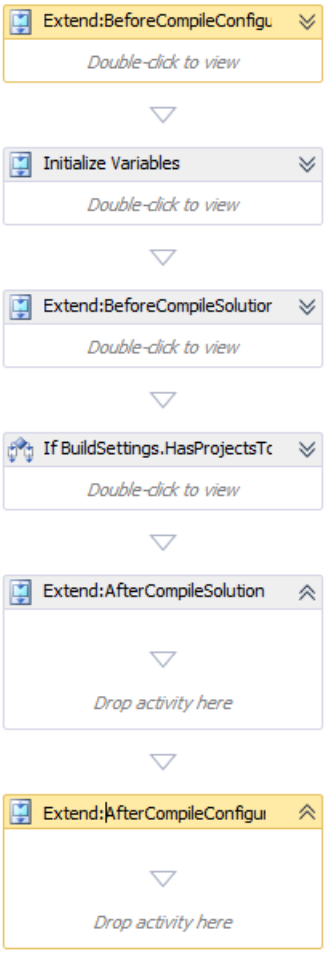
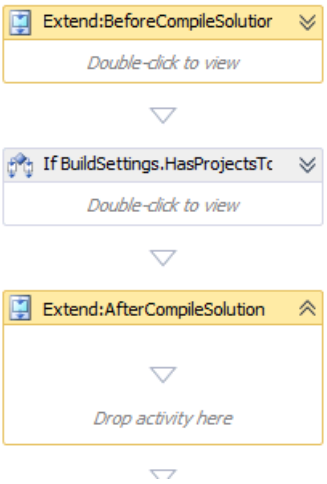
Refer to the example file **DefaultTemplateWithTargetMap.xaml**, which is included with this guidance. Alternatively, you can view this embedded version.

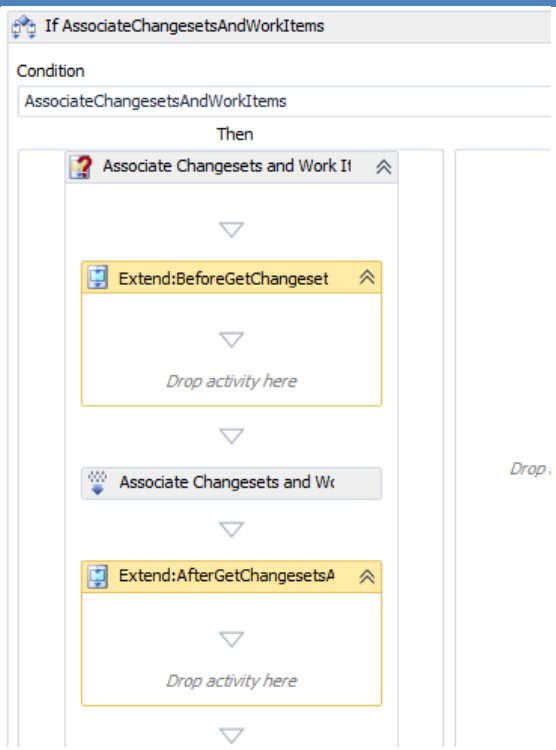
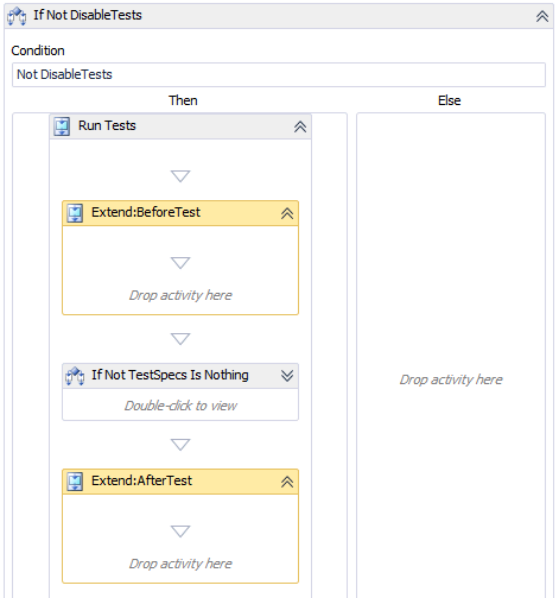
DefaultTemplateWithTargetMap.xml

Target	Build Template Area	Example
BuildNumberOverrideTarget	Update Drop Location Sequence	
BeforeInitializeWorkspace **AfterInitializeWorkspace**	Run On Agent Sequence	
BeforeGet **AfterGet**	Initialize Workspace Sequence	

Target	Build Template Area	Example
BeforeLabel **AfterLabel**	CreateLabel Condition	
BeforeClean **AfterClean**	Initialize Workspace Sequence	

Target	Build Template Area	Example
BeforeCleanConfiguration **AfterCleanConfiguration**	Initialize Workspace Sequence CleanWorkspace Condition	
BeforeCompile **AfterCompile**	Compile and Test Sequence	

Target	Build Template Area	Example
BeforeCompileConfiguration **AfterCompileConfiguration**	Compile and Test for Configuration Sequence	
BeforeCompileSolution **AfterCompileSolution**	Compile and Test for Configuration Sequence	 We could see the BeforeCompileSolution going inside of the ForEach as well, but then it is really a "before compile project."
BeforeGetChangesetsAnd	Compile, test and	

Target	Build Template Area	Example
UpdateWorkItems **AfterGetChangesetsAnd UpdateWorkItems**	Associate Changesets and Work Items Parallel Sequence	
BeforeTest **AfterTest**	If not Disable Tests Condition	

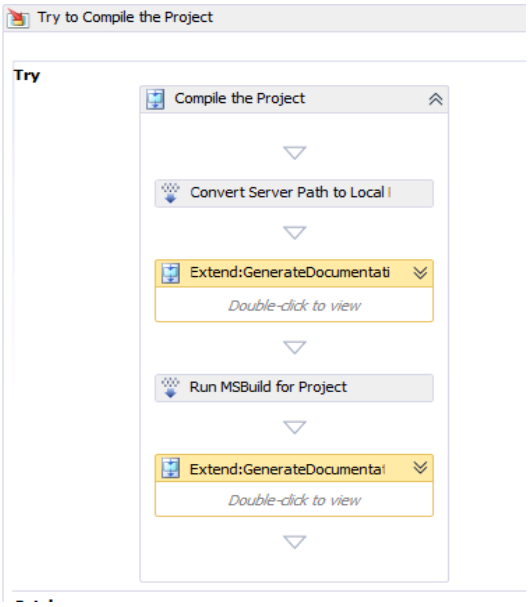
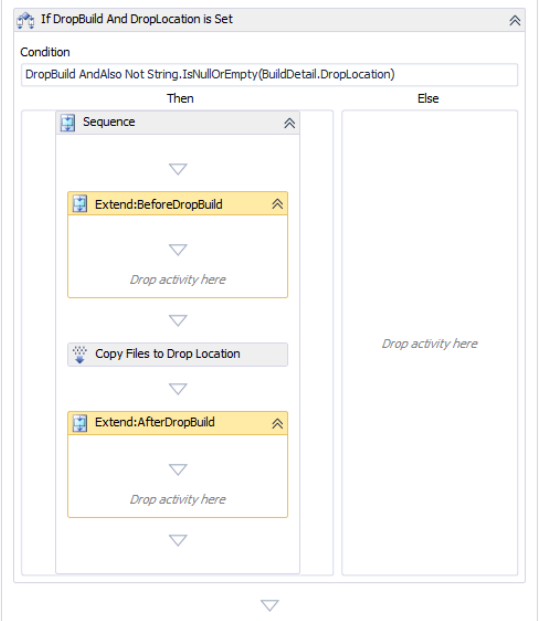
Target	Build Template Area	Example
GenerateDocumentation	Try to Compile the Project TryCatch	 A project setting before or after the run MSBuild for Project
BeforeDropBuild **AfterDropBuild**		

Table 13 – Extension Points compared to Extensibility Targets

Tracing Build Process Template and Custom Assembly Version

Goals

Team Foundation Server executes builds using several different aspects:

- The build process template, which describes the workflow and is checked in to source control.
- The build definition, which specifies parameter values that apply to the chosen build process template.
- Custom assemblies that are used in custom Windows Workflow Foundation activities.
- The build summary report, which comprises all relevant information about a build such as name, label, drop location, etc. after the build has finished.
- The build agent and controller, which are the execution infrastructure for the builds.

The default build process template does not store information about what build process template or custom assemblies were used. Further, parameter values set inside a build definition cannot be restored after a build has been executed. The functionality to allow a complete tracing back to the build definition has to be added using the extensibility features of Team Foundation Server.

Tracing back to original settings is important in case you want to re-run past builds after the build definition has been changed or in case you have to provide proof about what really has been done in the build, for example, in audits.

Solution

To keep track of the version and the name of the process template used for a build, create a custom activity that can be included in a build process template. This custom activity must output the process template that is used to run the build and its version.

```
// Log Process Template
var buildDetail = context.GetExtension<IBuildDetail>();
var vcs = context.GetExtension<TfsTeamProjectCollection>()
.GetService<VersionControlServer>();

var processTemplatePath = buildDetail.BuildDefinition.Process.ServerPath;
var processTemplateVersion = vcs.GetItem(processTemplatePath).ChangesetId;

context.TrackBuildMessage(string.Format("Used Process Template: {0};C{1}",
processTemplatePath, processTemplateVersion), BuildMessageImportance.High);
```

The code retrieves build detail data from the context and uses its properties to determine the Build Process Template and its version (as a changeset version) of that build process template file (.xaml) in source control. The code has to be included inside the Execute method of a custom build activity.

To track all custom assemblies used during the build, create a custom activity that outputs a list of the custom activity assemblies, along with their assembly version and file version.

You will need to add new activities because the necessary functionality is not provided with the standard Team Foundation Build activities.

```
// Log Custom Activities
var pathToTheCustomActivities = Path.Combine(
    Path.GetTempPath(),
    Path.Combine(
        "BuildController",
        LinkingUtilities.DecodeUri(buildDetail.BuildController.Uri.AbsoluteUri)
        .ToolSpecificId));

context.TrackBuildMessage(string.Format("Local path to custom assemblies: {0}"
                        , pathToTheCustomActivities)
                        , BuildMessageImportance.High);
foreach (var customActivityAssemblyPath in Directory
         .EnumerateFiles(pathToTheCustomActivities, "*.dll"))
{
    var customActivityAssembly = AssemblyName.GetAssemblyName(customActivityAssemblyPath);
```

```
        var version = FileVersionInfo.GetVersionInfo(customActivityAssemblyPath).FileVersion;
        var assemblyVersion = customActivityAssembly.Version;
        context.TrackBuildMessage(
            string.Format
                ("Used custom activity assembly: {0};Assembly Version:{1};File Version:{2}"
                , customActivityAssemblyPath, assemblyVersion, version)
            , BuildMessageImportance.High);
    }
```

The code will do the following tasks :

- Determine which folder contains custom assemblies on the build agent
- Iterate through all the assemblies inside that folder
- Get the assembly name, version and file version
- Track all this information inside the build log, using the TrackBuildMessage method

Again, the code has to be included inside the Execute method of a custom build activity.

Include the activity you just created inside your Build Process Template. When you are using the Default Template, you include the activity by adding it after the “Get the Build” activity). The following figure shows the “TraceAdditionalBuildInfo” activity integrated inside the Default Template.

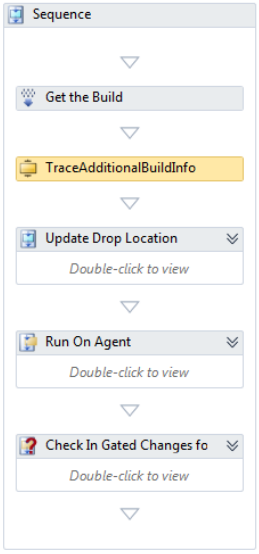


Figure 83 – “TraceAdditionalBuildInfo” activity integrated inside the Default Template

As a result, you should see the following inside your build log:

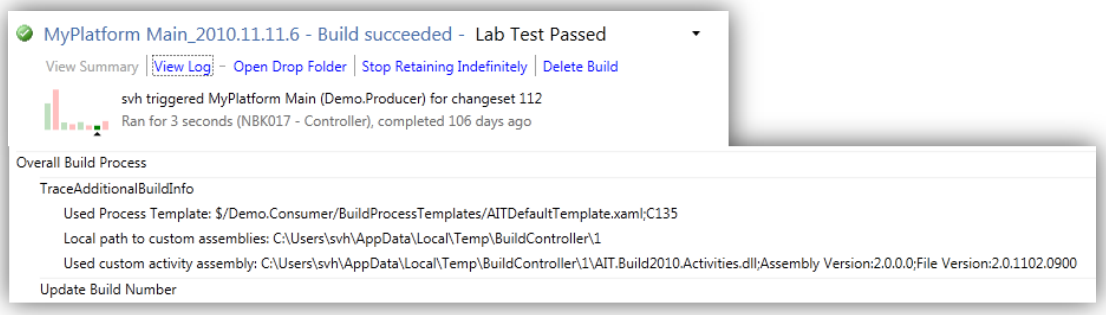


Figure 84 – Build Log Extract

Further Options

The option shown above provides a simple way to extend Team Foundation Build by tracing functions, as necessary.

To track information about other aspects, such as parameter values of the build definition, you could rely on diagnostic logging (see page 73) or create another activity that actually exports that information to a file in the drop location of the build.

NOTE

IBuildDetail.ProcessParameters contains parameters changed from build definition at queue time and IBuildDetail.BuildDefinition.ProcessParameters contains parameters changed from the template that was used in the build definition.

The Team Foundation Build API

Introduction

The key to automating the management of Team Foundation Build is to become familiar with the Team Foundation Build API. Any Team Foundation Build administrative task that can be done in Visual Studio can also be done using the Team Foundation Build API. In some cases, you can perform actions with the API that cannot be done from Visual Studio. The other advantage that the Team Foundation Build API provides is the ability to do actions in bulk that would otherwise be very labor intensive if you were to use the Visual Studio interface.

Resources

The Team Foundation Build API is one part of several that are covered on MSDN under Extending Team Foundation[36] where a growing resource of samples and documentation are being created by Microsoft. Other useful resources are the materials on the SDK[37] page and an ongoing API series by Shai Raiten[38]

Reference Application: Community TFS Build Manager

A reference application, Community TFS Build Manager, has been created to try and illustrate the usage of many parts of the API while providing a tool that many Build Engineers may hopefully find extremely useful. The Community TFS Build Manager is available from the Community TFS Build Extensions[39] site and provides an easy way to manage your builds in bulk.

Some of the growing list of features it provides includes

- View and change the State of multiple build definitions
- View and change the Build Templates for multiple build definitions
- View and change Retention Policies for multiple build definitions
- View and change Build Controllers for multiple build definitions
- Queue multiple build definitions
- Powerful Build cloning
- Delete multiple build definitions
- View Running Builds at various levels
- Visualize Build Resource Architecture
- Managing Build Process Templates

[36] http://msdn.microsoft.com/en-us/library/bb130146.aspx

[37] http://archive.msdn.microsoft.com/TfsSdk

[38] http://blogs.microsoft.co.il/blogs/shair/archive/tags/TFS+API/default.aspx

[39] http://tfsbuildextensions.codeplex.com

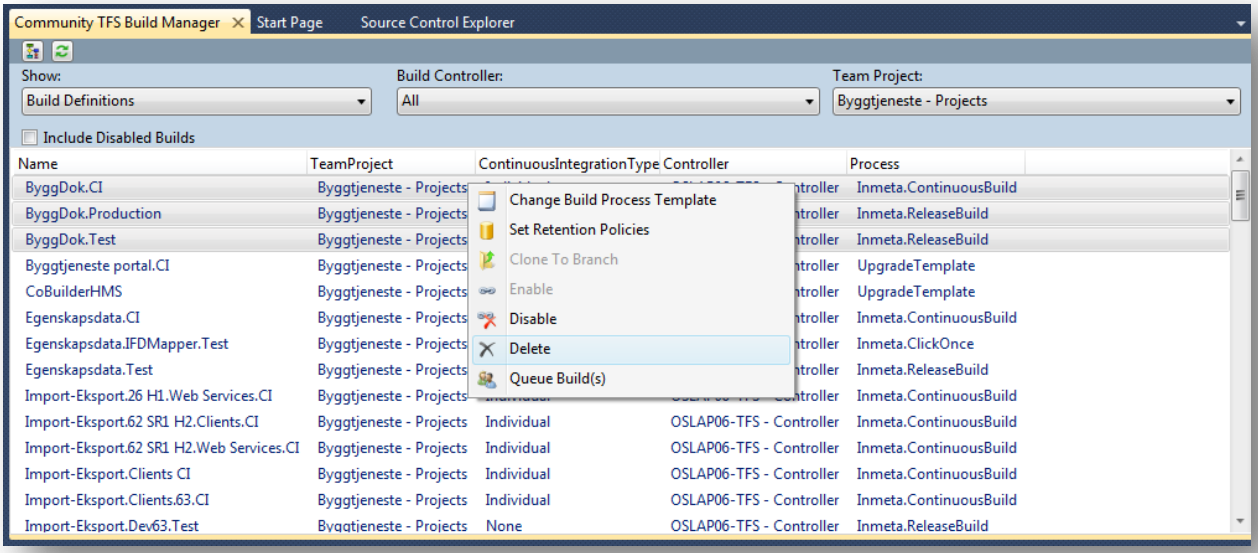


Figure 85 – Community TFS Build Manager as a Visual Studio Extension

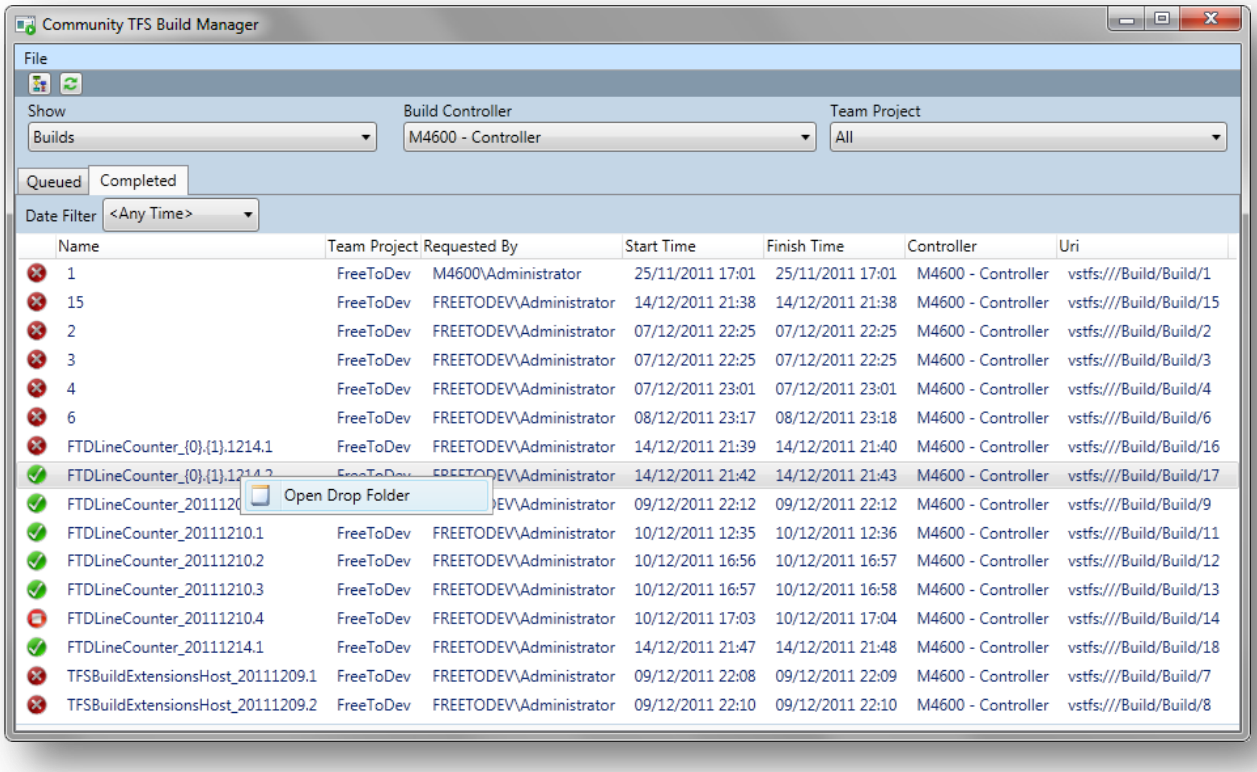


Figure 86 – Community TFS Build Manager as a Stand-alone Windows application

Getting Started

The first thing you need to do when programming against Team Foundation Server is to connect to it. You must first decide at what level you wish to connect; you have two options

- TfsConfigurationServer – you have access to services for the whole server
- TfsTeamProjectCollection – you have access to services for the team project collection. This is the connection class which we will use for build related actions.

Depending on which you use, your connection will expose different functionality as outlined in the table below

Service	TfsConfigurationServer (server-level)	TfsTeamProjectCollection (collection-level)
ITeamFoundationRegistry	✓	✓
IIdentityManagementService	✓	✓
ITeamFoundationJobService	✓	✓
IPropertyService	✓	✓
IEventService	✓	✓
ISecurityService	✓	✓
ILocationService	✓	✓
TswaClientHyperlinkService	✓	✓
ITeamProjectCollectionService	✓	
IAdministrationService	✓	✓
ICatalogService	✓	
VersionControlServer		✓
WorkItemStore		✓
IBuildServer		✓
ITestManagementService		✓
ILinking		✓
ICommonStructureService3		✓
IServerStatusService		✓
IProcessTemplates		✓

Table 14 – Scope of the TfsConnection classes

Basic Sample

To get started with the Team Foundation Build API, create a Console Application and add references to the following assemblies:

- Microsoft.TeamFoundation.Client
- Microsoft.TeamFoundation.Build.Client

Add the following imports to Program.cs:

```
using Microsoft.TeamFoundation.Client;
using Microsoft.TeamFoundation.Build.Client;
```

Finally, replace the body of Main with the following code. Replace the URL with the URL for your Team Project Collection:

```
var tpc = TfsTeamProjectCollectionFactory.GetTeamProjectCollection(
  new Uri("http://tfsrtm10:8080/tfs"),
  new UICredentialsProvider());

var buildServer = tpc.GetService<IBuildServer>();
```

The IBuildServer service will provide you access to all of the functionality available in the Team Foundation Build API. Most of the APIs are self-explanatory. You should be aware of some things:

1. A number of the Query APIs have an overload that takes a Spec object. This allows finer-grained querying than is available with the other overloads. You can create these Spec objects by calling the Create*Spec API, setting properties on the resulting object, and then passing it to the Query API. For example, the QueryBuilds API has an overload that takes an IBuildDetailSpec, which can be created using the CreateBuildDetailSpec API. For example:

   ```
   var buildDetailSpec = buildServer.CreateBuildDetailSpec("YourTeamProjectName");
   buildDetailSpec.Status = BuildStatus.Succeeded;
   buildDetailSpec.MaxBuildsPerDefinition = 10;
   buildDetailSpec.QueryOrder = BuildQueryOrder.FinishTimeDescending;
   var buildDetailResults = buildServer.QueryBuilds(buildDetailSpec);
   ```

2. Some of the Query overloads are inefficient so you should typically use the overload that takes a Spec object and set the QueryOptions property to specify what related information is returned. The use of MaxBuildsPerDefinitions property in the above example is important to limit the result.

   When you query builds, there is an additional consideration, which is the InformationTypes property. If this property is left as the default, then all information nodes attached to the build will be returned; this includes all log messages the build has created. Unless you need this information, you should set the property to an empty string array.

   In the next example, we restrict the above query to only returning information about the build itself and its definition, and to not return any information nodes:

   ```
   var buildDetailSpec = buildServer.CreateBuildDetailSpec("YourTeamProjectName");
   buildDetailSpec.Status = BuildStatus.Succeeded;
   buildDetailSpec.MaxBuildsPerDefinition = 10;
   buildDetailSpec.QueryOrder = BuildQueryOrder.FinishTimeDescending;
   buildDetailSpec.QueryOptions = QueryOptions.Definitions;
   buildDetailSpec.InformationTypes = new string[] { };
   var buildDetailResults = buildServer.QueryBuilds(buildDetailSpec);
   ```

3. If you have performed a query with a limited set of QueryOptions and InformationTypes that returned a large set of build or build definitions, you might want to get all information for some of these builds. For builds you can use the IBuildDetail.RefreshAllDetails() method that will perform a full reload of the

build. For build definitions there is no corresponding method, so you need to call IBuildServer.GetBuildDefinition(<buildDefinitionUri>) method.

4. After modifying most of the objects used in the Team Foundation Build API, call the Save () method to persist the data back to the database.

The following example applies these concepts to automate a management task that would otherwise be very time consuming if you were using Visual Studio. In this example, we update all definitions that use one controller (OldController) to use a different controller (NewController):

```
var buildControllerSpec = buildServer.CreateBuildControllerSpec("NewController", "");
var buildControllerResults = buildServer.QueryBuildControllers(buildControllerSpec);
var newBuildController = buildControllerResults.Controllers.First();

var buildDefinitionSpec = buildServer.CreateBuildDefinitionSpec("YourTeamProjectName");
buildDefinitionSpec.Options = QueryOptions.Controllers;

var buildDefinitionResults = buildServer.QueryBuildDefinitions(buildDefinitionSpec);
foreach (var buildDefinition in buildDefinitionResults.Definitions)
{
  if (buildDefinition.BuildController.Name.Equals("OldController",
    StringComparison.OrdinalIgnoreCase))
  {
    buildDefinition.BuildController = newBuildController;
    buildDefinition.Save();
  }
}
```

Accessing Build Process Parameters

Build process parameters are objects that are passed into the build process and determine what will be built, what tests to run (and how), and all the other switches that decide what happens during the build. These properties are not directly available on the IBuildDefinition interface; instead they are stored in a serialized dictionary that can be accessed using the IBuildDefinition.ProcessParameters property. To deserialize the properties into a dictionary that you can work with, you use a class called WorkflowHelper that is available in the *Microsoft.TeamFoundation.Build.Workflow.dll* assembly, that is located in the *%Program Files (x86)%\Microsoft Visual Studio 10.0\Common7\IDE\PrivateAssemblies* directory.

NOTE

Your application must target the full .NET Framework 4.0 to use this reference, not just the .NET Framework 4 Client Profile that is the default target framework for several project types in Visual Studio 2010.

Here is a short sample that lists the projects being built for a build definition:

```
IBuildDefinition bd = buildServer.GetBuildDefinition("uri");
var parameters = WorkflowHelpers.DeserializeProcessParameters(bd.ProcessParameters);
var buildSettings = parameters["BuildSettings"] as BuildSettings;

foreach( (int i = 0; i < buildSettings.ProjectsToBuild.Count(); i++)
{
        Console.WriteLine(buildSettings.ProjectsToBuild[i]);
}
```


The BuildSettings key referenced in the sample are one of the many process parameters that exist in the DefaultTemplate.xaml template. The following table lists all process parameters that are available in the DefaultTemplate:

Setting Name	Type
BuildSettings	Microsoft.TeamFoundation.Build.Workflow.Activities.BuildSettings
TestSpecs	Microsoft.TeamFoundation.Build.Workflow.Activities.TestSpecList
BuildNumberFormat	System.String
CleanWorkspace	Microsoft.TeamFoundation.Build.Workflow.Activities.CleanWorkspaceOption
RunCodeAnalysis	Microsoft.TeamFoundation.Build.Workflow.Activities.CodeAnalysisOption
SourceAndSymbolServerSettings	Microsoft.TeamFoundation.Build.Workflow.Activities.SourceAndSymbolServerSettings
AgentSettings	Microsoft.TeamFoundation.Build.Workflow.Activities.AgentSettings
AssociateChangesetsAndWorkItems	System.Boolean
CreateWorkItem	System.Boolean
DropBuild	System.Boolean
MSBuildArguments	System.String
MSBuildPlatform	Microsoft.TeamFoundation.Build.Workflow.Activities.ToolPlatform
PerformTestImpactAnalysis	System.Boolean
CreateLabel	System.Boolean
DisableTests	System.Boolean
GetVersion	System.String
PrivateDropLocation	System.String
Verbosity	Microsoft.TeamFoundation.Build.Workflow.BuildVerbosity

Table 15 – Build Process Parameters in DefaultTemplate.xaml

Note that any custom process parameters that you add to your build process template, will be accessible in the same way.

You can also access the process parameters for an IBuildDetail object. These will be the parameters that were actually passed when queuing the build, which is not necessarily the same as the ones in the build definition. You access them through the IBuildDetail.ProcessParameters property. This property will only contain the process parameters that were actually changed from the default values when queuing the build. That is, for a default build where a build definition is just queued without any changes to the process parameters, the IBuildDetail.ProcessParameters property will actually be *null*.

Managing Mega-Build Environments

Mega-builds are those builds that run for long periods, for example, over eight hours, or that run in large build labs. These types of builds pose unique challenges when you design and implement a build process. You can apply some of these techniques to shorter-running builds, but you should decide whether these add value to your build process before applying them.

Mega-build Challenges

In this section, we will discuss some of the challenges facing mega-builds and introduce some possible solutions to them. In the following section, we will explore some of the solutions in-depth, along with reference implementations that you can use in your build process.

Logging

Adequate logging is a huge issue in mega-builds. For example, if a problem occurs 18 hours into a 24-hour build, you do not want to re-run the build with diagnostic logging to determine the root cause of the issue. Conversely, too much logging can make viewing the logs through Visual Studio or Team Foundation Web Access slow and unreliable. For more information, refer to Team Foundation Server 2010 - Viewing the Build Details Log View in Visual Studio is very slow[40].

If you have been running builds for a while, one of the key factors in deciding what to log and with what level of detail is the history of previous failed builds. You should review the histories, identify what parts of the build process are prone to failure, and focus your logging efforts around them.

There are three principles you should follow when you implement logging in a mega-build process:

Minimize logging of things that do not change from build to build.

By default, Team Foundation Build logs the execution of every single activity in your workflow. As your build process grows, this will produce a large amount of unchanging log entries.

There is little value in logging certain activities that rarely fail, such as Sequence, If, Assign. By applying the BuildTrackingParticipant.Importance attribute to these activities, you can drastically reduce the amount of static noise in your build log. This is discussed in more detail on pages 147 and 151.

You might be confused by the inclusion of the If activity in the previous discussion. When the If activity is logged, only its static display name is logged. This does not tell you which branch of the If was taken. We recommend not logging the If activity itself, but instead, you should wrap each branch of the If in a Sequence activity, if it is not already. In addition, make sure its display name clearly indicates which branch was taken. Finally, make sure that those sequences are logged.

For example:

```
<If Condition="[String.IsNullOrWhiteSpace(BuildDetail.DropLocation)]"
  DisplayName="Prepare Drop Location" mtbwt:BuildTrackingParticipant.Importance="Low">
  <If.Then>
    <Sequence DisplayName="Not Dropping" mtbwt:BuildTrackingParticipant.Importance="High">
      <!-- ... -->
    </Sequence>
  </If.Then>
  <If.Else>
    <Sequence DisplayName="Dropping" mtbwt:BuildTrackingParticipant.Importance="High">
      <!-- ... -->
    </Sequence>
  </If.Else>
</If>
```

[40] http://blogs.msdn.com/b/jpricket/archive/2011/02/08/tfs-2010-viewing-the-build-details-log-view-in-visual-studio-is-very-slow.aspx

Maximize logging of things that do change from build to build.

Conversely, anything that is likely to be different in this build compared with prior builds should be logged. Logging the differences ensures there is enough information to help debug a failed build or unexpected build outputs. Consider logging the following:

- Process parameters.
- Commands that are executed (including the working directory, path, and arguments).
- Environment variables.
- Machine information, such as operating system, architecture, etc.

NOTE

Team Foundation Server 2012 now logs all builds to file using diagnostic verbosity by default. This can be very handy in diagnosing any build issues.

Log any action that can produce logs of unbounded size to a file rather than the Team Foundation Build log.

As we discussed on page 101 you can easily us the WriteBuildMessage activity to add log messages to the build detail window in Visual Studio. By using this activity in conjunction with reading lines from a file or the InvokeProcess activity, you can include output from commands run by your build process in the build detail window. This can be a very useful tool in fulfilling the “Maximize logging of things that do change from build to build” principle.

However, the build detail window was primarily designed to log the hierarchy of activities executed by the build process, along with a small amount of contextual information. If you log a large number of messages, you might find that the log is extremely slow to open and difficult to work with. To prevent this, any activity that might log large amounts of output should log to a file instead of to the build detail window. An example of this in the Default Template is the MSBuild activity, which logs the build outputs to a file and only logs the warnings and errors to the build detail window.

While some tools, such as MSBuild and Robocopy, create log files when given the correct arguments, others do not have this capability. In these situations, you will need to use the Handle Standard Output and Handle Error Output ActivityActions of the InvokeProcess activity to redirect the output to log files.

In this example, we create two StreamWriters (one for standard output and one for error output) and the InvokeProcess activity to call MSDeploy.exe and log its output to two files. This is wrapped in a TryCatch activity (although we only use Try and Finally) to Close the StreamWriters when the InvokeProcess completes.

```
<TryCatch DisplayName="Log MSDeploy Output">
  <TryCatch.Variables>
    <Variable x:TypeArguments="si:StreamWriter"
      Default="[New StreamWriter(&quot;msdeploy.out.log&quot;)]"
      Name="StandardOutputWriter" />
    <Variable x:TypeArguments="si:StreamWriter"
      Default="[New StreamWriter(&quot;msdeploy.err.log&quot;)]" Name="StandardErrorWriter"
/>
  </TryCatch.Variables>
  <TryCatch.Finally>
    <Sequence>
      <InvokeMethod DisplayName="Close Standard Output Writer" MethodName="Close">
        <InvokeMethod.TargetObject>
          <InArgument x:TypeArguments="si:StreamWriter">[StandardOutputWriter]</InArgument>
        </InvokeMethod.TargetObject>
      </InvokeMethod>
      <InvokeMethod DisplayName="Close Standard Error Writer" MethodName="Close">
        <InvokeMethod.TargetObject>
          <InArgument x:TypeArguments="si:StreamWriter">[StandardErrorWriter]</InArgument>
        </InvokeMethod.TargetObject>
      </InvokeMethod>
    </Sequence>
  </TryCatch.Finally>
```



```
<TryCatch.Try>
  <mtbwa:InvokeProcess DisplayName="Run MSDeploy" FileName="msdeploy.exe">
    <mtbwa:InvokeProcess.ErrorDataReceived>
      <ActivityAction x:TypeArguments="x:String">
        <ActivityAction.Argument>
          <DelegateInArgument x:TypeArguments="x:String" Name="errOutput" />
        </ActivityAction.Argument>
        <InvokeMethod DisplayName="Log To Standard Error Writer" MethodName="WriteLine">
          <InvokeMethod.TargetObject>
            <InArgument x:TypeArguments="si:StreamWriter">
                        [StandardErrorWriter]</InArgument>
          </InvokeMethod.TargetObject>
          <InArgument x:TypeArguments="x:String">[errOutput]</InArgument>
        </InvokeMethod>
      </ActivityAction>
    </mtbwa:InvokeProcess.ErrorDataReceived>
    <mtbwa:InvokeProcess.OutputDataReceived>
      <ActivityAction x:TypeArguments="x:String">
        <ActivityAction.Argument>
          <DelegateInArgument x:TypeArguments="x:String" Name="stdOutput" />
        </ActivityAction.Argument>
        <InvokeMethod DisplayName="Log To Standard Output Writer" MethodName="WriteLine">
          <InvokeMethod.TargetObject>
            <InArgument x:TypeArguments="si:StreamWriter">
              [StandardOutputWriter]
            </InArgument>
          </InvokeMethod.TargetObject>
          <InArgument x:TypeArguments="x:String">[stdOutput]</InArgument>
        </InvokeMethod>
      </ActivityAction>
    </mtbwa:InvokeProcess.OutputDataReceived>
  </mtbwa:InvokeProcess>
 </TryCatch.Try>
</TryCatch>
```

Recovering from Failures

Typically, when Team Foundation Build encounters a failure, it logs the error and stops the build. This is desirable behavior for small builds because you can resolve the problem and simply re-run the build.. However, in mega-build scenarios where the builds are long running, you usually prefer to recover from the failure and continue the build.

In this section we will discuss different techniques for recovering from failures, both automatically (where possible) and manually. Typically, you will want to use different techniques in different parts of your build process, depending on the types of failure you expect in that part.

Retries

The first technique is simply retrying the operation that failed in hope that it succeeds. The definition of insanity is repeating the same thing and expecting different results. However, when it comes to complex network infrastructure, temporary outages, and the like, repetition is often a perfectly valid technique.

One way to implement retries is simply to leverage the retry capabilities that are built in to the tools that are called by your build process. For example, your build process may call Robocopy to drop the outputs of the build. Robocopy has the following switches for controlling retries:

```
/R:n :: number of Retries on failed copies: default 1 million.
/W:n :: Wait time between retries: default is 30 seconds.
/TBD :: wait for sharenames To Be Defined (retry error 67).
```

This technique only works when you call tools that have built-in retry capabilities so it cannot be used for custom code activities. In this situation, we can implement retries in the workflow itself by either modifying the code activity to implement retries or wrapping it in a composite activity that implements the retries.

If you choose the approach of modifying the code activity to implement retries, it is very important that the code activity derives from AsyncCodeActivity and supports cancellation. Otherwise, it will block users who try to stop the build or who are running other activities in parallel.

Wrapping the activity in a composite is a simpler, although more verbose, technique. It provides cancellation for free, assuming the code activity you call is short running, because composites are automatically cancellable. If the activity is long running then it should still derive from AsyncCodeActivity to avoid the problems mentioned previously.

In this example, we wrap a fictitious UpdateDatabase activity in a DoWhile and TryCatch activity to implement retries. To minimize noise in the log, we only output the exception message if it differs from the last exception message we saw.

```
<DoWhile>
  <DoWhile.Variables>
    <Variable x:TypeArguments="x:Boolean" Default="False" Name="Succeeded" />
    <Variable x:TypeArguments="x:String" Name="LastExceptionMessage" />
  </DoWhile.Variables>
  <DoWhile.Condition>[Not Succeeded]</DoWhile.Condition>
  <TryCatch>
    <TryCatch.Try>
      <Sequence DisplayName="Try Update Database">
        <a:UpdateDatabase Text="..." DisplayName="Update Database" />
        <Assign DisplayName="Mark Succeeded">
          <Assign.To>
            <OutArgument x:TypeArguments="x:Boolean">[Succeeded]</OutArgument>
          </Assign.To>
          <Assign.Value>
            <InArgument x:TypeArguments="x:Boolean">True</InArgument>
          </Assign.Value>
        </Assign>
      </Sequence>
    </TryCatch.Try>
    <TryCatch.Catches>
      <Catch x:TypeArguments="s:Exception">
        <ActivityAction x:TypeArguments="s:Exception">
          <ActivityAction.Argument>
            <DelegateInArgument x:TypeArguments="s:Exception" Name="exception" />
          </ActivityAction.Argument>
          <Sequence DisplayName="Handle Exception">
            <If Condition="[exception.Message &lt;&gt; LastExceptionMessage]"
              DisplayName="If Exception Message Different">
              <If.Then>
               <Sequence DisplayName="Output Exception Message">
                  <mtbwa:WriteBuildWarning DisplayName="Write Build Warning"
                    Message="[exception.Message]" />
                  <Assign DisplayName="Capture Exception Message">
                    <Assign.To>
                      <OutArgument x:TypeArguments="x:String">
                        [LastExceptionMessage]
                      </OutArgument>
                    </Assign.To>
                    <Assign.Value>
                      <InArgument x:TypeArguments="x:String">
                                  [exception.Message]</InArgument>
                    </Assign.Value>
                  </Assign>
                </Sequence>
              </If.Then>
            </If>
            <Delay Duration="[TimeSpan.FromMinutes(1)]" />
          </Sequence>
        </ActivityAction>
      </Catch>
```



```
      </TryCatch.Catches>
    </TryCatch>
  </DoWhile>
```

We can simplify this pattern by creating a custom sequence activity that we can use in our workflows. The custom sequence activity automatically retries the activities in it if any of them fail. This reduces the noise in the workflow, ensures that the retry pattern is followed consistently, and improves the productivity of developers working in the workflow.

Below is a simple implementation of a Retry activity that will attempt to execute the activities it contains. If one of the contained activities throws an exception, it will retry **all** of the contained activities after a 1-minute delay. It will do this indefinitely until all of the activities succeed. It also prints the first exception that occurs and the total number of retries required before the activities succeeded.

```
using System;
using System.Activities;
using System.Activities.Statements;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Windows.Markup;
using Microsoft.TeamFoundation.Build.Workflow.Activities;

namespace Activities
{
  [ContentProperty("Activities")]
  [Designer("System.Activities.Core.Presentation.SequenceDesigner,
    System.Activities.Core.Presentation, Version=4.0.0.0, Culture=neutral,
    PublicKeyToken=31bf3856ad364e35")]
  public class Retry : Activity
  {
    public Collection<Variable> Variables { get; private set; }
    public Collection<Activity> Activities { get; private set; }

    /// <summary>
    /// Constructor
    /// </summary>
    public Retry()
    {
      Variables = new Collection<Variable>();
      Activities = new Collection<Activity>();
      base.Implementation = () => CreateBody();
    }

    private Activity CreateBody()
    {
      Variable<bool> needToRetry = new Variable<bool> { Name = "needToRetry" };
      Variable<int> numberOfRetries = new Variable<int> { Name = "numberOfRetries" };
      DelegateInArgument<Exception> ex = new DelegateInArgument<Exception>();

      return new Sequence()
      {
        Variables =
        {
          needToRetry,
          numberOfRetries,
        },
        Activities =
        {
          new Assign<bool>
          {
            Value = new InArgument<bool>(true),
            To = new OutArgument<bool>(needToRetry),
          },
          new Assign<int>
```

```
{
  Value = new InArgument<int>(1),
  To = new OutArgument<int>(numberOfRetries),
},
new While(env => needToRetry.Get(env))
{
  Body = new TryCatch
  {
    Try = new Sequence()
    {
      Activities =
      {
        CreateSequence(),
        new Assign<bool>
        {
          Value = new InArgument<bool>(false),
          To = new OutArgument<bool>(needToRetry),
        }
      }
    },
    Catches =
    {
      new Catch<Exception>()
      {
        Action = new ActivityAction<Exception>()
        {
          Argument = ex,
          Handler = new Sequence
          {
            Activities =
            {
              new If
              {
                // Print out exception stack only when it happens for the 1st time
                Condition = new InArgument<bool>(
                  env => numberOfRetries.Get(env) == 1),
                Then = new WriteBuildWarning
                {
                  Message = new InArgument<string>(env => string.Format("Retry
                    Operation '{0}' due to '{1}'. StackTrace: {2}.", DisplayName,
                    ex.Get(env).Message, ex.Get(env).StackTrace)),
                },
              },
              new Assign<int>
              {
                Value = new InArgument<int>(env => numberOfRetries.Get(env) + 1),
                To = new OutArgument<int>(numberOfRetries),
              },
              new Delay
              {
                   Duration = TimeSpan.FromMinutes(1),
              }
            },
          },
        },
      },
    },
  },
},
new WriteBuildMessage
{
  Message = new InArgument<string>(env => string.Format("Retried {0} times.",
    numberOfRetries.Get(env))),
},
```

```
            },

          };
        }

        private Sequence CreateSequence()
        {
          Sequence seq = new Sequence();
          foreach (var item in Activities)
          {
            seq.Activities.Add(item);
          }

          foreach (var item in Variables)
              seq.Variables.Add(item);

          return seq;
        }
      }
    }
```

In this example, we wrap this Retry activity around a fictitious UpdateDatabase activity:

```
<a:Retry>
  <a:UpdateDatabase Text="..." />
</a:Retry>
```

Regardless of how you implement retries, it is important that you can determine from the build log that an activity is being retried. Otherwise, the build might appear hung, especially if the activity has to be retried numerous times before it succeeds. To be configurable if you implement the Retry activity in the example, you might want to extend it to allow the maximum number of retries and delay between each retry. You may even want to send email automatically to the build's requestor if the activity does not complete within a configured threshold.

Waiting for Manual Input

If part of the build consistently fails when retried, it might be necessary to allow someone to manually intervene, complete the task manually, and then signal the build that it should proceed.

For example, we might have a step in the build workflow that runs a generated SQL script against a database server, which, for reasons beyond our control, can fail despite using retry technique in the example. Instead of failing the build, we can alert someone that manual intervention is required and allow them to tell the build when it is safe to proceed.

The pattern we will apply to solve this problem is:

```
Try
        Failing Step
Catch
        Log Warning / Send Mail
        Wait for Manual Input
End Try
```

To wait for manual input, we can either wait for a certain file to exist (known as a semaphore) or we can wait for an information node to be attached to the build.

The advantage of waiting for a file is that it is simple, requires little development effort, and it is easy for the person who wants to resume the build to create a file. The downside is that the person needs to know what file to create, where to create it, and that the file needs to be created somewhere that the build controller (or agent) can access it.

In this example, we call a fictitious UpdateDatabase activity. If the activity fails, we log a warning and then wait for the file %TEMP%\DbUpdated.sem to be created manually:

```
<TryCatch>
  <TryCatch.Try>
```



```
      <a:UpdateDatabase Text="..." />
    </TryCatch.Try>
    <TryCatch.Catches>
      <Catch x:TypeArguments="s:Exception">
        <ActivityAction x:TypeArguments="s:Exception">
          <ActivityAction.Argument>
            <DelegateInArgument x:TypeArguments="s:Exception" Name="exception" />
          </ActivityAction.Argument>
          <Sequence DisplayName="Update Database Manually">
            <mtbwa:WriteBuildWarning DisplayName="Write Build Warning"
                Message="[String.Format(&quot;Updating database failed, update manually and
                create the file %TEMP%\DbUpdated.sem on {0} to continue the build: {1}&quot;,
                Environment.MachineName, exception)]" />
            <DoWhile DisplayName="Wait For DbUpdated.sem">
              <DoWhile.Variables>
                <Variable x:TypeArguments="scg3:IEnumerable(x:String)" Name="SemaphoreFile" />
              </DoWhile.Variables>
              <DoWhile.Condition>[SemaphoreFile.Any()]</DoWhile.Condition>
              <Sequence>
                <mtbwa:FindMatchingFiles DisplayName="Find Semaphore"
                    MatchPattern="[Path.Combine(Path.GetTempPath(),
                                  &quot;DbUpdated.sem&quot;)]"
                    Result="[SemaphoreFile]" />
                <Delay Duration="[TimeSpan.FromMinutes(1)]" />
              </Sequence>
            </DoWhile>
          </Sequence>
        </ActivityAction>
      </Catch>
    </TryCatch.Catches>
  </TryCatch>
```

Information nodes are strongly-typed objects that can be attached to a build by adding them to the Information collection on the build's IBuildDetail object. The advantage of information nodes is that we can add and query them from anywhere that we can use the Team Foundation Build API. For example, we could send mail that contains a hyperlink that calls an ASP.NET page. The ASP.NET page uses the Team Foundation Build API to add an information node to resume the build. The disadvantage is that additional development effort is required to create both the information node and a tool that allows it to be set.

Do

Query Build

Check For Information Node

While (Information Node Doesn't Exist)

In these examples, we have used the semaphores and information nodes as binary operations. If they exist, then we continue; otherwise, we keep waiting. However, we can extend this technique to allow the person to provide input to the build process.

This example builds on the previous example by reading the contents of the semaphore file, and taking one of these actions:

1. If it contains "retry", delete the semaphore file, and retry the original operation by setting RetryDatabaseUpdate to True. This will cause the DoWhile activity to execute it again.
2. If it contains "continue", exit the DoWhile loop by setting RetryDatabaseUpdate to False.
3. If it contains "fail", fail the build by throwing an exception.
4. Otherwise, log that the semaphore contains an invalid value, delete the semaphore, and continue waiting for a new semaphore.

```
  <DoWhile>
    <DoWhile.Variables>
      <Variable x:TypeArguments="x:Boolean" Default="False" Name="RetryDatabaseUpdate" />
    </DoWhile.Variables>
```



```
<DoWhile.Condition>[RetryDatabaseUpdate]</DoWhile.Condition>
<TryCatch>
  <TryCatch.Try>
    <a:UpdateDatabase Text="..." />
  </TryCatch.Try>
  <TryCatch.Catches>
    <Catch x:TypeArguments="s:Exception">
      <ActivityAction x:TypeArguments="s:Exception">
        <ActivityAction.Argument>
          <DelegateInArgument x:TypeArguments="s:Exception" Name="exception" />
        </ActivityAction.Argument>
        <Sequence DisplayName="Update Database Manually">
          <mtbwa:WriteBuildWarning DisplayName="Write Build Warning"
            Message="[String.Format(&quot;Updating database failed, please update manually
              and create %TEMP%\DbUpdated.sem on {0}: {1}&quot;, Environment.MachineName,
              exception)]" />
          <DoWhile DisplayName="Wait For DbUpdated.sem">
            <DoWhile.Variables>
              <Variable x:TypeArguments="scg3:IEnumerable(x:String)"
                        Name="SemaphoreFile" />
            </DoWhile.Variables>
            <DoWhile.Condition>[SemaphoreFile.Any()]</DoWhile.Condition>
            <Sequence>
              <mtbwa:FindMatchingFiles DisplayName="Find Semaphore"
                MatchPattern="[Path.Combine(Path.GetTempPath(),
                              &quot;DbUpdated.sem&quot;)]"
                Result="[SemaphoreFile]" />
              <If Condition="[SemaphoreFile.Any()]" DisplayName="If Semaphore Exists">
                <If.Then>
                  <Switch x:TypeArguments="x:String" DisplayName="Switch Semaphore Value"
                    Expression="[File.ReadAllText(SemaphoreFile.First()).ToUpper()]">
                    <Switch.Default>
                      <Sequence DisplayName="Handle Invalid Semaphore Value">
                        <mtbwa:WriteBuildWarning DisplayName="Invalid Semaphore Warning"
                          Message="[&quot;Semaphore contained invalid value, must be
                          &quot;&quot;retry&quot;&quot;, &quot;&quot;continue&quot;&quot;,
                          or &quot;&quot;fail&quot;&quot;.&quot;]" />
                        <InvokeMethod DisplayName="Delete File" MethodName="Delete"
                          TargetType="si:File">
                          <InArgument x:TypeArguments="x:String">
                            [SemaphoreFile.First()]
                          </InArgument>
                        </InvokeMethod>
                        <Assign DisplayName="Reset Semaphore List">
                          <Assign.To>
                            <OutArgument x:TypeArguments="scg3:IEnumerable(x:String)">
                              [SemaphoreFile]
                            </OutArgument>
                          </Assign.To>
                          <Assign.Value>
                            <InArgument x:TypeArguments="scg3:IEnumerable(x:String)">
                              [New String() {}]
                            </InArgument>
                          </Assign.Value>
                        </Assign>
                      </Sequence>
                    </Switch.Default>
                    <Assign x:Key="&quot;CONTINUE&quot;" DisplayName="Suppress Retry">
                      <Assign.To>
                        <OutArgument x:TypeArguments="x:Boolean">
                          [RetryDatabaseUpdate]
                        </OutArgument>
                      </Assign.To>
                      <Assign.Value>
                        <InArgument x:TypeArguments="x:Boolean">False</InArgument>
                      </Assign.Value>
```

```
                    </Assign>
                    <Sequence x:Key="&amp;quot;RETRY&amp;quot;"
                              DisplayName="Retry Database Update">
                      <InvokeMethod DisplayName="Delete File" MethodName="Delete"
                        TargetType="si:File">
                        <InArgument x:TypeArguments="x:String">
                          [SemaphoreFile.First()]
                        </InArgument>
                      </InvokeMethod>
                      <Assign DisplayName="Request Retry">
                        <Assign.To>
                          <OutArgument x:TypeArguments="x:Boolean">
                            [RetryDatabaseUpdate]
                          </OutArgument>
                        </Assign.To>
                        <Assign.Value>
                          <InArgument x:TypeArguments="x:Boolean">True</InArgument>
                        </Assign.Value>
                      </Assign>
                    </Sequence>
                    <Throw x:Key="&amp;quot;FAIL&amp;quot;" Exception="[New Exception(
                      String.Format(&amp;quot;Build failure requested by {0}.&amp;quot;,
                      SemaphoreFile.First()))]" />
                  </Switch>
                </If.Then>
                <If.Else>
                  <Delay Duration="[TimeSpan.FromMinutes(1)]" />
                </If.Else>
              </If>
            </Sequence>
          </DoWhile>
        </Sequence>
      </ActivityAction>
    </Catch>
  </TryCatch.Catches>
</TryCatch>
</DoWhile>
```

Subdividing Build Processes

It is tempting to implement your entire build process as a single build process template, but there can be advantages to splitting it into multiple build process templates and multiple build definitions. Splitting your build process template into major phases, such as compile, test, and deploy, allows you to run each phase independently,(either automated or manually. You can even use different hardware.

To implement this technique there are three key concepts:

1. Create an activity that can queue a build. If you need to execute it asynchronously, you can create another activity to check the status of a build.
2. Add process parameters to your build definition that allow you to configure what definition, and possibly even which controller, to queue when the phase completes.
3. Establish a “process parameter contract” between your build process templates, that is, what information will be passed from one phase to the next.

There are multiple ways you could implement this technique, depending on your requirements. Some of the variations include:

- Each build definition could have a matching test and deploy build definition, such as Main, Main_Test, and Main_Deploy. Alternatively, you could share test and deploy definitions such as Main, Test, Deploy_Staging.

- You could have each phase queue the next phase , such as build queues test and test queues deploy, or you could have the build phase drive the process, such as build queues test, waits for it to complete, and then queues deploy.
- You could have some phases run automatically and others run manually, such as having build queues test automatically but specifying that someone has to manually queue the deploy definition.

In addition, the information in the "process parameter contract" will be very specific to your process templates and the inputs that they need.

To demonstrate how to implement this technique, we will have the build process template queue and wait for the test process template. We will allow the definition and controller names to be specified using process parameters. To keep the example simple, we will have a very simple "process parameter contract" that only includes the Build URI and Drop Location.

Firstly, here is the implementation for the QueueBuild activity we will use. A similar activity is available in the Community TFSBuild Extensions.[41] Notice that it passes the current build's URI and Drop Location to the queued build as process parameters named SourceBuildUri and SourceDropLocation, respectively:

```
using System;
using System.Collections.Generic;
using System.Activities;
using Microsoft.TeamFoundation.Build.Client;
using Microsoft.TeamFoundation.Build.Workflow;

namespace Activities
{
  [BuildActivity(HostEnvironmentOption.All)]
  public sealed class QueueBuild : CodeActivity<IQueuedBuild>
  {
    public InArgument<string> BuildDefinitionName { get; set; }
    public InArgument<string> BuildControllerName { get; set; }

    protected override IQueuedBuild Execute(CodeActivityContext context)
    {
      var buildDetail = context.GetExtension<IBuildDetail>();
      var buildServer = buildDetail.BuildServer;

      var buildDefinition = buildServer.GetBuildDefinition(buildDetail.TeamProject,
        BuildDefinitionName.Get(context));
      var buildController =
          buildServer.GetBuildController(BuildControllerName.Get(context));

      var processParameters = new Dictionary<string, object>();
      processParameters.Add("SourceBuildUri", buildDetail.Uri);
      processParameters.Add("SourceDropLocation", buildDetail.DropLocation);

      var buildRequest = buildServer.CreateBuildRequest(buildDefinition.Uri,
        buildController.Uri);
      buildRequest.ProcessParameters = WorkflowHelpers.SerializeProcessParameters(
        processParameters);

      return buildServer.QueueBuild(buildRequest);
    }
  }
}
```

We will now use that activity to queue the definition whose name is specified in the process parameter TestDefinitionName against the controller whose name is specified in TestControllerName:

```
<Sequence>
  <Sequence.Variables>
```

[41] http://tfsbuildextensions.codeplex.com


```
      <Variable x:TypeArguments="mtbc:IQueuedBuild" Name="TestBuild" />
      <Variable x:TypeArguments="mtbc:QueueStatus" Name="TestBuildStatus" />
    </Sequence.Variables>
    <a:QueueBuild BuildControllerName="[TestControllerName]"
      BuildDefinitionName="[TestDefinitionName]" DisplayName="Queue Test Definition"
      Result="[TestBuild]" />
    <While DisplayName="While Test Build Is Running"
      Condition="[TestBuildStatus &lt;&gt; QueueStatus.Completed]">
      <Sequence>
        <Delay DisplayName="Delay 5 seconds" Duration="[TimeSpan.FromSeconds(5)]" />
        <InvokeMethod DisplayName="Refresh Build" MethodName="Refresh">
          <InvokeMethod.TargetObject>
            <InArgument x:TypeArguments="mtbc:IQueuedBuild">[TestBuild]</InArgument>
          </InvokeMethod.TargetObject>
          <InArgument x:TypeArguments="mtbc:QueryOptions">
            [QueryOptions.All]
          </InArgument>
        </InvokeMethod>
        <Assign>
          <Assign.To>
            <OutArgument x:TypeArguments="mtbc:QueueStatus">
              [TestBuildStatus]
            </OutArgument>
          </Assign.To>
          <Assign.Value>
            <InArgument x:TypeArguments="mtbc:QueueStatus">
              [TestBuild.Status]
            </InArgument>
          </Assign.Value>
        </Assign>
      </Sequence>
    </While>
    <If Condition="[TestBuild.Build.Status &lt;&gt; BuildStatus.Succeeded]"
      DisplayName="If Build Failed">
      <If.Then>
        <Throw Exception="[New Exception(
          String.Format(&quot;Test build '{0}' has an unexpected status: {1}.&quot;,
          TestBuild.Build.BuildNumber, TestBuild.Build.Status))]" />
      </If.Then>
    </If>
  </Sequence>
  }
}
```

Dependency Management

A dependency exists when code is shared between two or more Visual Studio solutions and / or projects. There are two participants in a dependency. First, you have the source of the dependency. The source could be one or more Visual Studio projects (source code dependency), or it could be one or more .NET assemblies (binary dependency). Second, you have one or more dependent projects. These are Visual Studio solutions and / or projects that have references to the dependency source. The dependent project's reference might be a project reference (source code dependency) or assembly reference (binary dependency).

Dependency management is one of the most difficult parts of creating a reliable, repeatable, and low-maintenance build process. One challenge is that introducing dependencies is extremely easy (right-click a project and choose Add Reference). Validating that a dependency will not break other developers or a build is difficult. In other words, just building a solution on your machine is not sufficient to validate a dependency. Often, developers do not understand dependency issues and simply take dependencies to assemblies in their file system or GAC that are not in version control or available to the build server. It is easy to solve in a way that will increase the long-term cost of your build process. Just install the dependency on your build machines.

In an ideal world, the source of every dependency of your build process is stored in version control. In this way, performing any build can be as simple as synchronizing to the version you want to build and then running your build process. In reality, like all decisions, there are trade-offs to be made.

When developers properly manage dependencies, they will be happier because all they need to do to start working on a project is to synchronize that project's source code as well as the dependency sources, open the project, and start working. No more having to GAC assembly X, install application Y, and track down the assembly that Bob has in C:\Temp just to find he's replaced it with a new version that has breaking API changes in it. Unfortunately, some dependencies require installation for their design-time experience to work correctly.

Creating a Dependency Repository

A dependency repository is where the source (source code or assemblies) of a dependency is stored. The most developer- and build-friendly way to manage dependencies is to store them in version control alongside the projects that consume them. There are a few different strategies you can use. We will discuss these later, but they all use the following structure:

```
<Root Dependencies Folder>
  <Vendor>
    <Product>
      <Version>
        <Dependency Files>
```

Dependencies can be organized into folders by vendor, such as Microsoft; product, such as Enterprise Library; and version, such as 5.0. For dependencies that come from your team, such as a common database library, consider creating a vendor called "Internal" or your team name. Later we will discuss how your build process can automatically publish your dependencies.

This consistent structure helps developers easily find the files they need to take on a dependency and enables multiple versions to be stored side-by-side, which allows different projects to take dependencies on different versions. Dependency files in the repository can be source code, assemblies, readme files, license keys, shortcuts to the vendor's website, or anything that a developer might need to use the dependency.

You might not want to create a folder for each version. For example, if a dependency has a history of not making breaking changes in minor releases, and you always want dependent projects to target the latest minor version, you might just create a folder for the major release and "upgrade" the dependency source files in this folder each time a new minor version is released.

There are a few options for locating the Root Dependencies Folder (the source of the dependency):

1. A centralized, dedicated Team Project (for example $/Dependencies). This is a good approach for sharing dependencies across multiple team projects.
2. A dependencies folder in the root of a product's or project's Team Project, for example, $/Contoso/Dependencies. This is a good approach for centralizing dependencies and promoting re-use.it will also suit the needs of most teams.
3. A dependencies folder under each dependent product or project, for example, $/Contoso/ProjectX/Dependencies).
4. A dependencies folder under a branch in the dependent project's Team Project, for example,$/Contoso/ProjectX/Main/Dependencies).

In each of the options above, the dependent project(s) will create either creating project or assembly references to files in the respective Dependencies folder. For option 1, there should be sufficient trust across teams to allow dependencies to be centralized because dependency references will span Team Projects.

Options 3 and 4 might cause dependencies to be unnecessarily duplicated, but each offers benefits. For example, option 3 can be useful if a product's or a project's dependencies need to be isolated,(for example, to simplify workspace mappings or for security or management reasons. Option 4 offers the advantage of isolating all of the

dependencies needed for a particular version of a product or project. Also, because they will typically only reference a single version of a dependency at a time you could remove the Version level from the structure and always “upgrade” in-place.

Options 2-4 are also not necessarily mutually exclusive to having a centralized dependency repository. Consider branching from the centralized repository into the respective Dependencies folders.

Please also see the Integrating with NuGet section on page 199 for more dependency management options.

Referencing Dependencies

The easiest way to allow both developers and the build agents to build projects that use a central dependency repository is to use relative file (not GAC) references to the dependency sources. Relative references are relative from the location of the project file (*.csproj, *.vbproj, etc.) not to the solution (*.sln). When you add a reference to an assembly on the same drive as the project file, Visual Studio will automatically make it a relative reference.

If a dependency is in your GAC when you add a reference to it, its hint path,(which is used to locate the assembly, will not be populated even if you browse to the assembly on disk. This will cause build failures when you check in because the build process will not be able to resolve the location of the assembly. To prevent build failures with assemblies in the GAC, either ensure that dependencies are not in your GAC before you add a reference to them, or manually edit the project file after you add the reference and then add a <HintPath> element. To add the missing hint path:

1. Right-click the project in Solution Explorer and click **Unload Project**.
2. Right-click the project again and click **Edit <Project Filename>**.
3. Locate the <Reference Include=”<Assembly Name>”> element for the reference you want to update.
4. Add a <HintPath> element as a child of the <Reference> element with the relative path to the dependency, for example:

```
<Reference Include="SharedLibrary">
  <HintPath>..\..\Dependencies\Contoso\SharedLibrary\1.0\SharedLibrary.dll</HintPath>
</Reference>
```

For relative-reference paths to work effectively, follow these guidelines:

1. Developers and build machines must use the same workspace mappings. At the very least, keep the dependencies root location and the projects at the same relative depth with respect to each other. It is recommended that you keep your workspace mappings simple and have your local folder structure match the server folder structure.
2. If the dependencies are not contained within a branch (option 4), keep all branches for a given product/project at the same folder-depth with respect to each other. For example, branch **$/Contoso/ProjectX/Main to $/Contoso/ProjectX/ReleaseV1** rather than **$/Contoso/ProjectX/Releases/V1**. If the folder depth differs between branches, you will need to update all of the relative references, which will complicate creating the branch and ongoing merging between branches.

When you configure your build definitions, you will need to configure the workspace mappings to include both the source being built as well as the required external dependencies. Be sure to keep the relative folder structure consistent. Although you could configure the build definition to map the whole dependency repository, this will cause clean builds to have to sync the whole dependency repository. This will make it possible for different projects in the same solution to reference different versions of a dependency, which is rarely desirable. It will also cause gated and continuous integration builds to launch unnecessarily. Instead, consider mapping just the particular dependencies that are used by the product/project.

Important Note: A concern with option 3 or 4 for smaller teams, or teams with many applications, is that it may be massive overkill. Suppose a team is responsible for sixty (60) applications, all of which reference the same shared .NET Assembly DLL. Whenever this DLL is updated, the team would need to update it in the central repository and then merge it 60 times.

Alternatively, consider option 2. Use relative references to the shared dependencies folder. Here the shared assembly can be updated once. The primary downside of option 2 is you need to take precautions to make the relative references work, namely, keeping branches at the same folder depth with respect to each other.

Using Team Foundation Build in Heterogeneous Environments

WHAT'S IN THIS CHAPTER?

- A possible architecture for building applications in heterogeneous environments
- Building code in heterogeneous environments
- Using the BRD lite included build process template to build code in heterogeneous environments

Introduction

Many organizations face the need to develop in heterogeneous environments, that is, environments including both Microsoft Windows and non-Windows based operating systems. Their code not only has to be built/executed on Windows but also on non-Windows environments. The objective of this guidance is to help you understand how you can build your code in such a heterogeneous environment using Team Foundation Build.

To be clear, the target audience for this guidance is those who are using Team Foundation Build and want to build their code on non-Windows operating systems. Although the principles on this guidance are generic, our main focus is how to build code on UNIX based operating systems.

Using Team Foundation Server for heterogeneous development

There are two common misconceptions regarding Team Foundation Server.

- First, that it can only be used in conjunction with Visual Studio. This is not true; there are multiple non-Microsoft development environments running on Windows that can integrate with Team Foundation Server either using the Team Foundation Server MSSCCI Provider 2010[42], using Team Foundation Server SDK[43] or by integrating directly using the provided Web Services.
- While the first misconception is not very prevalent, the second is much more common; the idea that Team Foundation Server can only be used on Microsoft Windows operating systems; this is also not true. Team Explorer Everywhere[44] can be used on non-Windows[45] operating systems to develop code that will be executed on a non-Windows operating system. In addition the Team Foundation Server SDK for Java provides a full SDK for developing cross-platform applications using Java that talk to Team Foundation Server's web services

With this guidance however, we want to go a little further, we also want to be able to build code for development environments not directly supported by Team Explorer Everywhere, e.g. iPhone applications developed using XCode.

NOTE

This guidance is not about building Java code. To build your Java code (if you use Ant or Maven) you can use the Build Extensions Power Tool[46] to build your Java code on Windows based build agents without having to resort to the techniques described on this chapter. Due to the portable nature of Java, these projects can be deployed to non-Windows operating systems easily. If you are using Team Explorer Everywhere you can even generate the build definitions for your Java projects using the built in wizard in no time. You can read more about this at http://msdn.microsoft.com/en-us/library/gg490754.aspx

[42] http://visualstudiogallery.msdn.microsoft.com/bce06506-be38-47a1-9f29-d3937d3d88d6

[43] http://archive.msdn.microsoft.com/TfsSdk

[44] http://www.microsoft.com/visualstudio/en-us/products/2010-editions/team-explorer-everywhere

[45] http://www.microsoft.com/visualstudio/en-us/products/2010-editions/team-explorer-everywhere/system-requirements

[46] http://visualstudiogallery.msdn.microsoft.com/2d7c8577-54b8-47ce-82a5-8649f579dcb6

Building code on non-Windows environments

The fact that we can use Team Explorer Everywhere to develop on non-Windows environments doesn't mean we can use Team Foundation Build out of the box to build our non-Windows applications, since the Team Foundation Build only runs on Windows[47].

To build the code on a non-Windows environment we basically have three options

1. Use a third party build system that can talk to Team Foundation Server for version control but that runs on the non-Windows environment. Hudson/Jenkins and CruiseControl are both popular open source build systems in the non-Windows space and they can use Team Foundation Server for version control.
2. Use a build tool chain[48] that runs on Windows but produces code that can be executed on non-Windows environments. In its simplest form a cross compiler[49] could be enough. A cross compiler is typically used when we want to produce code that will be executed on an environment for which the compilation is too costly or doesn't have enough resource to perform a compilation (embedded systems).
3. Delegate the building of the code to an external machine, while the orchestration of the whole process is still done on a Windows machine so we can reap all the benefits of Team Foundation Build without having to implement in a non-Windows platform all of the goodness it currently provides (to name a few: code labeling, copying output to drop folders, gated check-ins, etc.).

Option 1 is outside the scope of the document. While the solution works well, it does not feedback data about that build process into Team Foundation Server. In addition using a third party build system makes it complex to integrate those build results with other builds that are occurring with-in Team Foundation Server. Option 2 is valid but, its usefulness is limited since these scenarios are more unusual; we will therefore solely focus on option 3 in this guidance.

At the risk of oversimplifying, our workflow will be implementing something along the following lines (omitting a lot of steps that a typical build template would do):

1) Place the code on the external machine
2) Invoke the build scripts on the external machine (those build scripts written in the appropriate build orchestration language for that platform and deliverable such as make, Ant, Maven or even just a collection of shell scripts)
3) Place the result of the build on the output folder

To be a little more specific, a build template definition to build code on non-Windows machine, could be something along the lines of the build definition template pictured.

[47] http://msdn.microsoft.com/en-us/library/dd578619.aspx

[48] http://en.wikipedia.org/wiki/Toolchain

[49] http://en.wikipedia.org/wiki/Cross_compiler

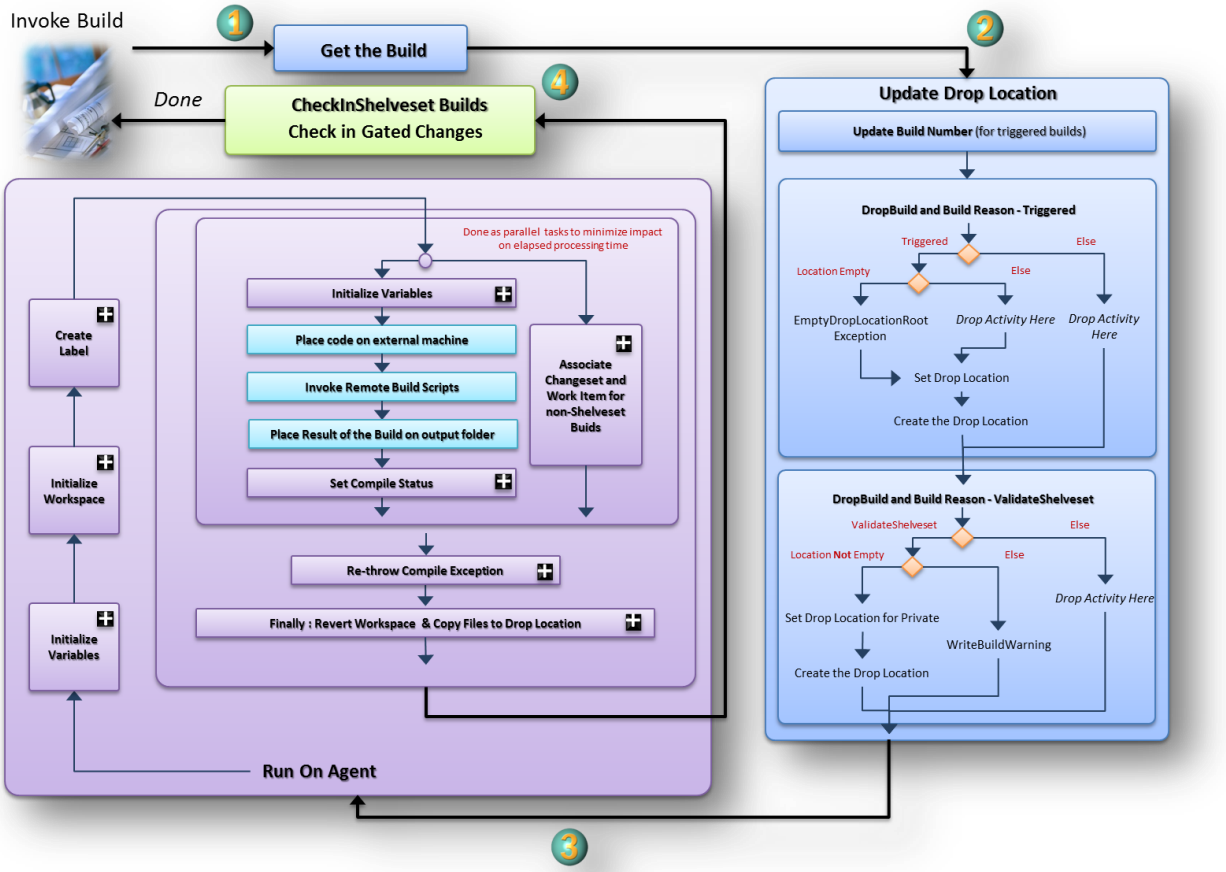


Figure 87 – (basic) Build definition template for building code on non-Windows machines

We are basically (at a conceptual level) extending the notion of the build agent to a non-Windows platform. We mean conceptually, because although they can be considered part of the build infrastructure they are not bound nor registered to be part of it. Therefore they are not part of the formal build infrastructure but they are part of the build farm that is used and leveraged to build our code on UNIX machines[50]. Stretching the official build architecture we could say (a possible) build architecture for heterogeneous environments could be the one pictured in **Error! Reference source not found.**

[50] We are not limited to use UNIX machines, we can use any machine for which we can remotely execute code and copy files from/to but since we are focused mainly on Unix scenarios we will refer to Unix machines

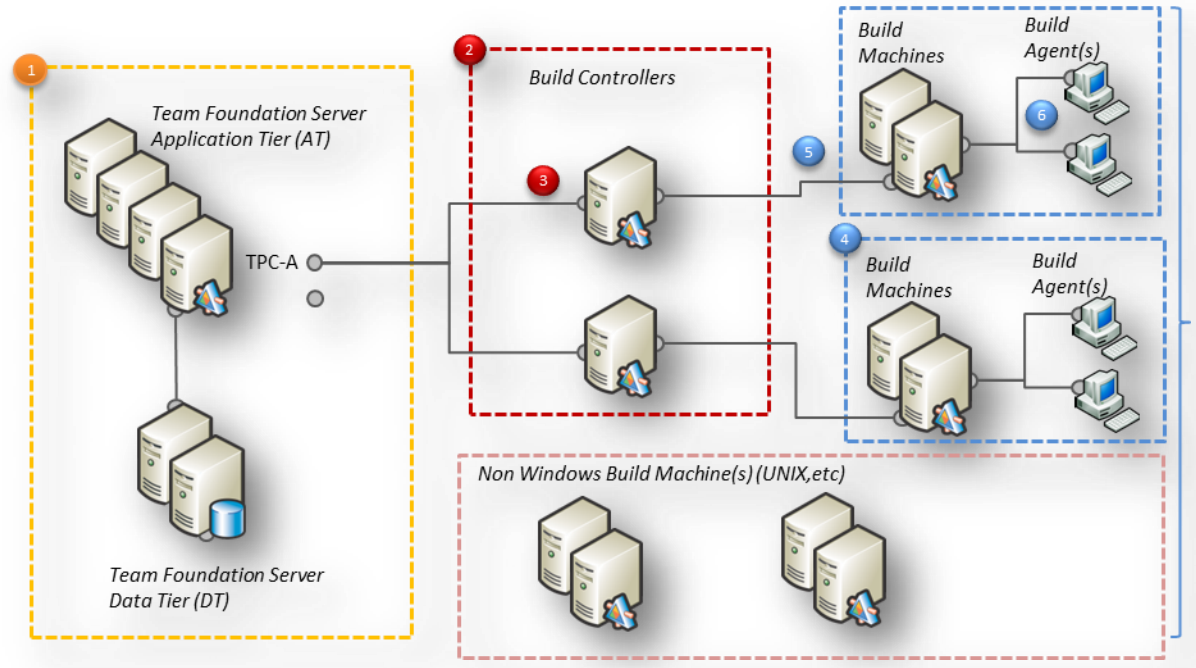


Figure 88 – A possible build infrastructure for heterogeneous environments

Placing the code on the external machine

Before building the code, we need to have the source on the external machine. To place the code into the external build machine we have two options

1. **Push the code** – After getting the code from source control on the build agent the code will be pushed onto the external build machine.
2. **Pull the code** – The code will be pulled from the external build machine directly from Team Foundation Server source control (for example using Team Explorer Everywhere command line[51])

While the pull method may be more efficient (since the Team Foundation Server client will only pull files that have been changed), it not only requires a more complicated setup (we have to maintain and configure the workspaces), but it also adds a dependency on Team Explorer Everywhere to the external build machine (reducing the number of dependencies reduces the servicing effort, which increases exponentially with the number of machines.).

Pull method also has the advantage over push that UNIX file attributes are preserved (the execute bit may be important if we wish to execute scripts) whereas they can be lost if we push files from Windows to UNIX via a file transfer protocol. But this can be easily overcome in the push method if we manually set the necessary attributes prior to executing any script.

NOTE

In Team Foundation Server 2012 enhancements have been made to both Team Foundation Server and Team Explorer Everywhere to preserve UNIX file attributes

[51] http://msdn.microsoft.com/en-us/library/gg413282.aspx

Therefore we recommend the use of the push method since it will require less maintenance although we may face a penalty hit for larger codebases (which can be mitigated by tools that only copy changed files).

We have several options to push the code to the remote machine; all the listed options are commonly available on UNIX machines and accepted by most system administrators (the more standard the methods is used, the less resistance we face with system administrator, so we place our focus on standard methods which are familiar to system administrators) as a valid way to transfer files

- rcp[52] - remote copy is the original UNIX command for copying files to remote systems. It is very pervasive, but it is showing its age because it was created in an era where trust and authentication between machines wasn't an issue. System administrators tend to disable this service due to these issues.
- ftp[53] - it is probably the most universal method for transferring files between machines. It can not only transfer files between machines but perform other file operations (like creating directories or deleting files). Although it is the most common in use, administrators tend to prefer sftp[54] (a secure replacement for ftp) since ftp doesn't guarantee encryption and transmits passwords in plaintext.
- scp[55] - in order to overcome the limitations of rcp, scp was created. It relies on ssh[56] to guarantee authentication and a secure channel between the two machines. It is typically the preferred method for file transfers.
- rsync[57] - rsync is an incremental file transfer protocol which can only skip files that have not been changed but it can also only transmit the deltas for files that have changed. For larger code bases (rsync by default works over SSH so it is a secure and fast option) in which we can keep the build files on the external machine between builds so it may be a way to reduce the build times by minimizing the time spent on copying files between machines

Although you can use any of these methods, this guidance will be based around an SCP/SFTP activity since it is the most pervasive available option.

Invoke the build scripts on the external machine

The execution of the build is delegated on the remote build machine, so we need a way to

- remotely execute the build command
- get the output of the build (so we can log and monitor the result of the building)
- get the result of the build as whole to determine if the build has been successful

For our purposes we don't need to know how the code is built remotely. The build definition template will know how to remotely invoke a remote shell script responsible for building the code. We treat building as a black box and are totally abstracted of the way it's done. We only need to know if the build was successfully or has failed. The code can be built using Make[58], Ant[59], Rake[60] or any other tool of your choice that can be invoked via shell commands and doesn't require user interaction.

In order to remotely execute code we typically have several options at our disposal

[52] http://en.wikipedia.org/wiki/Rcp_(Unix)
[53] http://en.wikipedia.org/wiki/Ftp
[54] http://en.wikipedia.org/wiki/SSH_file_transfer_protocol
[55] http://en.wikipedia.org/wiki/Secure_copy
[56] http://en.wikipedia.org/wiki/Secure_Shell
[57] http://rsync.samba.org/
[58] http://en.wikipedia.org/wiki/Make_(software)
[59] http://en.wikipedia.org/wiki/Apache_Ant
[60] http://en.wikipedia.org/wiki/Rake_(software)

- rsh[61] - The oldest method of remotely invoking shell code on a Unix system. The use of this option has been widely deprecated since it uses an unencrypted channel and if user credentials are used the passwords are plaintext
- ssh[62] - the most used and pervasive method of invoking shell code remotely. It is secure and supports several methods of authentication.

Although both methods are valid, this guidance will be based around a SSH workflow activity to remotely invoke the build scripts.

Place the result of the build on the drops folder

After the build is finished we need to copy the output of the build (an executable, shared libraries or whatever artifacts the build produces) into the drops folder

In order to achieve this objective we have two different methods at our disposal

1. **Pull the outputs** - Pull the outputs from the remote machine into the machine where the build agent is executing and then copy them to the output folder as the default build templates do (we place it on the outputs folder, since this is same method as used by the default template (which will copy them to the drops folder if the option to copy to drops folder is enabled). We can simplify this and opt to copy the output directly to the drops folder).
2. **Push the outputs –** The push method is faster than the pull method since it requires less copying, however it requires that extra management overhead, since it not only requires a client that can copy files to a SMB/CIFS[63] folder (the drops folder resides in a network share), while this can easily be achieved by using Samba[64] it may also require extra steps to manage permissions.

While the pull method may require some overhead with the extra copying of data typically this is not an issue, so the pull method will be used, in order to eliminate the need to perform installations or configuration on the remote build machine. (Again we choose to minimize servicing needs and configuration on the remote build agents in order to minimize friction for this process)

Building code on non-Windows environments using the BRD Lite Template

BRD lite includes a build process template called **HeterogeneousEnvTemplate** which can be a starting point to build code using in a now windows environment (we say starting point, since the template while autonomous and is fully capable of build code on a non-windows machine, it represents one way of doing. We are not advocating it's the only way to do it).

This template uses the "push" method describe previously, in order to copy and invoke build scripts on remote machines it uses the SSH. It has mainly three requirements to be used

- A machine to build the code with an SSH daemon installed (designated remote build agent from now on)
- Community TFS Build Extensions[65] activities (December 2011 release or later) on the build machines
- PuTTY[66] SSH client installed on the build agent machine(s)

To recap these are the actions (in broad and simplistic terms) the build process template performs

- Get the files from source control
- Label the fetched files (if not disabled)
- Push source code to remote build machine

[61] http://en.wikipedia.org/wiki/Remote_shell

[62] http://en.wikipedia.org/wiki/Secure_Shell

[63] http://en.wikipedia.org/wiki/CIFS

[64] http://www.samba.org/

[65] http://tfsbuildextensions.codeplex.com/

[66] http://www.chiark.greenend.org.uk/~sgtatham/putty/

- Invoke build script(s) on remote build machine
- Pull compilation result from result build machine and into the drops folder (optional)

The build process template supports gated check-ins

NOTE

Although PuTTY doesn't require any special installation (it supports xcopy installation) it should be installed using the provided MSI on the official site.

The SSH activities rely on the either installation being done by the MSI (which registers its location on the registry) or PuTTY executables being available on the path.

Remote Agent Authentication

Before communication is possible between the build agent(s) and the remote build agent(s) some steps need to be taken care off. The first step is guaranteeing that the remote build agent is known to the build agent machine.

This is a step required by SSH in order to insure that we are connecting to a machine which is trusted in order to avoid spoofing attacks.

This requires logging on to the remote agent from the build machine (it *has to be done* with the account under which the build agent runs) and accept the host key[67] as valid (this is only required to be done once. After that the host key will be stored locally and the machine will be trusted (no globally to the machine but only for the user who accepted the machine).

This manual step is not practical since it requires you to manually login on each build agent machine, making it unpractical if you have several build machines or you will add new build agent machine(s) later. It may also prove an action impossible to be performed if your build agent account is an account that doesn't allow interactive logins (e.g.: the Network Service which is the default build agent account (see Figure 89)).

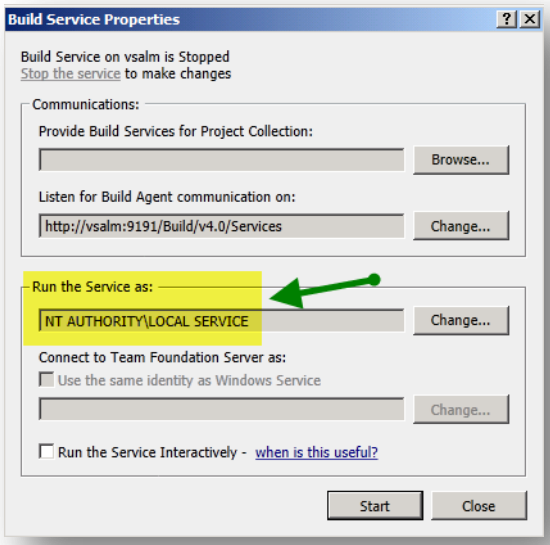


Figure 89 – Build Agent running with Network Service account

In order to ease manageability of the build agents the SSH activities included in Community TFS Build Extensions allow you to specify a known hosts file (which can either be stored in source control or in a well-known directory of

[67] http://the.earth.li/~sgtatham/putty/0.62/htmldoc/Chapter2.html#gs-hostkey

the build agent machine). We still have to create this file manually, but it can be done in a machine of your choice and under any login. This is optional, you only need to do this if you don't want to log in interactively on the build agent(s) machines and recognize the hosts manually.

To create this file run PuTTY and connect to the remote build agent machine(s) by entering the host name/address and pressing **Open** (Figure 90)

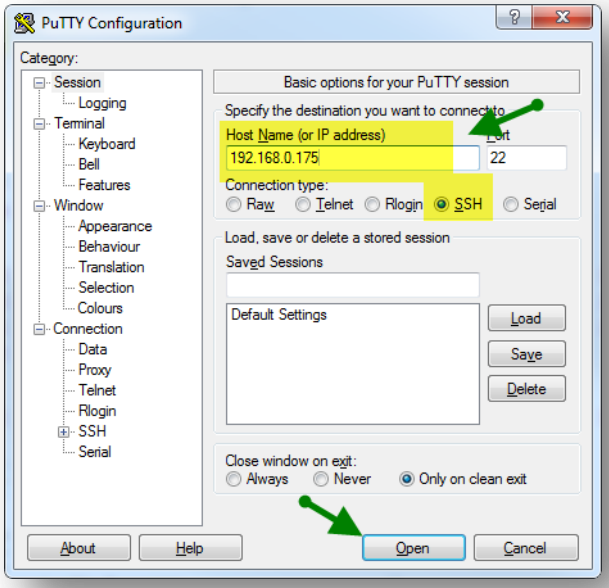


Figure 90 – SSHing to a host using PuTTYs

After you click Open (assuming you have never connected to the particular host) you will be presented with a windows stating that the host is now known (see Figure 91).

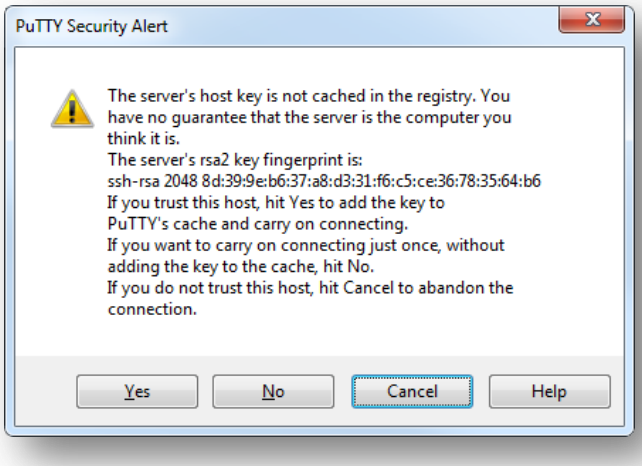


Figure 91 – PuTTY security alert that we are connecting to an unknown host

After checking that the host private key fingerprint is correct you should select **Yes.** After you do this, proceed normally or just close the PuTTY window. At this time the host fingerprint is already stored on the registry. Repeat this for all the hosts that you intend to use as remote build agents.

After repeating accepting all hosts that will be used as remote build agents we need to export this information to a file (we call this file the well-known hosts). Since PuTTY stores this information on registry we can export this information to a file using regedit.exe or reg.exe

You can export this information to a file by running the following command at a command prompt (Figure 92) the filename (knownhosts.reg) is merely an example you can use whatever name it suits you.

reg.exe export HKEY_CURRENT_USER\Software\SimonTatham\PuTTY knowhosts.reg

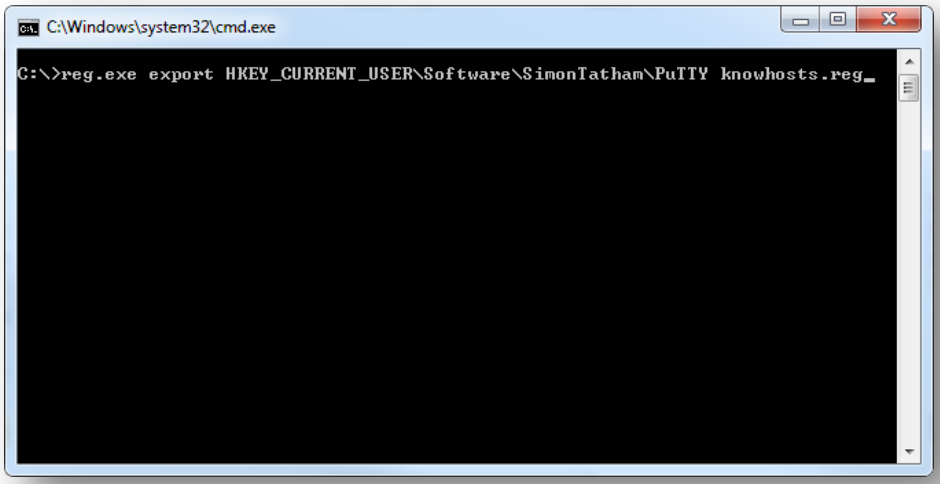


Figure 92 – Exporting the known hosts information to a file

After storing this info on a file you have two options.

- Copy these file to a well-defined directory to all build agent machine(s)
- Store this file on source control (the recommended method). This way the file is centrally machine and you do not need to keep updated version on all build agent machine(s). You can also add new build agent machines without having to manually configure them. It is recommended that this file is stored in a source control folder with restricted permissions so only authorized persons can add remote build agents (not only this file can be used to add unauthorized remote build agents but it can also be used to load arbitrary registry keys since we do not perform any validation on the content of this file)

NOTE

Every time you add a new remote build agent, you will have to repeat this procedure for that particular host.

When supplied with this file the SSH activities will automatically add this information to the registry (with the credentials of the build agent running account).

To use this file (optional) all you need to do is specify it in the **Know Hosts file** parameter in the build definition parameters (Figure 93). You can either specify a file stored in the build agent file system or stored in source control. The format will be automatically picked up and the SSH activities will use the file where it is stored.

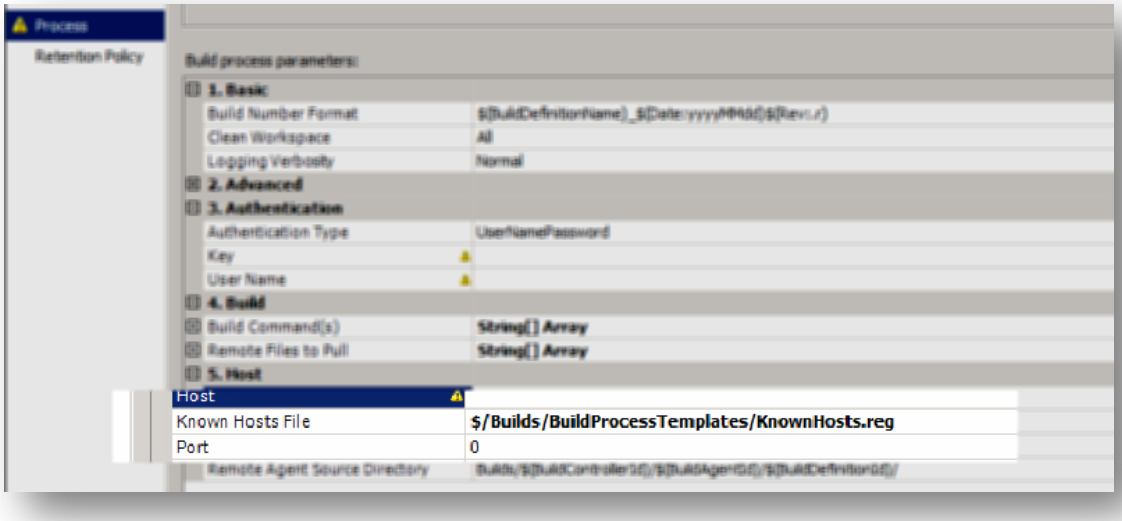


Figure 93 – Specifying the Known hosts file in the build definition parameters

Authentication

Now that we are able to try to connect to a host we still have to authenticate before we are allowed to perform any operations on it.

All operations performed on the remote build agent, will be perform with a user (of the remote machine), it is recommended that this user doesn't have any special privileges on the remote machine (unless the build process requires it). This user will have to be authenticated before it is allowed to perform any operation on the remote machine. We support two kinds of authentication

- **Password** – The simplest authenticating mechanism. You just have to specify the password for the user that will be used to execute commands on the remote build agent. This was the disadvantage of the password being visible to anyone who can see a build execution report.
- **Private Key file** - The user is authenticated using a private key file. This requires that the private key file be accessible to the build process, therefore either stored in version control or loaded onto the build agent machines. With the private key the build service does not require a password to authenticate on the remote machine as the user, therefore if the remote users password changes no action is required to the build process.

The user is specified in the User Name parameter (Figure 94)

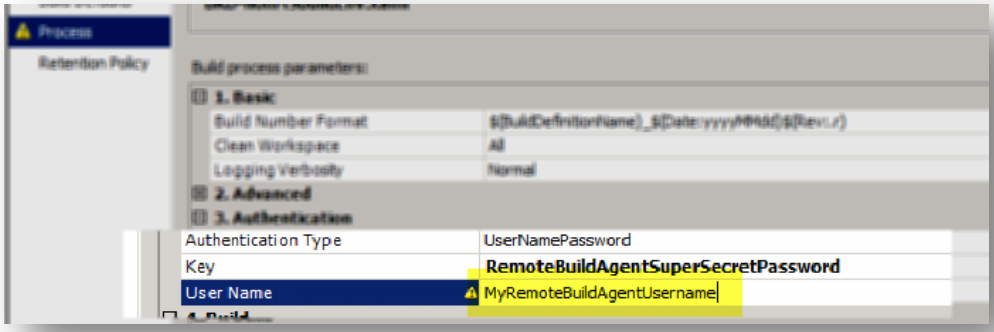


Figure 94 – Specifying the remote user name

Password authentication

To use password authentication, we need to select the **UserNamePassword** option in the Authentication type parameter (Figure 95) and specify the password in the **Key** parameter (Figure 96)

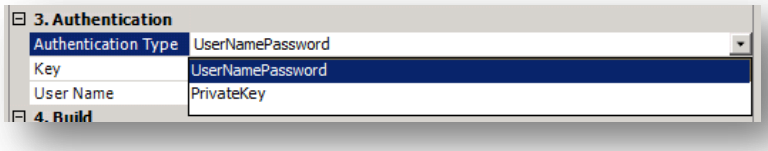


Figure 95 – Authentication type selection

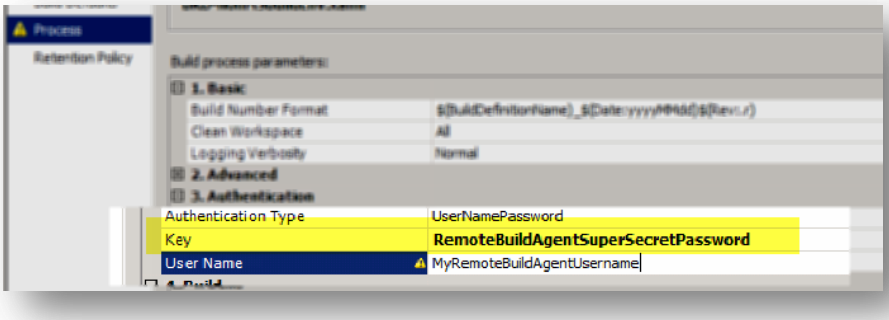


Figure 96 – Specifying a password for username authentication

Private Key authentication

If you wish to use private key you will have to generate a private key on a machine (using PuTTYGen), export it to a file and place the private key file on the build agent file system (which will have to be replicated among all build agent machine(s)) or place it on source control (it is recommended that this file is placed on a folder with restricted permissions so only a few users are allowed do view/edit this file).

We will omit how you can generate your key for authentication, since it is thoroughly explained in Chapter 8 of PuTTY documentation[68]

[68] http://the.earth.li/~sgtatham/putty/0.62/htmldoc/Chapter8.html#pubkey

After generating your key and exporting it to a file (section 8.2.8[69]) you either save the file on the build agent machine(s) file system or (recommended) add it to source control and reference it on the build definition parameters.

In order to use a private key, two parameters need to be defined. **Authentication Type** (Figure 95) will have the **PrivateKey** value and the **Key** parameter will hold a reference to the private key file (Figure 97) (either located in the filesystem or the in a source control folder)

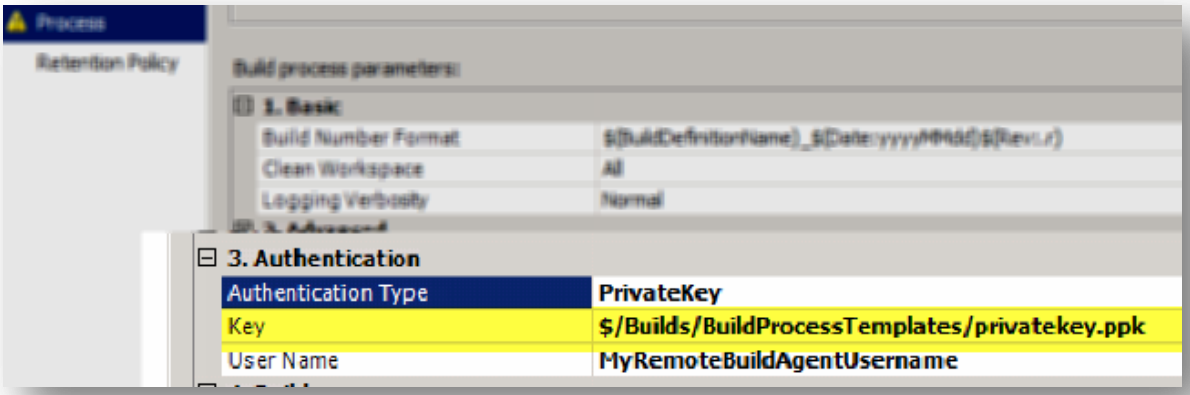


Figure 97 – Specifying private key authentication file

NOTE

Don't forget the public keys will have to be copied to the remote build machine(s) user account SSH folder (the exact procedure depends on the SSH daemon you are using, but typically it is stored under **~/.ssh/** (Figure 98))

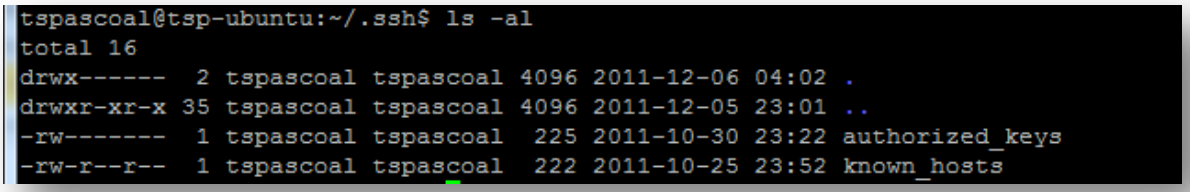


Figure 98 – SSH authorized hosts files

Push source code to remote build agent

Pushing the source code is pretty automatic; all the folders that are mapped in Workspace parameter (Figure 99) are automatically pushed to the remote build agent when the build is triggered.

[69] http://the.earth.li/~sgtatham/putty/0.62/htmldoc/Chapter8.html#puttygen-savepriv

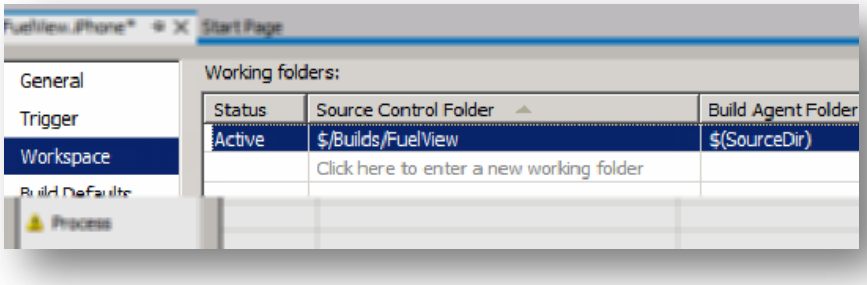


Figure 99 – Workspace parameter configuration

The files will be copied to a directory located in the home directory of the remote build agent user (Figure 94). The base directory is specified in the parameter **Remote Agent Source Directory** (Figure 100)

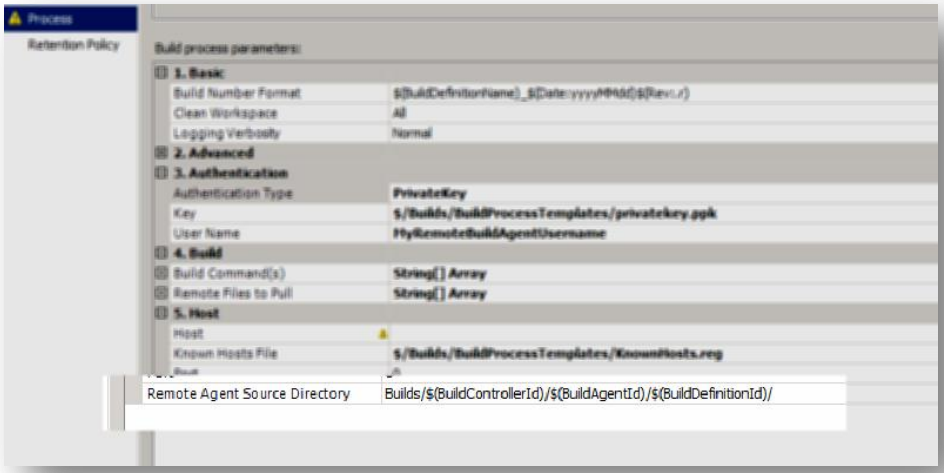


Figure 100 – Remote Agent Source Directory

By default the base directory is located at **Builds/$(BuildControllerId)/$(BuildAgentId)/$(BuildDefinitionId)/** unless you have specific reason you shouldn't change this value (see the section Available Variables for the list of available variables for use in these fields)

While at first this may seem more complicated than it should, it allows us to have concurrent builds in the same build remote agent coming from more than one build agent (and even different build controllers).

We have opted to use the Identifiers (Build controller identifier and build agent identifier) instead of using the more (human) readable names (e.g. $(BuildAgentName)). While this certainly help looking at the directories in the remote build agent to understand where the files are coming from using the more human readable has two issues

- It adds unnecessary complication, since the more human readable forms usually have spaces. This may require quoting when executing scripts and copying files.
- It may even render the building of some projects impossible, since some build scripts don't work when they are inside a directory that has spaces.

NOTE

The result of the expansion of the remote build source directory (Figure 100) will be placed in the variable **$(RemoteAgentTargetSourcePath)** that can be used to refer to the remote build sources directory without having redundant references.

If you change this value, don't forget to add the trailing slash (/) to the directory. It is advisable that this directory ends with a slash so scripts don't have to worry about it when constructing paths.

This path can be either absolute or relative (the default value is relative to the home directory of the user running the remote builds). It is advisable that the path is relative for easier maintenance (no need to deal with permissions for example)

Building the code remotely

After the source code is pushed to the remote build agent, we are ready to build it (in our examples building is restricted to compilation, but there is nothing to prevent us to do more than just compilation (e.g.: documentation generation, running unit tests, deploying, etc.).

NOTE

Keep in mind that all remotely executed commands are done in the user home folder (~) while the source code for a given build definition are stored by default on ~/Builds/$(BuildControllerId)/$(BuildAgentId)/$(BuildDefinitionId)/

In the **Build Command(s)** parameter (Figure 102) you can specify the command(s) to be executed on the remote build agent. You can specify has many shell commands as you want, but keep in mind that each line will be executed independently (different connection) in a sequential manner (if one fails the remaining ones will not be called) so a command can't depend on any context (nonpersistent) created by a previous command (e.g.: change of working directory or environment variable setting). The commands are nothing more than shell commands that will be executed remotely, if a command returns a value other than zero than it is considered that its execution has failed.

The output for the standard output is placed in the log file and the output for standard error is written to the build log as a build error message (the fact that an error appears on the build (Figure 101) doesn't mean the execution has failed. Only the return value determines if the execution has failed.

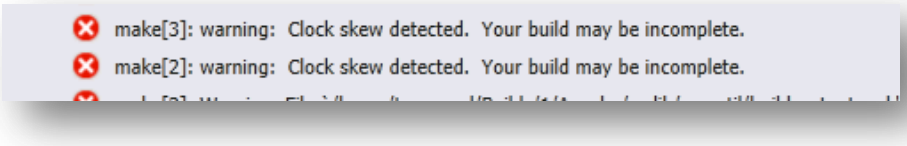


Figure 101 – A warning appearing as an error message in the build log

Assuming there is a shell file called **build.sh** in the sources folder and that can be executed, using the following commands (one per line) will not work.

cd "\"$(RemoteAgentTargetSourcePath)\""
./build.sh

If you want to call more than one command in the same call, you can separate them with a semicolon (;) (Assuming the remote build agent user default shell is C Shell[70], Bourne Shell[71] or Bourne Again Shell[72] other shells may use a different command separator)

[70] http://en.wikipedia.org/wiki/C_shell
[71] http://en.wikipedia.org/wiki/Bourne_shell
[72] http://pt.wikipedia.org/wiki/Bash

cd "\"$(RemoteAgentTargetSourcePath)\"" ; ./build.sh

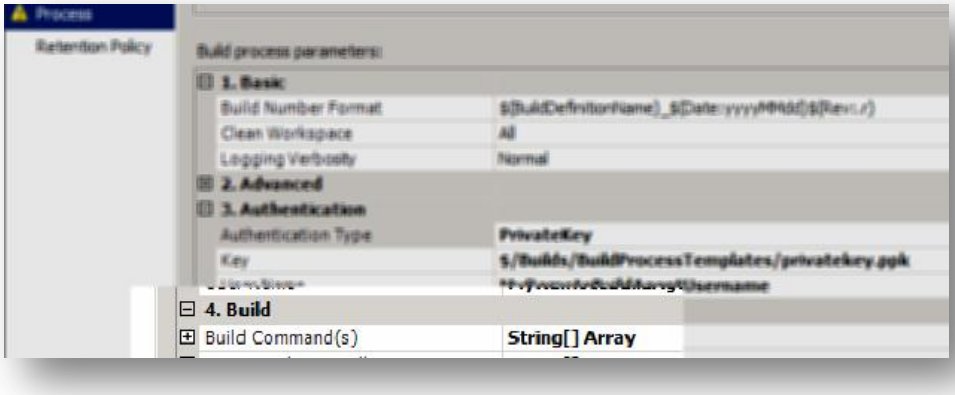


Figure 102 – Build Command(s)

If you recall earlier, we said the pull method has the drawback of not preserving UNIX file permissions, so the correct command in this case would be

cd "\"$(RemoteAgentTargetSourcePath)\"" ; chmod –f +x build.sh ; ./build.sh

Although you can specify all your compilation commands in the build definition parameters it is recommended that you keep your build script in a source control file and your commands only sets the execute flags, changes the current directory to the sources directory and executes the script. This method has several advantages

- Much easier to test build scripts (every developer can test it on their own machine). Writing a full shell script as parameters can be a daunting task
- Full traceability of the build scripts
- As an added bonus no need to quote parameters

So for example (to build Apache HTTP Server[73]) we could have (Figure 103)

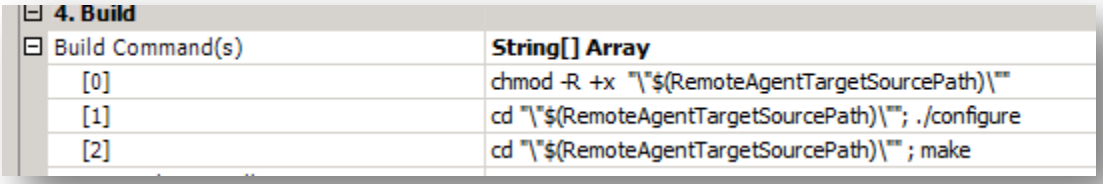


Figure 103 – Commands to build Apache Web Server

But we should instead have

[73] http://projects.apache.org/projects/http_server.html

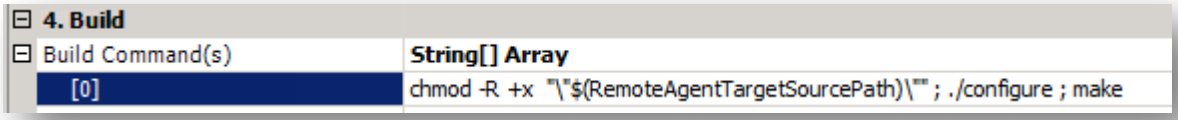


Figure 104 – Invoking a script to build Apache Web Server

And the **BuildApache** file would have the following content

```
#!/bin/sh

chmod -R +x .
./configure
make
```

NOTE

If the command parameters have spaces in them, it is the caller responsibility to properly quote them.

In the following command

cd "\"$(RemoteAgentTargetSourcePath)\""

Notice the double quoting. Once for the executing of the command on the local machine and another one for the command (which is escaped so the quote is properly interpreted at the remote machine.

Assuming the $(RemoteAgentTargetSourcePath) contains **My Directory With Spaces**. The command cd "\"My Directory With Spaces\"" would be seen to the SSH local executor as cd \"My Directory With Spaces\" and executed remotely has cd "My Directory With Spaces".

You can read more about command quoting in PuTTY documentation

Pulling outputs

In this stage we copy the output of the build in the build agent binaries folder (so it gets copied to the drops folder). This is the hardest part of defining a build definition, since the outputs vary wildly depending on what we are building. We may have to cherry pick directories or files from multiple locations.

In order to pull the files from the remote build agent (Figure 105) you have to specify the file location or the directory that you wish to copy (directories are copied recursively). You can specify as many items as you want (one per line) and of mixed types. You can also use wildcards (notice these are UNIX type regular expression wildcards).

If no files/directories are specified this step is skipped.

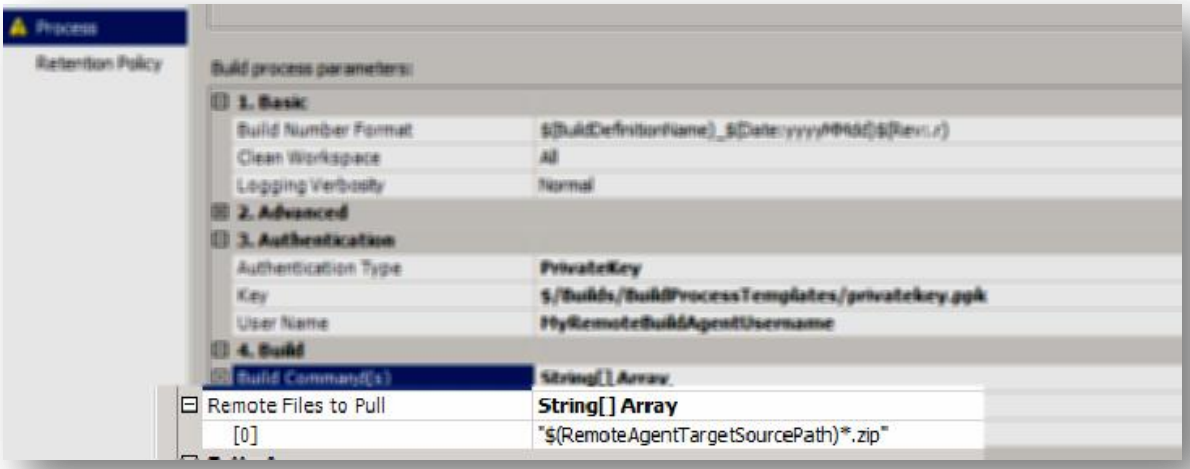


Figure 105 – Specifying the location of the output files on the remote build agent using the Remote Files to Pull parameter

NOTE

In order to facilitate this task it is advisable that the build script gathers all outputs (or packages it in a single file like a TAR[74], a ZIP an RPM[75], a Debian Package[76] or a Solaris Package) in a single directory to facilitate the pull task. A package may be necessary for other reasons, for example to preserve symbolic links[77] in case your build process generates them.

The files are copied using SFTP protocol so double quoting isn't necessary

Build Process Template Parameters

Besides the usual parameters that are common to most build process templates (grouped under Basic and Advanced) this particular build process template has parameters that are grouped under three categories (**Authentication**, **Build** and **Host** (Figure 106)). These parameters have been already discussed; this section just aggregates all of them for easier reference.

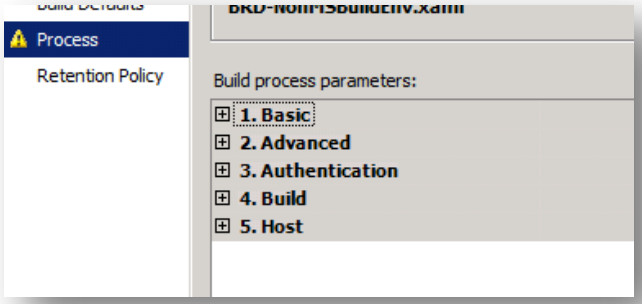


Figure 106 – Parameter Groups

[74] http://en.wikipedia.org/wiki/Tar_archive

[75] http://en.wikipedia.org/wiki/RPM_Package_Manager

[76] http://en.wikipedia.org(wiki/Debian_package

[77] http://en.wikipedia.org/wiki/Symlink

Authentication

This group has the parameters that allow the authentication on the build remote host, so we can remotely execute scripts and copy files. It has three parameters

- **Authentication Type** – Which can be either **UserNamePassword** or **PrivateKey**
- **User Name** – The username name under which the code will be executed on the remote build agent
- **Key** – The password if using username and password authentication or a reference to the private key file if using private key authentication (either a reference to a file in source control or a reference to a file in the build agent local file system).

Build

This group has three parameters related the build process. How to build the code and which files we which to collect

- **Build Command(s) –** the list of commands to be executed remotely so the code is built
- **Remote Files to Pull** - The files/directories on the remote system that we wish to copy to the drops folder

Host

- **Host** – the remote build agent machine in which the build will be remotely performed
- **Known Hosts File** (optional) the file that contains the hosts that we are allowed to connect to.
- **Port** (optional) the communication port used to connect to the server. This parameter only needs to be filled if a non-default port is being used.
- **Remote Agent Source Directory** the directory that will be used on the remote build agent to store the files for the executing build definition. (The list is available in the section Available Variables). It is advisable that this directory doesn't contain spaces in any part of its path.

Available Variables

The parameters **Build Command(s)**, **Remote Files to Pull** and to an extent **Remote Agent Source Directory** support variables that will be expanded with a value during runtime execution. To implement variable expansion it uses the **ExpandVariables** activity from Community TFS Build Extensions[78] (this activity also supports the expansion of environment variables).

The variables will be expressed in the format $(*variablename*). At the time of this writing the following variables are supported:

- BuildNumber
- BuildId
- BuildDefinitionName
- BuildDefinitionId
- TeamProject
- DropLocation
- BuildAgentName
- BuildAgentId
- BuildControllerName
- BuildControllerId
- BuildControllerCustomAssemblyPath

The variable (not part of the available variables from the **ExpandVariables** activity) **RemoteAgentTargetSourcePath** is also available in the parameters **Build Command(s)** and **Remote Files to Pull**. It represents the expanded value of the field **Remote Agent Source Directory**

An example from end to end

To wrap this up, we would like to present this with an example from end to end to better explain the process in a consistent way instead of using isolated examples like we have done so far.

[78] http://tfsbuildextensions.codeplex.com/

In order to test this example, we tested building Apache Web Server[79], Ruby Interpreter[80] and Python Interpreter[81] (namely the CPython implementation) all of them on Linux (tested on both Ubuntu[82] and OpenSUSE[83]). These are stock installations as downloaded for their respective sites. No customizations where made besides making sure the prerequisites for compilation were installed (SSH daemon and the necessary GNU compiler tool chain).

For our end to end example we chose to show how we can compile an iPhone application using a Mac OS X (10.7.1 Lion) as the remote build agent. No special customizations were made to compile the code on Mac OS X besides making sure the prequisites were installed (SSH Daemon and Apple developer tools[84]).

Instead of opting for a synthetic demo application we used a real application downloaded from the Web. We used the Fuel View application[85] since not only the full source code is available but how the code was built is fully explained in the author weblog.

The first step we made was to unzip the source code and place it under Team Foundation Server source control (Figure 107)

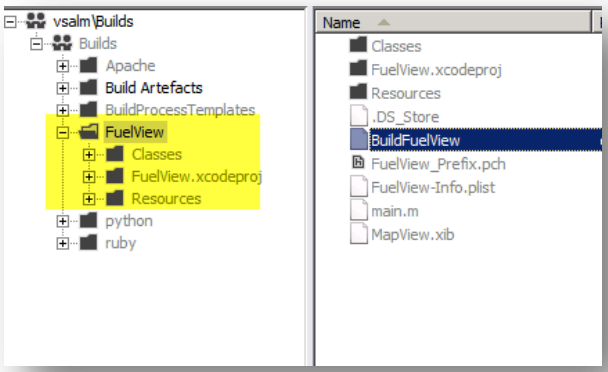


Figure 107 – Tested projects under source control (FuelView highlighted)

Second we created a build definition so we can compile it (Figure 108)

[79] http://httpd.apache.org/

[80] http://www.ruby-lang.org/en/

[81] http://python.org/

[82] http://www.ubuntu.com/

[83] http://www.opensuse.org

[84] http://developer.apple.com/technologies/tools/

[85] http://cocoawithlove.com/2011/06/process-of-writing-ios-application.html

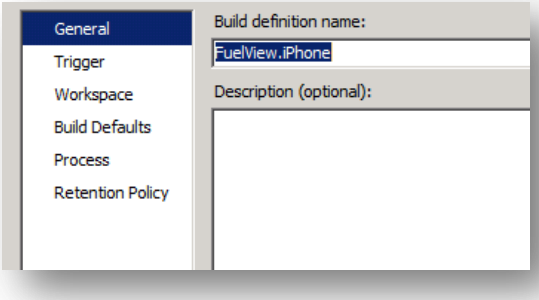


Figure 108 – FuelView.iPhone build process template

In the Workspace we mapped the FuelView source code just like a regular Team Build build definition (Figure 109)

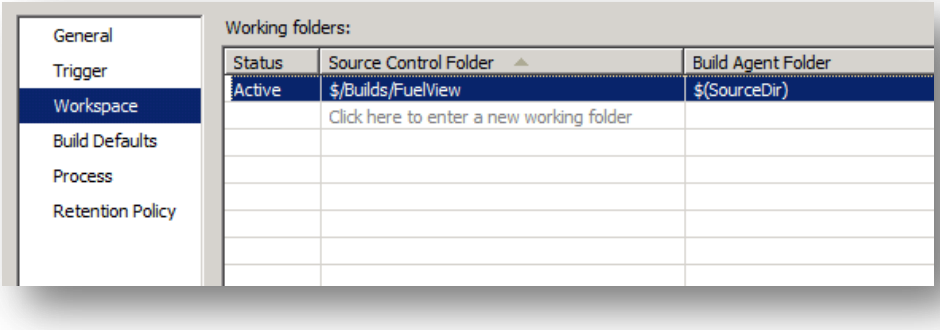


Figure 109 – mapping the Workspace for the FuelView.iPhone build definition

We added to source control a shell script to build and package the code and then added it to source control (we opted to call it **BuildFuelView**)

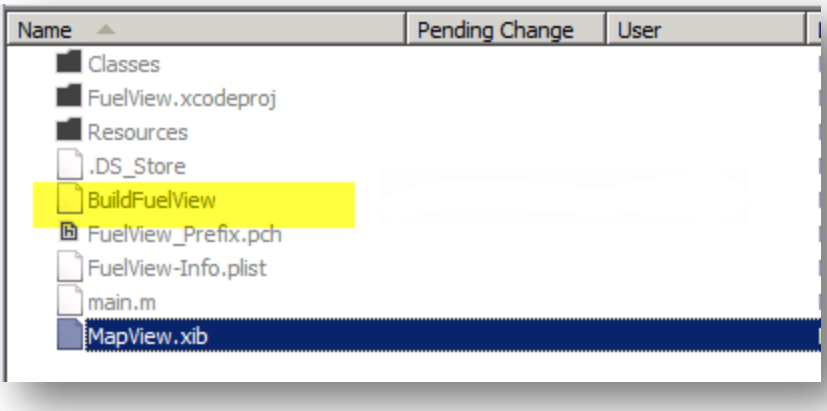


Figure 110 – The build script added to source control tree

Figure 111 contains the content of the build script that will be executed on the remote build agent. We have opted to compile the code just for the iOS simulator to keep the example simple. This way we don't have to deal with certificates.

We build the code using XCode and package it as a ZIP; the build script could build the code for the simulator and also sign it with a certificate so the application can be submitted to the app store.

```
#!/bin/sh

xcodebuild -project FuelView.xcodeproj -sdk iphonesimulator5.0
cd build/Release-iphonesimulator
zip -r -T -y "FuelView_ReleaseSimulator.zip" *.app*
mv *.zip ../..
```

Figure 111 – BuildFuelView shell script

Now we are ready to configure the Process step from the build definition (Figure 112)

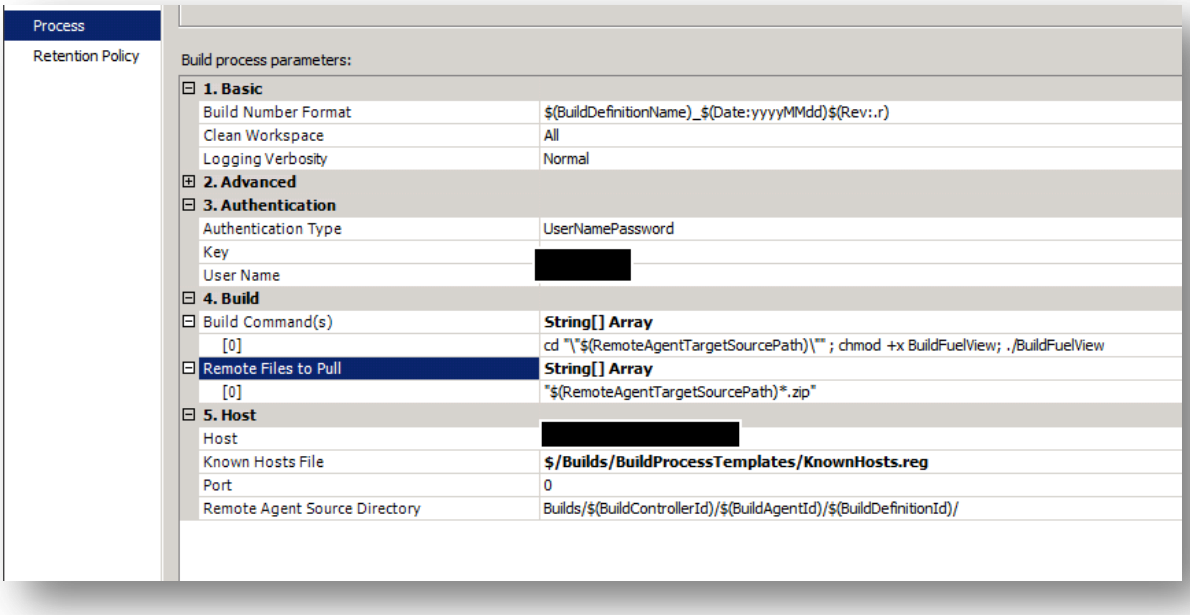


Figure 112 – Process parameters for FuelView.iPhone

Besides the host and authentication parameters which are just administrative parameters we just need to define two parameters: **Build Command(s)** and **Remote Files to Pull**.

Since the build process has been delegated to a script (**BuildFuelView**) stored in source control, we just need to invoke it, in order to invoke perform (in a single execution) the following operations: change the working directory to the base path where the source code is stored for this build, add the execute attribute (chmod +w) and then execute it (./BuildFuelView) :

cd "\"$(RemoteAgentTargetSourcePath)\"" ; chmod +x BuildFuelView; ./BuildFuelView

After compilation we want to get the result of the compilation to the drops folder. So we fill the **Remote Files to Pull** parameter with:

"$(RemoteAgentTargetSourcePath).zip"*

So all zip files (from the base source directory) are copied to the drops folder (Figure 113)

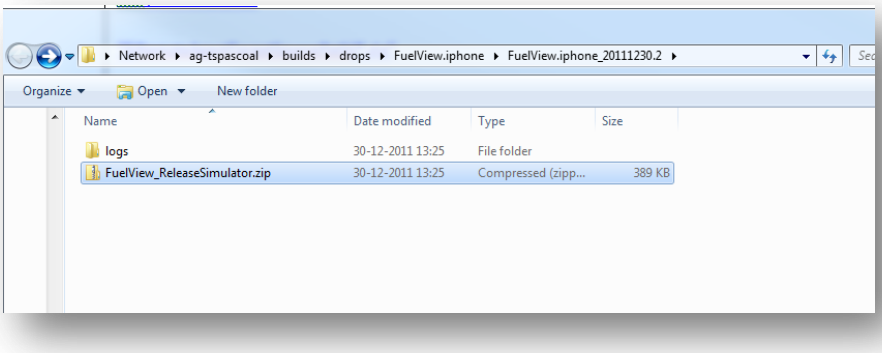


Figure 113 – Drops folder for the build of type FuelView.iPhone

We are now in condition to manually trigger our newly created build definition. After the build finishes successfully (Figure 114)

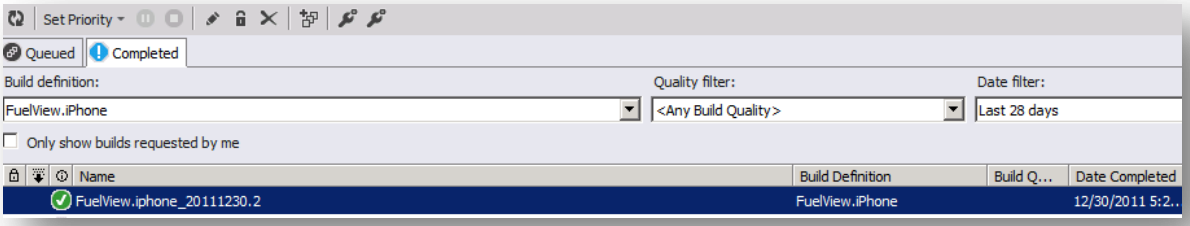


Figure 114 – FuelView.iPhone build finished successfully

We can examine the build logs (Figure 115).

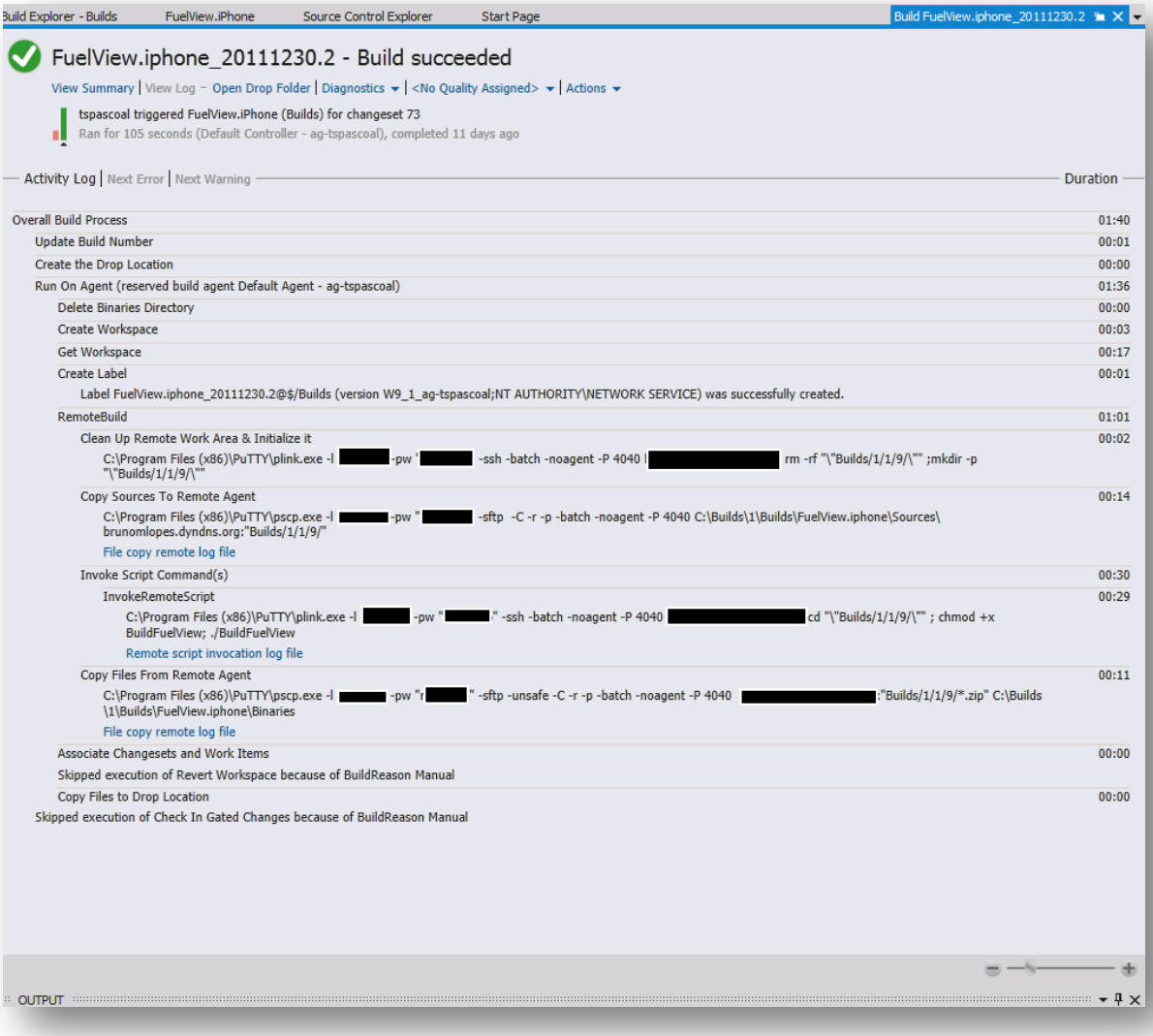


Figure 115 – FuelView.iPhone build execution log

We would just like to highlight a few features of the build definition template visible on the build log.

After the execution of the push files (**Copy Sources to Remote Agent**), the build Commands (**Invoke Script Command(s)**) and pull files (**Copy Files from Remote Agent**) there is a link that if clicked will automatically open the respective log file(see for example Figure 116) (the log files are stored in logs files of the drops folder (Figure 113))

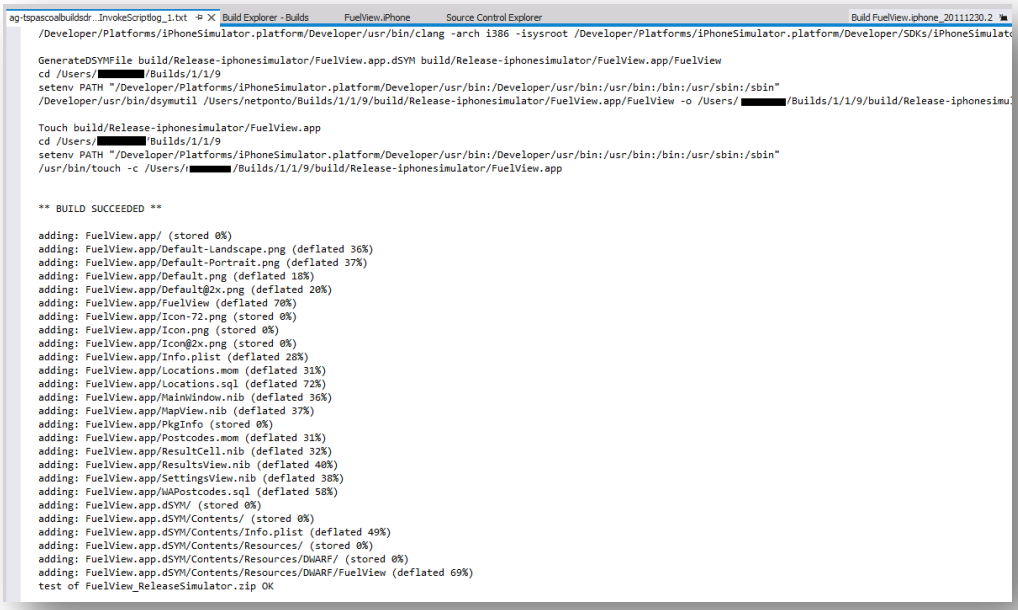


Figure 116 – portion of the log file of the remote script invocation

If we try to use the app with the iPhone simulator we can see it running (Figure 117). The package has been installed into the simulator from the file that has been placed on the drops folder (Figure 113).

Figure 117 – FuelView running on the simulator (fetched from the drops folder)

And this concludes our demonstration of compiling an iPhone application using Team Foundation Build and the BRD Lite **HeterogeneousEnvTemplate** build definition template

Empowering developers and build engineers with build activities

Introduction

In Visual Studio 2005 Team Foundation Server and Visual Studio Team System 2008 Team Foundation Server, you may have created MSBuild tasks to extend your build processes. MSBuild provided the Windows Workflow Foundation engine for Team Foundation Build in Visual Studio 2005 and Visual Studio Team System 2008. In Team Foundation Build in Visual Studio 2010, MSBuild is still used for compilation, but the workflow logic has been replaced by Windows Workflow Foundation (WF) 4.0[86].

Windows Workflow Foundation provides the declarative framework for building application and service logic. It gives developers a higher-level language for handling asynchronous, parallel tasks, and other complex processing.[87] This can be extended by creating custom Workflow activities.

Recommended Resources

Like any development task, before you attempt to solve an extensibility problem, you should be aware of available resources so you avoid re-inventing the wheel.

Team Foundation Server Code Activities

Visual Studio Team Foundation Server is released with the following Windows Workflow Foundation activities to use in your build.

Visual Studio Team Foundation Server 2010 Product Activities		
AgentScope	FindMatchingFiles	MSTest
AssociateChangesetsAndWorkItems	GetBuildAgent	OpenWorkItem
CheckInGatedChanges	GetBuildDetail	PublishSymbols
ConvertWorkspaceItem	GetBuildDirectory	RevertWorkspace
ConvertWorkspaceItems	GetBuildEnvironment	SetBuildProperties
CopyDirectory	GetImpactedTests	SharedResourceScope
CreateDirectory	GetTeamProjectCollection	SyncWorkspace
CreateWorkspace	GetWorkspace	TfsBuild
DeleteDirectory	IndexSources	UpdateBuildNumber
DeleteWorkspace	InvokeForReason	WriteBuildError
DownloadFile	InvokeProcess	WriteBuildInformation<T>
DownloadFiles	LabelSources	WriteBuildMessage
EvaluateCheckInPolicies	LabelWorkspace	WriteBuildWarning
ExpandEnvironmentVariables	MSBuild	

Table 16 – Visual Studio Team Foundation Server Product Activities

Guidance for these build activities can be found on MSDN[88]. In addition to these activities, Visual Studio Team Foundation Server also ships a collection of Lab Management activities. These will be covered in the upcoming ALM Rangers Lab Management Guide

Community TFS Build Extensions

In this chapter we will also use and refer to custom activities used in the "Community TFS Build Extensions"[89].

[86] http://msdn.microsoft.com/en-us/netframework/aa663328.aspx

[87] A Developer's Introduction to Windows Workflow Foundation (WF) in .NET 4 - Matt Milner, Pluralsight

[88] http://msdn.microsoft.com/en-us/library/gg265783.aspx

[89] http://tfsbuildextensions.codeplex.com

Community TFS Build Extensions

The "Community TFS Build Extensions" is an open source project that is hosted on CodePlex. The "Community TFS Build Extensions" project provides a place for build engineers to share Windows Workflow Foundation activities, build process template files, and various tools for Team Foundation Server.

WF4 Activity Best Practices

If you are new to Windows Workflow Foundation activities, you will also find some great sample code and over two hours of video on the "WF4 Activity Best Practices" [90] MSDN page.

Books

For even more information about Team Foundation Build in Visual Studio, you can read the following publications:

1. Professional Team Foundation Server 2010[91]
2. Inside the Microsoft® Build Engine: Using MSBuild and Team Foundation Build. 2nd Edition[92]

Using Existing MSBuild tasks

It is important to remember that your investment in MSBuild is not wasted when you upgrade from Team Foundation Server 2008. Existing builds as a whole will work as usual, using the Upgrade Template. If you choose to move to a WF-based build, you can still make use of your MSBuild tasks by using the MSBuild Activity[93], which was released with Team Foundation Server 2010.

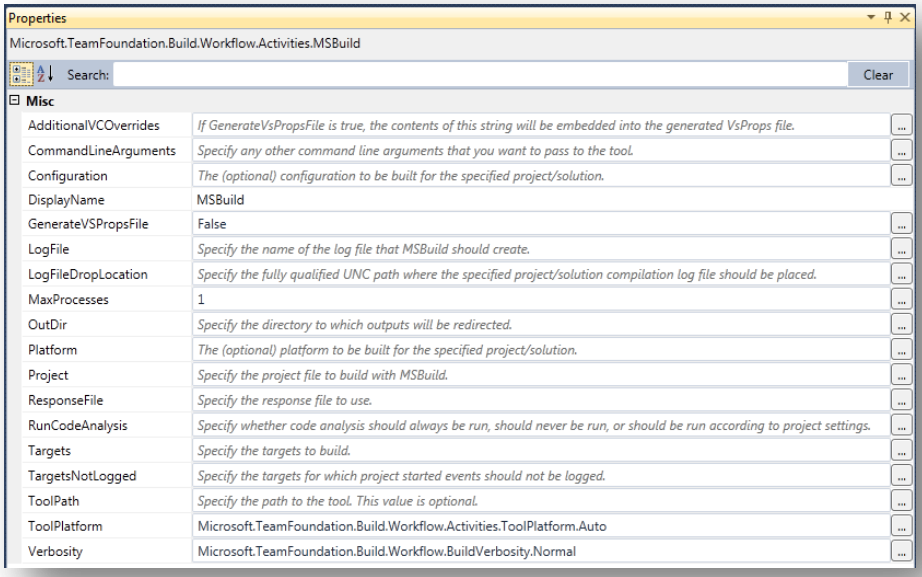


Figure 118 – MSBuild Activity properties

Jim Lamb has written an interesting blog post that is available here: "Windows Workflow Foundation vs. MSBuild in TFS 2010"[94]. In this post, he suggests the following general guidance:

[90] http://code.msdn.microsoft.com/Project/Download/FileDownload.aspx?ProjectName=wf4BP&DownloadId=13066

[91] http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470943327.html

[92] http://go.microsoft.com/fwlink/?LinkID=206999&clcid=0x409

[93] http://msdn.microsoft.com/en-us/library/gg265783.aspx#Activity_MSBuild

[94] http://blogs.msdn.com/b/jimlamb/archive/2010/06/09/windows-Workflow-vs-msbuild-in-tfs-2010.aspx

- If the task requires knowledge of specific build inputs or outputs, use MSBuild.
- If the task is something you need to happen when you build in Visual Studio, use MSBuild.
- If the task is something you only need to happen when you build on the build server, use WF unless it requires knowledge of specific build inputs/outputs.

William Bartholomew provides additional guidance in this blog post: Upgrade Paths for Custom MSBuild Tasks[95] where he covers the pros and cons of four different ways of leveraging your current investment:

1. Use MSBuild Activity to call MSBuild wrapper around MSBuild task.
2. Wrap MSBuild task in a custom Workflow Activity.
3. Rewrite MSBuild task as a Workflow Activity.
4. Extract custom task logic into a POCO class and provide an MSBuild Task and Workflow Activity adapters.

Remember that before you re-write an existing MSBuild task; if you are using the MSBuild Extension Pack[96] tasks please consult the "Community TFS Build Extensions"[97] CodePlex project, which provides a port of many of the MSBuild Extension Pack tasks into Workflow activities. You might also wish to request or contribute a new activity.

Creating a Custom Activity

Microsoft developed Team Foundation Build in Visual Studio as a WF 4.0 application that does the following:

- Implements the default Team Foundation Build Workflow.
- Supports parameterization of the Workflow.
- Supports customization of the Workflow.
- Supports extension of the Workflow.

Activities are the building blocks of a WF application. They drive the processing of the application's Workflow. The Team Foundation Build in Visual Studio WF 4.0 application implements the sequence and configuration of activities executed during a build. The build process template's exposed parameters, which are editable in the build definition or when you queue a new build, are used by the application to perform the build according to the customer's requirements. However, when the default behavior and definition parameters are unable to meet all of the customer's requirements, the default Workflow can be customized, and if needed, extended.

Parameterization vs. Customization vs. Extension

Customizing the default Team Foundation Build Workflow involves the use and configuration of the activities that are available in the Team Foundation Build Process Template. The Team Foundation Build Workflow is controlled by the Build Process Template. This is the XAML file containing the WF implementation code. Every Build Definition uses one Build Process Template and a set of parameter values to control the Workflow based on activities and conditions implemented inside the Build Process Template.

When you create a Team Project that is based on a Team Project Process Template, Team Foundation Server creates a BuildProcessTemplates folder under the Team Project and populates it with the following Build Process Template files:

- DefaultTemplate.xaml – the default Team Foundation Build Workflow in Visual Studio
- UpgradeTemplate.xaml – supports running pre-2010 Build Definitions
- LabDefaultTemplate.xaml – supports automated build deployment using Visual Studio Lab Management

[95] http://blogs.msdn.com/b/willbar/archive/2009/11/12/upgrade-paths-for-custom-msbuild-tasks.aspx

[96] http://www.msbuildextensionpack.com

[97] http://tfsbuildextensions.codeplex.com

NOTE

We recommend you branch or copy these templates to create any customizations and leave these as reference / starting point templates.

In this section, we will discuss the DefaultTemplate Workflow. For more information, refer to Default Template on page 39.

NOTE

You might change the target folder and the files deployed inside the Team Project Process Template's "Build.xml" under the folder "Build".

The default Workflow uses some but not all of the activities provided by the Team Foundation Build in Visual Studio. To customize the XAML file means to do one or both of the following:

- Change the configuration of activities already implemented by DefaultTemplate.xaml.
- Add and configure one or more of the activities from the Team Foundation Build WF 4.0 application to the XAML file used for your Build Definition.

The Team Foundation Build WF 4.0 application can also be extended. To extend the application means to create new activities and add them to the build definition's XAML file.

Workflow Data

In addition to the flow of work performed, WF 4.0 supports data handling. The three main WF 4.0 concepts regarding data are:

- Variables – for storing data
- Arguments – for passing data
- Expressions – for manipulating data[98]

Choosing a Base Class

In general, almost all your custom activities will derive from CodeActivity or a base class that you have derived from CodeActivity.

You should derive from **CodeActivity** when your activity:

- Executes in a single pulse of execution
- Does not need to schedule other activities
- Does not need advanced WF Runtime features
- Does not need to execute asynchronously

You should derive from **AsyncCodeActivity** if the work has to be done asynchronously.

You should derive from **NativeActivity** when your activity:

- Executes in multiple pulses
- Schedules other activities
- Needs advanced WF Runtime features

Remember that you can use the derivatives of Activity<TResult> to author activities with a return value, so rather than exposing a property or OutArgument, the return value is simply stored in the Result.

[98] Ibid

Base Class	Pros	Cons
Activity (designer)	• Visual representation of the process • 'Drag, Drop and Configure' authoring • Executing other activities is easy • Tool support for authoring expressions	• Each of the Contained activities is scheduled – potential performance penalty against CLR code • No first class designer experience for ActivityDelegate • Can be less succinct then plain CLR code
Activity	• Reuse – new functionality composing existing • Executing other activities is easy • No limitation for ActivityDelegates	• Each of the Contained activities is scheduled – potential performance penalty against CLR code • Verbosity • Some activities don't provide a good code experience because they use references
CodeActivity	• The activity describes itself to the runtime • Can leverage WF validation framework • Express behavior in a concise manner • Code is executed in a single pulse of execution • Can port any piece of existing behavior to WF	• Requires creating a new type • No declarative description of behavior
AsyncCodeActivity	• Same as CodeActivity • Async execution	• Same as CodeActivity
NativeActivity	• The Activity describes itself to the runtime • Can leverage WF validation framework • Full Access to the runtime (bookmarks scheduling, cancellation, abortion, etc.) • Express behavior in a concise manner • Semantics for children activities execution • Can port any piece of existing behavior to WF	• Requires creating a new type • No declarative description of behavior • Opaque at runtime • More complex

Table 17 – Pros and Cons of the available base classes (*Source: WF4 Activity Best Practices*[99])

InArgument vs. Properties

If you are writing your first code activity, you may be struggling with whether to use a **Property** or an **InArgument**. The difference between the two is that an **InArgument** can be specified by an expression and a Property can only accept a value. Being able to evaluate expressions at runtime is a key and powerful feature of using arguments. This is an important distinction. Keep in mind that your code activity may be used in other processes, possibly by different people. In most cases, you will be better served by using an **InArgument**.

[99] http://code.msdn.microsoft.com/Project/Download/FileDownload.aspx?ProjectName=wf4BP&DownloadId=13066

Keep in mind that you should not simply make every input an argument; there is a minor usability difference. In the case of a Boolean, properties are rendered as check boxes in the UI, which are generally better accepted by your end user.

In the Workflow Designer properties selection shown below, **SetAssemblyFileVersion** and **SetAssemblyVersion** are Boolean properties. They are rendered as check boxes. **TreatWarningsAsErrors** is defined as an **InArgument** and rendered as a text box that will accept an expression.

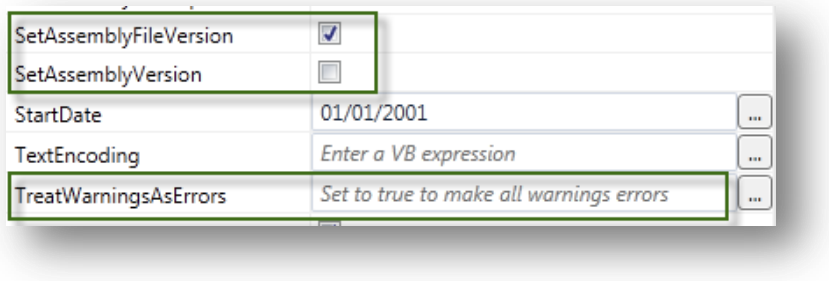


Figure 119 – Boolean properties render as checkboxes whereas Boolean arguments will accept an expression

GetValue vs. Get

If you have not worked with Windows Workflow Foundation before, you might notice that the assignment of values to arguments is a little different from that of standard class properties. Depending on what samples you read, it might even be done in different ways. Let us take, for example, this simple **InArgument**:

```
public InArgument<bool> UseUtcDate { get; set; }
```

To get the value of this **InArgument**, you might see samples do it in two ways; either

```
bool myValue = context.GetValue(this.UseUtcDate)
```

or alternatively

```
bool myValue = this.UseUtcDate.Get(context)
```

where *context* is the current activity context.

The end result is identical, and it is a matter of your personal preference. Whichever you choose, use it consistently so that your code reads consistently.

Specifying Default Values for Arguments

Another quirk of working with code activities is specifying default values for your **InArguments**. Default values can make your code activity a lot easier to consume by the end user. In your MSBuild Tasks, you might have specified a default for a property as follows:


```
private string delimiter = ".";

public string Delimiter
{
   get { return this.delimiter; }
   set { this.delimiter = value; }
}
```

When it comes to using an **InArgument,** you can the same pattern. Simply change the input type as appropriate.

```
private InArgument<string> delimiter = ".";

public InArgument<string> Delimiter
{
   get { return this.delimiter; }
   set { this.delimiter = value; }
}
```

There is a helpful side effect of declaring your default like this. It is evaluated at design time and the default will be rendered for the user in the properties grid.

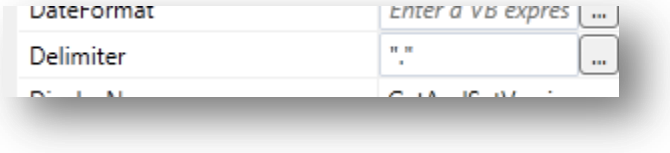


Figure 120 – Default value rendered in properties box.

Please note that although you might find samples that use an attribute to specify the default value. This code will even compile without error, but the behavior of DefaultValue may not be as you expect. For example:

```
[DefaultValue(".")]
public InArgument<string> Delimiter { get; set; }
```

The DefaultValue above will not be serialized to your XAML and will have no effect on your activity. The only 'valid' use of DefaultValue is to use `[DefaultValue(null)]` when you want to avoid serializing a null. If you would like more information, refer to the ShouldSerialize and Reset methods on MSDN at http://msdn.microsoft.com/en-us/library/53b8022e(v=VS.100).aspx.

If your code activity requires a bit more logic for determining a default value, then you can also check to see whether the user has provided a value by using the Expression property. For example:


```
if (this.Format.Expression == null)
{
   // your logic here to determine a default value…

   this.Format.Set(this.ActivityContext, YourDeterminedValue);
}
```

General Naming Guidelines

Consistency is a great asset for your code to have. These are some suggested guidelines for naming your activities:

- Do not add the postfix Activity.
- Do not add the postfix Argument or Arg to Arguments.
- Do not add the postfix Variable or Var to Variables.
- Do use the suffix "Scope" for activities which are designed to decorate other activities and add execution semantics.

Examples of Bad Naming

```
public sealed class GenerateCertificateActivity : CodeActivity // Activity postfix
{
   public InArgument<string> MessageArgument { get; set; } // Argument postfix

   public int EncryptionLevelVar { get; set; } // Var postfix
```

Examples of Good Naming

```
public sealed class GenerateCertificate : CodeActivity
{
   public InArgument<string> Message { get; set; }

   public int EncryptionLevel { get; set; }
```

Working with Visual Basic Expressions

Those who have made the move to C# would be forgiven for looking through the Options to change "Use VB Expression" to "Use C# Expression" in the Workflow Designer. Unfortunately, you will find no such option. C# and custom language expressions are not supported. Why Visual Basic you may ask? Visual Basic was chosen because it has in-memory compilation support while C# does not.

Here are some great resources if you need some refreshing:

- How To: Use the Expression Editor[100]
- Operators and Expressions in Visual Basic[101]

[100] http://msdn.microsoft.com/en-us/library/dd797416.aspx

[101] http://msdn.microsoft.com/en-us/library/a1w3te48(VS.100).aspx

- End-to-end expression editing feature deck[102] by Cathy Dumas

Logging

Logging to the Build Output from Code Activities

Just as you may have used the BuildStep[103] task in your Team Foundation Server 2008 builds, you will want to log your own messages, warnings, and errors to the build output in the new Team Foundation Server Workflow builds. There are two methods of achieving this. The first is to use the Track method provided with the active context that you are using. For example:

```
protected override void Execute(CodeActivityContext context)
{
   this.ActivityContext.Track(new BuildInformationRecord<BuildMessage>
   {
      Value = new BuildMessage()
      {
         Importance = BuildMessageImportance.Normal,
         Message = "Hello Workflow!",
      },
   });
```

The second and easier method is to add a reference to Microsoft.TeamFoundation.Build.Workflow.dll. This is typically installed to %PROGRAM FILES%\Microsoft Visual Studio 10.0\Common7\IDE\PrivateAssemblies if you are using Visual Studio 2010. Add the following using statement:

```
using Microsoft.TeamFoundation.Build.Workflow.Activities;
```

You will then be able to use the context.TrackBuildMessage, context.TrackBuildWarning and context.TrackBuildError extension methods. For example:

```
context.TrackBuildMessage("Hello WorkFlow!", BuildMessageImportance.Normal);
```

Note that if you track a build error, the build will not stop but will be marked as partially succeeded, assuming all further activities succeed. You might want to add a helper method to your base code activity class that you use to write build errors. By exposing a "FailBuildOnError" argument, your end users can then have a consistent way to stop the build if any of your activities fail. You could further extend this logic to areas like 'TreatWarningsAsErrors' or 'TreatErrorsAsWarnings'. The "Community TFS Build Extensions" implement this logic.

[102] http://blogs.msdn.com/b/cathyk/archive/2010/01/22/end-to-end-expression-editing-feature-d eck.aspx

[103] http://msdn.microsoft.com/en-us/library/bb399129(v=VS.90).aspx

Managing Log Verbosity

If you have been using MSBuild, you will be familiar with the /verbosity switch which is used to manage the level of logging by the build engine. In Team Foundation Build, you can set the default logging verbosity in the process section of the build definition, as shown in the following illustration:

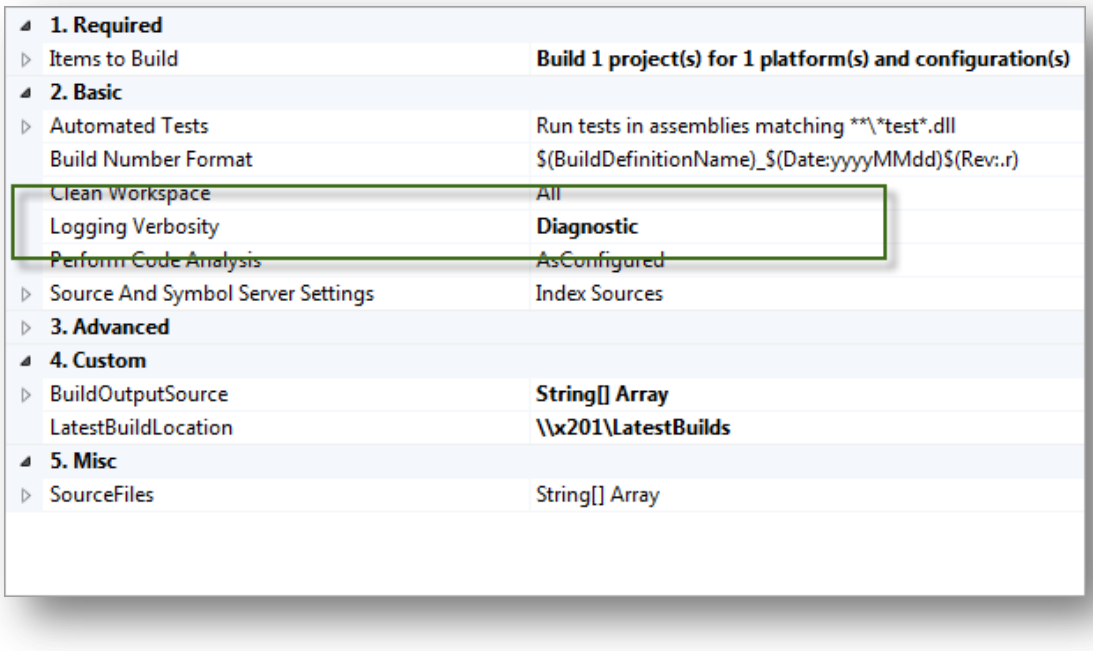


Figure 121 – Setting the Logging Verbosity in a Build Definition

When you queue a build, you also have the option of overriding the default logging verbosity, as shown in the following illustration:

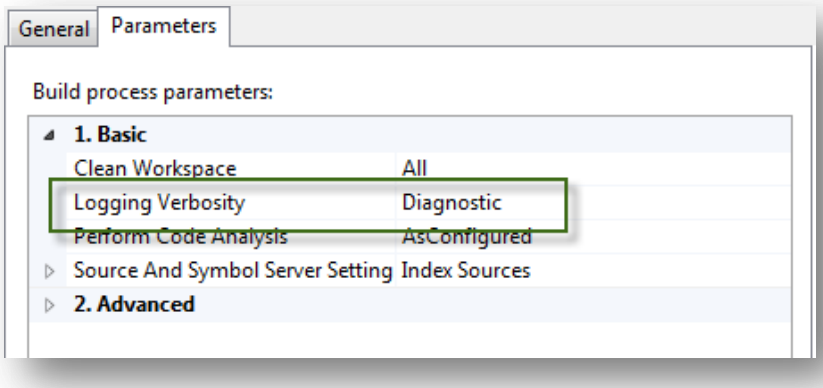


Figure 122 – Overriding the Build Definition Logging Verbosity when you queue a build

This setting will control whether your `context.TrackBuildMessage` statements appear in the build in accordance to the `BuildMessageImportance` that you have specified.

Let us look at how the verbosity affects the logging in our Hello activity. If we add the activity to our build and configure as follows:

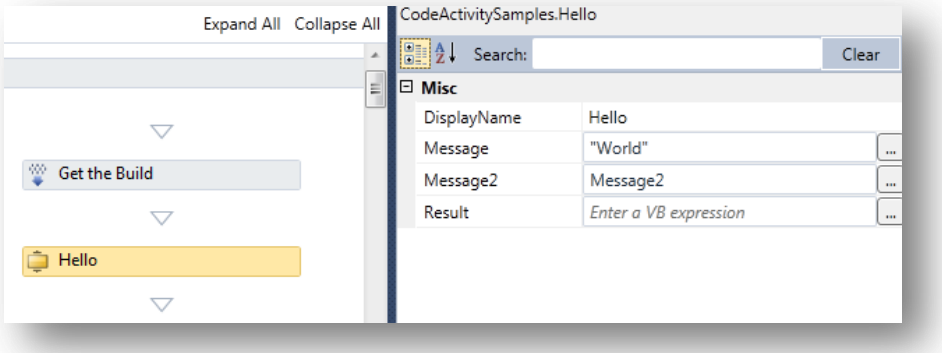


Figure 123 – Sample Hello activity with arguments populated, Message2 is a property set to "hope you are well"

Remember that within the activity we have

```
context.TrackBuildMessage(myMessage, BuildMessageImportance.High);
```

We get the following output:

Logging Verbosity: Minimal

Result:

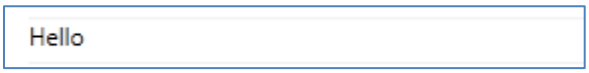


Image: Only the activity name is emitted.

Logging Verbosity: Normal

Result:

Image: The activity name and the logging are shown.

'Hello' is provided by the task, 'World' is provided by the Message property, ' hope you are well' is provided by the Message2 argument and 'from 100' is provided by the task where 100 is simply the build number.

Logging Verbosity: Detailed

Result:

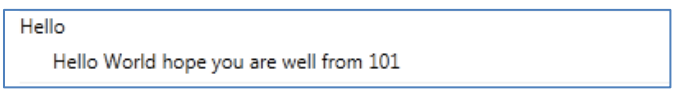


Image: The activity name and the logging are shown

Logging Verbosity: Diagnostic

Result:

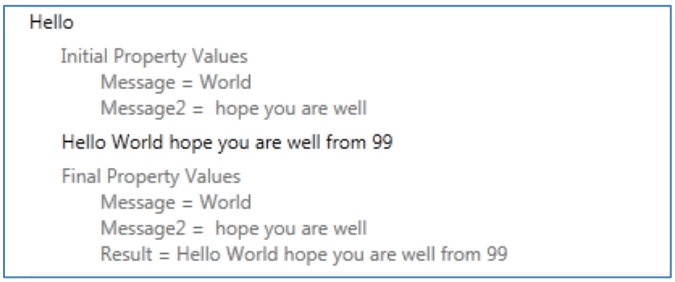


Image: The activity name, logging and detailed property / argument values are shown

Clearly, the Diagnostic verbosity level gives us the most information. However, in real-world scenarios, this amount of logging could have performance issues. The result could be logs that are hard to diagnose, due to the sheer amount of data.

We recommend the following:

Verbosity	When to Use
Minimal	Stable non-critical builds
Normal	Stable builds
Detailed	We recommend the usage of Normal and when necessary switch to Diagnostic rather than use Detailed
Diagnostic	New builds where diagnostic information helps to debug Issues All builds where the extra logging is not an issue.

Table 18 – Recommended logging verbosity usage

NOTE

As an author of activities, you should keep in mind how people will use your activities and the logs they produce. Do not log all information messages with high importance. Do not overuse the default logging importance. If your information will help debug an issue, you should log the messages with Low importance so that they are only shown in diagnostic builds where the Build Engineer is trying to get more information.

Note that if you open the DefaultTemplate.xaml build template using a text editor, you will notice that many components of the build have a property called mtbwt:BuildTrackingParticipant assigned to them. This also affects how much is logged to the build output. We do not recommend adding this property to your custom activities, but rather use it on the **Control Flow** activities.

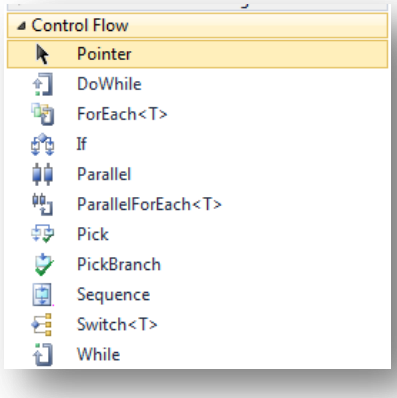


Figure 124 – Control Flow Activities

In general, you should use and expect your end users to rely on the diagnostic level to control the build log.

Attributes

Class Level Attributes

There are a few key attributes to be aware of which can be set at the class level.

- **BuildActivity**

  This indicates that an activity should be instantiated and loaded into the Workflow designer on the client. It also indicates whether it should be loaded on the build controller, the agents, or both as specified by the HostEnvironmentOption setting. For example:

```
[BuildActivity(HostEnvironmentOption.Agent)]
public sealed class StyleCop : BaseCodeActivity
```

- **BuildExtension**

  This indicates that a service should be instantiated and loaded into the Workflow runtime on the build controller, the agents, or both, as specified by the HostEnvironmentOption setting. For example:

```
[BuildExtension(HostEnvironmentOption.Agent)]
public sealed class StyleCop : BaseCodeActivity
```

  Activities that are marked with the BuildExtension attribute are stateful and can be queried using GetExtension[104].

- **ActivityTracking**

[104] http://msdn.microsoft.com/en-us/library/dd486210.aspx

Composed activities can participate in workflow tracking, meaning that their internal progress can be easily tracked as they execute. You can control this behaviour, that is, the amount of information logged to the build report, by using the ActivityTrackingOption setting. For example:

```
[ActivityTracking(ActivityTrackingOption.ActivityOnly)]
public sealed class StyleCop : BaseCodeActivity
```

The ActivityTrackingOption enumeration supports the following values:

- **None** – No tracking will be performed. An example would be the **WriteBuildMessage** activity for which additional tracking information would not provide any additional value.
- **ActivityOnly** – The activity will be tracked, but no tracking of implementation activities.
- **ActivityTree** – This is the default value and indicates that the activity and all implementation activities will be tracked.

Argument Level Attributes

There is an extensive set of attributes to use on arguments. You can find a complete list here http://msdn.microsoft.com/en-us/library/z82ykwhb.aspx. Some of the most common and useful are:

- **Browsable**
  This indicates that the argument can be modified at design time. Note that the default is true and the designer will load all arguments where this attribute is declared as true or not declared.

```
[Browsable(false)]
public InArgument<string> Arguments { get; set; }
```

There is no need to use this attribute if the argument should be modifiable at design time. For example, **don't** do this:

```
[Browsable(true)]
public InArgument<string> Arguments { get; set; }
```

- **DefaultValue**
  This attribute is not supported and you should avoid its use. For more details, see Specifying Default Values for Arguments in this guide.

- **Description**
  This indicates the text to show in the designer textbox for the property. For example:

```
[Description("The action for this activity.")]
public InArgument<string> ActivityAction { get; set; }
```

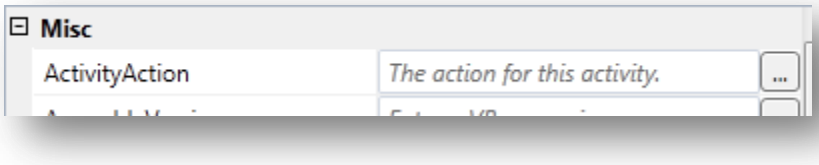


Figure 125 – Description showing in the property grid

- **RequiredArgument**
  This indicates that the argument is required. For example:

```
[RequiredArgument]
public InArgument<string> SmtpServer { get; set; }
```

Storing Custom Activities in Team Foundation Server

NOTE

Refer to Branching Process Templates on page 78 for information about branching of Build Process Templates and Custom Activities.

Adding Custom Assemblies to Version Control

To add a custom activity to the version control system, follow these steps:

1. Open Team Explorer.
2. Double-click the **Source Control** node to open the **Source Control Explorer** tab.
3. In the Source Control Explorer tab, locate the parent folder location in which you wish to create the folder in to store the custom activities.
4. Make sure this folder is mapped to a local path. Otherwise, you cannot create a new folder.
5. Create the new folder to hold the custom activities.
6. Copy the custom activity assemblies to this new folder and add them in the Source Control Explorer tab.
7. Check in the pending changes.

Configuring Build Controllers to use Custom Activities

After the custom activities are stored within a TPC, the Build Controller(s) can be configured to point to their location. If this is NOT done, an error will occur that the Custom Activity cannot be found in form whenever a Build Agent tries to perform a build that contains a Custom Activity. For example:

TF215097: An error occurred while initializing a build for build definition [BuildName]: Cannot create unknown type '{http://schemas.microsoft.com/netfx/2009/xaml/activities}Variable({clr-namespace:[BuildActivity Namespace];assembly=[Build Activity Assembly Name])'.

Figure 126 – Example build error message

You can configure a Build Controller to use Custom Activities from within Team Explorer; you do not have to be on the Build Controller itself. To configure the Build Controller, follow these steps:

1. Open Team Explorer.
2. Right-click **Builds** and select **Manage Build Controllers**.
3. Select the controller to configure and select **Properties**.
4. Set the **Version control path to custom assemblies**. You can use the ellipsis (…) to browse to the location under version control.

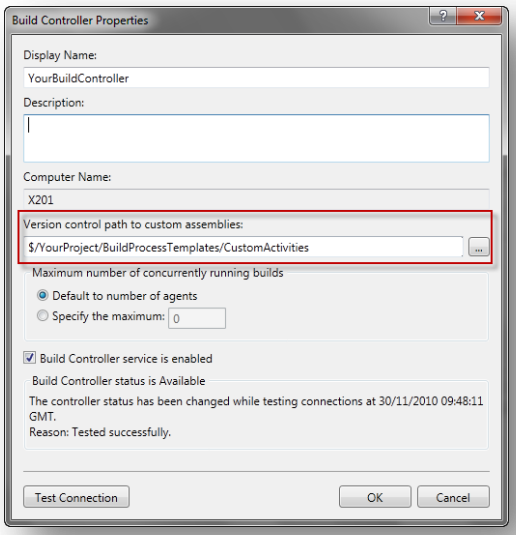


Figure 127 – Setting the path the custom assemblies for a build controller

Alternatively, you can make the same configuration changes via the Team Foundation Administration Console on the host running the Build Controller.

When this path is set, all the Build Agents attached to the controller will be able to load any of the custom assemblies stored in this location on the TPC and make use of them for builds.

Making a Custom Activity available in the Visual Studio toolbox

Getting an error while viewing a Windows Foundation Workflow that uses a custom Workflow

If you try to view or edit a build workflow inside Visual Studio that uses a custom activity, and you do not have that custom activity in the correct location on your development PC, you will see a red error bar as shown below. This is because the custom assembly cannot be loaded by Visual Studio.

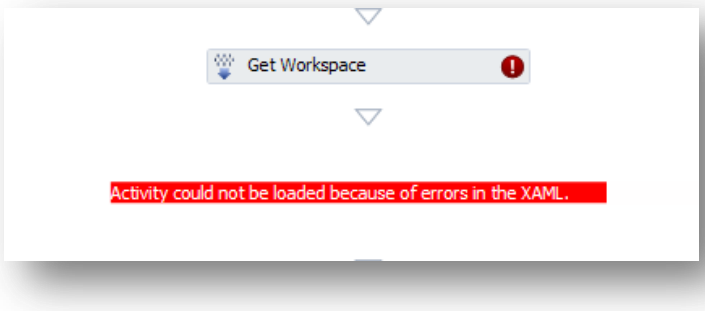


Figure 128 – Error message "Activity Could not be loaded because of errors in the XAML."

This error does not mean that the workflow will not work on the Build Agents; they have their own mechanism to load custom assemblies from version control. For more information, see Storing Custom Activities in Team Foundation Server. The issue is that for Visual Studio to be able to load the assembly containing the custom activity, the assemblies must be a in a location that is known to Visual Studio.

Options to avoid an error when viewing a Workflow with custom activities in Visual Studio

Unfortunately, there is no simple deployment model that allows a custom activity to be accessed in a project other than the one in which it was developed. In most cases, making this project available to all solutions that require it is obviously impractical. Custom code activities should be deployable using only a .NET assembly, not a complete solution with source.

However, there are a number of options to get around the problem. You can place any custom activity assemblies in locations that Visual Studio can access them.

Branching the Workflow template

The recommended solution to this problem is to always create a branched version of any process template that needs to contain custom activities. This .XAML file should be branched into a suitable Visual Studio project[105]. This project must also contain references to the assemblies that contain the custom build activities.

If this is done, when the .XAML process template is opened in Visual Studio, the IDE will be able to load the assemblies and display the custom activities on the design surface.

Placing the custom activity assemblies in the GAC

The Global Assembly Cache (GAC) is a machine-wide store of .NET assemblies. Any assembly stored in the GAC is accessible to all .NET applications such as Visual Studio. However, adding the custom activity assemblies to the GAC is not a recommended solution, because many users will not have user rights to the GAC.

Placing the custom activity assemblies in the Probing Path

The probing paths are folders that Visual Studio uses to load assemblies. They are defined in the devenv.exe.config[106] file as follows:

[105] A standard class library is a good choice


```
<probing privatePath=
"PublicAssemblies;PrivateAssemblies;CommonExtensions\Microsoft\TemplateProviders;
PrivateAssemblies\DataCollectors;PrivateAssemblies\DataCollectors\x86;
CommonExtensions\Microsoft\Editor;CommonExtensions\Platform\Debugger"/>
```

When an assembly is in a folder that is in the probing path, it becomes accessible by Visual Studio. Therefore, if you placed a custom build activity assembly in a probing path folder[107] and reloaded the workflow, the missing assembly errors will be replaced by the custom activity.

NOTE

Remember that Visual Studio will find and load any assemblies in the GAC or on a probing path in preference to any other folders. This becomes a problem if you are writing a new version of a custom build activity and wish to debug it on a PC that has the older version of the activity in the GAC or on a Probing Path.

For Visual Studio to load your new version from your project's working folder, you will need to delete the older versions in the GAC or on Probing Paths.

For this reason neither the GAC or Probing Paths can be recommend to store custom build activity assemblies.

The branch model should always be the first choice solution.

Adding a custom activity to your Visual Studio toolbox

There is no automated way to add a custom activity to the Visual Studio toolbox. You must do this manually.

An assembly can be added to the toolbox from any folder. However, if it is not in a folder that Visual Studio will search (see above) you will encounter the problem that although you can add the activity to the toolbox and drag it onto the Workflow design surface, you cannot drop it anywhere in the Workflow. You will just get a circle cursor with line through it like the following: ⃠

Therefore, it is vital to add the assembly from a location that is known to Visual Studio. After you do this, it can be added to the Visual Studio toolbox and then dropped onto the build process design surface.

The recommended process to add custom activities to a build process template, when you have the custom activities as compiled assemblies ,is as follows:

1. Copy your custom activity assemblies to a local folder on the development PC.
2. Open Visual Studio.
3. Create a new class library project (or add project to existing solution).
4. Add references to the locally stored assemblies that contain the custom build activities.
5. Check the new project into Team Foundation Server.
6. In Source Control Explorer branch, the process template you wish to add the custom activities to into the newly created project. Alternatively, you might want to link directly to the original template rather than branching.
7. In Solution Explorer, add the newly branched process template file to the solution.
8. Set the **Build Action** property of the branched XAML file to **None**.
9. Open the Workflow in the graphical designer.

   The context based toolbox changes to the show the build tabs
10. Right-click the toolbox.
11. Select **Add Tab** and create one for **Custom Activities** if it does not already exist.

[106] In C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE for Visual Studio 2010 or C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE for Visual Studio 2012

[107] Visual Studio only 'probes' the probing paths on start up, so if an assembly is placed in any of these directories it will not be found without a restart of Visual Studio.

12. While you are still working in the new tab, right-click and select **Choose items**. You will see dialog box like the following one:

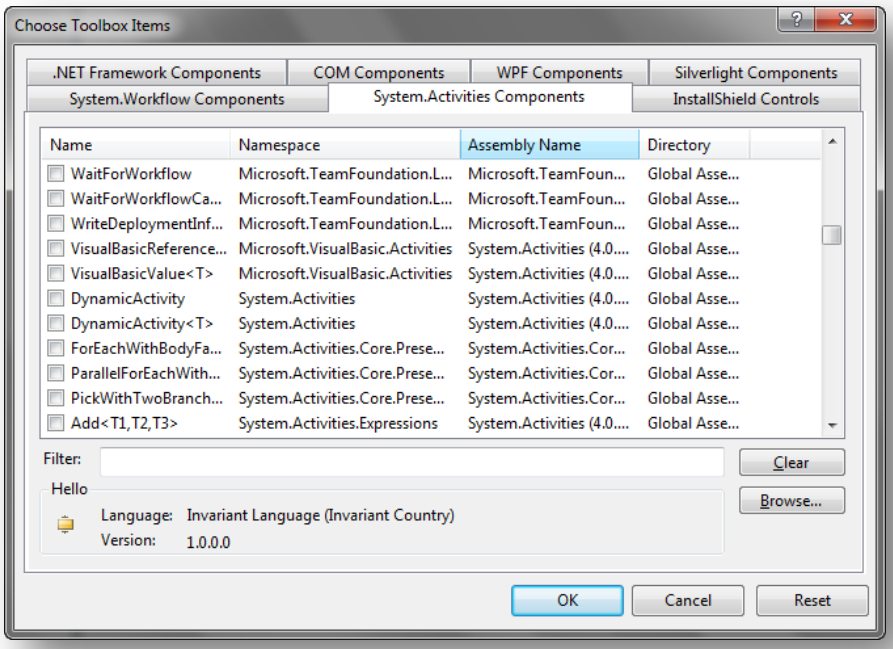


Figure 129 – Choosing additional activities for the Toolbox

13. Either enter the path to the assembly that contains the custom activity or browse to the custom activity.
14. The activities contained in the assembly will be added to the list. Click OK to close the dialog box.
15. The custom activity should now be shown in the toolbox and can be dragged onto the Workflow design surface.
16. When you have edited the Workflow as you wish save it, merge the branched file back into the original so it can be used on the build agents.

NOTE

Activities can be added on any tab, but placing them on a custom tab makes it easier to find them in the future.

Testing and Debugging Custom Activities

Like any code development, being able to easily test and debug your code is an important part of the development process. The code activities you write will have to work on the build server. However, testing it on the build server is not feasible for various reasons, including:

- You might not have access to the build server.
- If you do have access, you do not want to risk interfering with current builds.
- It is a slow process working with build templates.

Fortunately, you do not need access to your build server or build template to test your activity adequately. Let us take the following activity as a starting point. This activity accepts an input message, logs it to the build output, and returns the full message.

```
using System.Activities;
using Microsoft.TeamFoundation.Build.Client;
using Microsoft.TeamFoundation.Build.Workflow.Activities;

[BuildActivity(HostEnvironmentOption.All)]
public sealed class Hello : CodeActivity<string>
{
   public InArgument<string> Message { get; set; }

   protected override string Execute(CodeActivityContext context)
   {
      // Get the message
      string myMessage = "Hello " + this.Message.Get(context);

      // log it to the build output
      context.TrackBuildMessage(myMessage, BuildMessageImportance.High);

      // return the message
      return myMessage;
   }
}
```

Test this activity by creating a unit test class that uses a `WorkflowInvoker`[108] class to execute the activity.

```
using System.Activities;
using Microsoft.VisualStudio.TestTools.UnitTesting;

[TestClass]
public class HelloTest
{
   public TestContext TestContext { get; set; }

   [TestMethod]
   [DeploymentItem("MyAssembly.dll")]
```

[108] http://msdn.microsoft.com/en-us/library/dd484932.aspx


```
    public void ExecuteTest()
    {
      // Initialize Instance
      var target = new Hello { Message = "World" };

      // Invoke the Workflow
      var actual = WorkflowInvoker.Invoke(target);

      // Test the result
      Assert.AreEqual("Hello World", actual);
    }
  }
```

Tip: if you are working with non-value types, see this knowledge base article: Windows Workflow throws error: 'Literal<type>': Literal only supports value types and the immutable type System.String[109].

You now have the ability to easily execute the unit test and step through your code for real-time debugging. This is a simplified example, so let us cover some other areas you might encounter.

Passing output arguments

Out and InOut arguments cannot be passed in via object construction as used in the previous example. If we add the following argument,

```
    public InOutArgument<string> Message2 { get; set; }
```

We need to create a dictionary object and pass in the named parameters. We then pass the parameters to the `WorkflowInvoker`

```
    [TestMethod]
    [DeploymentItem("MyAssembly.dll")]
    public void ExecuteTest2()
    {
      // Initialise Instance
      var target = new Hello { Message = "World" };

      // Declare additional parameters
      var parameters = new Dictionary<string, object>
      {
        { "Message2", "this is message 2" },
      };

      // Invoke the Workflow and pass the parameters
      var actual = WorkflowInvoker.Invoke(target, parameters);

      // Test the result
      Assert.AreEqual("Hello World", actual);
```

[109] http://support.microsoft.com/kb/2013194


```
    }
```

Working with Extensions

We have `WorkflowInvoker` working nicely for us, but what if our activity uses objects in the context of the build? For example, what if it outputs the name of the build along with the message? We do not have a build! This is where extensions and mock objects are helpful.

Here is a revised sample, which uses GetExtension[110] to get the current build associated with the context and queries the build number from it.

```
using System.Activities;
using Microsoft.TeamFoundation.Build.Client;
using Microsoft.TeamFoundation.Build.Workflow.Activities;

[BuildActivity(HostEnvironmentOption.All)]
public sealed class Hello : CodeActivity<string>
{
   public InArgument<string> Message { get; set; }

   public InOutArgument<string> Message2 { get; set; }

   protected override string Execute(CodeActivityContext context)
   {
      // Get the message
      string myMessage = "Hello " + this.Message.Get(context);

      // Get the second message
      myMessage += this.Message2.Get(context);

      // Get the current BuildNumber using the GetExtension method
      string tfsBuildNumber = context.GetExtension<IBuildDetail>().BuildNumber;

      // add the build number to the message
      myMessage += " from " + tfsBuildNumber;

      // log it to the build output
      context.TrackBuildMessage(myMessage, BuildMessageImportance.High);

      // return the message
      return myMessage;
   }
}
```

To test this, we need to create mock `IBuildDetail` and `IBuildDefinition` objects. The `IBuildDetail` object will be passed as an extension to the Workflow invoker. You will find mock implementations in the "Community TFS Build Extensions"[111].

[110] http://msdn.microsoft.com/en-us/library/dd486210.aspx

NOTE

If you create your own IBuildDefinition implementation, you might encounter the following error when you compile your class:

'Microsoft.TeamFoundation.Build.Client.IProjectFile' is obsolete: 'This interface has been deprecated. Please remove all references.'

To resolve this, you must mark your method as obsolete, too. For example:

```
[Obsolete("This method has been deprecated. Please remove all references.", true)]
public IProjectFile CreateProjectFile()
{
   throw new NotImplementedException();
}
```

The updated unit test is shown in the following illustration:

```
using System.Activities;
using System.Collections.Generic;
using Microsoft.TeamFoundation.Build.Client;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using TfsBuildExtensions.Activities.Tests;

[TestClass]
public class HelloTest
{
   public TestContext TestContext { get; set; }

   [TestMethod]
   [DeploymentItem("MyAssembly.dll")]
   public void ExecuteTest2()
   {
      // Initialise Instance
      var target = new Hello { Message = "World" };

      // Declare additional parameters
      var parameters = new Dictionary<string, object>
      {
         { "Message2", " hope you are well" },
      };

      // Create the Workflow object
      WorkflowInvoker invoker = new WorkflowInvoker(target);

      // Create a Mock build detail object
      IBuildDetail t = new MockIBuildDetail { BuildNumber = "MyBuildNumber" };
      invoker.Extensions.Add(t);
```

[111] http://tfsbuildextensions.codeplex.com


```
        // Invoke the Workflow
        var actual = invoker.Invoke(parameters);

        // Test the result which is now accessed via the named Result key
        Assert.AreEqual("Hello World hope you are well from MyBuildNumber", actual["Result"]);
    }
}
```

GetExtension can query the following objects:

- Microsoft.TeamFoundation.Build.Client.IBuildAgent
- Microsoft.TeamFoundation.Build.Client.IBuildDetail
- Microsoft.TeamFoundation.Build.Workflow.BuildEnvironment
- Microsoft.TeamFoundation.Build.Workflow.Tracking.BuildTrackingParticipant
- Microsoft.TeamFoundation.Client.TfsTeamProjectCollection

For more information about GetExtension, please see "Inside the Microsoft® Build Engine: Using MSBuild and Team Foundation Build. 2nd Edition"[112].

Designer

If you prefer to work within a designer, you can add a XAML activity to your test project and drag your activity onto it. You may author the Workflow as necessary, with easy access to additional activities, arguments and properties.

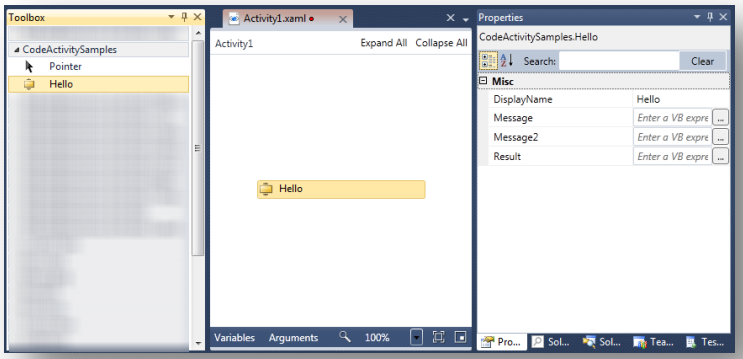


Figure 130 – Designer showing activities and properties

You would then use Workflow invoker to initiate your Workflow as you did with the previous samples.

Remote Debugging

Because activities are generally isolated components with well-defined inputs and we have the ability to provide extension data, it should be rare that you need to perform remote debugging on a build server. Given this worst-case scenario though, you should follow these steps to perform remote debugging.

Prerequisites

- To perform remote debugging you must have a Pro, Premium or Ultimate SKU of Visual Studio 2010.
- Please see the MSDN guidance to setup remote debugging here - http://msdn.microsoft.com/en-us/library/bt727f1t.aspx and Remote Debugging Summary[113]

[112] http://go.microsoft.com/fwlink/?LinkID=206999&clcid=0x409

- We will follow the 'share' method here.
- The build server must have the program database files (pdbs) with full debug info.

Process

1. On your **Developer Workstation**:
   a. Share your Remote Debugger folder. This folder is located here

Visual Studio 2010
• **%ProgramFiles%\Microsoft Visual Studio 10.0\Common7\IDE\Remote Debugger**
Visual Studio 2012
• **%ProgramFiles%\Microsoft Visual Studio 11.0\Common7\IDE\Remote Debugger**

   b. Ensure that the user from the build server has access to the share.

2. On the **Build Server**:
   a. Access the shared folder from the Build Server.
   b. Navigate to the flavor of your operating system and start msvsmon.exe.

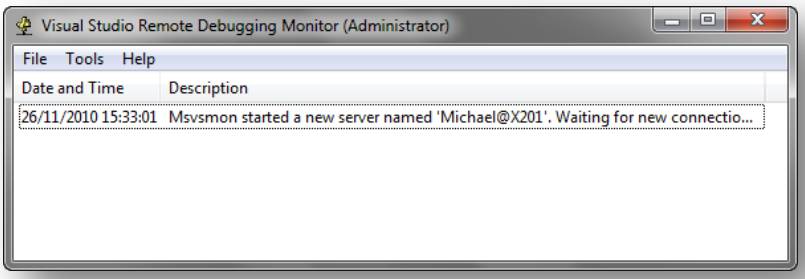


Figure 131 – Remote Debugger User Interface

3. On your **Developer Workstation**:
   a. Open your activities project / solution in Visual Studio.
   b. Click **Debug** and then click **Attach to Process**.

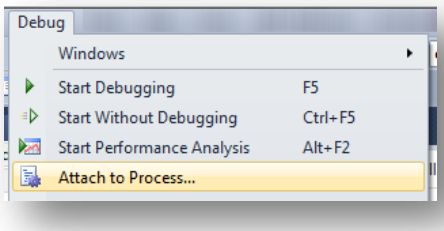


Figure 132 – Attach to Process from the Debug menu

   c. In the Attach to Process window:

[113] http://msdn.microsoft.com/en-us/library/ff678494.aspx

i. Browse to your Build server.
ii. Select the TFSBuildServiceHost.exe process. If you do not see the process, select the **Show processes from all users** and **Show process in all sessions** check boxes.
iii. Click **Attach**.

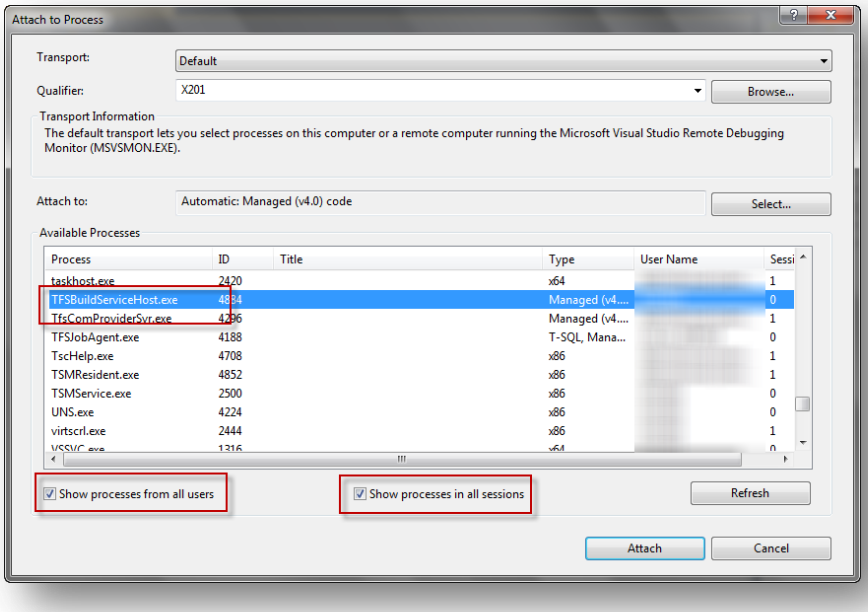


Figure 133 – Attach to Process Dialog

d. Set a breakpoint in your code.
e. Start the applicable build.

Versioning Assemblies

Understanding the different versioning attributes

One single topic has garnered an incredible amount of discussion, debate and passion over the years; that is the topic of versioning code during Team Foundation builds. First, let us clarify that by versioning code we are referring to setting the numbers in the version attributes of your assembly information file. We are not referring to the Team Foundation Build number or labeling, although we will illustrate that these are, or at least should be, logically linked.

A version number consists of four components, <major version>.<minor version>.<build number>.<revision>. Each component must be an integer greater than or equal to zero and each is restricted to a maximum value of UInt16.MaxValue - 1. A compilation error will occur if any part exceeds this limit.

We mentioned “the version attributes” and this is perhaps a good place to start. You should be aware of three attributes:

Available Attributes

AssemblyVersionAttribute[114] - MSDN provides a great explanation of this attribute. The key parts to take from this are

- The Common Language Runtime (CLR) cares very much about your AssemblyVersionAttribute.

[114] http://msdn.microsoft.com/en-us/library/zb298d28.aspx

- It plays a key part in binding to the assembly and in version policy.
- The default version policy for the runtime is that applications run only with the versions they were built and tested with, unless overridden using Assembly Binding Redirection[115].
- Version checking only occurs with strong-named assemblies. For this guidance, we assume that you are strong-naming your assemblies.

An AssemblyVersionAttribute should be scoped to the <major version> and <minor version> components with the other components set to zero. For example:

```
[assembly: AssemblyVersion("1.5.0.0")]
```

Note that you may use non-zero values for the <build> and <revision> components. However this could introduce confusion because they will differ from those in the AssemblyFileVersion attribute. The AssemblyVersionAttribute should **not change** during the development of your release and you may even choose to maintain the same version across releases if you do not require side by side installations.

AssemblyFileVersionAttribute - Used to identify a specific build of your assembly. The CLR does not care about this attribute and does not attempt to examine it. If this attribute is not specified, the value for AssemblyVersionAttribute is used.

The AssemblyFileVersionAttribute should be scoped to the <build number> and <revision> components with the <major version> and <minor version> components being the same as the AssemblyVersionAttribute values. For example:

```
[assembly: AssemblyFileVersion("1.5.1234.1")]
```

The AssemblyFileVersionAttribute **should change** during every Team Foundation Build. How it changes is up to you and we will not offer a preferred numbering scheme because there is none. Regardless of your numbering scheme, the objective is the same; **you should be able to associate the versioned assembly to a particular Team Foundation Build**. Typically, the <build number> component will change each day and the <revision> component will change with each Team Foundation Build.

AssemblyInformationalVersionAttribute essentially represents the product version. The CLR does not care about this attribute and does not attempt to examine it. If this is not specified, the value for AssemblyVersionAttribute is used. If you have avoided using this in the past due to the warning *"Assembly generation -- The version 'XYZ' specified for the 'product version' is not in the normal 'major.minor.build.revision' format"*, you will be happy to know that this is resolved from Visual Studio 2010, so you can use something like '2.0 RC' with no issues.

It's up to you how and if you choose to use and update this attribute.

Using Wildcards in Attributes

When you start a new project, you will see some template code like the following:

```
// Version information for an assembly consists of the following four values:
//
//      Major Version
//      Minor Version
```

[115] http://msdn.microsoft.com/en-us/library/2fc472t2.aspx


```
//      Build Number
//      Revision
//
// You can specify all the values or you can default the Build and Revision Numbers
// by using the '*' as shown below:
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

If you are building code which is strongly named, then you should avoid using wildcards due to the issues we have discussed about the way the runtime resolves assemblies. If you use a wildcard, the Build Number will be set as the number of days elapsed since 01 January 2000 and the Revision will be set as the number of seconds elapsed since midnight. You can find information about using a custom date, rather than 01 January 2000 this blog post: http://blogs.msdn.com/b/grantri/archive/2004/04/13/112837.aspx

Common Versioning Practices

Now that you understand the different versioning attributes, let us look at some common practices for changing them during team daily Team Foundation Builds. Note that there is no reason to version your assemblies during continuous integration builds because the output of these builds should not be used for deployment.

<major version>

The **AssemblyVersion** and **AssemblyFileVersion** will share the **same** major version.

The major version will typically remain the same for the lifecycle of a major release of the software. Therefore, in release two of your product the AssemblyVersion will by **2**.0.0.0 although the AssemblyFileVersion may be **2**.0.1234.1. Remember that the runtime uses the AssemblyVersion for binding, so changing this version will usually result in a high code churn to ensure that all references are updated.

This component will not be changed by builds.

<minor version>

The **AssemblyVersion** and **AssemblyFileVersion** will share the **same** minor version.

The minor version may be increased for point or smaller releases, although you must consider the effect on runtime binding and the associated cost in code churn.

This component will not be changed by builds.

<build number>

The **AssemblyVersion** and **AssemblyFileVersion** will **not** share the same build number.

The AssemblyVersion will again remain static and usually be zero.
The AssemblyFileVersion is where most people stamp their unique versioning algorithm. This number will usually be unique for any given day. It can be anything from zero to UInt16.MaxValue - 1, which is 65534. The two most common values for this component are a formatted date or an elapsed calculation. For example:

- An assembly built on Dec 25: 2.0.**1225**.1
- An assembly built on Nov 17: 2.0.**1117**.1
- An assembly built 300 days after a specified start date: 2.0.**300**.1

This component will be changed on a daily basis by the build process.

<revision>

The **AssemblyVersion** and **AssemblyFileVersion** will **not** share the same revision number.

The AssemblyVersion will again remain static and usually be zero. For the AssemblyFileVersion, this is usually taken from the revision provided by the Team Foundation build. Sometimes additional calculated numbers are prefixed or suffixed to the provided revision, but this number is usually unique for every build, regardless of quality.

This component will be changed for each build by the build process.

In summary

	<major version>	<minor version>	<build number>	revision
AssemblyVersion	=	=	Static in daily builds	Static in daily builds
AssemblyFileVersion	=	=	changes per day	changes per build

Table 19 – Versioning Assemblies Summary

Avoiding Source Control Operations in Versioning

It is disappointingly common to see a build process checking out many versioning files, typically AssemblyInfo.cs, during the build process, incrementing the AssemblyFileVersion attribute and then checking in the files. By 'versioning files' we mean those files which contain the versioning attributes. For example:

```
[assembly:AssemblyVersion("1.0.0.0")]
[assembly:AssemblyFileVersion("1.0.0.0")]
[assembly:AssemblyInformationalVersion("1.0.0.0")]
```

There are several downsides to this approach:

- It takes time, code and effort to perform these operations.
- The developer needs to ensure that they use ***NO_CI*** in the check-in comment so as not to trigger Continuous Integration (CI) builds.
- Depending on the source control structure and frequency of builds, the history over the source control tree might be littered with these version control changesets.
- You will have to deal with unnecessary merge candidates.
- You should be able to distinguish between a build that is built on a build server and one that is built on a developer's machine. By updating the AssemblyInfo.cs file as it is stored in version control, you could lose the distinction.

Version control is not about ensuring that your AssemblyInfo.cs files in source control have the correct versions. Rather, version control ensures that the build **output** is correctly versioned and can be correlated to a build. Your versioning should not be determined by the contents of your versioning files, but rather based on the robust revision logic provided out the box by Team Foundation Server since the initial Team Foundation Server 2005 release.

Another good practice is to use a single linked file across all your products' projects, for example, VersionInfo.cs, so that the single file can be updated during the build process and all your output is at the same version.

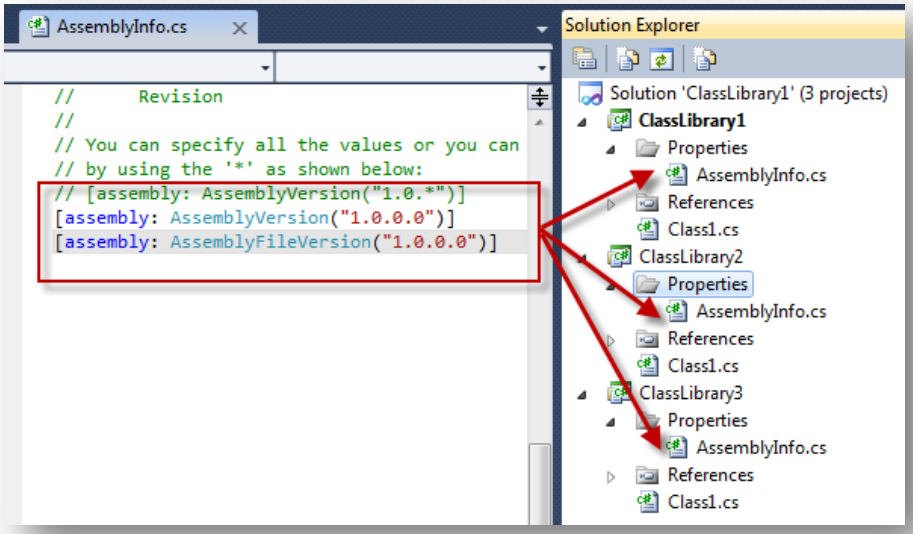


Figure 134 – Not Ideal. Projects all containing attributes which should be common, such as versioning attributes

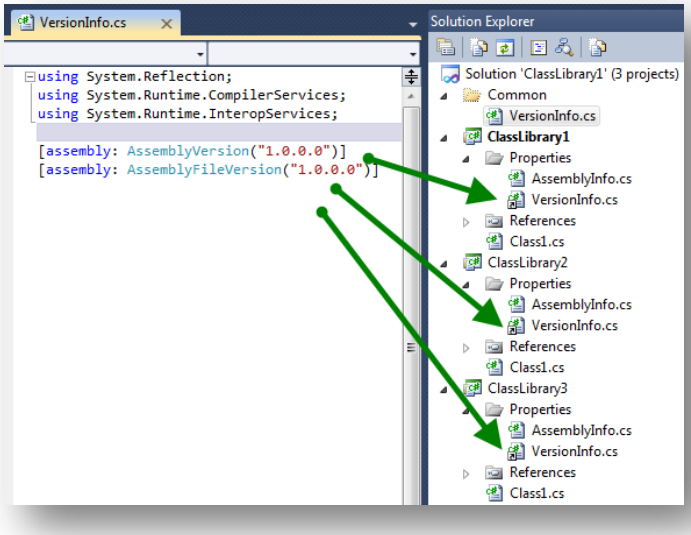


Figure 135 – Better. Projects linking a shared attribute file

The TfsVersion activity in the "Community TFS Build Extensions"[116] project provides an easy way to apply consistent versioning. This activity uses the unique revision logic provided by Team Foundation Server and various user configurable preferences to provide a suitable versioning strategy. The TfsVersion activity is covered in more details here.

Versioning non-AssemblyInfo code

You might encounter code that does not use the standard versioning attributes found in the AssemblyInfo file, such as C++ and BizTalk projects.

[116] http://tfsbuildextensions.codeplex.com

The guidance for this code is the same as that for code that uses AssemblyInfo style versioning, which is to isolate your versioning information to a common linked file, if possible, and ensure that all code is versioned at the same version. To maintain the same version, your language might have an API to call, or most likely, you might have to edit file contents at the time of the build.
As an example, let us assume that your solution consists of various C# projects that share a linked VersionInfo.cs file and a collection of BizTalk projects (.btproj) which have version information contained in each .btproj project file. For example:

```
<Build>
  <Settings>
    <EditorCommon/>
    <ProjectCommon
    …
      AssemblyInformationalVersion = "1.1.0.0"
      AssemblyFileVersion = "1.1.0.0"
    …
```

Assuming you have calculated your version, you could write an activity that parses the file and updates the AssemblyFileVersion with the new version. A code activity to do this can be found in the "Community TFS Build Extensions"[117] project.

The sample below shows how this would look in a Workflow.

1. Get Workspace is the standard Team Foundation Build activity that runs on the Agent.
2. We use the FindMatchingFiles activity to find files that match our common versioning file. For example:

```
MatchPattern: String.Format("{0}\**\AssemblyInfo.cs", SourcesDirectory)
```

3. We use the "Community TFS Build Extensions" TfsVersion activity to get our version and set it in the matching files.
4. We use the FindMatchingFiles activity to find our BizTalk files. For example:

```
MatchPattern: String.Format("{0}\**\*btproj", SourcesDirectory)
```

5. We use the "Community TFS Build Extensions" File activity replace the version information in the BizTalk Files.

[117] http://tfsbuildextensions.codeplex.com

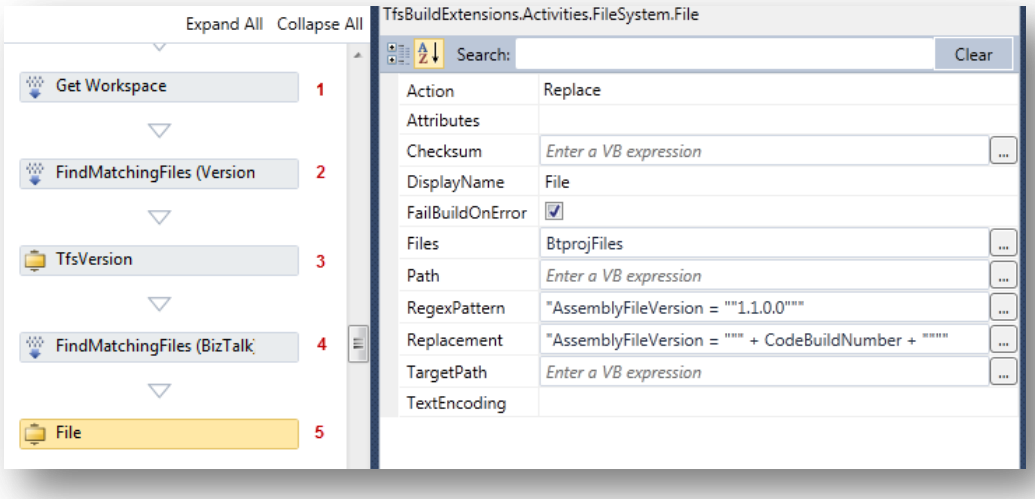


Figure 136 – Versioning code using the "Community TFS Build Extensions" activities

In this sample, you can see that we are calling the File activity with an Action of Replace. We provide the files, which the FindMatchingFiles activity returns, and we provide a pattern to search for with replacement text, which contains our calculated CodeBuildNumber.

Create a Versioning Code Activity

As we have illustrated, there are several best practices for versioning code. If you need to create a bespoke activity, you should keep this guidance in mind.

Rather than create another versioning activity here, we will cover the TfsVersion task, which is available in the "Community TFS Build Extensions" project on CodePlex.

TfsVersion Activity

The TfsVersion Activity is a general-purpose versioning task based on the TfsVersion MSBuild task provided in the MSBuild Extension Pack. If you use this MSBuild task in your current build processes, then it will be an easy migration when you move a build based on the Default XAML template in Team Foundation Build in Visual Studio. The activity exposes the following interface:

Property / Argument	Comments
Action	Provides a list of actions which the activity supports. The activity currently supports GetVersion, SetVersion and GetAndSetVersion. ~ GetVersion will use a combination of the current build information and the properties you provided to generate the complete version number. ~ SetVersion will apply the calculated version number to the specified files. ~ GetAndSetVersion will perform both operations from the single activity.
AssemblyDescription	Text to set the AssemblyDescriptionAttribute value to.
AssemblyVersion	Specifies the AssemblyVersion. This defaults to Version if not set.
Build	Gets or Sets the Build component of the version number.
CombineBuildAndRevision	Specifies whether to make the revision a combination of the Build and Revision. For example, if the build number is 1208 and the revision is 1, then the revision is altered to be 12081.
DateFormat	Specifies the date format to use when using VersionFormat="DateTime". For example: MMdd

Property / Argument	Comments
Delimiter	The Delimiter to use in the version number. The default is a . (period)
FailBuildOnError	Specifies whether to fail the build if the activity logs an error. The default is false.
Files	Specifies the files to version. Typically theses would be passed in as a result of using the FindMatchingFiles activity.
ForceSetVersion	Set to true to force SetVersion action to update files that do not have AssemblyVersion or AssemblyFileVersion attributes present. The default is false. ForceSetVersion does not affect AssemblyVersion when SetAssemblyVersion is false.
LogExceptionStack	Specifies whether to log the full exception stack in the event of an exception. The default is false.
Major	Gets or Sets the Major component of the version number.
Minor	Gets or Sets the Minor component of the version number.
PaddingCount	Specifies the number of padding digits to use in the Build component of the version number.
PaddingDigit	Specifies the padding digit to use.
Revision	Gets or Sets the Revision component of the version number.
SetAssemblyDescription	Specifies whether to set the AssemblyDescriptionAttribute value. The default is false.
SetAssemblyFileVersion	Specifies whether to set the AssemblyFileVersionAttribute value. The default is true.
SetAssemblyVersion	Specifies whether to set the AssemblyVersionAttribute value. The default is false.
StartDate	Specifies the start date to use when the VersionFormat is Elapsed.
TextEncoding	Specifies the encoding to use when saving the files which have been changed. The default is UTF8.
TreatWarningsAsErrors	Specifies whether to treat any warning as an error. If FailBuildOnError is true, then a warning will fail the build
UseUtcDate	Specifies whether to use UTC date / time in calculations. The default is false.
Version	Gets or sets the Version.
VersionFormat	Specifies the version format to use. Supports Elapsed and DateTime.
VersionTemplateFormat	Specify the format of the build number. A format for each part must be specified or left blank, e.g. "00.000.00.000", "..0000.0"

Table 20 – TfsVersion Interface

It is a large interface, but the tasks ability can be summarized as follows:

Build Component	Comment
Major	Supports any value
Minor	Supports any value
Build	Supports any value Can be calculated as days since a given date Can be any date format Can be padded with any digit
Revision	Supports any value Generally taken from Team Foundation Build revision number Can be combined with the build number

Table 21 – Summary of TfsVersion's capabilities

NOTE

Additional functionality is being added to this task on an on-going basis. For the latest capabilities, it is best to consult the online documentation for the task.

Below is an example of the activity in Visual Studio:

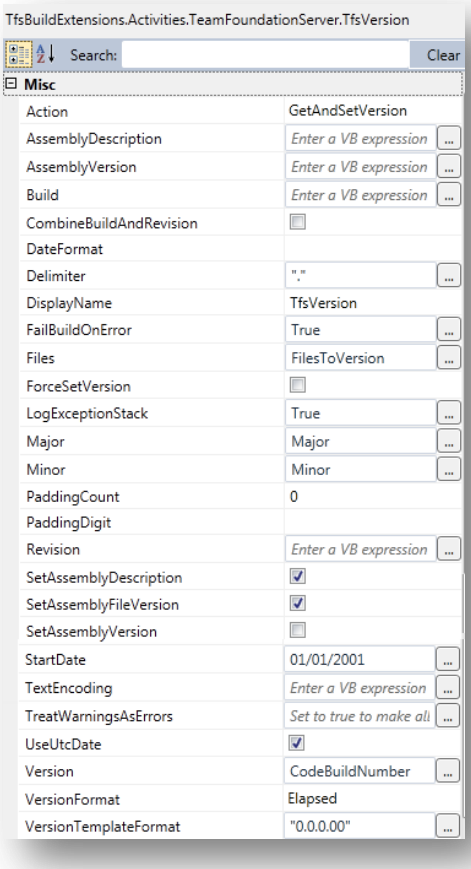


Figure 137 – TfsVersion Property Grid

Sample Build Numbers

- **1.0.11208.1**
  - A build taken on December 12th using a padding digit of 1.
  - As revision is zero based, this is the second build of the day.
- **1.0.355.2**
  - The third build on a day that is 355 days after a given starting date.
- **1.0.1298. 12983**
  - The fourth build on a day that is 298 days since a given date and using a padding digit of 1.
  - The revision is combined with the build number.

Managing Virtual Machines

Even if you are not using Lab Management, being able to manipulate virtual machines during a build can play an import part in your development process. You could start a virtual machine during the build, deploy to it, perform a fully automated test run, and then snapshot it or discard the server. The "Community TFS Build Extensions"[118] project is creating a library of activities to manipulate the various different host technologies of today.

Hyper-V

The Hyper-V activity is based on the samples provided on MSDN in the second 'Using the Hyper-V WMI Provider' (http://msdn.microsoft.com/en-us/library/cc723875(v=VS.85).aspx).

The WMI provider for Hyper-V enables developers, and scripters, to quickly build custom tools, utilities, and enhancements for the virtualization platform. The WMI interfaces can manage all aspects of the Hyper-V services.

The WMI provider is used to send commands to a Hyper-V host to manage a Virtual Machine. This Hyper-V host can be local, on the build host, or on a remote Hyper-V host, assuming that the build user account the build is running under has suitable rights to administer.

The `HyperV` activity supports the following actions:

Hyper-V Activities	
Activity	Usage
ApplyLastSnapshot	Restores last snapshot of the Virtual Machine.
ApplyNamedSnapshot	Restores the first snapshot of the Virtual Machine found that matches the name provided in the SnapshotName property.
Pause	Pauses the running Virtual Machine.
Restart	Restarts the Virtual Machine.
Shutdown	Performs a shutdown of the guest operating system on the Virtual Machine.
SnapShot	Takes a snapshot of the current state of the Virtual Machine.
Start	Starts a stopped or paused Virtual Machine.
Suspend	Saves and stops a running Virtual Machine.
Turnoff	Immediately halts a running Virtual Machine.

Table 22 – Hyper-V Activity Actions

Windows VirtualPC

The VirtualPC activity is based on the VirtualPC task in the MSBuild Extension Pack for Team Foundation Server 2008 (http://msbuildextensionpack.codeplex.com/). It provides the same functionality via the same set of operation commands.

The VirtualPC activity achieves this using the Microsoft.VirtualPC.Interop assembly. As with the Hyper-V activity, this allows commands to be issued to the VirtualPC Virtual Machine to perform certain actions. The key difference is that the VirtualPC instance must be hosted locally on the same PC as the build is being run on.

The `VirtualPC` activity supports the following actions:

Windows VirtualPC Activities	
Activity	Usage
AddHardDiskConnection	Adds a new VHD to a stopped Virtual Machine.
DiscardSavedState	Discards the saved state of a stopped Virtual Machine.
DiscardUndoDisks	Discards the undo disk of a stopped Virtual Machine.
IsHeartBeating	Checks if the Virtual Machine is running.

[118] http://tfsbuildextensions.codeplex.com

Windows VirtualPC Activities	
IsScreenLocked	Checks if the Virtual Machine's screen is locked.
List	Lists the Virtual Machines on the host.
LogOff	Logs off from the guest OS on a Virtual Machine.
MergeUndoDisks	Merges the undo disk of a Virtual Machine into the parent VHD.
Pause	Pauses a running Virtual Machine.
RemoveHardDiskConnection	Removes a VHD from a stopped Virtual Machine.
Restart	Restarts a Virtual Machine that is not currently off.
Resume	Resumes a paused Virtual Machine.
RunScript	Runs a script containing text and mouse input on the guest OS.
Save	Saves a running Virtual Machine.
Shutdown	Shutdowns the guest OS on a running Virtual Machine.
Startup	Starts a stopped Virtual Machine.
TakeScreenshot	Takes a screenshot of the Virtual Machine desktop and saves as a BMP.
Turnoff	Immediately stops a Virtual Machine.
WaitForLowCpuUtilization	Pause the Workflow until the Virtual Machine's CPU reaches a set level.

Table 23 – Windows VirtualPC Activity Actions

Test Impact Analysis

Refer to TFS 2010 Build - Only run impacted tests[119] for more information.

[119] http://scrumdod.blogspot.com/2011/03/tfs-2010-build-only-run-impacted-tests.html

Deployment of Applications and Data Stores

Deploy Environments

Overview

This section describes our reference scenario for deployment. In current practice, we have found many variations but, due to limited space, we will cover only some cases, which the reader can easily adapt to his/her situation.

Our hypothetical IT department has five environments.

Environment	Description
Developer	This is composed of developer workstations and virtual machines. These could be part of organization's domain. Developers' accounts have administrative rights on these machines. We suggest an *isolated* development model for most scenarios, in which a developer has all the required software installed on his own machine with no shared database.
Integration	Used for Integration test. Builds are deployed here first. This environment reproduces some fundamental traits of the Production environment. Some or all developers' accounts may have administrative rights in order to troubleshoot and occasionally to debug. Testers have access to this environment.
QA	(Quality Assurance) Used for Acceptance test. QA mimics Production in a scaled-down version to limit cost using the same topology with reduced resources. Developers have limited access such as read-only. Testers have access to this environment. This environment should be on its own domain (AD, DNS). In some cases, QA shares the production domain, for example, when it doubles as a staging environment. Deployment is done by Infrastructure team (Jane).
Production	Only the Infrastructure team (or IT Operation pros) has administrative access to this environment. Developers get log and trace files in an ad-hoc or automated way; however such files must not contain sensitive information, such as passwords, credit card numbers, etc.
Service	We consider Team Foundation Server, build server(s) and lab host to be part of this environment; it includes utilities like file sharing, collaboration, etc.; it can managed by Developers (e.g. Garry) or IT Operation depending on scale.

Table 24 – Hypothetical IT Department Environments

Although this arrangement is immediately applicable to an *on-premises* scenario, it can be easily adapted to the *cloud*.

In this document, we suppose the following set of machines and domains. They will be referenced in the examples and HOLs:

Environment	Domain	DNS suffix
Developer	INTERNAL	.internal.contoso.com
Integration	TEST	.test.contoso.com
QA	QA	.qa.contoso.com
Production	PROD	.contoso.com
Service	INTERNAL	.internal.contoso.com

Table 25 – Hypothetical IT Department Environment Domains and DNS Details

Role	Environment						Note
	Developer	Integration	QA		Production		
Web server	DEV*nn*	WEB	WEB01	*WEB*	EPS45WR01	*WWW*	*Load balanced*
			WEB02		EPS45WR02		
			n/a		EPS45WR03		
DB server		SQL	SQL01	*SQLCLU*	EPS41DR01	*EPS41DCL01*	*Clustered*
			SQL02		EPS41DR02		
Management server		n/a	MNGT01		EPS41MR01		—

Table 26 – Hypothetical IT Department Environment Roles

- Note that Production machines comply with a naming convention that does not convey the machine role —a common scenario.
- The Production web servers are exposed to the Internet through a firewall.
- In a real-world setup, there will be many more machines with more roles, but the preceding and following tables are sufficient for the purpose of this document.

Service Environment	Note
TFS	We suppose a single Team Foundation Server instance with a single AT machine; DT is not shown.
BUILD01	Build server for .NET.
BUILD02	Build server for Java.
FILES01	File share server hosting *drop* folders. The file shares on this machine are accessible from all environments. Although this may be unrealistic, adding hops to move a file from one environment to another is not practical for our purposes.

Table 27 – Hypothetical IT Department Environment Notes

Garry's machine is DEV01.internal.contoso.com. The name of the second production web server is EPS45WR02.contoso.com.

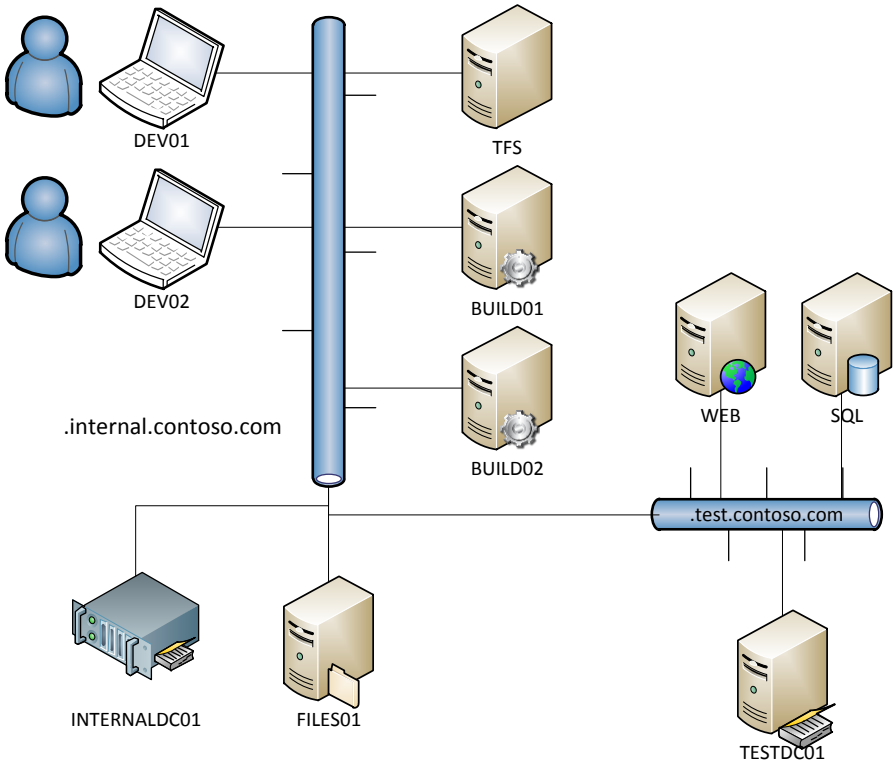


Figure 138 – Hypothetical Developer & Infrastructure Environment

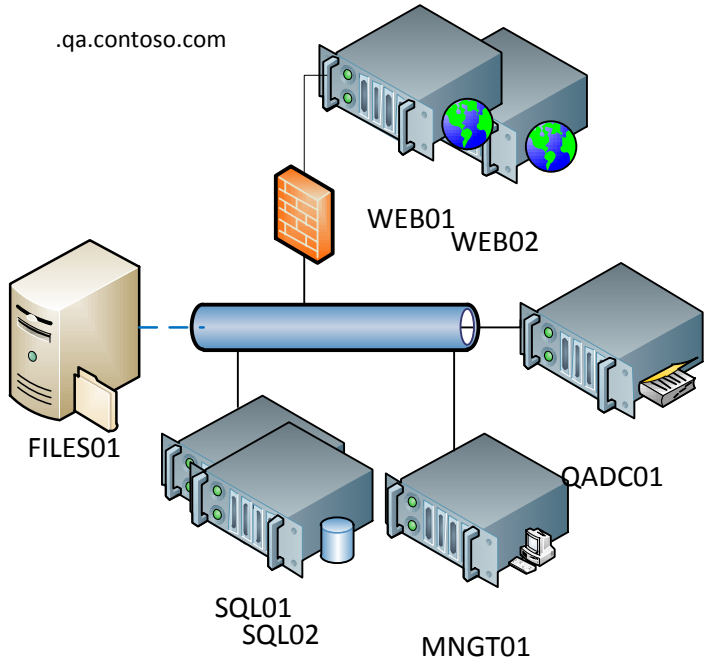


Figure 139 – Hypothetical Quality Assurance Environment

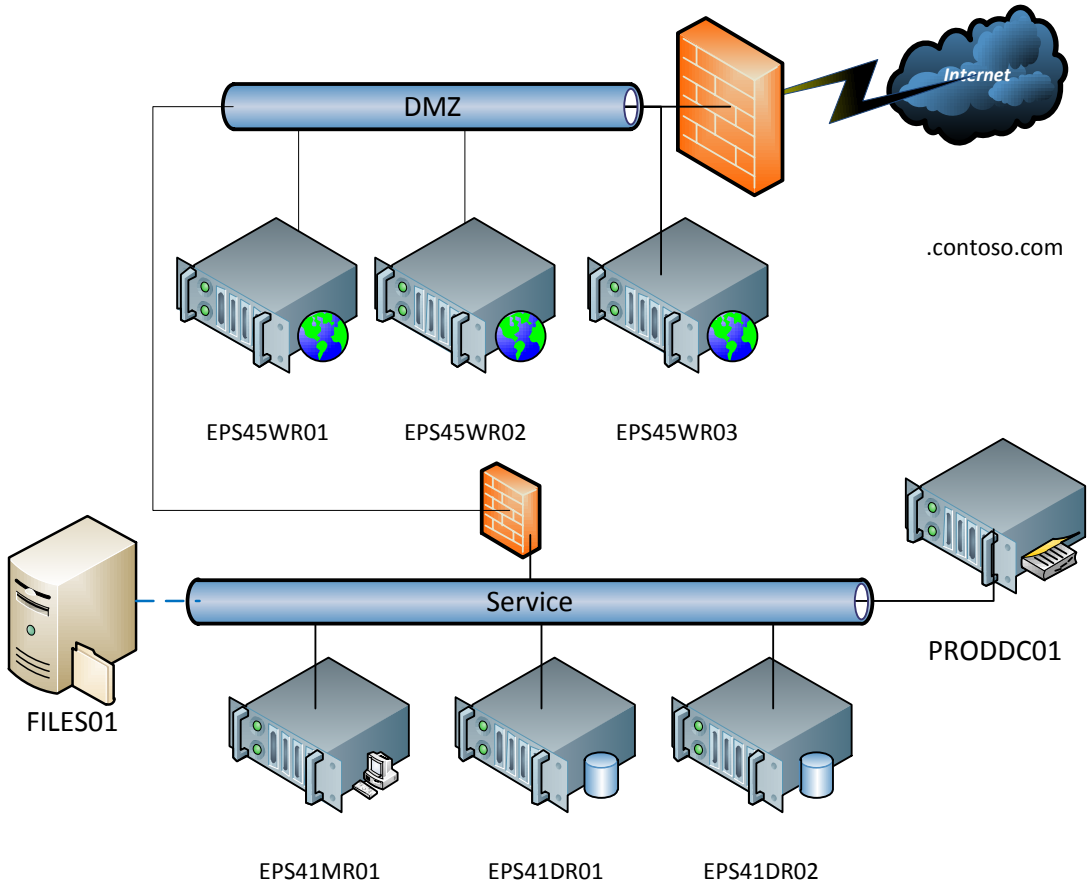


Figure 140 – Hypothetical Production Environment

Software configuration

The following table lists the software installed on our hypothetical systems.

Role / Server	Installed software	Note
TFS	Windows Server 2008 R2 Standard Edition x64 Team Foundation Server — AT	
BUILD01	Windows Server 2008 R2 Standard Edition x64 Team Foundation Server— Build agent Visual Studio	Visual Studio is required for some build scenarios like Code Analysis, Test Impact, etc.
BUILD02	Windows Server 2008 R2 Standard Edition x64 Team Foundation Server — Build agent Java Development Kit 6 Apache Ant 1.8 Apache Maven 3.0 JUnit 4.8 FitNesse 20101101	
FILES01	Windows Server 2008 R2 Standard Edition Core x64	

Role / Server	Installed software	Note
Web server	Windows Web Server 2008 R2 Web Deployment Tool (Web Deploy)	
DB server	Windows Server 2008 R2 Enterprise Edition x64 SQL Server 2008 R2 Enterprise Edition x64 (DB,AS)	
Management server	Systems Center Operations Manager 2007 R2	

Table 28 – Hypothetical Environment Software

Users

In the next section, we will use a number of users and accounts from our hypothetical IT department, whose characteristics are summarized in the following table.

Person	Role	Accounts	Permission	Note
Doris	Dev	INTERNAL\doris	Admin rights on DEV04; Read access to FILES01	
Garry	DevLead	INTERNAL\garry	Admin rights on DEV01; Read access to FILES01; TP Admin*	
		TEST\garry	Read access to FILES01	
Abu	Build Master	INTERNAL\abu	Admin rights on Team Foundation Server, BUILD01 and BUILD02	
Sam	Release Manager	TEST\sam		Knows enterprise admin password
		QA\sam		
Jane	Infrastructure specialist	QA\jane	QA Domain Admin	
		PROD\jane	PROD Domain Admin	
Dave	Team Foundation Server Admin	INTERNAL\dave	Admin rights on Team Foundation Server, BUILD01 and BUILD02	
Christine	Tester	INTERNAL\chris		
		QA\chris		
	DBA	QA\tbd	SysAdmin on SQLCLU	
		PROD\tbd	SysAdmin on EPS41DCL01	

Table 29 – Hypothetical Environment Users

We will also refer to a number of technical accounts that do not correspond to a user. These are listed in the following table.

Domain	Account	Permission
INTERNAL	TFSSERVICE	See *Team Foundation Installation Guide for Visual Studio 2010*[120]
INTERNAL	TFSBUILD	See *Team Foundation Installation Guide for Visual Studio 2010*
TEST	WEBSERVICE	Granted login on SQL; belongs to WEB\IIS_IUSRS group
TEST	SQLSERVICE	See Setting Up Windows Service Accounts[121] in SQL Server Books Online[122]
QA	WEBSERVICE	Granted login on SQLCLU; belongs to WEB01\IIS_IUSRS and WEB02\IIS_IUSRS groups
QA	SQLSERVICE	See Setting Up Windows Service Accounts in SQL Server Books Online
PROD	WEBSERVICE	Granted login on EPS41DCL01; belongs to EPS45WR01\IIS_IUSRS, EPS45WR02\IIS_IUSRS and EPS45WR02\IIS_IUSRS groups
PROD	SQLSERVICE	See Setting Up Windows Service Accounts in SQL Server Books Online

Table 30 – Hypothetical Environment Service Accounts

[120] http://www.microsoft.com/downloads/en/details.aspx?displaylang=en&FamilyID=2d531219-2c39-4c69-88ef-f5ae6ac18c9f

[121] http://msdn.microsoft.com/en-us/library/ms143504.aspx

[122] http://msdn.microsoft.com/en-us/library/ms130214.aspx

Database deployments

This chapter describes how to include the database deployment in the context of a Team Foundation Build using a Database project (.dbproj) as the source. Team Foundation Build embraces and supports databases, which are a fundamental part of most solutions, as part of the build process. Supporting databases creates a unified compile, build, and deployment model.

For an overview of the structure of database projects, visit the Visual Studio Database Project Guidance[123] project on CodePlex.

There are different levels of Build and Deployment you can use with MSBUILD / Team Foundation Build:

- **Build only**
  Compile and build the projects and relevant artifacts needed to deploy of the database.

NOTE

While deploying different stages during the application lifecycle, such as having a development / staging / testing / production we must handle different "versions" of the database, meaning they differ in their schema. While we normally don´t know which database version we are targeting, the built artifacts produced by the build are considered to be version agnostic. This leaves us the freedom to take the artifacts and deploy them to any database version schema and downgrade or upgrade the schema as needed. The deployment engine VSDBCMD.EXE will handle this and produce the relevant schema motion scripts.

- **Build and Deploy**
  Compile and build the projects and relevant artifacts, and deploy the database to a configured target database server.
- **Build & Deploy & Run Automated Unit Tests**
  In addition to build and deploy, run automated unit tests, which are typically database unit tests.

Build Only

Build process and needed files

Including a database project in a build is similar to including other project types. Whether you only build the project file (.dbproj) or a whole solution (.sln) the output artifacts typically include:

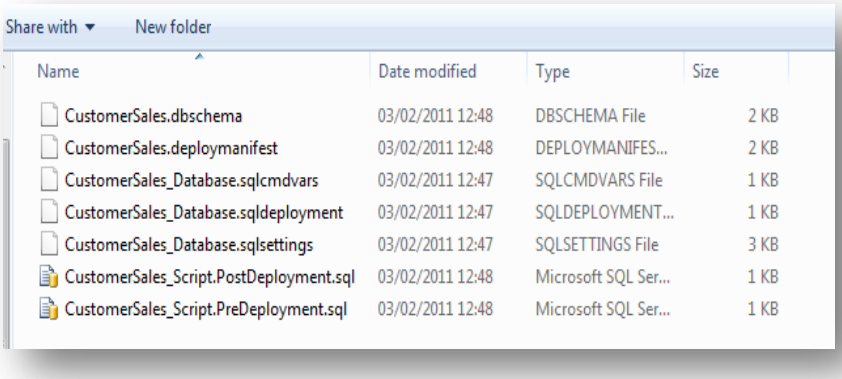


Figure 141 – Sample build output of a database project

The most important artifact from the preceding list is the .dbschema file. It contains the complete conceptual model of the database, which can be used to create or upgrade a database using the deployment engine

[123] http://vsdatabaseguide.codeplex.com

VSDBCMD. The other files are used to manage non-modeled database objects and pre/post executed scripts as well as settings and runtime parameters. For more information, refer to the *"Deployment Configuration and Pre/Post Deployment Script"* section of the database project guidance. It contains a brief description of the file and its purposes.

Deployment

Automated Deployment

Automated deployments from Team Foundation Build are valuable in many situations:

- **Checking for deployment issues**
  Deployment of the database can discover additional errors that weren´t found during the regular build. For example:
  - Issues with variables in scripts, for example, are typically only found during a real deployment.
  - The incompatibility of certain features on target platforms (e.g. using Enterprise Edition features on a target Standard edition) or the lack of the installed Full-text engine feature for Full-text enabled database.
- **Testing deployment feasibility**
  Although almost all changes can be propagated to the database and the target database schema, there might be options selected for your deployment that are not compatible with the changes you want to do, such as:
  - Using the option "Block incremental deployment if data loss might occur" with having existing data in the database.
  - Changing a NULLABLE column to be NOT NULLABLE with already existing NULL values for the attribute and no default assigned to the attribute.
- **Getting rule-of-thumb figures for upgrades to the database**
  - For a large database, the time needed to upgrade the database can be on the critical path. The deployment engine might decide to run data movement scripts, which securely copy data over, to a new table with the schema changes, dropping the old data. Note that this might introduce massive IO processing and physical storage utilization, exceeding expectations.
- **Getting information for needed deployment prerequisites**
  - Development machines might already have the needed bits installed on the machine, so you might want to deploy the database from a machine, which is equal to the machine used in the subsequent environments like Testing / Staging / Production.

Reading through all the above-mentioned situations, you might ask if deploying / upgrading an empty database could the best solution for the automated deployment. The answer is **no**. Always use an existing database, which is as similar as possible to the target database, in terms of data volume, data complexity, and deployment time and complexity. If you have problems getting a production database because it contains sensitive data, define a process to de-sensitize the data and use a backup of this database in subsequent deployments.

If you have a backup of a database, a good build and deployment process would be to:

1. Restore the database.
2. Build the project.
3. Deploy the database.
4. Run sanity checks on the database to make sure the database is consistent.
5. Run Unit Tests on the database.

Although the build and deploy can be done using the general targets for deploying database in your Team Foundation Build, it is preferable using additional targets in your customized target file to include such tasks as described in points 1 and 4. These are custom tasks that cannot be executed through the project system. Specifically, to include in your Workflow the command to restore a database, which you want to upgrade, use an InvokeProcess activity, such as in the following XAML fragment.


```
    <mtbwa:InvokeProcess Arguments="[String.Format(&quot;-S. -E -i {0}&quot;,
    RestoreCommandScript)]" DisplayName="Restore database before upgrade"
    FileName="sqlcmd.exe" Result="[ExitCode]">
      <mtbwa:InvokeProcess.ErrorDataReceived>
        <ActivityAction x:TypeArguments="x:String">
          <ActivityAction.Argument>
            <DelegateInArgument x:TypeArguments="x:String" Name="errOutput" />
          </ActivityAction.Argument>
        </ActivityAction>
      </mtbwa:InvokeProcess.ErrorDataReceived>
      <mtbwa:InvokeProcess.OutputDataReceived>
        <ActivityAction x:TypeArguments="x:String">
          <ActivityAction.Argument>
            <DelegateInArgument x:TypeArguments="x:String" Name="stdOutput" />
          </ActivityAction.Argument>
          <Sequence mtbwt:BuildTrackingParticipant.Importance="Low">
            <Assign mtbwt:BuildTrackingParticipant.Importance="Low">
              <Assign.To>
            <OutArgument x:TypeArguments="x:String">[ErrorMessage]</OutArgument>
              </Assign.To>
              <Assign.Value>
                 <InArgument x:TypeArguments="x:String">
                      [If(Not String.IsNullOrEmpty(ErrorMessage),
                      Environment.NewLine + ErrorMessage, "") + stdOutput]</InArgument>
              </Assign.Value>
            </Assign>
            <mtbwa:WriteBuildMessage
    Importance="[Microsoft.TeamFoundation.Build.Client.BuildMessageImportance.Low]"
    Message="[stdOutput]" mva:VisualBasic.Settings="Assembly references and imported
    namespaces serialized as XML namespaces" />
          </Sequence>
        </ActivityAction>
      </mtbwa:InvokeProcess.OutputDataReceived>
</mtbwa:InvokeProcess>
```

If you are using the Upgrade template, add a target to your TFSBuild.proj with similar code:

```
    <Exec Command='sqlcmd.exe  -S. -E -i RestoreCommandScriptHere
       Condition="%(RestoreDatabases.DatabaseName) != ''"/>
```

If you want to run sanity checks against the database to ensure database integrity, you would also use the mentioned syntax in the Exec Command from the previous code and run your individual check scripts.

We recommend using a VSDBCMD command to execute a deployment and to process the results .sql file.

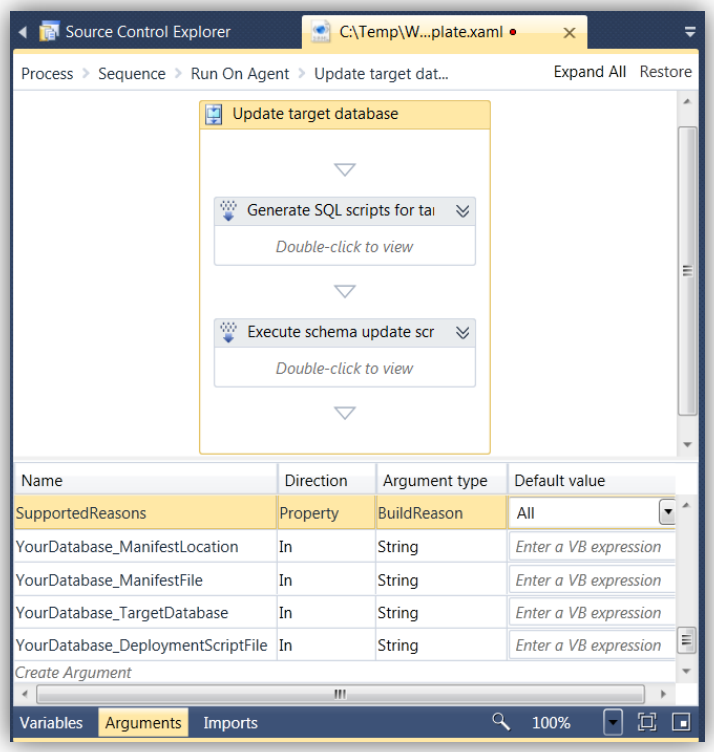


Figure 142 – Calling VSDBCMD and SQLCMD using InvokeProcess

XAML code example:

```
    <Sequence DisplayName="Update target database">
      <mtbwa:InvokeProcess Arguments="[&quot;/a:Deploy
/p:SqlCommandVariablesFile=&quot;&quot;&quot; + YourDatabase ManifestLocation +
&quot;\Database_Deployment.sqlcmdvars&quot;&quot; /DeployToDataBase-
/Connectionstring:&quot;&quot;Data Source=.;Integrated
Security=True;Pooling=False&quot;&quot; /dsp:Sql /ManifestFile:&quot;&quot;&quot; +
YourDatabase_ManifestFile + &quot;&quot;&quot; /p:AlwaysCreateNewDatabase=True
/p:TargetDatabase=&quot; + YourDatabase_TargetDatabase + &quot;
/DeploymentScriptFile:&quot;&quot;&quot; + YourDatabase_DeploymentScriptFile +
&quot;&quot;&quot;&quot;]" DisplayName="Generate SQL scripts for target DB"
FileName="[Environment.GetEnvironmentVariable(&quot;ProgramFiles(x86)&quot;) +
&quot;\Microsoft Visual Studio 10.0\VSTSDB\Deploy\VSDBCMD.exe&quot;]">
        <mtbwa:InvokeProcess.ErrorDataReceived>
          <ActivityAction x:TypeArguments="x:String">
            <ActivityAction.Argument>
              <DelegateInArgument x:TypeArguments="x:String" Name="errOutput" />
            </ActivityAction.Argument>
          </ActivityAction>
        </mtbwa:InvokeProcess.ErrorDataReceived>
        <mtbwa:InvokeProcess.OutputDataReceived>
          <ActivityAction x:TypeArguments="x:String">
            <ActivityAction.Argument>
              <DelegateInArgument x:TypeArguments="x:String" Name="stdOutput" />
            </ActivityAction.Argument>
          </ActivityAction>
        </mtbwa:InvokeProcess.OutputDataReceived>
      </mtbwa:InvokeProcess>
      <mtbwa:InvokeProcess Arguments="[&quot;-S. -E -i &quot;&quot;&quot; +
YourDatabase_ManifestLocation + &quot;\&quot; + YourDatabase_TargetDatabase +
&quot;.sql&quot;&quot; -V 11 -f 65001&quot;]" DisplayName="Execute schema update scripts"
FileName="SQLCMD.exe">
```

```
        <mtbwa:InvokeProcess.ErrorDataReceived>
          <ActivityAction x:TypeArguments="x:String">
            <ActivityAction.Argument>
              <DelegateInArgument x:TypeArguments="x:String" Name="errOutput" />
            </ActivityAction.Argument>
          </ActivityAction>
        </mtbwa:InvokeProcess.ErrorDataReceived>
        <mtbwa:InvokeProcess.OutputDataReceived>
          <ActivityAction x:TypeArguments="x:String">
            <ActivityAction.Argument>
              <DelegateInArgument x:TypeArguments="x:String" Name="stdOutput" />
            </ActivityAction.Argument>
          </ActivityAction>
        </mtbwa:InvokeProcess.OutputDataReceived>
      </mtbwa:InvokeProcess>
    </Sequence>
```

In case you are using the Upgrade template, add a target to your TFSBuild.proj with similar MSBuild code:

```
    <Exec Command='"$(VSEnvVar)..\..\VSTSDB\Deploy\VSDBCMD.exe" /a:Deploy
    /p:SqlCommandVariablesFile="%(YourDatabaseVars.ManifestLocation)\Database_Deployment.sqlcm
    dvars" /DeployToDataBase- /Connectionstring:"Data Source=.;Integrated
    Security=True;Pooling=False" /dsp:Sql /ManifestFile:"%(YourDatabaseVars.ManifestFile)"
    /p:AlwaysCreateNewDatabase=True /p:TargetDatabase=%( YourDatabaseVars.TargetDatabase)
    /DeploymentScriptFile:"%( YourDatabaseVars.DeploymentScriptFile)"'/>
```

Follow this with:

```
    <Exec Command='sqlcmd.exe  -S. -E -i "%( YourDatabaseVars.ManifestLocation)\%(
    YourDatabaseVars.TargetDatabase).sql" -V 11 -f 65001'"/>
```

Note: All above fragments assume that the account running the build has administrative privileges on the target SQL Server instance.

RECOMMENDATION

Make sure that you test and validate the deployment of your database frequently, and that you measure and record the time needed for the process steps 3 (Deploy the database) and 4 (Run sanity checks on the database to make sure the database is consistent) in the process on page 141. Assuming that you have almost, if not the same database content as in the production system, and that your production system might perform even better than your test deployment system you can get a good understanding of how much time is needed to do a schema version upgrade.

Build, Deployment and Testing

If you have already set up your build in Team Foundation Build and created an automated deployment to your build machine, you have already gone half way.

Doing additional consistency checks to check the stability of your build and the validity of your deployment is very important to having a consistent, reusable, and reproducible baseline. Because other solutions typically consume the database, we recommend you develop isolated tests that validate a unit of work, such as a stored procedure, using the database unit tests.

If you are not familiar with writing database unit tests, please refer to the **Visual Studio Database Projects Guidance** document, chapter *"Database Testing and Deployment Verification".*

Related chapters:

- The chapter *"Data Generation and Reference Databases"* guides you through options of populating your database with test data, needed for meaningful, reliable and consistent testing.
- The chapter *"How to create a team build"* guides you through the options of integrating your database unit tests in a Team Foundation Build.

NOTE

You should consider running your UnitTests separately from other UnitTests. Although the Visual Studio Database projects guidance states that tests can run in a transactional scope you want to minimize the effects of other UnitTests in the Build with the database builds. This can be done whether you serialize the UnitTests or deploy a separate instance of your database for testing.

Build and deploy a database project to a new database / upgrading an existing database

Scenario Description

In many cases, just building the databases is not enough for validating developed source code. Although all the artifacts might be consistent, you might have additional pre/ post-deployment scripts that run before or after the build. If these are not built to be idempotent or if you simply assume a state that is not present at the target server, the deployment will fail. As mentioned in the previous section, it is always advisable testing to deploy the database from scratch, thus rebuilding all objects. In addition, you should also deploy the database schema to an existing schema, upgrading the existing schema to the new schema. The scenario will build the database and deploy to a fresh new database as well as upgrade an existing database in order to support an upgrade path.

Supported targets

The supported targets for database projects are:

- Build (Compile the projects if any changes occurred)
- Rebuild (Compile the projects even if no changes occurred)
- Deploy (Deploy the existing build artifacts to a database server / script output)
- Clean (Remove created deployment artifacts)

We will concentrate on the Build and deployment, because they are the heart of an automated build and deployment.

Build

A build is not version agnostic and produces common artifacts:

- **. dbschema** file (containing the database schema model)
- **.deploymanifest** (containing information and how to deploy to the database and which references to include)
- **.sqldeployment** (Setting for the properties for the deployment)
- **.sqlpermission**s (XML File with definition of the to be applied permissions)
- **.sqlsettings** (Common database settings)
- **.sqlcmdvars** (Runtime values that can be used to replace placeholders in your script)
- **Script.PreDeployment.sql** / **Script.PostDeployment.sql** that contain SQL statement to be executed before / after the deployment

Refer to MSDN article "Property Files in Database and Server Projects"[124] for more detailed information about the preceding files.

Although the other files can make the deployment easier by using predefined values and encapsulating environmental differences, the most important file is the .dbschema file that contains the schema of all the objects within the database. Building the project file within Team Foundation Build is not any different from building other project types.

Deploy

Deploying a database has two flavors:

- Generate a script to deploy the database
- Generate a script to deploy the database and deploy the script to the assigned TargetConnection

Configure these settings through the *DeployToScript* and *DeployToDatabase* properties; you might want to set that to only create a script. You can create the script, but run it manually to assure that you will capture all the runtime information or possible error information on the fly.

Similar to how any other Team Foundation Build or MSBuild process works, you can pass in the properties that need to be set and are typically referenced in the files mentioned above that are part of the output in the build process. As you usually do not want to define all the properties manually, you can take advantage of the Build flavors of the build process. Using a special flavor will allow you to decide which properties to use during a deployment. Although you can overwrite the properties from the referenced files at any time, the hierarchy of the files is as follows:

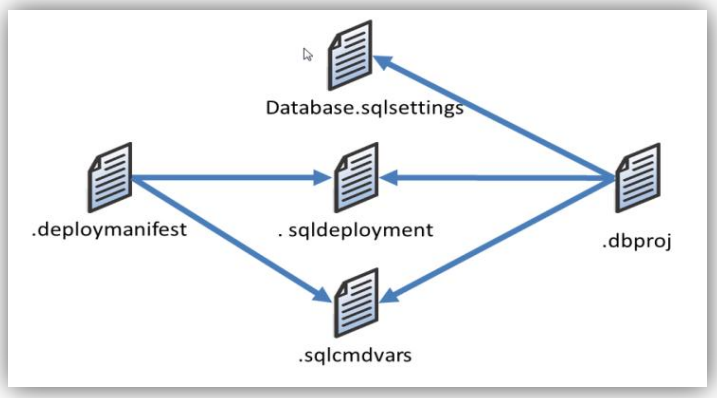


Figure 143 – Deployment Script

Deployment configuration files

The preceding illustration shows the dependencies between the different deployment files. Although you can use a .deploymanifest that references the .sqldeployment file and the .sqlcmdvars, you can also overwrite references of the files during the deployment.

.sqlcmdvars configuration file

This file will be used to substitute placeholders (SQL command variables) in your script during build / deployment time. This file can be referenced from the .deploymanifest file, or directly from the VSDBCMD command line itself to overwrite a referenced file. This is often done and is useful if you have different environments and/or customer dependent runtime values, which need to be reflected in your individual deployment. Referencing the .deploymanifest from VSDBCMD can be done using the /manifest:<PathToManifest> switch, e.g. `/manifest:c:\AdventureWorks\AdventureWorks.deploymanifest`. If you want to overwrite referenced .sqlcmdvars files mentioned in the .sqldeploymanifest file, you can do this by using the

[124] http://msdn.microsoft.com/en-us/library/dd193289(VS.100).aspx

`/p:SqlCommandVariablesFile=<PathToSQLCmdvarsfile>` switch, e.g. `/p:SqlCommandVariablesFile=c:\AdventureWorks\Properties\Database.sqlcmdvars`. Using this switch, VSDBCMD will ignore the .sqlcmdvars file referenced in the .deploymanifest file.

RECOMMENDATION

If you only want to overwrite single SQL Command variables with specific values, you can also use the switch **/p:** *SqlCmdVariableName* **=** *<Value>. This makes it easier to reuse predefined .sqlcmdvars files with infrequently changing values and as well as changing the deployment parameters individually.*

.sqldeployment configuration file

The same overwrite mechanism applies to the .sqldeployment file, where you need to use the /p:DeploymentConfigurationFile=<filename> switch. Although it might be already referenced in the .deploymanifest information, you can force VSDBCMD to use another file instead. One of the practical uses of doing this is when you have different settings (depending on the environment) for the .sqldeployment file. You might want to deploy a database in one environment with dropping the database first and creating it from scratch. In other environments, you might want to upgrade the database instead by only applying needed schema changes to the database. For that you would need to change the "AlwaysRecreateDatabase" to either false (=upgrade) or true (=Drop and create database).

NOTE

For all possible command line parameters of VSDBCMD, visit the MSDN article "Command-Line Reference for VSDBCMD.EXE (Deployment and Schema Import)"[125]

Manual Deployment

Prerequisites for the manual deployment

Because build deployments are typically disconnected from the target environments, you need to take the output from the build, copy it over to the target environment, and start the deployment engine at the target. As mentioned before, the deployment engine is VSDBCMD.EXE, which is a small command-line utility that is able to create database scripts from the .dbschema file and optionally take the other build artifacts of the database deployment into account. While it runs without having Visual Studio installed, it has some prerequisites for running it.

RECOMMENDATION

You don´t need to install the prerequisites on the database machine you want to deploy to. You can take any other machine, which is able to connect to the database system. This makes it easier to maintain a clean database environment and avoid maintenance windows for installing prerequisites.

For the Visual Studio 2010 edition of the database projects, the following prerequisites must be installed. (Refer to the MSDN Article **How to: Prepare a Database for Deployment From a Command Prompt by Using VSDBCMD.EXE**[126].)

[125] http://msdn.microsoft.com/en-us/library/dd193283.aspx

[126] http://msdn.microsoft.com/en-us/library/dd193258.aspx

- The directory Deploy located under %ProgramFiles%\Microsoft Visual Studio 10.0\VSTSDB\Deploy or %ProgramFiles(x86)%\Microsoft Visual Studio 10.0\VSTSDB\Deploy from any machine having Visual Studio 2010 installed.
- Microsoft.SqlServer.BatchParser.dll that matches your version of SQL Server, which is included in different standalone packages of SQL Client. See **How to: Prepare a Database for Deployment From a Command Prompt by Using VSDBCMD.EXE** for more information.
- Microsoft .Net Framework 4
- SQL Server Compact Edition 3.5 SP1 (if you are deploying from a 64bit system, you will also need to have the 64bit version installed)

NOTE

In many cases, ISVs do not have direct control over the deployed database at a customer site, and they need a generic way of deploying the sources without knowing which version a customer has actually installed. Visual Studio Database projects and the VSDBCMD deployment engine help you to not be version agnostic. Using an MSI installer and installing the previously mentioned prerequisites prior to deploying the database will give you the option of a minimum-to-zero effort touch installation. A sample for installing database projects using a Windows Installer XML (WiX) is shown in the Visual Studio Database Project Guidance document in the chapter *"Automating Database deployments using VSDBCMD and WiX".*

The process of manual deployment

Manual deployment simply means taking the build artifacts from the build output and applying them using the VSDBCMD.EXE utility. A sample call for this could be:

```
VSDBCMD.exe /a:Deploy /cs:"Data Source=MyServerName;Integrated Security=True;Pooling=False"
/dsp:Sql /model:c:\AdventureWorks\AdventureWorks.dbschema /p:TargetDatabase= AdventureWorks
/p:SqlCommandVariablesFile= c:\AdventureWorks\Properties\Database.sqlcmdvars /manifest:
c:\AdventureWorks\AdventureWorks.deploymanifest /script: c:\ AdventureWorks\AdventureWorks.sql
```

Files, other than the .dbschema file, can be referenced from the command line call as well as being referenced from the .deploymentmanifest file.

NOTE

Having these deployment artifacts, the DBA or in general the person in charge for the deployment has full control over the actions against the target database. Using the switch /DeployToDatabase+ (or the short form /dd+) one can either deploy directly to the target database (using the + sign as showed above) or create a .sql schema motion script file, which will make sure that all appropriate actions are taken on the target database schema to move to the new version.

For more information about the deployment settings for a database, the difference when upgrading a database vs. creating a .sql file for applying the database schema manually, and the differences when deploying the database directly with the command you can refer to the Visual Studio Database Project Guidance[127]

[127] http://vsdatabaseguide.codeplex.com

RECOMMENDATION

In some cases and situations, you expect a certain database schema version to exist, such as when you tested the upgrade thoroughly to make sure to meet certain SLAs and maintenance windows. Changes to that expected database version might cause a severe change in the timeline and might cause you to roll back an upgrade. For that reason, you should make sure to compare the actual database to the expected database schema version and see if changes exist. You can find more information in the Visual Studio Database Project Guidance under the topic "Finding Model drifts."

Create and configure an IIS Web Application

Before you can start deploying your Web Application or Web Service, you need to have a configured IIS Web Application in place. Often this is done as a manual step because, after it is set up and configured, it rarely changes. Unfortunately, the details on what to configure is easy to forget and therefore it makes sense to automate this step as well.

A flexible solution to this problem is to use a Windows PowerShell script. IIS exposes a PowerShell commandlet, which makes is quite simple to automate the creation and configuration of any IIS Web Application. The following sample script will create a new application pool and then create an IIS application bound to the application pool. We provide a few settings for the identity of the web application as well as the location of the virtual directory containing the application files.

Sample PowerShell script to create and configure an IIS application

```
# Settings
$newApplication = "service_name"
$poolUserName = "user"
$poolPassword = "password"
$newVDirName = "W3SVC/1/ROOT/" + $newApplication
$newVDirPath = "C:\" + $newApplication
$newPoolName = $newApplication + "Pool"
# Create Application Pool
$appPoolSettings = [wmiclass] "root\MicrosoftIISv2:IISApplicationPoolSetting"
$newPool = $appPoolSettings.CreateInstance()
$newPool.Name = "W3SVC/AppPools/" + $newPoolName
$newPool.PeriodicRestartTime = 0
$newPool.IdleTimeout = 0
$newPool.MaxProcesses = 2
$newPool.WAMUsername = $poolUserName
$newPool.WAMUserPass = $poolPassword
$newPool.AppPoolIdentityType = 3
$newPool.Put()
# Create the virtual directory
mkdir $newVDirPath
$virtualDirSettings = [wmiclass] "root\MicrosoftIISv2:IIsWebVirtualDirSetting"
$newVDir = $virtualDirSettings.CreateInstance()
$newVDir.Name = $newVDirName
$newVDir.Path = $newVDirPath
$newVDir.EnableDefaultDoc = $False
$newVDir.Put()
# Create the application on the virtual directory
```



```
$vdir = Get-WmiObject -namespace "root\MicrosoftIISv2" -class "IISWebVirtualDir" -filter "Name = '$newVDirName'"
$vdir.AppCreate3(2, $newPoolName)
# Updated the Friendly Name of the application
$newVDir.AppFriendlyName = $newApplication
$newVDir.Put()
```

To bind this script to a Team Foundation Build process you will have to customize your build process template and add an activity to execute the PowerShell script. The simplest way to achieve this is to use the InvokeProcess activity. To make the extension easy to use you can also add Workflow arguments to the build process template so that the arguments needed in the script can be set in the build definition.

For more information, refer to:

- Getting Started with PowerShell[128]
- Web Administration Provider for Windows PowerShell[129]

ASP.NET Web Application in Integration and QA Environments

Scenario Description

A build often needs to deploy the same version of an application to two or more different environments. The following guidance discusses the integration between Team Foundation Build and the Web Deployment tool. Then proceeds to describes two common ways to perform deployments of Web Deploy packages to multiple environments. The first approach uses Web.Config Transformations with the second approach using parameter files for each environment. Each approach has particular advantages and are both effective.

Integration between Team Foundation Build and the Web Deployment Tool

The Web Deployment Tool, or Web Deploy, simplifies a deployment to IIS Web Server. This tool can be used from the Visual Studio IDE in the Publish option or using the command line. Here, we explore how the Web Deployment Tool can be integrated with Team Foundation Build providing flexibility for deploying a web application.

In most cases, the Team Foundation Build agent will not be the same machine where you will publish your application. Considering that scenario, the Web Deployment Tool needs to provide a remote service that the build agent can connect to and publish the application.

Additional references:

- Overview of Web Deploy[130]
- Installing Web Deploy[131]
- Web Deploy Command Line Reference[132]
- Visual Studio 2010 Web Deployment[133]

When you publish a web application from Visual Studio using the integration with Web Deploy, there are some options available (Figure 144). The same options, and more, are available when you deploy integrating Team Foundation Build and Web Deploy. However, instead of being available in the user interface, they are passed as parameters to the MSBuild.

[128] http://technet.microsoft.com/en-us/library/ee308287(WS.10).aspx

[129] http://technet.microsoft.com/en-us/library/ee909471(WS.10).aspx

[130] http://learn.iis.net/page.aspx/426/overview-of-web-deploy

[131] http://learn.iis.net/page.aspx/421/installing-web-deploy

[132] http://technet.microsoft.com/en-us/library/dd568991(WS.10).aspx

[133] http://weblogs.asp.net/scottgu/archive/2010/07/29/vs-2010-web-deployment.aspx

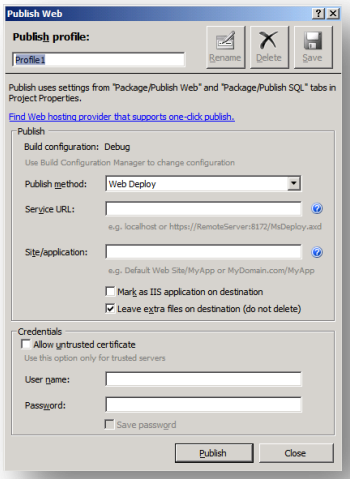


Figure 144 – Web Deploy Publish Setup Dialog

Table 31 shows the basic parameters that must be passed to the Team Foundation Build to deploy an application using Web Deploy.

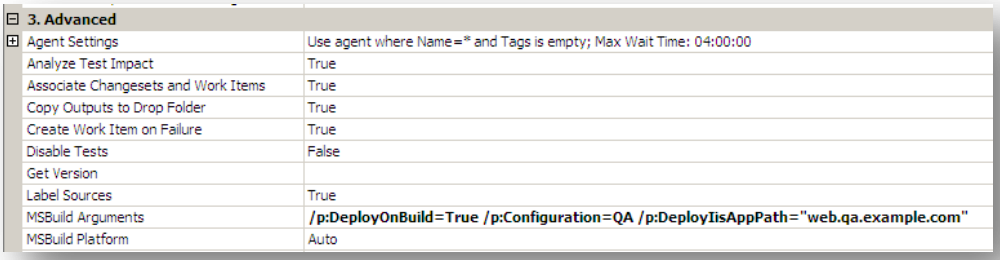
Name	Description
MSDeployServiceUrl	This is the address of the remote agent service installed in the Web Deploy. Using this URL the Web Deploy will be contacted to publish the application.
AllowUntrustedCertificate	By setting this parameter to true, untrusted certificates will be allowed to use the deployment service.
UserName	The Username used to contact the remote agent service.
Password	The Password that used to contact the remote agent service.
DeployIISAppPath	This is the application path where the publishing will occur.
DeployOnBuild	This parameter specifies that, after a successful build, the target for deploying will be called.
DeployTarget	This parameter indicates the target that will be called in order to publish the application. This value must be set to MsDeployPublish in order to publish.
TransformWebConfigEnabled	This parameter must be set to true to enable WebConfig transformation if it exists to the active build configuration.
Configuration	Specifies the solution configuration for the build and what web config transformation to use.
MsDeployPublishMethod	This parameter specifics the method for MsDeployUse **InProc** if the server is local or **RemoteAgent** if deploying to a remote server.

Table 31 – Web Deploy Parameters

There are many more parameters available. If you need to access more parameters, you should look at the Microsoft.Web.Publishing.targets file. Refer to http://weblogs.asp.net/scottgu/archive/2010/07/29/vs-2010-web-deployment.aspx for more information.

If you understood the parameters shown previously, it will be simple to publish a web application using the Team Foundation Build in a similar fashion to publishing from Visual Studio. The steps to perform are:

1. First, we need to create a build definition that will build the application. Create a new build using the Default Template and choose your web application solution as the item to build.
2. Next, we can pass in the Web Deploy parameters to Team Foundation Build through the MSBuild Arguments property.



3. Advanced	
Agent Settings	Use agent where Name=* and Tags is empty; Max Wait Time: 04:00:00
Analyze Test Impact	True
Associate Changesets and Work Items	True
Copy Outputs to Drop Folder	True
Create Work Item on Failure	True
Disable Tests	False
Get Version	
Label Sources	True
MSBuild Arguments	**/p:DeployOnBuild=True /p:Configuration=QA /p:DeployIisAppPath="web.qa.example.com"**
MSBuild Platform	Auto

Figure 145 – Configure MSBuild Arguments

   a. Note that not all of the MSBuild Arguments are visible. The following is the complete list to support deploying the application.
      i. /p:DeployOnBuild=True
      ii. /p:Configuration=QA
      iii. /p:DeployIisAppPath=”web.qa.contoso.com”
      iv. /p:MsDeployServiceUrl=”https://web01:8172/MsDeploy.axd”
      v. /p:MSDeployPublishMethod=RemoteAgent
      vi. /p:DeployTarget=MsDeployPublish
3. Finally, queue a new build and the application will deploy to the QA web server

Web.Config Transformations

Now that we have built and deployed the web application to a single environment, how do we manage the configurations, builds, and deployments across all of the environments? Web Deploy includes a feature called Web.Config Transformations to simplify the configuration differences between environments (see Vishal Joshi’s blog, Web Deployment: Web.Config Transformation[134], for more on Web.Config Transformations). In the past, managing these configurations has required manual and error prone steps for changing things such as connection strings and service references. The Web.Config Transformations typically use a simple transform for replacing these values.

The Web.Config Transformations are driven by the configurations in the web application. Create Configurations for all of the environments for your application such as Staging, QA, Test, etc. To keep things consistent with other projects, usually have Debug = Development and Release = Production. If you prefer to create Development and Production configurations, remove the debug and release to avoid confusion.

[134] http://vishaljoshi.blogspot.com/2009/03/web-deployment-webconfig-transformation_23.html

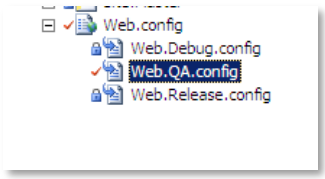


Figure 146 – Web.Config Transformation Files

The following example shows a QA transform that replaces the default PrimaryDatabase connection string with the QA environment's connection.

```
<connectionStrings>
  <add name="PrimaryDatabase"
    connectionString="Data Source=SQL01;Initial Catalog=AppDb;Integrated Security=True"
    xdt:Transform="SetAttributes" xdt:Locator="Match(name)"/>
</connectionStrings>
```

Figure 147 – Sample transformation

Deploying in Multiple Environments using Web.Config Transformation Files

The web application now contains the configurations for each environment. We can use Web Deploy to deploy to multiple environments. In the Team Foundation Build example above, the application built and published on fly. There was no staging package built that could be tested and used for the other environments. A sound Software Configuration Management (SCM) process would consist of managing these packages as releases and ensuring that same code and deployment process across the environments. Web Deploy provides the features to do this. The following steps are a more typical build, stage, and deployment process:

1. Team Foundation Build compiles and creates a Web Deploy package for each environment that includes that environment's web.config file differences. Because a Web Deploy package is built for a specific environment, the build process must build a package for each configuration.
2. These packages are copied to a versioned Deployment Packages / Releases folder where they are staged for future deployments.
3. A scripted deployment process or a separate Team Foundation Build definition takes the package and deploys it to the target environment.

To have Web Deploy create a deployment package from Team Foundation Build, again pass the values to through the MSBuild arguments. In this instance the key property is DeployTarget=Package. This creates the package that can be copied to the versioned release folder.

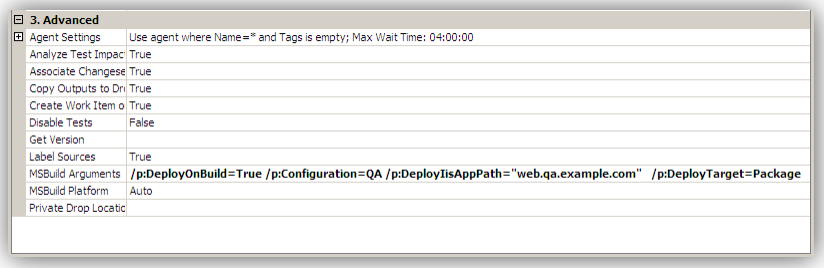


Figure 148 – Specifying MSBuild Arguments

With the deploy package created, it can be deployed to the QA environment using a PowerShell script. See the section on Production Deployments for specifics about the PowerShell script and using tempAgent to avoid having to install the Web Deploy agent on the QA servers.

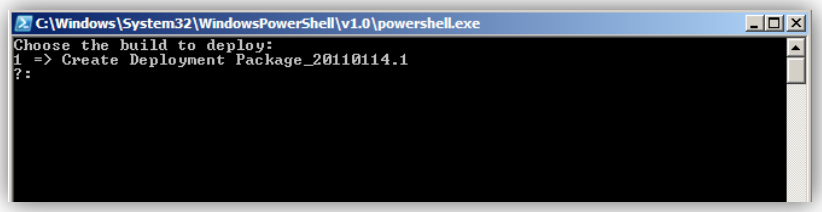


Figure 149 – Deploying using PowerShell

Deploying in Multiple Environments using Set Parameters files

The second approach for using Web Deploy to deploy across multiple environments is to use set parameters files. As the first approach explained, the Web Deploy package is compiled for a particular environment. However, if you open the Web.Config file inside the Web Deploy package, it includes a parameterized value for the connection string and IIS application name that is not set until the package is deployed. Notice the connection string setting in Figure 146 below. This doesn't have the actual environment's value but instead has the parameter token. This allows an environment specific SetParameters.xml files to be used to set the values as shown below.

```
<configuration>
  <connectionStrings>
    <add name="CoolMvcDb"
         connectionString="$(ReplacableToken_CoolMvcDb-Web.config Connection String_0)"
         providerName="System.Data.SqlClient" />
  </connectionStrings>

  <appSettings>
    <add key="webpages:Version" value="1.0.0.0" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
    <add key="CustomSetting" value="5"/>
  </appSettings>
```

Figure 146 – Parametertized Settings in Web.Config

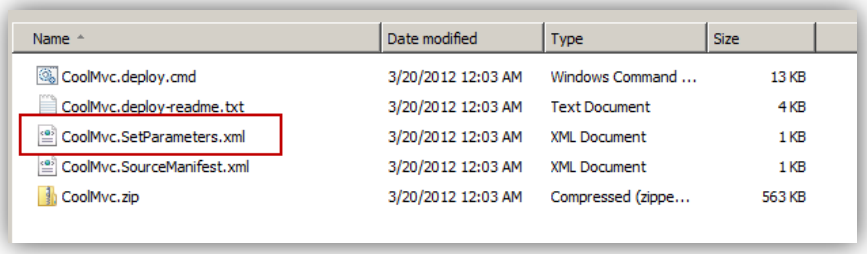


Figure 147 – Default Set Parameters created

This feature provides the primary benefit of this approach and that it allows you to compile the application once and reuse the package for multiple environments. This provides the most flexibility while providing the highest guarantee that the same code is deployed to all of your environments. Figure 148 shows that the connection string token will be replaced by this value in the SetParameters.xml file.

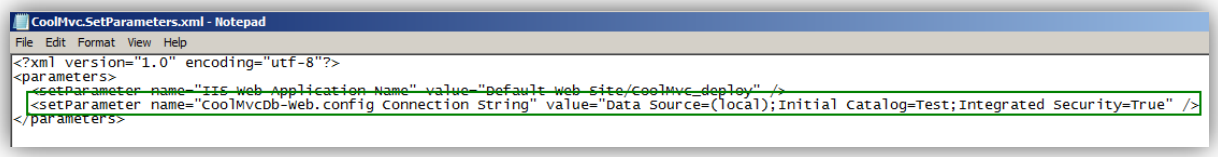


Figure 148 – SetParameters.xml File

The set parameters approach provides automatic parameterized values for the connection string and IIS application however, often there are additional settings that are unique per environment. Just as the Web.Config transformations provide a mechanism for setting unique values for each environment, you can create additional settings to be parameterized. To parameterize a setting the setting has to be explicity specified in a parameters.xml file. The parameters.xml file must be manually created in the root of the web application. This file contains the XSL transformation on what to set in the Web.Config file.

```
<?xml version="1.0" encoding="utf-8" ?>
<parameters>
  <parameter name="CustomSetting" description="Set Custom Setting" defaultValue="1" tags="">
    <parameterEntry kind="XmlFile" scope="\\web.config$" match="/configuration/appSettings/add[@key='CustomSetting']/@value" />
  </parameter>
</parameters>
```

Figure 149 – Custom Parameters in the Parameters.xml FileThe Parameters.xml file above shows adding an appSetting called CustomSetting as outlined in red in Figure 146 to point out that it wasn't automatically parameterized. The parameter element includes several attributes to specify what and where the setting is located that is going to be parametertized. The key attributes include:

- **kind** – specifies what kind of resource where the parameter is to be applied. This is XmlFile since Web.Config file is XML.
- **scope** – is a regular expression that specifies the path to the file that is going to be changed.
- **match** – specifies the XPATH expression that points to the value that is going to be parameterized.

See How to: Use Parameters to Configure Deployment Settings When a Package is Installed[135] for more information on using the parameters in the Parameters.xml file.

After repackaging the application, the SetParameters.xml file now includes the custom setting that was created with the Parameters.xml file.

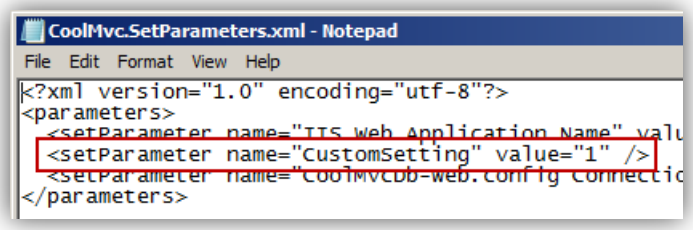


Figure 150 – SetParameters File with custom setting

The Parameters.xml file does not need to be packaged into the XML file so you can exlude this from the packaging by setting the build action of the file equal to None. The next step for this approach is to create the parameters files for each environment. These files should be named with the appropriate environment, stored in source control, and be copied to the drop location with the other Web Deploy package files. In this example, the folder is

[135] http://msdn.microsoft.com/en-us/library/ff398068.aspx

created under the solution called EnvironmentParameters that contains the SetParameters files for each environment.

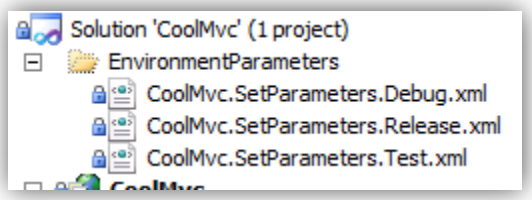


Figure 151 – Set Parameters files for each environment in source control

To build and deploy this from Team Build, this requires some simple modifications to the default build process template. Add two CopyDirectory activities to copy the Set Parameters files and the PowerShell script to do the deployment. Finally call the PowerShell script to do the deployment. This is shown in the figure below.

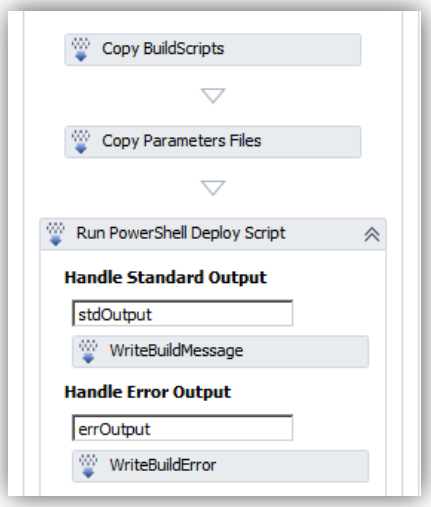


Figure 152 – Build Process Template changes for deploying

The PowerShell script needs to accept the drop location of the build as the parameter and then know the relative path of the Web Deploy package and Set Parameters file. Once these are set, MSDeploy can be called as shown below. The attribute is the setParamFile. This will be set to the full path of the Set Parameters file in the drop location.

```
    cmd.exe /C $("msdeploy.exe -verb:sync -source:package=`"{0}`" -
    dest:auto,computername=`"{1}`",username=`"{2}`",password=`"{3}`",authtype=Basic -
    allowUntrusted -setParamFile:`"{4}`" -retryattempts:20" -f $webdeployPackage,
    $destinationComputer, $user, $pass, $paramsFile )
```


Integrating with NuGet

What is NuGet?

NuGet[136] is an open source package manager that provides a developer with a way to manage assembly references in a project for assemblies that are not within their solution. It is most commonly used to manage externaly used community libraries such a nHibernate or JQuery, although you can also use it manage a team's own internal shared libraries[137].

Installing NuGet on Visual Studio 2012

NuGet installs via a Visual Studio Extension

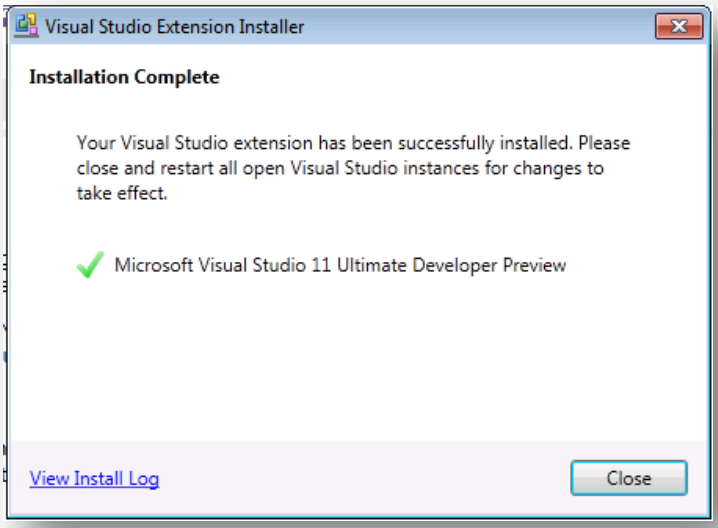


Figure 150 – NuGet extension installer

Using NuGet in the Enterprise

- Setup a Company Repository
- For a single location use a UNC share
- Distributed companies should use an http server
- Start by manually packaging up existing company shared libraries and components and their configurations
- Create meta packages to share better practices
    - A Meta package is a package that does not contain assemblies or code. It only combines a set of dependent packages
        - Testing package could have unit tests, assertions, and a mocking framework
- Define a process for when your NuGet packages are published to the official company repository

Using NuGet in Visual Studio

Once NuGet has been installed into a developer's Visual Studio via the Visual Studio Gallery, references to packages can be made by right-clicking a project and selecting **Manage NuGet Packages**… The NuGet management screen will be shown as in the figure below.

[136] http://visualstudiogallery.msdn.microsoft.com/27077b70-9dad-4c64-adcf-c7cf6bc9970c

[137] http://docs.nuget.org/docs/creating-packages/hosting-your-own-nuget-feeds

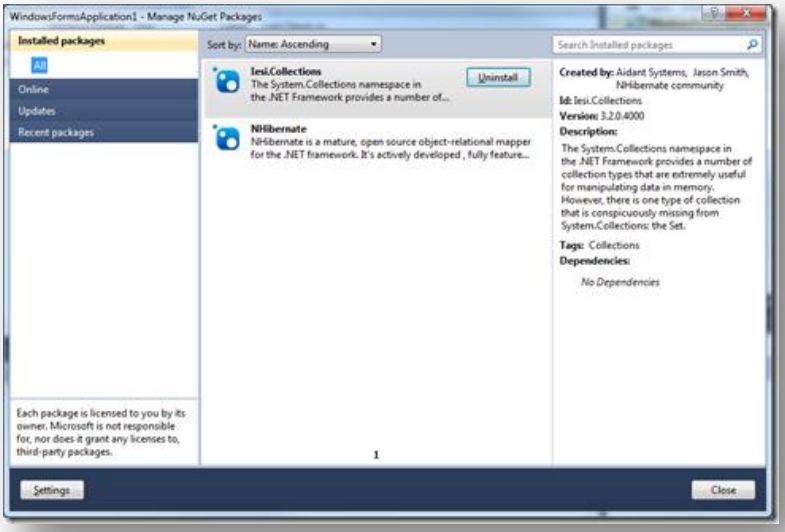


Figure 151 – NuGet management screen

When a package is added to a project via NuGet a **packages.config** file that lists the NuGet dependencies is added to the project. This is in the form

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
   <package id="Iesi.Collections" version="3.2.0.4000" />
   <package id="NHibernate" version="3.2.0.4000" />
</packages>
```

The package is also available when viewing the Installed Packages tab:

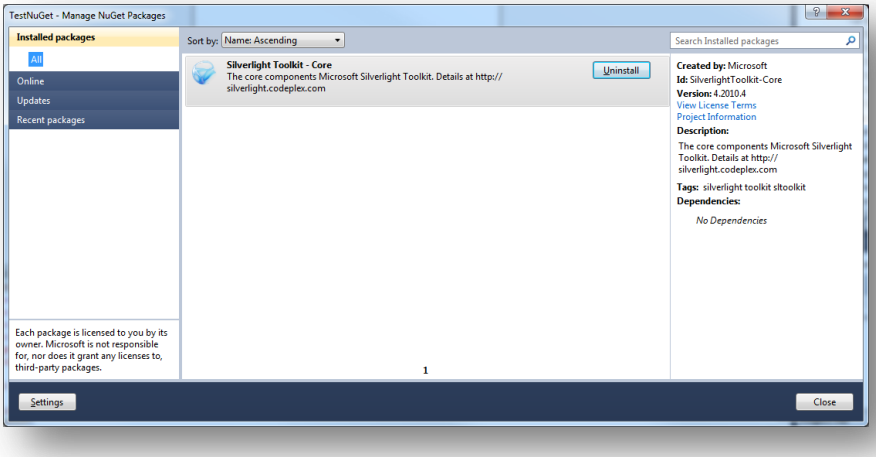


Figure 152 – NuGet installed packages

As this **packages.config** file is part of the Visual Studio project, when the project is put under source control so will this file.

As well as creating the **packages.config** file a local copy of the required assemblies is made. This is done under the solution folder

```
Solution Directory
        Packages – the root of the local cache of assemblies created by NuGet
        Project Directory
```

The packages folder is **NOT** placed under source control by default.

Adding a Package Source Folder to a project

To integrate an internally built NuGet package into a project, simply select a source folder to integrate into the solution.

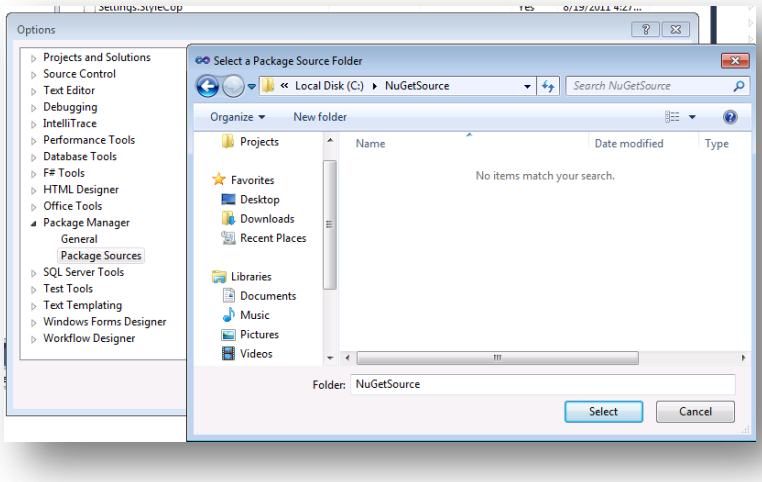


Figure 153 – Select a Package Source folder

As you can see from this screenshot, multiple sources can be made available as package sources:

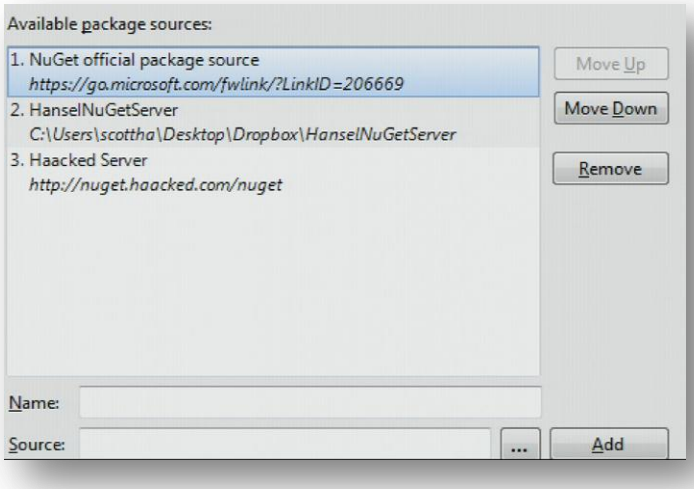


Figure 154 – Available package sources

Using NuGet in a Team Foundation Server Build

ALERT

If a build is created to build a solution with projects using NuGet build errors will be seen in the form

Form1.cs (16): The type or namespace name 'NHibernate' could not be found (are you missing a using directive or an assembly reference?)
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Microsoft.Common.targets (1490): Could not resolve this reference. Could not locate the assembly "Iesi.Collections". Check to make sure the assembly exists on disk. If this reference is required by your code, you may get compilation errors.
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Microsoft.Common.targets (1490): Could not resolve this reference. Could not locate the assembly "NHibernate". Check to make sure the assembly exists on disk. If this reference is required by your code, you may get compilation errors.

This is because though the **packages.config** file is under source control the locally cached assemblies are not. To fix this issue there are two options.

Option1: Add the packages folder to source control manually

To add the missing assemblies to source control in Visual Studio

1. Select Team Explorer > Source Control
2. Navigated to the solution folder.
3. Press the add files button and added the whole **Packages** folder.

NOTE

When you add the whole folder structure the default is to exclude .DLLs (and .EXEs). You need to make sure that the required assemblies are not excluded when you add the folder structure

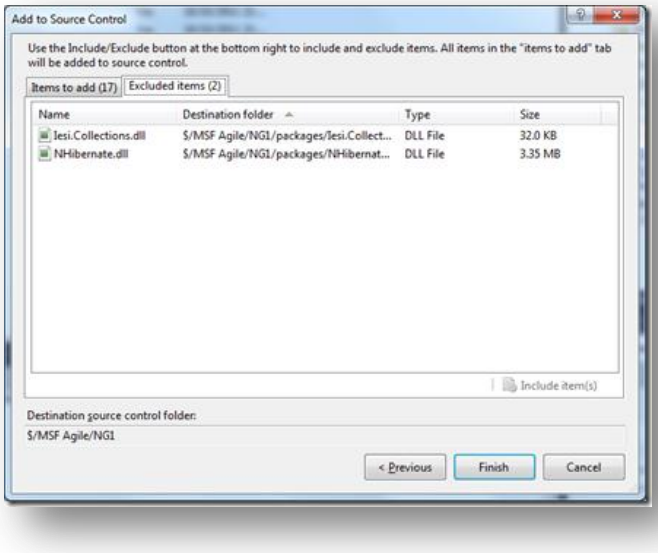


Figure 155 – Add file source control dialog

Once **ALL** the correct files are under source control the build will works as expected.

Option 2: NuGet Package Restore

The other option is to enable the NuGet Package restore feature in NuGet 1.6 and later. For more info, see Using NuGet without committing packages to source control.[138]

Future of NuGet in Team Foundation Server 2012

Currently, the Team Foundation Build 2012 environment is not aware of NuGet, though Martin Woodward from the product team has released a screen shot of a template with NuGet support built in.

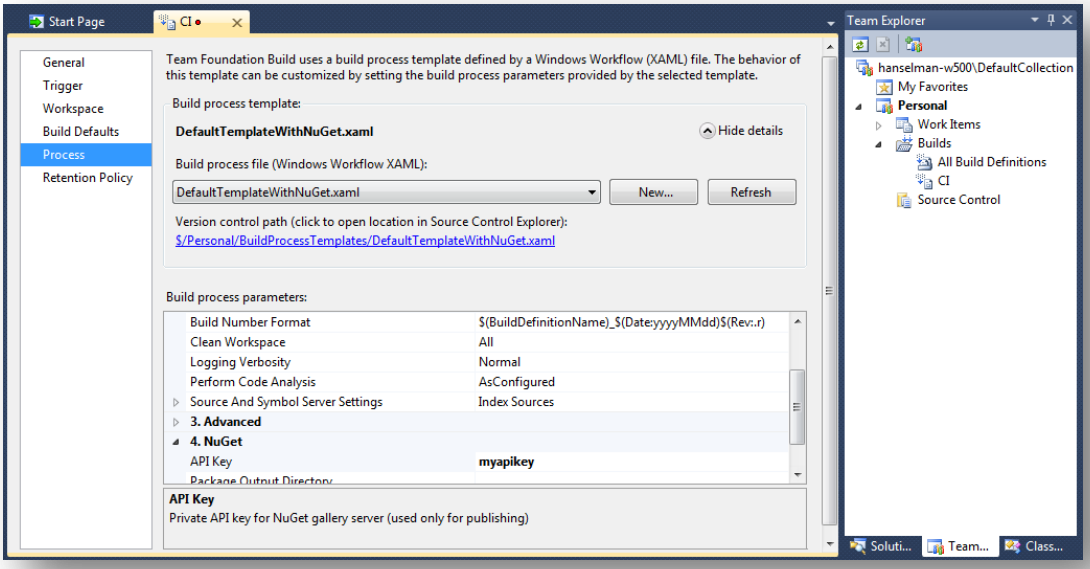


Figure 156 – NuGet support

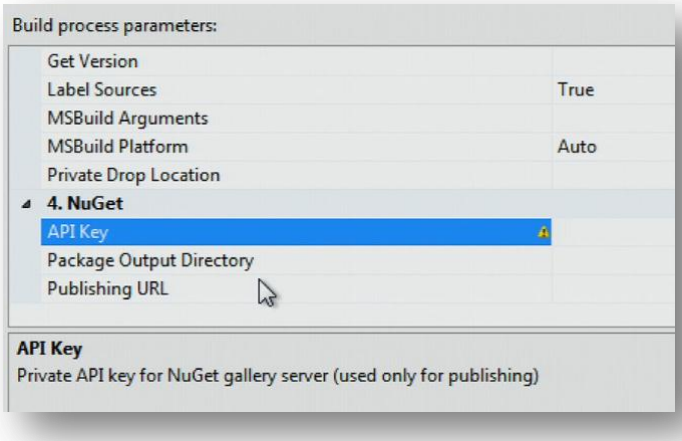


Figure 157 – Build process parameters

[138] http://docs.nuget.org/docs/Workflows/Using-NuGet-without-committing-packages

Link to a Scott Hanselman blog post concerning his presentation from TechEd 2011: http://www.hanselman.com/blog/NuGetForTheEnterpriseNuGetInAContinuousIntegrationAutomatedBuildSystem.aspx. Scott made a presentation about how to use NuGet for the Enterprise: NuGet in a Continuous Integration Automated Build System. The video is located here: http://channel9.msdn.com/Events/TechEd/NorthAmerica/2011/DEV338.

Integrating with Windows Azure

Personas

The personas most likely interested in the Windows Azure build and deploy customization are Abu the Build Master and Doris the Developer when they are associated with Windows Azure cloud application development projects.

Summary

This section describes the process of building Windows Azure applications, pushing them into the cloud and then deploying them to a Windows Azure's staging or production slot. It demonstrates how to create and setup this process via an automated Team Foundation Build template and build machine configuration.

Keep in mind the process described below is simply one way to perform the action of build and deploy of Windows Azure applications. Also, Windows Azure allows the deployment of applications written in your choice of languages (e.g. .NET, node.js, java, php, python, C++, etc.) and the common thread with all of them is the creation of the .cspkg (package) and .cscfg (configuration) files; once those files are created, the deployment to the cloud is consistent. There may be differences in the way your IDE and environment present the process but ultimately, the process itself is the same. This guidance will show you how to use the automated build capability of Team Foundation Server to deploy those Windows Azure applications – although the concepts can be translated to any environment.

Requirements for Using this Guidance

- **Windows Azure SDK and Development Tools**: There is tooling for several development environments available on the Windows Azure web site. Start here and download/install the appropriate set of tools for your environment.
  - http://www.windowsazure.com/develop/
- **Windows Live Account**: This email account (Hotmail, Gmail, etc.) will be used to identify you and authorize use of the Windows Azure cloud environment i.e. the Windows Azure Management Portal. You can set up your email account as a Windows Live Account as part of the Windows Azure Account setup or go to the signup page: https://signup.live.com/signup.aspx?lic=1
- **Windows Azure Account**: Although with Windows Azure you can run your applications in a locally emulated "Azure compute environment" deployment to the cloud is different than executing locally so you will need an account within the Windows Azure environment.
  - Windows Azure Management Portal: https://windows.azure.com/Default.aspx This is where you manage your Windows Azure services, SQL Azure, Service Bus, Blob Storage, etc.
  - Free Trial Offer: http://www.windowsazure.com Go to the Windows Azure site home page and click "free trial" in the upper right of the screen.
- **Windows Azure Management Certificate**: These are X.509 v3 certificates that are uploaded to Windows Azure. At least one is required for authentication/authorization while the deployment process is working with Windows Azure to get the application up and running in Windows Azure. To create a certificate (as described in this guidance), you will need "makecert.exe". This is a command-line application included with the Windows SDK: http://www.microsoft.com/download/en/details.aspx?id=8279
- **Windows Azure Storage Explorer**: This is a utility application, similar to Windows Explorer that will let you create a storage "container" which is used to temporarily store you deployment files. Several

explorers can be found on the internet – search for "Windows Azure cloud storage explorer". This application will also allow you to review/verify the package files sent up for deployment.

The Windows Azure Build and Deploy Process

The build and deploy process of a Windows Azure application should follow along these steps.

Compile

The Compile and unit test portion of the build is fairly standard across all development environments. For .NET it is performed by an MSBuild process and the output of this step is a compiled (and hopefully, unit tested) application ready to run in Windows Azure (or locally if you wish).

Package

An extra step provided by the Windows Azure SDK is the packaging step. Packaging makes the application consumable by the Windows Azure environment. Here it makes a callout to the CSPack.exe console application to do the work of generating a package file containing the Windows Azure application. This package file ends in ".cspkg" and is actually a "zip" format file containing the entire Windows Azure application file and folder structure minus the configuration files (those ending in ".cscfg"). Your application can have multiple configuration files and during deployment you indicate the one to use in Windows Azure for the environment you are operating in (staging, test or production).

Upload

Once the package has been created it must be transferred to the Windows Azure environment where blob storage is the destination. This upload step is a relatively simple file transfer process to get the package file from the build machine up to the Windows Azure environment. Blob storage is used to temporarily store the package and once stored, it is immediately available to deploy into a staging or production "slot." It is not necessary to upload the configuration file at this time because that step is performed later but keeping both files together as a group is not a bad idea.

Deploy

Next, the actual Deploy process starts. This procedure starts the "spin-up" of the virtual machine(s) (VM) and copies the application files to the VM. The VM can exist in the staging location (slot) or production.

Start

Finally, once your application is deployed, it is not actually in a "running" state so it must be started. The last step in this process starts the application and attempts to makes it available for use. Assuming all of the steps succeeded, you should then be able to access the application.

Although this guidance suggests a defined way of handling all the process steps described above, the actual process you develop and use may look different since there are multiple ways of performing the steps. The point to take away from this is that the processes in general will remain fairly consistent while the way you decide to execute them may vary widely. For example, the process in this guidance was derived from console applications that are called from within a Team Foundation Build Template. You may decide to take advantage of custom workflow activities to perform the steps. You may decide to bundle upload, deploy and start within a PowerShell script; how you do it is very much up to you. This guidance will show you one of the possible ways to educate you on the process and the things to think about as you develop your Windows Azure build/deploy process.

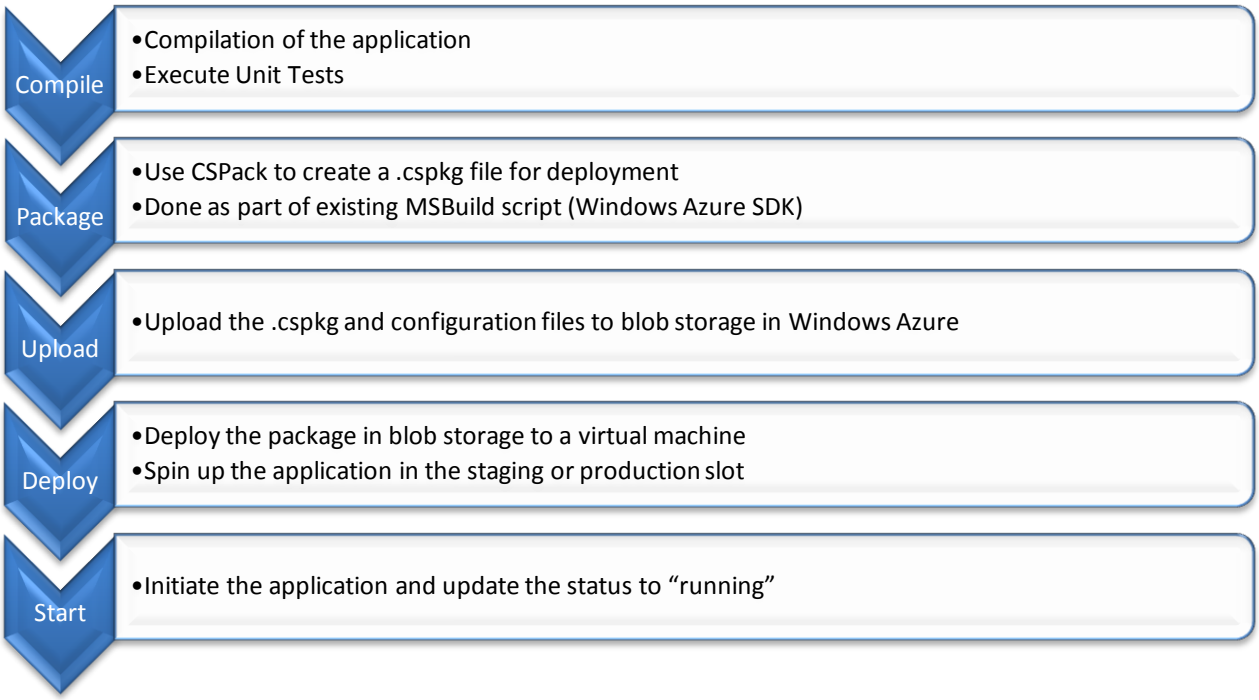


Figure 158: Windows Azure Build / Package / Deploy Process

This guidance will use the following to perform the outlined steps:

- Setup – Windows Azure and the build machines must be ready for the build/deploy process
- Compile – standard Team Foundation Build process using MSBuild
- Package – CSPack as provided by the Windows Azure SDK (already part of the Windows Azure SDK)
- Upload – ImportExportBlob application found on the Windows Azure Code Sample site
- Deploy – CSManage application found on the Windows Azure Code Sample site
- Start – CSManage application found on the Windows Azure Code Sample site

The process is managed by and additional functionality is provided by a custom Team Foundation Build workflow template.

RECOMMENDATION

Automated deploy of any application type (including Windows Azure applications) should not be done directly to production. It makes total sense to automate deployment into a test/staging environment so that a final suite of tests can be performed to validate the suitability of the application to exist in production. Automated tests are a great thing to have in your arsenal but they should not be relied upon solely as the decision maker of whether to deploy to production. Instead, Windows Azure has an inherent staging location from which your application can be deployed to and tested.

Set Up Windows Azure

If you do not already have a Windows Azure account set up, you can do this through a fairly easy online process. If you want to test the environment first, you have the option of setting up a trial account. The trial account gives you a full 90 days to work with and play around with Windows Azure including compute time (that is, time that your test virtual machines are up and running), SQL Azure database space, blob storage space, storage transactions and in/out bandwidth for testing your application in the real environment. For more details on this option, go to http://www.windowsazure.com/.

Install the Windows Azure SDK on the Build Machine

The Windows Azure SDK must exist on the build machine to provide the necessary environment including build scripts and assemblies. Go to http://www.windowsazure.com/develop/ to download/install the SDK on the build machine.

Setup Steps for Windows Azure Automated Build and Deploy

Once you have a Windows Azure online account, there are a few steps to ensure that you will be able to upload and deploy to Windows Azure in an automated manner. This assumes that you have already created a Windows Azure account.

1. Record your "Subscription ID" from the Windows Azure portal
2. Create a blob storage account and container
3. Create and deploy a Management Certificate
4. Install the Windows Azure SDK on the Build Machine

Record Your "Subscription ID"

The subscription ID is a globally unique ID (GUID) that identifies your particular Windows Azure environment. It can be found within the management portal (http://windows.azure.com/) after you log in. To find it, click the **Hosted Services, Storage Accounts and CDN** button in the lower left of the portal. Then click the **Hosted Services** tab i. On the right side of the scree, you will see a list of properties. Toward the bottom you will see the **Subscription ID**. If you right-click this property, you can copy the value. Store it in a safe location – you will need to provide that number later during the upload/deploy process.

Create a Storage Account and Blob Container

Blob storage is a file store. It can be used to store files used in your Windows Azure application or, as in this case, you can use it as an interim storage location for your deployments. You will need a storage account and a container. The storage account is location for table, queue and blob storage. In this case we will use the account for blob storage.

NOTE

When you create an account, you create a URL so that your files can be found online. Perform the following steps to create a storage account and container so you can store your Windows Azure package deployment files.

1. If you're not already there from the steps above, click on the "Hosted Services, Storage Accounts and CDN" button on the portal.
2. Click the "Storage Accounts" tab.

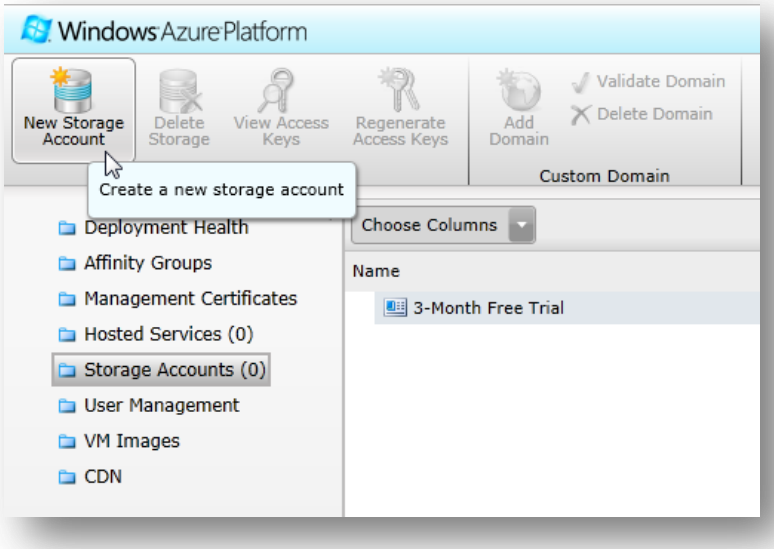


Figure 159: Create new Storage Account

3. Click **New Storage Account**.–A storage account creation dialog box appears

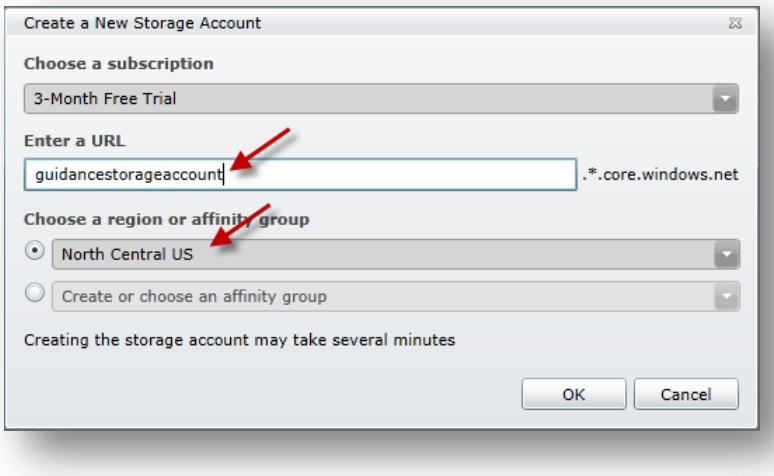


Figure 160: Enter Storage Account details

   a. Enter a name to give the account. This will be used as the beginning of the URL used to access the files. It must be unique, lower case/numbers, and between 3 and 24 characters long.
   b. Select a region that is the location of the data center where you want to create the storage. Pick the location closest to you in the dropdown.
   c. Click **OK**. It will take a few moments for the creation to complete.
4. During the deployment process, you will upload the package files to a "Container" in your storage account. The container can be created ahead of time or on the fly during the deployment process. Think of the container as a disk or a folder. It can contain files or more folders. If you want to create the container manually, you can use a Windows Azure storage explorer application.

You now have a deployment storage location for transferring the Windows Azure package up to the Windows Azure cloud.

NOTE

If you're wondering where to go to find a good Windows Azure Storage Explorer, you can check out this link: http://blogs.msdn.com/b/windowsazurestorage/archive/2010/04/17/windows-azure-storage-explorers.aspx. It is a list of Windows Azure explorers a feature comparison and their links.

Create a Hosted Service

A Hosted Service is a Windows Azure instance and your application runs within it. Unless you need or want a URL specific to you or your company, the hosted service also provides the URL for accessing the application via the name you give it. A hosted service needs to be in place to receive you application during deployment so you need to set it up ahead of time.

1. From within the portal, click the **Hosted Services, Storage Accounts and CDN** button Click **Hosted Services** .

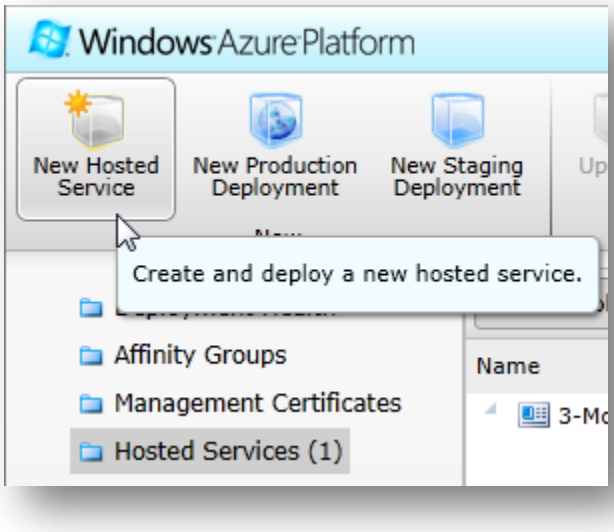


Figure 161: Create Hosted Service

2. Enter a name for your service.

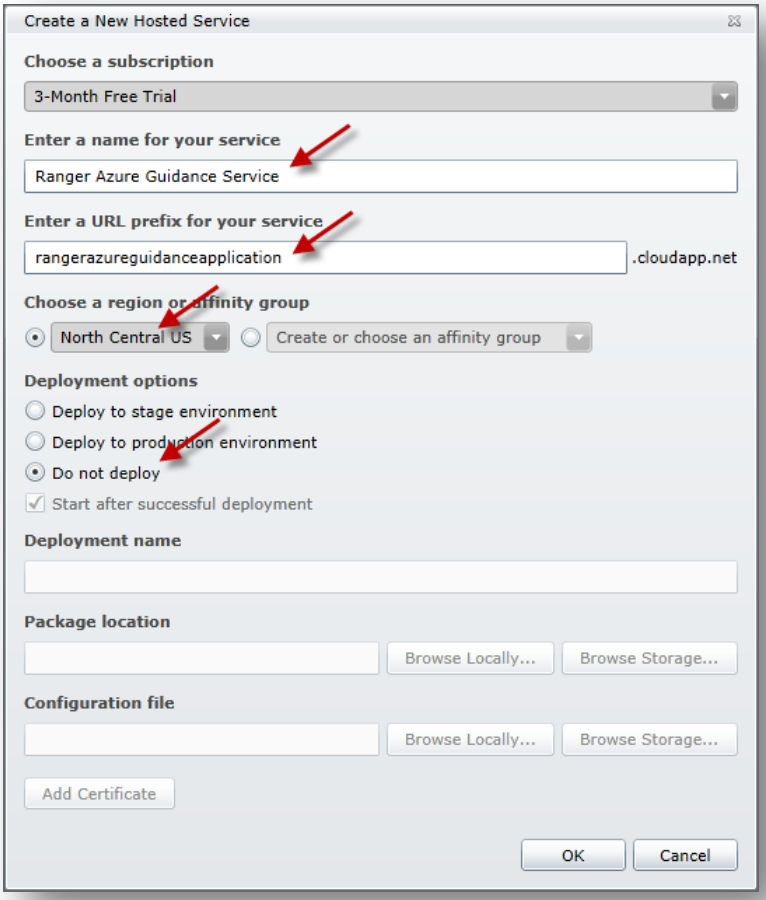


Figure 162: New Hosted Service details

3. Enter a URL prefix (also known as DNS Prefix) for your service. This must be unique across all Windows Azure applications – it will tell you if it is not.
4. Choose a region (data center location) for your application to be hosted.
5. In this case, click **Do not deploy** because we will be automatically deploying later as part of the build process.
6. Click **OK**.

This will create your hosted service environment.

Create and Deploy a Management Certificate

As described above, this certificate is used to ensure that you have the rights to deploy. A common certificate must exist on the machine deploying to Windows Azure and it also must exist in your Windows Azure account. We will first create the certificate and then we will push it up to your account using the portal. The certificate must be accessible by the person/account deploying to Windows Azure.

NOTE

You need management certificates locally and in Windows Azure if you are on a development machine and deploy through the Visual Studio interface. Likewise, if you are trying to automate build and deploy using a build server, you need to have matching certificates on the build machine and Windows Azure. This certificate must be

accessible by the service account performing the build and deploy – not the account of the person who initiated the build.

There are several ways to create a certificate. We will use the MakeCert command-line application to do the work. If you are creating this certificate for the build server, you will need to log in to the build machine using the build service account to install and perform these actions.

The MakeCert.exe command-line application is part of the Windows SDK and you can find it here: %programfiles(x86)%\Microsoft SDKs\Windows\(VERSION)\Bin\makecert.exe. If you do not have this installed on your build machine, you can run it from your development machine if you "remote desktop" into the build server and link your dev machine drives as resources to the remote machine.

Steps to Create a Management Certificate

1. Open a Visual Studio Command Prompt.

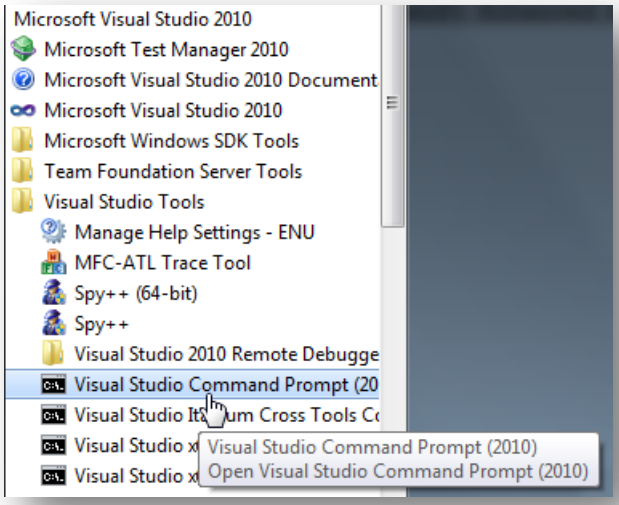


Figure 163: Visual Studio Command Prompt

2. Create a folder to contain the management certificate and name it "Azure Certificates." Then navigate to that folder:
   a. `MD \AzureCertificates`
   b. `CD \AzureCertificates`
3. Enter the following command that uses the MakeCert application to create a certificate. It will be named TFS-Azure Build Cert.cer:

```
makecert -r -pe -a sha1 -n "CN=Windows Azure Mgmt Authentication
Certificate" -ss My -len 2048 -sp "Microsoft Enhanced RSA and AES
Cryptographic Provider" -sy 24 "Tfs-Azure Build Cert.cer"
```

   The command above will automatically import the certificate into the "Personal" certificate store.

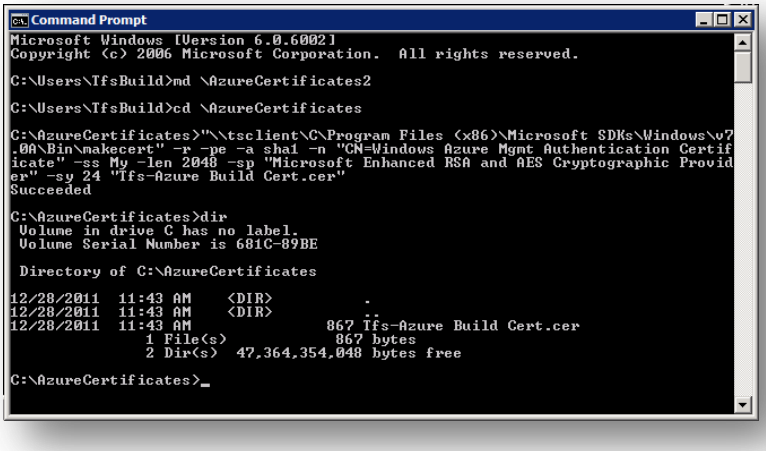


Figure 164: Certificate configuration

4. To deploy the certificate to Windows Azure, log in to the Windows Azure Portal. Note: You can copy the certificate file to another machine if it is easier to use your browser to log into the portal.
    a. In **Hosted Services, Storage Accounts & CDN** click **Management Certificates** and then click the **Add Certificate** button.
    b. Use the **Browse** button to identify the certificate file.
    c. Click **OK** to import the certificate into the Windows Azure Management Certificates

Build, Package and Deploy

Now that the build server is set up and ready, you need to modify the build process to provide the required upload and deploy functionality.

The full process from build (compile) to deploy can be broken down into the following steps:

1. Compile
2. Package – Creation of the CSPKG file
3. Upload – Copy the CSPKG file into blob storage
4. Deploy – Spin up of the Windows Azure VM and install of your application
5. Start – Start your application

Compile is done the standard way, using MSBuild scripts. In Team Foundation Build, even a build workflow template triggers the compilation of a .NET application using MSBuild. Included, as part of the Windows Azure SDK, is an extension MSBuild script that performs the package process using the CSPack command-line application.

To initiate this package process in Team Foundation Build and tell the build workflow what to use as a configuration file, you need to pass the information to the MSBuild script. In Team Foundation Build, this is done in the Build Definition using the **MSBuild Arguments** parameter. You also tell the script to use a particular Windows Azure configuration file using a naming convention. This is an example of the **MSBuild Arguments** parameter in a build definition:

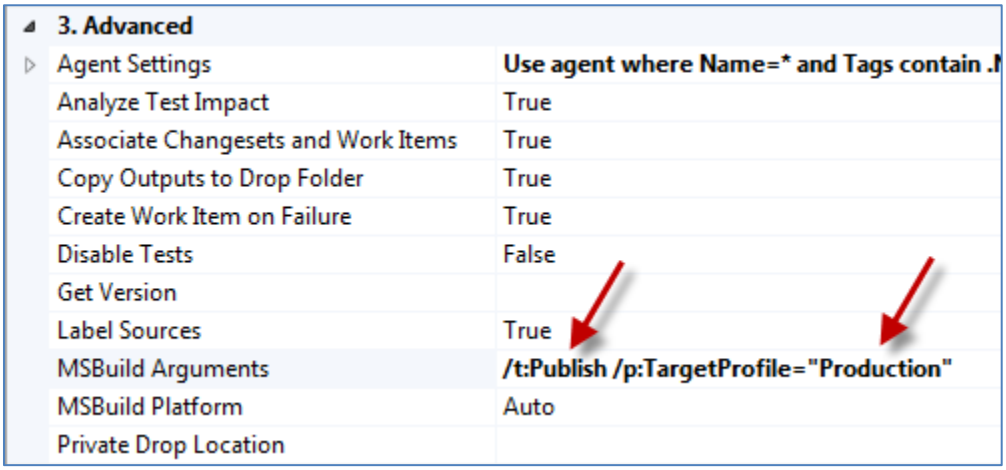


Figure 165: Example of the MSBuild Arguments

/t:Publish tells the MSBuild scripts to execute the **Publish** targets. This simply means that the CSPack application will be used to create a Windows Azure cspkg package file out of your compiled application.

/p:TargetProfile="Production" sets a parameter which tells the scripts which configuration file to use. In the example here, the process will use the configuration file named ServiceConfiguration.Production.cscfg.

What You Need to Create

So, the package process is provided for you by the SDK. This means to complete the automation of the entire build and deploy process, you need to create the upload, deploy and start actions. This can be done in several ways and all of them are valid. Some of the options are: create an MSBuild script, create a PowerShell script, or create an extended Team Foundation Build workflow script. Combinations of these options are valid as well. The real answer depends on your needs, environment and expertise. The solution described below utilizes a couple modified command-line applications and an updated Team Foundation Build workflow template. Note: The full details of this approach can be found in the associated Windows Azure Guidance Hands-on Lab.

Here is a zoomed out view of the Team Foundation Build template process. The additional steps are to be added at the end of the **Run On Agent** sequence. This is immediately after the compilation and the extension that creates the package file.

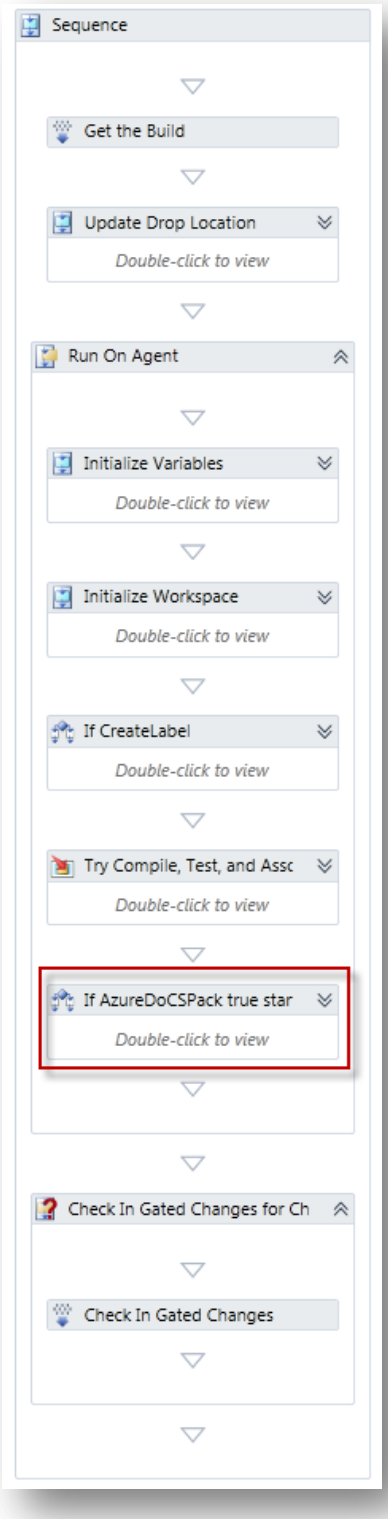


Figure 166: Location of the Windows Azure Build and Deploy Process Extension (box in red)

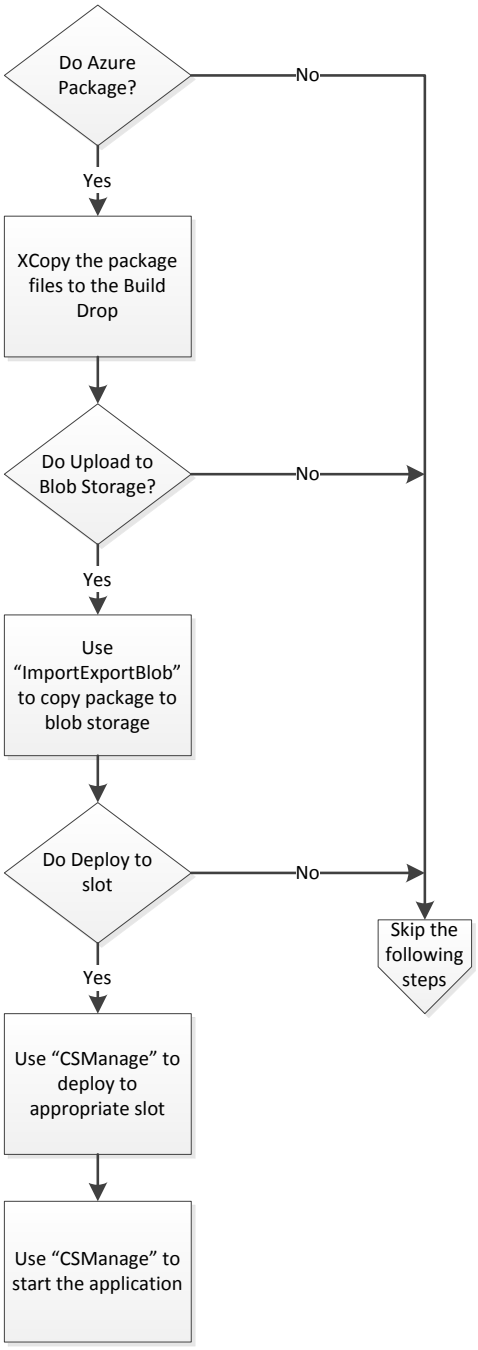


Figure 167: Windows Azure Deploy Decision and Process Flow

As you can see, there are three main decision points here. They are included to provide control over the entire deployment process.

1. The first point asks if we should do any of the deployment process at all.
2. The second point indicates if we should try to upload the package files to blob storage.
3. The last point asks if we should try to deploy the package to a Windows Azure slot.

Each of these is in place so you can test and verify the various portions of the deploy workflow.

Step 1: Copy to Build Drop

The extended MSBuild/CSPack process (unfortunately) does not copy the resulting package (cspkg) file to the build drop location. Therefore, to make all of the resulting build files easy to find and set the stage for the upload and deploy steps, you have to copy the cspkg and configuration files to the build drop location. The standard Windows XCopy.exe application is used for this step therefore it is available on all Windows machines.

Step 2: Upload the Packaged Application to Blob Storage

Here, we are using a command-line application that will perform the upload steps for us. The application used in this solution is called ImportExportBlob.exe. It can be found, along with the source code, on the Windows Azure Code Samples site (http://code.msdn.microsoft.com/windowsazure/). This application is not a custom build workflow activity so it can be used in a build workflow template or straight from a script.

Although it is not necessary, the package (cspkg) and the configuration (cscfg) files are both uploaded to blob storage. This is not entirely necessary because, in the next step, the configuration file is sent from the local machine. That is, it is not used directly from blob storage. However, they are always kept together in so you can tell what configuration was used if you look at the files in blob storage.

In the Hands-on Lab we will make some fairly minor modifications to the code to make it more scalable and easier to use in a process. The choice to use a command-line application here was made on the basis of available applications, documentation and relative ease of use. Again, you can choose a different approach.

Step 3: Deploy and Start the Package from Blob Storage

CSManage is used to deploy the application to a Slot. Like the ImportExportBlob application, it is available in the Windows Azure Code Samples site (http://code.msdn.microsoft.com/windowsazure/). CSManage is also used to start the application since deploying the app does not automatically start it. Like the ImportExportBlob application, some minor code modifications are made to make the application more scalable and easier to use in a process. The Deploy and Start steps are handled through separate calls to CSManage within the build template.

Minor Code Changes?

To be a little clearer on that point, the changes to the ImportExportBlob and CSManage applications include allowing the subscription ID and certificate thumbprint to be included as parameters. The applications both assume that you will place that information in an app.config file and although that would work in a build scenario, it makes it tougher to create multiple builds because you have to maintain or dynamically modify the config files. The modifications (shown in detail in the associated Hands-on Lab) allow the information to be provided as command-line parameters and therefore easier to pass on within the build template.

Locations for Build Process Artifacts

You should create a location within source control to hold all of your build process artifacts. Although it is not a requirement, if you create a Team Project devoted to holding all of your build process artifacts, it can make management of your builds easier across project teams. This project would act as a centralized location for all project teams to find the components that they need to manage their builds. For example, custom activities need a common source code location to store and find assemblies during the build process. This can be within a common Team Project. Likewise, the applications needed by the Windows Azure build process can be stored in the same Team Project. They are not custom activity assemblies so a different folder makes sense for their storage.

Reusable build templates can be another folder. Obviously, this makes them easy to find by everyone creating a build definition. The only real requirement is that the folders in the Team Project need to be readable by everyone using the common components.

As part of the build template you will see in the Hands-on Lab, there is code designed to pull the required artifacts from source control at the time of build. This has multiple benefits of making the files easy to manage and find for build setup AND you don't have to install the required command-line applications on the build machine. They are made available when they are needed.

If you look at the build process template in the Hands-on Lab you will see the following:

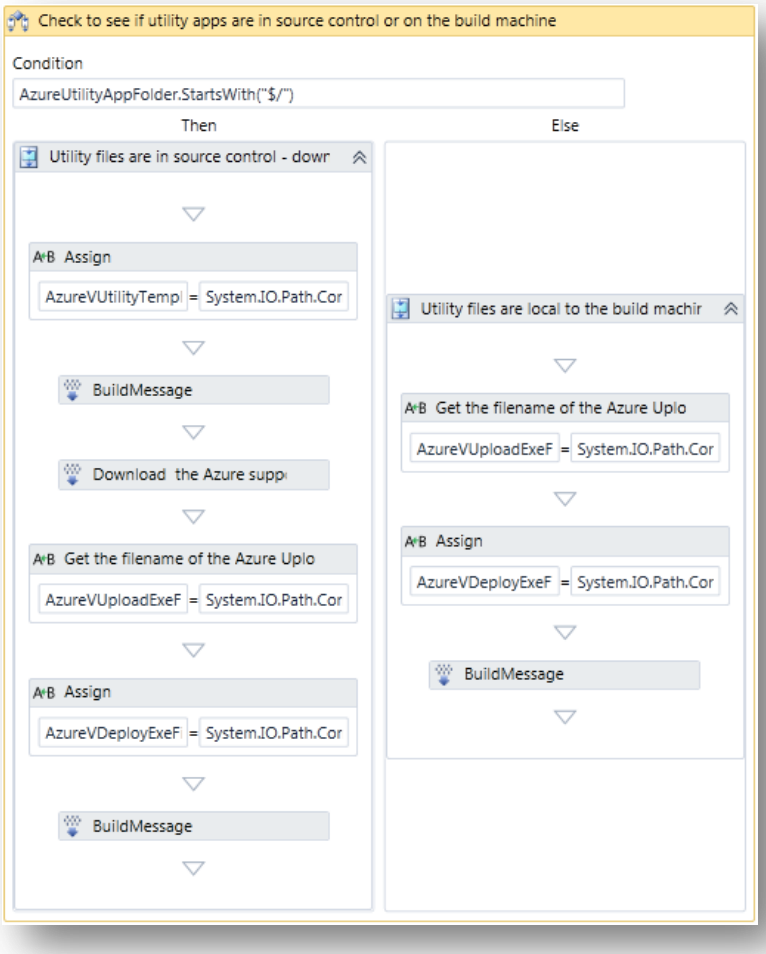


Figure 168: Conditional checking in the workflow

This code checks the argument that contains the location folder for the command-line applications. If it starts with "$/" then they reside in source control and the files are downloaded as part of the build and therefore ready for use below. If the name of the folder does not start with "$/" then it needs to exist on disk somewhere. This could be local to the build server or a network share but either way the files would need to be copied to that location and security set up to allow the build service to access them.

Calling the Command-Line Applications

The bulk of the added workflow steps in the build template are setting up, executing and validating the execution of command-line applications that do all the work. In all four cases of using XCopy, ImportExportBlob and CSManage applications, the Team Foundation Build InvokeProcess activity is used. The file path and application arguments (input) are provided as strings and the result (output) is captured in an integer variable. That variable is then evaluated for success (result == 0). If the execution of the application fails, then the process throws an exception, which should stop the build process and provide some detail about the failure in the build log.

The Build Definition

Once the build template is in place, you can create a build definition. Use the standard steps to load the updated build template in build definition process section. When you do, you will see several additional parameters under the **Misc** category.

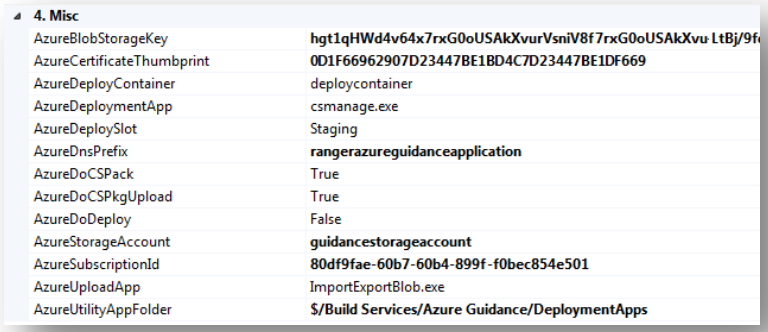


Figure 169: Team Foundation Build Definition configuration

The three Boolean values (**AzureDoCSPack**, **AzureDoCSPkgUpload** and **AzureDoDeploy**) control the three progressive decision points described above. Luckily, they are in the same order in the build definition as they appear in the workflow. Change any or all of them to control how much of the package/deploy process gets executed. AzureDoDeploy is defaulted to **False** so as not to accidentally deploy an application over another. Set it to **True** if you do want the application deployed. This parameter can also be exposed to the queue build definition window by checking the **Always show parameter** option in the Metadata Editor. This provides flexibility to control the decision point while queuing the build. Also, it doesn't require you to edit build definition permissions.

AzureBlobStorageKey comes from the Windows Azure management portal. You can get to a primary and secondary key in the portal and either will work. Just navigate in the portal to the Storage Accounts and then click the storage account you created. In the properties on the right side, there are buttons to see the primary or secondary access keys. Click either one; they both bring up the same dialog box that has copy buttons, so you can easily copy the values and paste the value where you want.

The **AzureUtilityAppFolder** indicates where to get the utility apps that are used in the workflow. In this case, they are in source control and all of the applications and their dependencies are included. The workflow downloads them all at the time of build, uses them and even deletes them from the build machine when done.

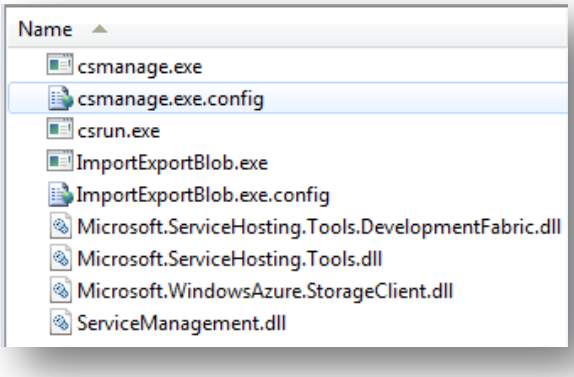


Figure 170: Utility Apps

AzureDeploymentApp and **AzureUploadApp** hold the file names for the applications that do the work. In this case the names are (respectively) CSManage.exe and ImporteExportBlob.exe

The **AzureCertificateThumbprint** communicates to Windows Azure that the build process has authorization to upload and deploy the application. This number is used to look up the local certificate and that is then compared with the Windows Azure management certificate for verification. You can get the value of the thumbprint on the Windows Azure portal by clicking the certificate and then copying the thumbprint value from the properties on the right side of the page.

The **AzureStorageAccount** is the account name that you created to set up blob storage. **AzureDeployContainer** is the name of the blob container that will store your package file. It is the name you created in the Windows Azure storage explorer application.

AzureDnsPrefix is the name you gave your Hosted Service. If the URL for your application is **myazuresite.cloudapp.net** then your DNS Prefix is **myazuresite**.

AzureDeploySlot is where your application is to be deployed. You have the choice of **Staging** or **Production**. It is recommended to not deploy straight to production. Test it first! Unless, of course, this is a test account and Production is still just a test location.

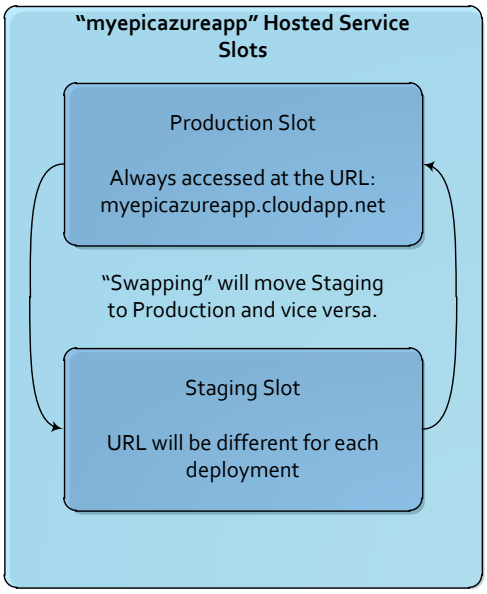
More on Windows Azure "Slots":

When a hosted service (destination location for your Windows Azure application) is created, two locations are configured to accept an application. They are called "Staging" and "Production". Production is where your application needs to reside to be seen through the DNS name that you give it when creating the hosted service. For example, if you create a hosted service and give it the name "myepicazureapp", then deploy an application to the Production slot, you will be able to access the site through the URL "myepicazureapp.cloudapp.net". However, if you deploy your application to the Staging slot, it will not be available through your new URL since only one deployment can use that URL at a time. You will be able to access the Staging slot application through a dynamically generated name created during deployment. It will be something like this: "3F2504E0-4F89-11D3-9A0C-0305E82C3301.cloudapp.net".

Then you ask, why the two slots? Staging can and should be used for testing your application before moving it to production. This way you can deploy an application, test it, and then decide to push it into Production.

Moving an application to Production from Staging is a simple task. You just need to perform a "Swap" using the Windows Azure Portal. You just right click on the deployment that you want to move to Production and select "Swap". If no application already exists in Production then the DNS entry for "myepicazureapp.cloudapp.net" is changed to point at your running application. If an application does exist in Production, the same process occurs but when completed, the app that was in Production is now in Staging and vice versa. To access the app in Staging you will need to find the new URL in the Windows Azure Portal. Just click on the app and look at the "DNS name" property.

All this was put in place so you can manage your deployments and give you the capability of testing an application before moving it to Production. It also gives you the ability to swap applications back if you push something into Production but then determine you need to roll back to the previous application. To do this, just perform another swap. Production will be back to the way it was.



The **AzureSubscriptionId** is a GUID that uniquely identifies your Windows Azure account. You can find it by clicking **Hosted Services** or **Storage Accounts**. It will then be in the properties section.

Verifying the Build Process

Create a build definition using the included build workflow template, set all of the Boolean values to indicate how much of the deployment process to execute and fill in the rest of the properties. Queue up the build and let it go. As the build process is executing, you can watch the Hosted Service area within the Windows Azure portal. When the build process gets to the deployment step, you should be able to see the notification that the VM is starting to spin up.

You should also be able to use your Windows Azure storage explorer to see the package and configuration file get uploaded to blob storage.

If the deploy does not work, verify the parameter values again. Also, review the build log. It should give you valuable insight into what went wrong.

If you see an error like this in the build log: **Deploy Azure Package: Client certificate cannot be found. Please check the config file**. then your management certificate is not deployed to the personal certificate store on the build machine. That is, it is not deployed for the service account that is executing the build process.

Ultimately, you should see the running application in the Windows Azure Portal:

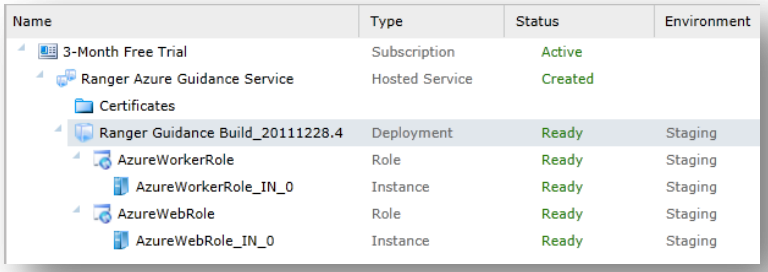


Figure 171: Application in Windows Azure Portal

If you deployed a web application, like in the Hands-on Lab, you should be able to click the link in the properties section below the **DNS name**…

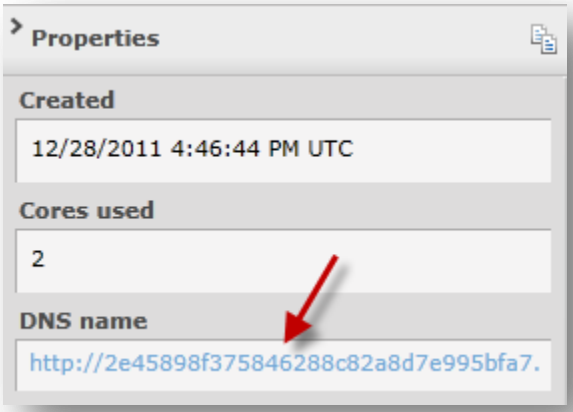


Figure 172: DNS name

and a browser session should launch and take you to your running application:

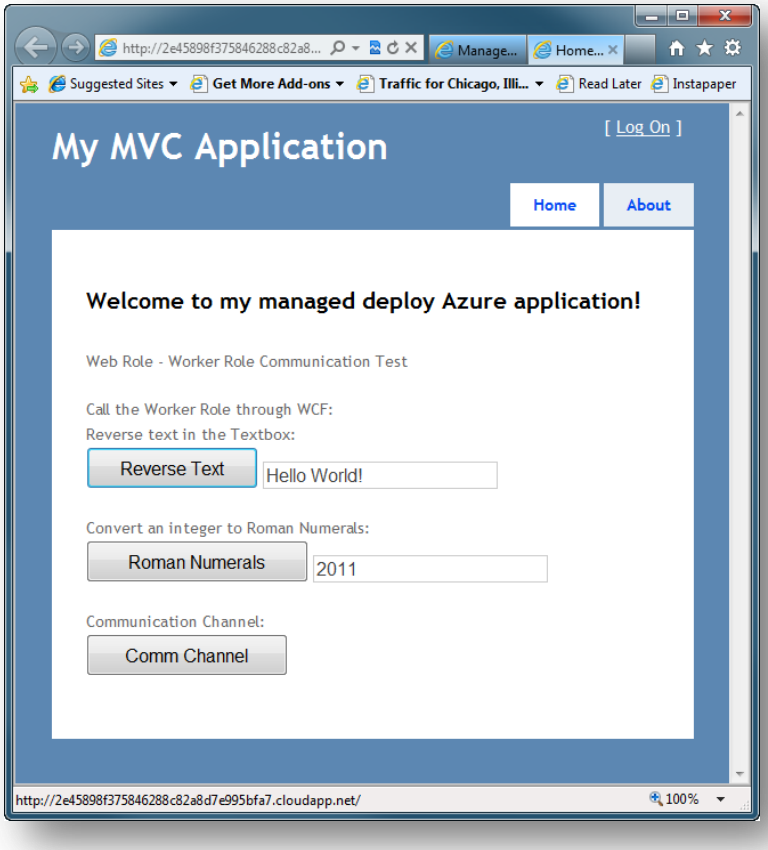


Figure 173: Application Deployed and Running

Windows Azure Resources

Windows Azure Site: http://www.windowsazure.com/

Getting Started with Windows Azure: http://go.microsoft.com/fwlink?LinkID=160765

Windows Azure Developer Center: http://www.windowsazure.com/develop/

Windows Azure Management Portal: https://windows.azure.com/

Windows Azure Samples (Code): http://code.msdn.microsoft.com/windowsazure/

Microsoft Windows SDK for Windows 7 and .NET Framework 4:
http://www.microsoft.com/download/en/details.aspx?id=8279

Deploying SharePoint Packages

How is a solution deployed to SharePoint?

The SharePoint deployment package is called a WSP file. This is actually a CAB file with the alternate extension of WSP. A WSP can contain one or more SharePoint components that need to be installed together as a SharePoint solution. These can be targeted at a SharePoint Farm or an individual Web Application.

NOTE

If you wish to see the contents of a WSP, rename the file to have the extension CAB and then you can browse into it using Windows Explorer

Creating a WSP in Visual Studio for SharePoint 2010

Using Visual Studio 2010 onwards

Since Visual Studio 2010, Visual Studio provides projects templates for SharePoint 2010 development that automatically creates WSPs when the project is built.

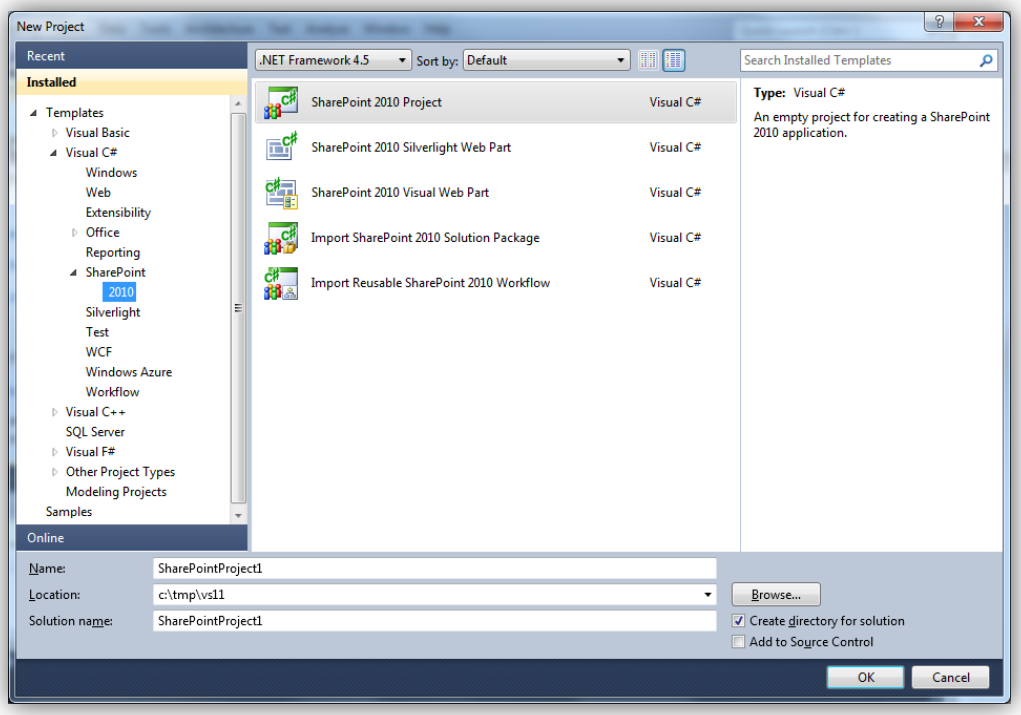


Figure 174 – SharePoint 2010 templates in Visual Studio 2012

These templates require that SharePoint 2010 be installed on the development PC. This can be done on a computer using either 64-bit Windows server and client operating systems, see MSDN[139] for details of the installation process.

Using these Visual Studio 2010 templates a WSP is created whenever the project is rebuilt, and it is automatically deployed to the local SharePoint server when F5 is pressed or the deploy option selected by right clicking on a project in Solution Explorer.

[139] http://msdn.microsoft.com/en-us/library/ee554869.aspx

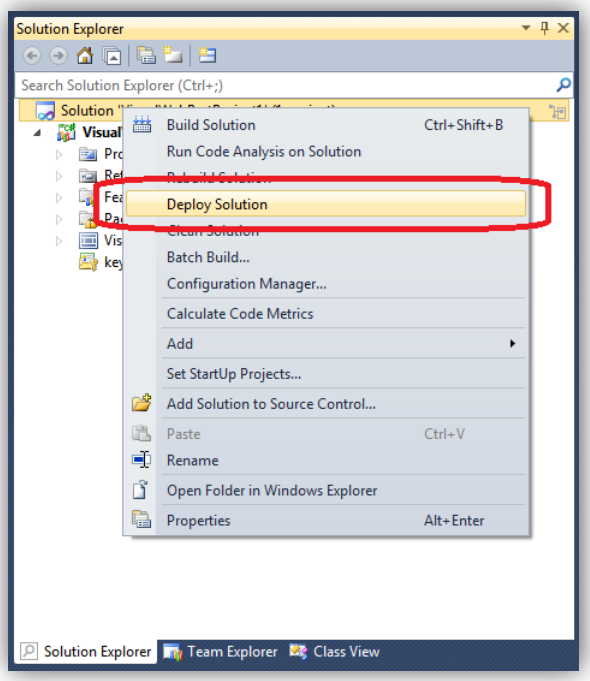


Figure 175 – Deploying a SharePoint solution

NOTE

If you are using SharePoint 2007 no version of Visual Studio provide templates for the creation of WSPs. A developer has to 'hand craft' their own mechanism. This involves creating a number of configuration XML files and running the MAKECAB.EXE tool as detailed on MDSN[140].

The hand editing of XML files and the management of the related GUIDs are prone to human error. Therefore it was advisable to use tools such as WSPBuilder[141] to manage the process. This tool automates the creation of the various XML configuration files, centralizing settings in a single file.

It is possible using WSPBuilder to create a Visual Studio projects that builds SharePoint components, either within the same project or another project in the solution, and then build a WSP. This is achieved by calling the required commands within the Post Build events of the projects, as detailed on the WSPBuilder documentation[142].

Using Team Foundation Build

By default building a SharePoint project on a Team Foundation Build Agent will not create a WSP. This is for two reasons:

1. Referenced assemblies are missing
2. The default build command does not create the WSP

Configuring Team Foundation Build to build SharePoint WSPs

By default a Team Foundation Build Agent will not be able to build a Visual Studio SharePoint project due to missing references. There are a number of options to address this issue:

[140] http://msdn.microsoft.com/en-us/library/bb466225.aspx

[141] http://www.codeplex.com/wspbuilder

[142] http://keutmann.blogspot.com/2009/04/wspbuilder-documentation.html

1. Install SharePoint on the Build Agent using the same process as for a development workstation (see references above[143]).
2. Create a 'Referenced Assemblies' folder under the solution directory in Visual Studio. Copy in all the required SharePoint assemblies[144] into this folder and reference them from this folder in all the SharePoint projects. Place this folder, along with the rest of the solution under Team Foundation Server source control. By using this model the required SharePoint assemblies will be automatically copied to the Build Agent with the other files in the solution.
3. Follow the instructions on MSDN[145] to install just the required assemblies onto the Build Agents.

The second option, although flexible because the build agents need no special servicing to support SharePoint, means that it is easy to scale out the number of build agents in use, has the limitations that the developer must manage the referenced assemblies and also the binaries are stored under source control (a solution not all teams like to adopt).

It is therefore recommended that the third option be adopted to allow for more flexible use of Build Agents for SharePoint projects, as all SharePoint assemblies are present, but without the overhead of installed and configuring SharePoint itself. To further ease this installation process a PowerShell script has been provided via the SharePoint Development Blog[146] to automate the process.

NOTE

It is possible within a build process template to detect the dependencies that are not installed and deployment them from a central resource using any scripting technology.

However, using NuGet is probably a more flexible approach.

However, a limitation of the second and third option is that SharePoint is not installed and configured and so tests that require SharePoint cannot be run on the Build Agent as part of the build process. This will not usually be an issue as

- Integration Tests can be run against remote SharePoint servers.
- True unit tests, tests that can be run within the build process without reference to external systems, can be run locally on the Build Agent as long as calls to SharePoint are mocked out[147].

Telling the Build to Create the WSP

To cause the build to create a WSP then an extra flag needs to be passed to MSBuild. To do this, pass the parameter in via an MSBuild argument for the build definition.

1. In Team Explorer, select the build definition you wish to edit.
2. On the **Process** tab select the advanced options.
3. Set the MSBuild Arguments to ***/p:IsPackaging=true***.
4. Save the edited build process definition.

[143] Note this works for SharePoint 2010 irrespective of the Build Agent operating system, but does require a server operating system if SharePoint 2007 is being targeted

[144] These files can be found in the SharePoint 12 (2007) or 14 (2010) hive e.g. C:\Program Files\Common Files\Microsoft Shared\Web Server Extensions\14\ISAPI

[145] http://msdn.microsoft.com/en-us/library/ff622991.aspx

[146] http://blogs.msdn.com/b/vssharepointtoolsblog/archive/2010/04/14/building-visual-studio-sharepoint-projects-using-team-foundation-build.aspx

[147] To mock out SharePoint for pure unit testing then Typemock Isolator (http://www.typemock.com) or Microsoft Pex & Moles (http://research.microsoft.com/en-us/projects/pex/) can be used

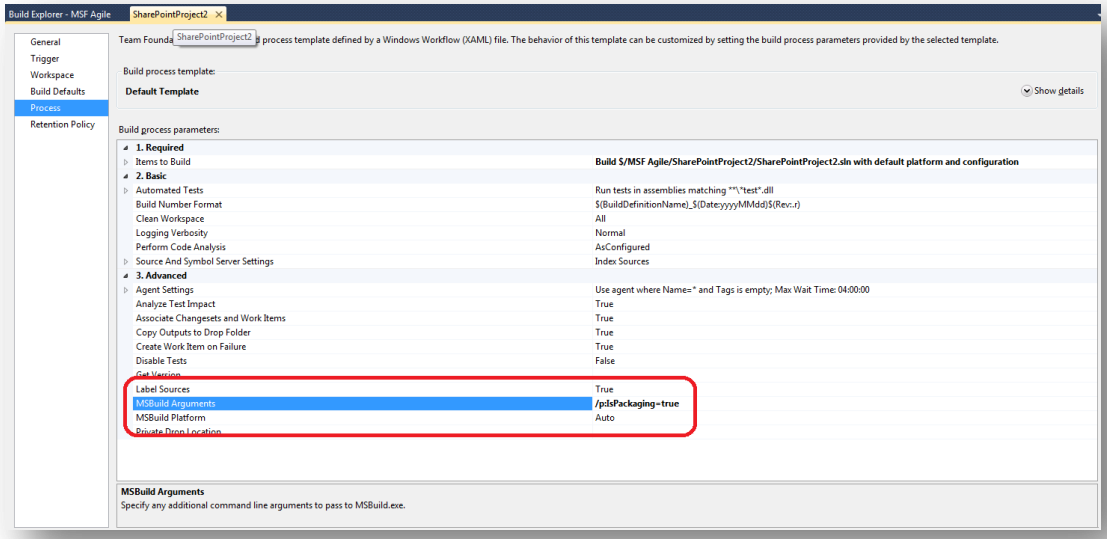


Figure 176 – Setting the MSBuild Parameter to create a WSP in Team Foundation Server Build

Once the Team Foundation Build has completed you will find the WSP files created by each SharePoint project within the solution have been created and the resultant files copied to the builds drop location.

Other activities to include in the build

As well as creating the WSP you can also use the Team Foundation Build to perform other tasks.

Versioning Features

Incrementing the feature versions within a SharePoint project can be automated within the build. There is no standard activity for this but the action can be done by editing the .csproj or .vbproj file that contains the SharePoint solution.

Details of the process can be found on the SharePoint Developers Team Blog[148]. The key step is to add an inline code fragment that updates the feature version in the project.

MICROSOFT SHAREPOINT ONLINE CODE ANALYSIS FRAMEWORK (MSOCAF)

MSOCAF[149] is an analysis tool that can be used to check that custom SharePoint solutions meet the requirements to run on SharePoint Online. The framework leverages existing tools like FxCop, CAT.Net, and SPDisposeCheck to analyze custom solutions.

The framework is distributed as Click-Once desktop application and appears to have no command line interface. Hence it cannot be integrated directly with the build process, but the tools it makes use of can be as discussed below. However, it remains a useful tool for checking and deploying the output of an automated build to SharePoint Online

SPDisposeCheck

SPDisposeCheck[150] is a tool for SharePoint developers to make sure that they are disposing of resources correctly. The problem is that it is a bit slow to run, so developers will tend not to run it as often as they should. A good

[148] http://blogs.msdn.com/b/sharepointdev/archive/2011/04/28/incrementing-the-feature-version-through-msbuild.aspx

[149] https://caf.sharepoint.microsoftonline.com/

[150] http://code.msdn.microsoft.com/SPDisposeCheck

solution to the problem is to run it as part of the Team Foundation Build process. There is no specific Team Foundation Build activity to do this, but the tools can be run using an InvokeProcess activity. Follow these steps:

1. Open the build workflow template.
2. Create a variable to store the number of issues found by the SPDisposeCheck command. This needs to be of the type Int32.
3. After the compile and before the test step, add an InvokeProcess activity .

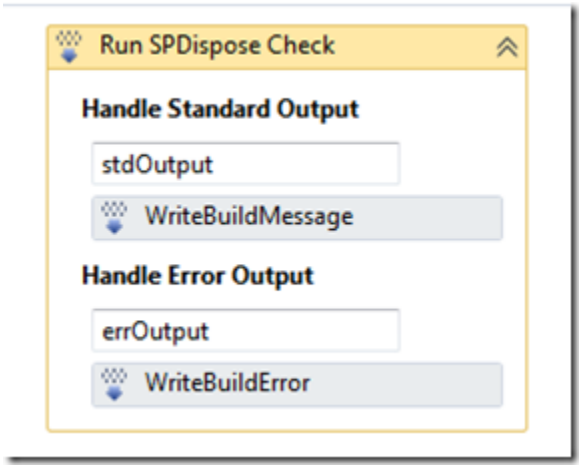


Figure 177 – Modified workflow showing InvokeProcess activity

4. Set the InvokeProcess properties as shown below.
   - Arguments: String.Format(“”””{0}””””, outputDirectory) (remember you need the enclosing “ as your path could have spaces in it)
   - Filename: To the location of the **SPDisposeCheck.exe** file
   - Result: A previously created build variable of type Int32

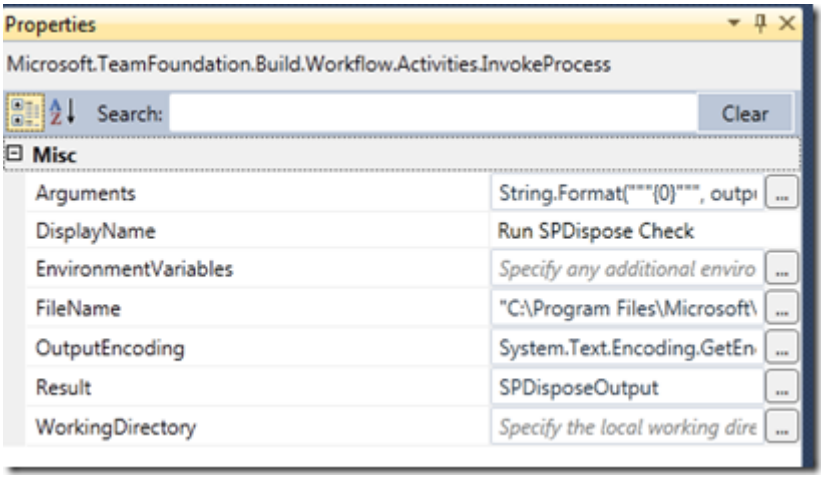


Figure 178 – Configured InvokeProcess activity to run SPDisposeCheck

5. Add a WriteMessage activity for both the Standard and Error output to make sure the console output from **SPDisposeCheck** is written to the build log file.

6. **SPDisposeCheck** reports the number of errors returned by the command line exe, so the **InvokeProcess** is able to store this in the **SPDisposeOutput** variable. We are able to use this to fail the build.

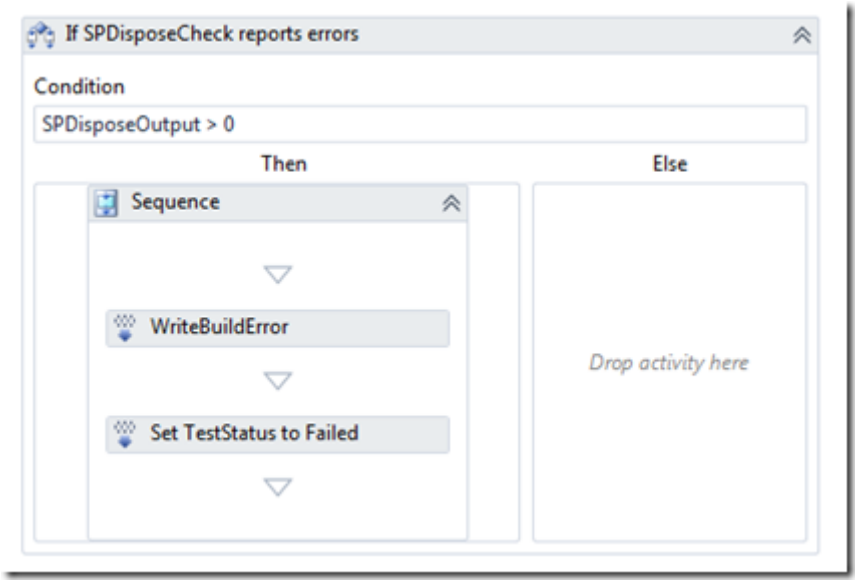


Figure 179 – Failing the build if SPDisposeCheck does not return zero

7. If there are any errors found a build error message is written and the **TestStatus** set to failed. You might choose to set the build status to fail or any other flag you wish. The potential problem with this simple solution is that the **TestStatus** value could be reset by the tests that follow in the build process, so a check of other uses of this variable should be made.

If we use this means of running **SPDisposeCheck** then we do not need to parse any text or XML results file. This makes the integration into the build process easier.

WSP Deployment

After the Team Foundation build server has created the WSP, it must be deployed. This can be done by hand using either the SharePoint STSADM command or using PowerShell. PowerShell is the recommended solution for SharePoint 2010 and most current Microsoft server technologies; however, it is possible to modify the Team Foundation Build process template to deploy the WSP to any target SharePoint server using either of these technologies by running either of these commands:

- The STSADM command (via the *InvokeProcess* activity)
- Running PowerShell commands (via the Community TFS Build Extensions[151] *PowerShell* activity).

Irrespective of the tool used, the underlying process to deploy a SharePoint features is usually as follows:

Add Solution to SharePoint Farm
Deploy Solution to Site(s)
Activate Feature(s)
[Use the feature(s)]
Deactivate Feature(s)
Retract Solution from Site(s)
Delete Solution from Farm

[151] http://tfsbuildextensions.codeplex.com

Manual Deployment via STSADM

In SharePoint 2007 the STSADM command was used to deploy WSP files. See TechNet for details[152].

Manual Deployment via PowerShell

In line with other Microsoft server technologies, SharePoint 2010 is moving towards management via PowerShell. Details of the management process can be found on MSDN[153].

Deployment via PowerShell and Team Foundation Build

There is no reason why single PowerShell commands have to be issued within a Team Foundation Build activity using the Community TFS Build Extensions[154] *PowerShell* activity. A whole script can be run to manage the complete deployment process. The exact details of the script to use will be dependent on the project's needs.

A good starting point can be found on Gary Lapointe's blog, "Deploying SharePoint 2010 Solution Packages Using PowerShell (Revisited)"[155].

The SharePointDeployment Activity

As part of the Community TFS Build Extension[156] CodePlex project a **SharePointDeployment** activity has been created that wrappers PowerShell commands to perform deployment actions. This activity can be used to deploy the SharePoint solution to an instance of SharePoint hosted the local build box, or to the more likely scenario of an existing SharePoint farm.

To deploy to a remote SharePoint instance, Windows Remote Management (WinRM[157]) is used.

NOTE

To run PowerShell commands on a remote machine, WinRM needs to be installed and configured on the remote server. On Windows Server 2008 the WinRM starts automatically and on Window Server 2003 R2, it must be installed. However, it is available as the Hardware Management feature through the Add/Remove System Components feature in Control Panel under **Management and Monitoring Tools**.

Actions Available

The activity can perform the following actions:

Action	Description
AddSolution	Add a new solution by issuing an **add-spsolution** command
InstallSolution	Deploys a solution by issuing an **install-spsolution** command
UpdateSolution	Updates an already installed solution by issuing an **update-spsolution** command
UnInstallSolution	Retracts a solution by issuing an **uninstall-spsolution** command
RemoveSolution	Removes a solution by issuing a **remove-spsolution** command
EnableFeature	Enables a feature by issuing an **enable-spfeature** command
DisableFeature	Disables a feature by issuing a **disable-spfeature** command
GetFeature	Either gets a list of all features installed or a single named feature by issuing a **get-**

[152] http://technet.microsoft.com/en-us/library/cc261956(office.12).aspx
[153] http://msdn.microsoft.com/en-us/library/ms442691.aspx
[154] http://tfsbuildextensions.codeplex.com
[155] http://blog.falchionconsulting.com/index.php/2011/04/deploying-sharepoint-2010-solution-package-using-powershell-revisited/
[156] http://tfsbuildextensions.codeplex.com
[157] http://msdn.microsoft.com/en-us/library/aa384372(VS.85).aspx

Action	Description
	spfeature command
GetSolution	Either gets a list of all solutions installed or a single named solution by issuing a **get-spsolution** command

The usage documentation can be found on the project site[158].

Other Tools and Technologies Applicable to SharePoint

Microsoft Online Code Analysis Framework (MSOCAF)

To be able to submit a SharePoint solution to the SharePoint Online[159] environment it must meet certain requirements. To enable a developer to validate their solution prior to submission, Microsoft has built a framework called the Microsoft SharePoint Online Code Analysis Framework. Part of this framework is the MSOCAF client[160], a tool that a developer runs on their own development PC to check for common issues in SharePoint development.

The framework leverages existing tools like FxCop[161], CAT.Net[162] and SPDisposeCheck[163] to statically analyze custom solutions and then can also perform a local test install using STSADM and PowerShell commands.

Unfortunately, the MSCOCAF wizard cannot be wired directly into the TFS build process because it is UI driven, and provides no command line options.

However, the MSOCAF client obviously has a good deal of overlap with the topics discussed in this guidance. The same analysis rules and PowerShell scripts used within the MSOCAF wizard can be applied to the TFS build process, but the extraction of these rules must be done manually from the directory structure that is created by the wizard.

SharePoint/TFS Continuous Integration Starter Pack

An alternative means to manage SharePoint within the TFS build process can be found in the SharePoint/TFS Continuous Integration Starter Pack[164] Codeplex project. This is discussed in the detail on SharePoint Developer Team Blog[165].

The basic process used in this toolkit is to run a set of PowerShell scripts that perform the deployment. Hence, this is a toolkit that can deliver deployment using the methods outlined in the Manual Deployment via PowerShell section of this guidance.

Summary

In this section we have seen how Team Foundation Build can be modified to create a SharePoint WSP and deploy the file to either a local or remote SharePoint farm.

[158] http://tfsbuildextensions.codeplex.com/wikipage?title=How%20to%20integrate%20the%20SharePointDeployment%20build%20activity&referringTitle=Documentation

[159] http://www.microsoft.com/en-us/office365/sharepoint-online.aspx#fbid=RhbBiysrZf4

[160] https://caf.sharepoint.microsoftonline.com/

[161] FXCop – Also known as Code Analysis in Visual Studio 2010 and later

[162] Cat.net – A security checking tool http://visualstudiogallery.msdn.microsoft.com/8ef8d7ba-422a-428d-86ed-74fc864a7697/

[163] http://code.msdn.microsoft.com/SPDisposeCheck

[164] http://sharepointci.codeplex.com/

[165] http://blogs.msdn.com/b/sharepointdev/archive/2011/08/04/continuous-integration-for-sharepoint-2010-mike-morton.aspx

Using Microsoft Dynamics CRM with Team Foundation Build

In this release of the guidance we had hoped to have in-depth guidance about using Team Foundation Build to automate the build and deployment processes involved with Microsoft Dynamics CRM. However, we now plan to provide this in a future update.

Even though we can't bring you in-depth guidance at this time, if you are using Microsoft Dynamics CRM and Team Foundation Build, we recommend that you download the Microsoft Dynamics CRM 2011 Software Development Kit (SDK)[166]. Once the download is extracted, install the *CRM Developer Toolkit*[167] using $ \sdk\tools\developertoolkit\ crmdevelopertools_installer.msi.

The *CRM Developer Toolkit* is "a set of Microsoft Visual Studio 2010 integration tools focused on accelerating the development of custom code for Microsoft Dynamics CRM 2011 and Microsoft Dynamics CRM Online. The Developer Toolkit supports creation and deployment of plug-ins, custom workflow assemblies, XAML workflows and Web resources. A developer can write custom code within Visual Studio and deploy the code to an unmanaged solution on a Microsoft Dynamics CRM server." A good introduction to the toolkit can be found on the team blog[168].

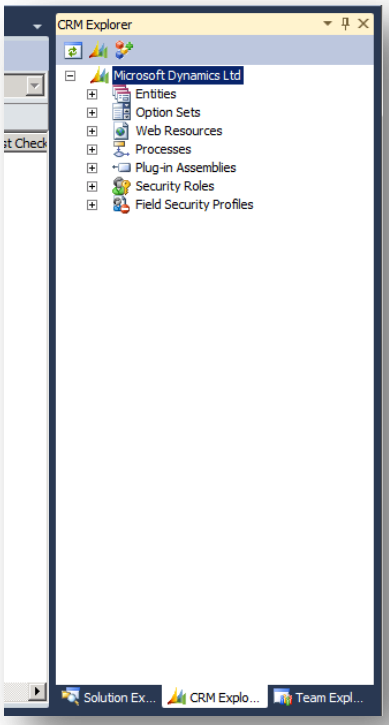


Figure 180 – CRM Explorer in Visual Studio

[166] http://www.microsoft.com/download/en/details.aspx?id=24004

[167] http://msdn.microsoft.com/en-us/library/hh372957.aspx

[168] http://blogs.msdn.com/b/crm/archive/2011/11/03/developer-toolkit-for-microsoft-dynamics-crm.aspx

Using ClickOnce with Team Foundation Build

What is a ClickOnce Application?

ClickOnce is a deployment technology that enables you to create self-updating windows based applications such as Windows Presentation Foundation (WPF), Windows Forms, Office Solutions, and console applications that can be installed and run with minimal user interaction. For a complete overview of ClickOnce, please visit the ClickOnce Deployment Overview article on MSDN.[169]

Why use ClickOnce?

Client applications are usually deployed to client machines by an installer package such as a MSI. For IT departments this means either manually or systematically touching each machine to deploy the package. Initial installations can be challenging but updates can be even more complex to uninstall previous versions and make sure the application is properly reinstalled with the updated version. ClickOnce allows client applications to simply be deployed to a URL or Network share where users can install the applications from this location. Be aware that ClickOnce deployments are by default restricted to a set of permissions and actions that are defined by the security zone[170]. In addition, these applications are installed in a dynamic location under the user's folder. ClickOnce can even be configured to check for updates and perform self-updates.

ClickOnce and Team Foundation Build

Typically, ClickOnce applications are packaged from Visual Studio. In Visual Studio you can click the **Publish Now** button from the project properties to package and deploy a ClickOnce application. There are also options to change the settings on that same dialog box. This makes deployment from Visual Studio straightforward. The figure below shows where to configure ClickOnce in Visual Studio and to initiate the publish operation.

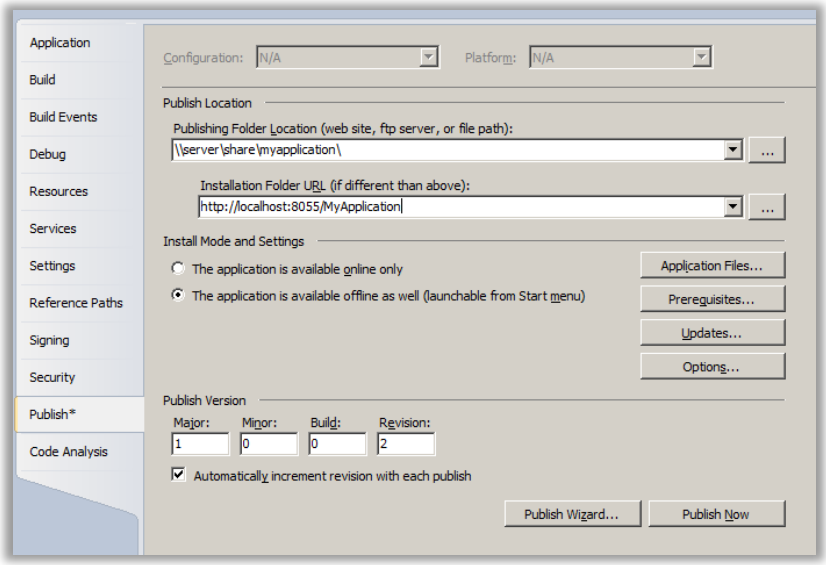


Figure 181 – ClickOnce Settings tab in Visual Studio

As recommended throughout this guidance, performing builds from a centralized build server instead of a developer PC is always preferred; however, from Team Foundation Build, there is currently no simple out of the box solution that gives you this same control. For accomplishing a simple build, parameters can be passed in on the MSBuild Arguments property in the build definition's process tab. The example below specifies a web url to run the installation from. To install from a network share, simply remove the PublishUrl and InstallUrl properties.

[169] http://msdn.microsoft.com/en-us/library/t71a733d.aspx

[170] http://msdn.microsoft.com/en-us/library/76e4d2xw.aspx

/Target:Publish /property:PublishDir=\\server\share\myapplication\
/property:PublishUrl="http://localhost:8055/MyApplication"
/property:InstallUrl=http://localhost:8055/MyApplication

The above parameters are added to the build definition's process tab below:

Figure 182 – Adding ClickOnce Publishing to a build definition

The first challenge that you will quickly notice is that there is no way to change the configuration file. Furthermore, the manifest that is created prevents you from modifying or replacing any files so there isn't a way to update the configuration file after the fact. In addition, signing files must be done before the manifest is created. The following section explains how to accomplish these at a project level that can be used by the build process.

Project File versus Build Template

To provide an enterprise worthy solution you need a method for applying environment-specific requirements, including replacing the configuration files. There are various solutions available for accomplishing this task that are either project file-specific or build template-specific. The project file-specific solutions use customizations to the project's .csproj or .vbproj file. Within the project file, the replacement of the configuration can be specified through conditional ItemGroup elements. Advantages of this solution include better separation of what are project specific requirements versus what is a global build workflow need. Because the configuration is project-based and a solution or group of solutions could contain any number of ClickOnce applications, having the project manage its own configuration files provides clean separation.

The other option is handing the configuration file replacement from within the build definition. This is a straightforward process that overwrites the appropriate configuration file after the source code has been retrieved for the build server workspace. This works fine for a single project but it becomes difficult to manage the configuration file relationships because multiple ClickOnce projects are being built.

The configuration file maintenance by the project file is the preferred solution and will be explained in the solutions below.

Configuration File Project Changes

First, create the folders for each environment and add an App.config file to each folder. Unlike the Web.config Transformations, each environment's configuration file will need to contain all of the settings and be maintained when settings change. Modify the environment specific settings, such as a database connection, to be the appropriate values.

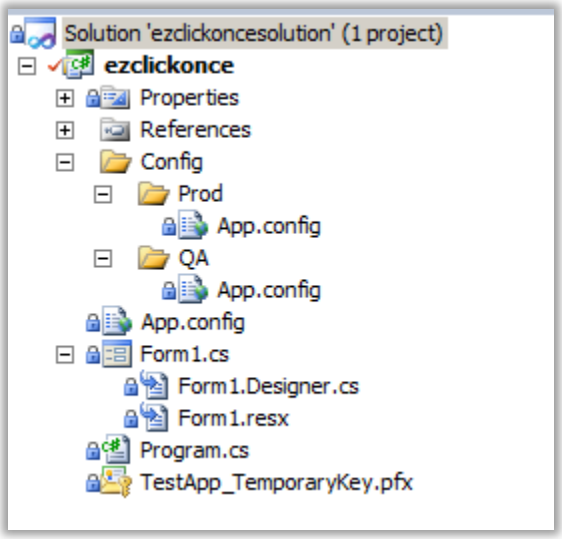


Figure 183 – Solution with multiple configuration files

To update the project XML in Visual Studio, right-click the project file and choose **Unload Project**. Then right-click the project and choose **Edit *projectname*.csproj** to open project as XML.

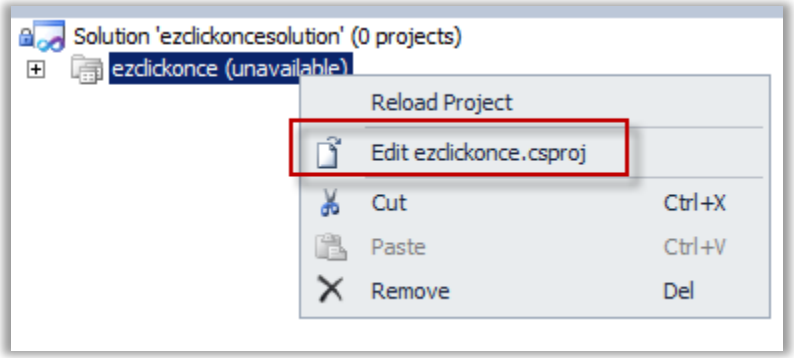


Figure 184 – Edit Project XML

In the XML, locate the configuration files that you created for the project. Remove these entries from the XML. They will be replaced by conditional items.

```
</Compile>
<None Include="App.Config" />
<None Include="Config\Prod\App.config" />
<None Include="Config\QA\App.config" />
```

Figure 185 – Remove existing references to the App.config files

Add the following conditional ItemGroup elements to handle replacing the App.config file, based on the configuration option selected.

```
<ItemGroup Condition="'$(Configuration)|$(Platform)' == 'Debug|x86' Or '$(Configuration)|$(Platform)' == 'Release|x86'">
  <None Include="App.config" />
</ItemGroup>
<ItemGroup Condition="'$(Configuration)|$(Platform)' == 'QA|x86'">
  <None Include="Config\QA\App.config" />
</ItemGroup>
<ItemGroup Condition="'$(Configuration)|$(Platform)' == 'Prod|x86'">
  <None Include="Config\Prod\App.config" />
</ItemGroup>
```


Now, when you select the configuration in the **Items to Build** dialog box, in the Build Definition, it will use the environment-specific configuration file in the build.

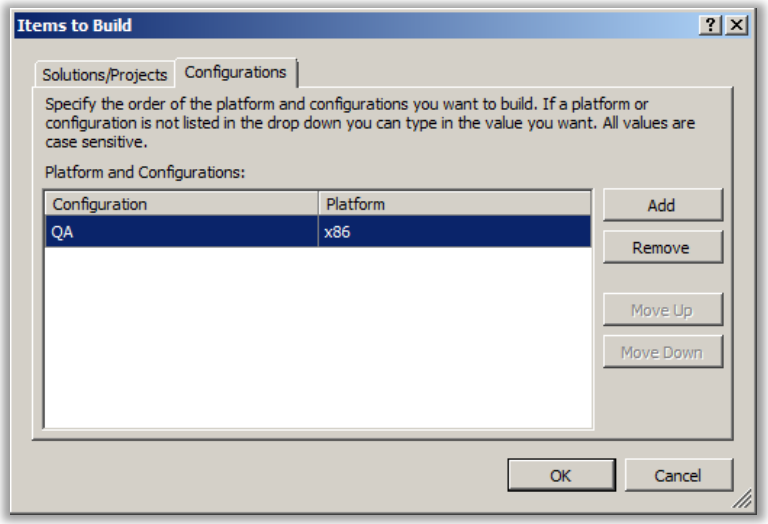


Figure 186 – Configuration determines which App.config is used

Signing ClickOnce Applications

ClickOnce applications can be deployed using two types of signing.

1. **Code Signing** – Code signing is the process of digitally signing assemblies and executable to confirm the software author and guarantee that the code has not been altered or corrupted because it was signed with a cryptographic hash.
2. **Manifest Signing** - To publish an application using ClickOnce deployment, the application and deployment manifests must be signed with a cryptographic x509 certificate. ClickOnce deployment uses the RSA private key encryption scheme; the manifests are signed with the key information from the certificate.

Before you can sign the ClickOnce application and manifest, you must make the certificate available to the build server. The certificate should not be stored in Team Foundation source control. To make the certificate available, add the certificate to the Personal Certificate Store on the build server. Once the certificate is installed to the Personal Certificate Store, it can be referenced by the Certificate Thumbprint Hash as shown in the following. See *How to: Create Temporary Certificates for Use During Development*[171] for more information about creating certificates and adding them to the Personal Certificate Store.

Just like the configuration file changes, the code signing must happen before the manifest is created. Furthermore, it would be difficult to manage the signing of all of the assemblies in the build workflow. This can be dynamically accomplished in the project file by adding the following sections before the </Project> element at the end of the file.

[171] http://msdn.microsoft.com/en-us/library/ms733813.aspx

```
<PropertyGroup>
  <SignClientAssemblies Condition="'$(SignClientAssemblies)'==''">false</SignClientAssemblies>
</PropertyGroup>
<Target Name="SignClientAssemblies" BeforeTargets="BeforePublish" Condition="'$(SignClientAssemblies)' == 'true'">
  <Message Text="Signing Client Assemblies" />
  <CreateItem Include="$(OutDir)\**\*.dll">
    <Output TaskParameter="Include" ItemName="ClientDlls" />
  </CreateItem>
  <SignFile CertificateThumbprint="$(ManifestCertificateThumbprint)" TimestampUrl="$(TimestampUrl)" SigningTarget="%(ClientDlls.FullPath)" />
  <Message Text="Client Assemblies Signed Successfully" />
</Target>
</Project>
```

Figure 187 – Added sections for signing assemblies.

The example above creates two new properties to pass to MSBuild.

- **SignClientAssemblies** – Specify true/false (false is the default) to perform the Code Signing.
- **ManifestCertificateThumbprint** – This property is the sha1 hash code of the certificate from the Personal Certificate Store to use. This is also the same property that ClickOnce uses to sign the manifest. If you need to specify different certificates, simply use a different name above.

These two additional properties can be passed in MSBuild arguments property. This would look like the following:

/T:Publish

/p:PublishDir=\\server\share\myapplication\

/p:PublishUrl="http://localhost:8055/MyApplication"

/p:InstallUrl=http://localhost:8055/MyApplication

/p:SignClientAssemblies=true

/p:ManifestCertificateThumbprint=C01D1C7C90B5BC2D60ABFBEA746497B4D1EAF578

Functionally, this will perform all of the ClickOnce options we need to do for a deployment. However, with some minor modifications to a copy of the default build template, these MSBuild properties could be exposed as properties to the build template and create a cleaner solution. Other areas of this guidance show how to extend a build template and create a build definition. Here are the specific changes for ClickOnce.

Add the following Arguments and modify the Metadata with friendly names and ClickOnce group:

- PublishDir
- PublishUrl
- InstallUrl
- SignClientAssemblies (boolean)
- ManifestCertificateThumprint

Modify the **Run MSbuild for Project** under Process > Sequence > Run on Agent > Try Compile, Test, and Associate Changesets and Work Items > Sequence > Compile, Test, and Associate Changesets and Work Items > Try Compile and Test > Compile and Test > For Each Configuration in BuildSettings.PlatformConfigurations > Compile and Test for Configuration > If BuildSettings.HasProjectsToBuild > For Each Project in BuildSettings.ProjectsToBuild > Try to Compile the Project > Compile the Project

Modify the CommandLineArguments property with the following value:

```
String.Format("/p:SkipInvalidConfigurations=true {0} /Target:Publish
/property:PublishDir={1} /property:PublishUrl={2} /property:InstallUrl={3}
/property:SignManifests=true /property:SignClientAssemblies={4}
/property:ManifestCertificateThumbprint={5}", MSBuildArguments, PublishDir, PublishUrl,
InstallUrl, SignClientAssemblies, ManifestCertificateThumbprint)
```

The end result looks like the figure below. This template can be found with the deliverables for the guidance.

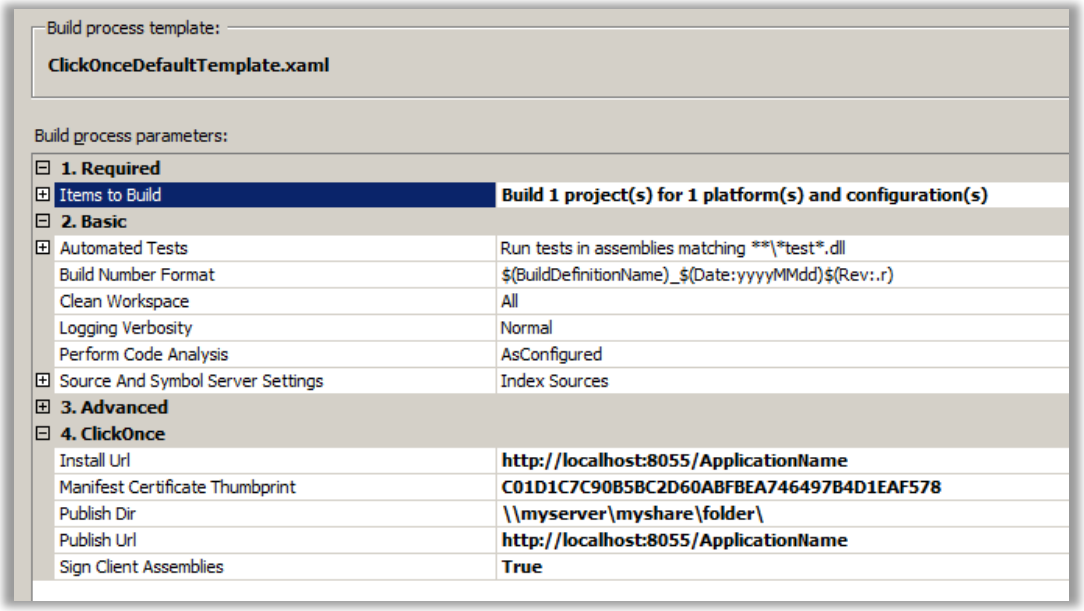


Figure 188 – Final build definition for ClickOnceDefaultTemplate

ClickOnce Versioning

The version number for ClickOnce applications specifies the version of the deployment package. Within Visual Studio this number can be configured to auto increment with each publish. However, when you create the ClickOnce deployment outside of Visual Studio, this must be manually set. Versioning of the application has already been discussed in this guidance. See *Versioning Assemblies* section for specific information about versioning the application and assemblies. The ClickOnce version should be in sync with the AssemblyFileVersion and incremented with each build. However, one feature of ClickOnce is that it can be configured to only download files that have changed. If all of the assembly versions are incremented, then all of the files in the ClickOnce deployment manifest will be re-downloaded. This could seem excessive or unnecessary but when you are building solutions with multiple projects, it is a good practice to rebuild and increment the version of all assemblies in order to keep the deployment in sync.

To specify the version for the ClickOnce deployment package, specify the version using the ApplicationVersion property. This can be applied to the MSBuild arguments. In the example below, the ApplicationVersion property is being specified to set a specific version.

```
String.Format("/p:SkipInvalidConfigurations=true {0} /Target:Publish
/property:PublishDir={1} /property:PublishUrl={2} /property:InstallUrl={3}
/property:ApplicationVersion={4} /property:SignManifests=true
/property:SignClientAssemblies={5} /property:ManifestCertificateThumbprint={6}",
MSBuildArguments, PublishDir, PublishUrl, InstallUrl, Version, SignClientAssemblies,
ManifestCertificateThumbprint)
```

To calculate the build number base on date or days since last release, update the version numbers of all of the assemblies, and set the Version variable to update the ClickOnce Application. In addition, use the TfsVersion Activity in the Community TFS Build Extensions[172] project. The sequence can be placed just before the "Try Compile, Test, and Associate Changesets and Work Items" sequence.

[172] http://tfsbuildextensions.codeplex.com/

The first Activity below in the Version Assemblies sequence is the FindMatchingFiles Activity. This is set to find the AssemblyInfo.cs files.

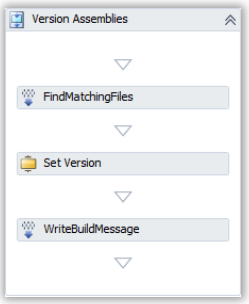


Figure 189 – Version Assemblies Sequence

Next the TfsVersion Activity is set to use the date format MMdd for the version. This can be set to almost any format, however, the ClickOnce version is more limited in digits. Illustrated below are all of the settings with the highlighted fields indicating those that were updated.

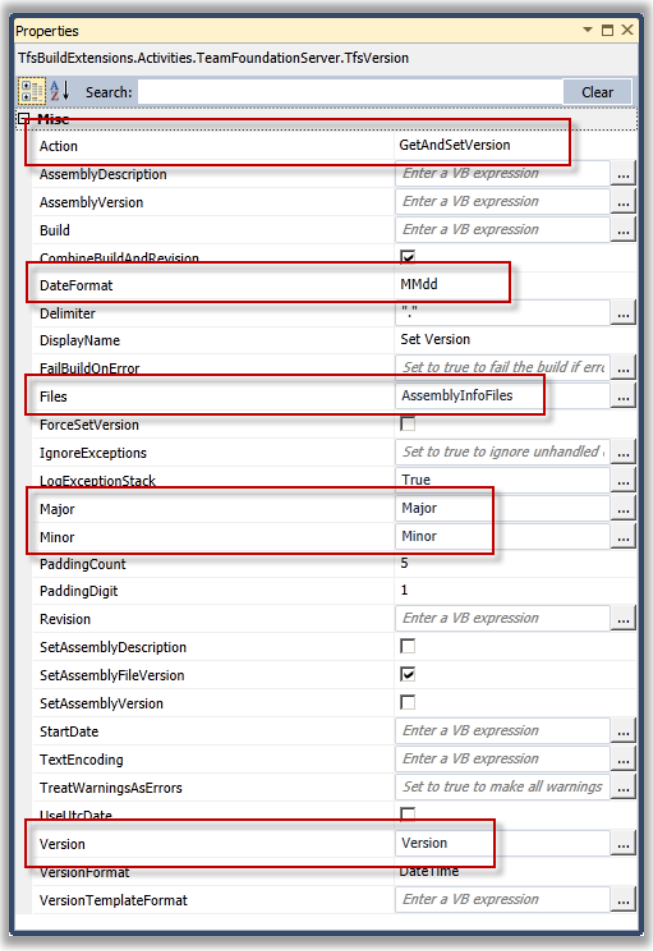


Figure 190 – TfsVersion settings

Fields set in TfsVersion Activity include:

- **Action** – GetAndSetVersion specifies that this activity should calculate the version and output the version number.

- **DataFormat** – Field that specifies the format of the versioning.
- **Files** – This indicates the array of AssemblyInfo files to be updated.
- **Major** and **Minor** – These are the static version numbers that are passed in through the Build Definition process settings.
- **Version** – This is the output variable that will be used for the ClickOnce ApplicationVersion.

With these settings, the version number will be something like: 1.5.10108.101082 when the Major is set to 1 and the Minor is set to 5. Illustrated below is the final process definition for ClickOnce with Versioning.

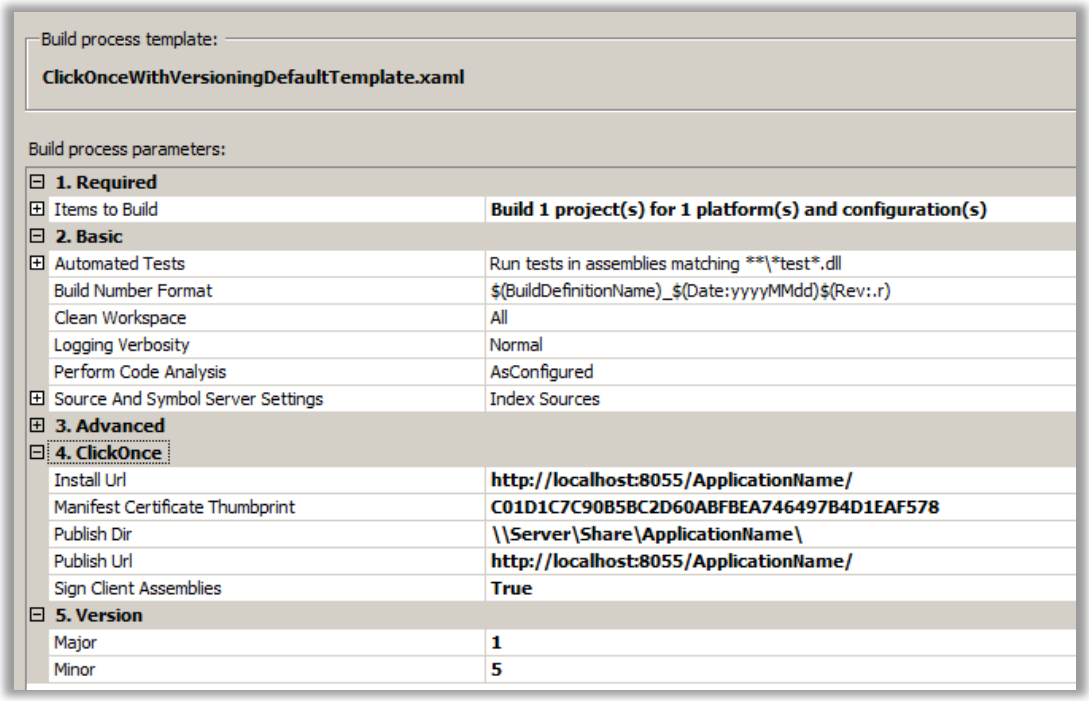


Figure 191 – Final ClickOnceWithVersioningDefaultTemplate Process Settings

ClickOnce Deployment Build Activity

The advanced solution is a custom ClickOnce build activity. The ClickOnce Deployment activity is available in the Community TFS Build Extensions[173] project. The primary benefit of the ClickOnce build activity is that it offers the most flexibility. Most of the options available from Visual Studio 2010 are available to be set through this build activity. The following table shows properties that are available to be set in the activity.

Property	Is Required	Description
MageFilePath	True	Local location of the mage.exe. Included in the .NET SDK
Version	True	Version of the ClickOnce deployment package. Can be independent of the Application version.
BinLocation	True	Source location for files to be included in the ClickOnce deployment package.
ApplicationName	True	Name of Application (without .exe)

[173] http://tfsbuildextensions.codeplex.com/

Property	Is Required	Description
CertFilePath	False	Local path and filename to the certificate file (pfx).
CertPassword	False	Password for the certificate file.
ManifestCertificateThumbprint	False	Certificate Hash that is stored in the certificate store.
PublishLocation	True	Publish location for ClickOnce Deployment package. This can be different than the InstallLocation.
InstallLocation	True	Public URL or UNC where application is going to be installed.
Publisher	True	Name of Publisher.
OnlineOnly	True	Determines if application is to be only used online. Offline applications will install a shortcut.
TargetFrameworkVersion	True	Target .NET Framework version. IE 4.0

Table 32 – ClickOnce Deployment Activity Properties

To use the build activity, see the Using the Community TFS Build Extensions section. This will show you how to reference the Activity's assembly and begin using all of the activities including the ClickOnceDeployment Activity.

The ClickOnceDeployment Activity requires the Manifest Generation and Editing Tool (mage.exe) found on MSDN[174]. Mage.exe is part of the .NET 4.0 SDK and can be found here[175].

In the figure below, the ClickOnceDeployment activity has been added to a copy of the Default Template.

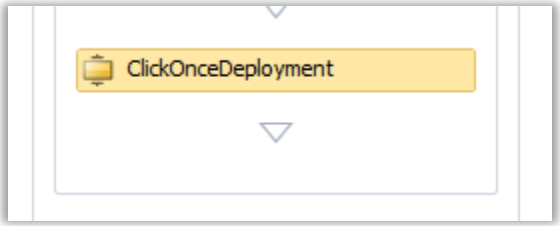


Figure 192 – ClickOnceDeployment Activity

Below are examples of values for each of the properties for the ClickOnceDeployment Activity. These properties could be set by Arguments in the template or other activities.

[174] http://msdn.microsoft.com/en-us/library/acz3y3te(v=VS.100).aspx

[175] http://www.microsoft.com/download/en/details.aspx?id=8279

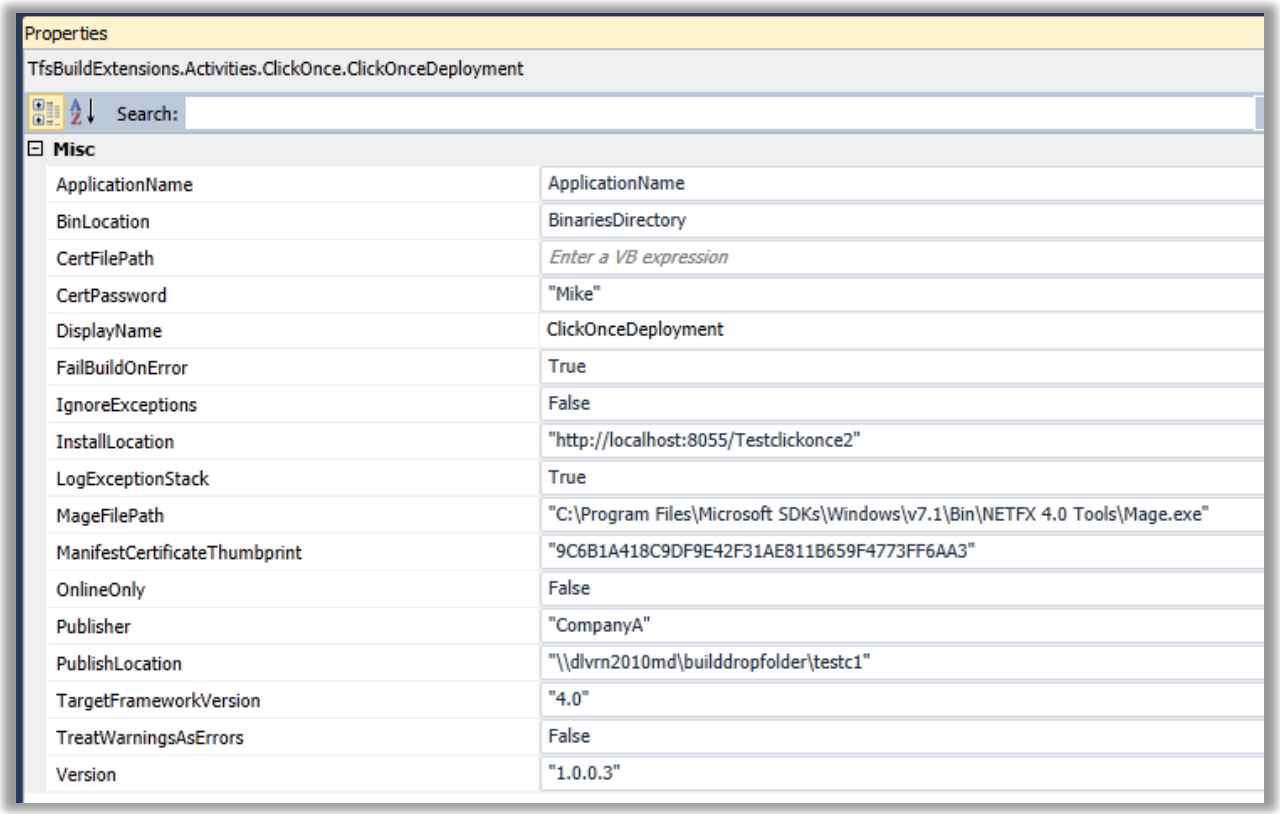


Figure 193 – ClickOnceDeployment Activity Properties

Which ClickOnce option to choose?

This section provided two separate approaches for building ClickOnce applications. Each approach has different advantages. Refer to the table below to help decide which approach is the best solution for your particular needs.

ClickOnce Default Template Advantages	ClickOnce Custom Assembly Advantages
Quicker to implement	Provides more control
Doesn't require any customization	More portable and can be added to any existing build definition
Covers 80% to 90% of scenarios	Exposes most of the options available in Visual Studio

Table 32 – ClickOnce Default Template and Custom Activity Comparison

Silverlight 4 Applications

Silverlight is Microsoft's cross-platform and cross-browser application framework for rich internet applications. It features a rich user interface typically found in client technologies such as Windows Forms and WPF without the complexity of deployment like ASP.NET. Silverlight applications are typically hosted on a web server and delivered to the users through a web browser. The Silverlight plug-in is the only pre-requisite for running Silverlight applications in the browser on machines with Windows, Macintosh, and Linux operating systems.

Silverlight continues to evolve and gain adoption at a tremendous rate. Silverlight 4 is the current version and it supports a number of features for packaging and deploying applications. These features include Out-of-Browser support, utilizing Web Deploy, and calling services from a separate domain. In this section, we will walk through building a Silverlight package and deploying it.

Silverlight Business Application Template

The example application used for the following guidance was created using the Silverlight Business Application Template. This template creates many of the features that might be needed for a business application. This includes navigation and user registration/login pages. The application also utilizes WCF RIA Services for separation of the application logic to the middle tier.

NOTE

Refer to the walkthrough Using the Silverlight Business Application Template[176] on MSDN.

Silverlight Packages

A Silverlight application is compiled to a XAP file (pronounced *zap*). It is essentially a compressed file that contains a number of files to support the Silverlight application. These files include the manifest and required assemblies. Sample contents of a XAP file for Silverlight Business Application are shown in the following illustration:

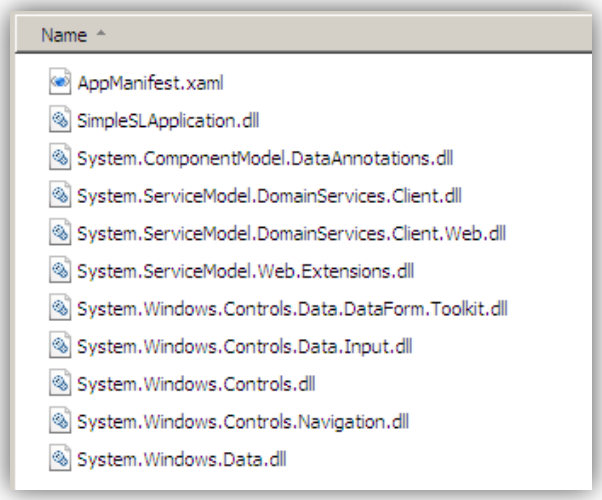


Figure 194 – Example XAP file

[176] http://msdn.microsoft.com/en-us/library/ee707360%28VS.91%29.aspx

Build Silverlight 4 Packages with Web Deploy

In a Silverlight 4 Business Application, the Silverlight application is hosted by a web application that is found in the ClientBin folder. The web application that hosts the Silverlight application can be packaged and deployed using Web Deploy.

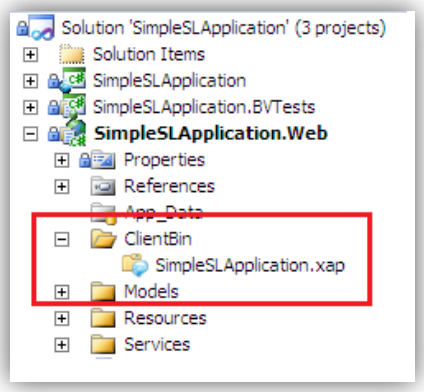


Figure 195 – ClientBin Folder

The Web Deploy package that contains the web application and the Silverlight application can use the same Web Deploy features, such as web.config transformations and deploying to production environments. These features are described in the Package and Deployment sections (page 192). The anatomy of a Web Deploy package that contains a Silverlight application is shown in Figure 189.

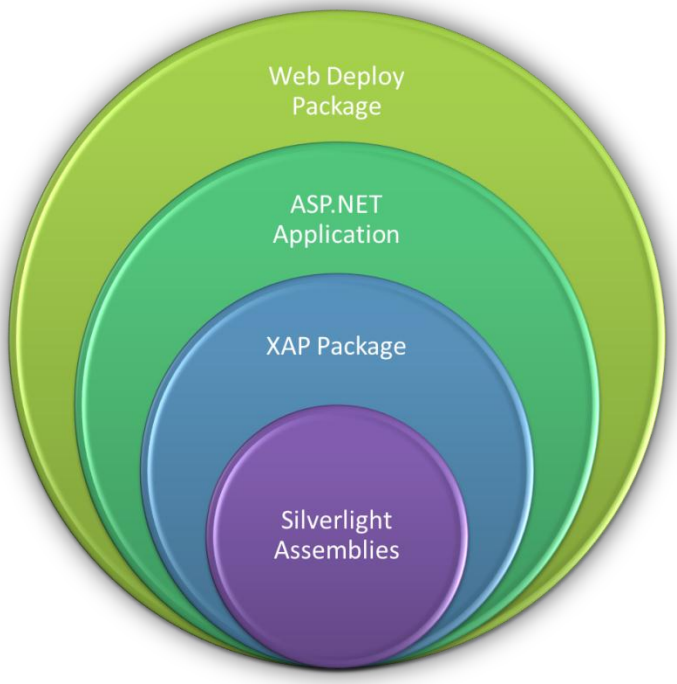


Figure 196 – Silverlight Web Deploy Package

Building Silverlight 4 Applications with Team Foundation Build in Visual Studio 2010

Building Silverlight 4 application on Team Foundation Build requires the following be installed on the build server:

- Visual Studio 2010
- Microsoft Silverlight 4 SDK

If the build server is a 64-bit Server, you might receive this error if the MSBuild platform is set to **Auto**.

> C:\Program Files (x86)\MSBuild\Microsoft\Silverlight\v4.0\Microsoft.Silverlight.Common.targets (104): The Silverlight 4 SDK is not installed.

To resolve the error, under **Process**, **Advanced** settings, change the MSBuild Platform to X86.

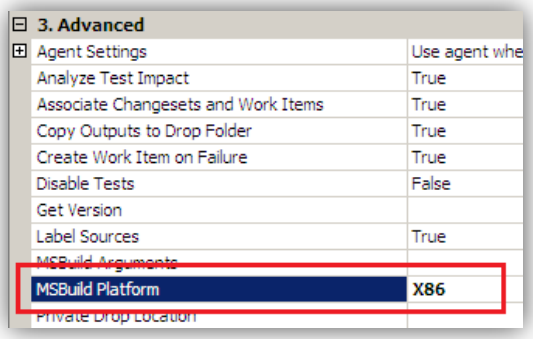


Figure 197 – Setting MSBuild Platform to X86

Deployments

Web Deploy packages containing Silverlight applications can be deployed exactly the same way as applications without Silverlight. See the Web Deployment section (page 192) in the section above for more information. Web servers hosting the Silverlight application may require configuring the MIME Types so the file extensions can be correctly served to the clients. IIS 7 in Window Server 2008 these settings are present. Other platforms and previous versions may require these to be set.

- .xap application/x-silverlight-app
- .xaml application/xaml+xml
- .xbap application/x-ms-xbap

Refer to the article *Configuring IIS for a Silverlight application*[177] for the step-by-step guide about how to configure these settings.

Other Options for Packaging and Deploying Silverlight 4 Applications

The example Silverlight 4 application used in the guidance is hosted by a web application. This web application hosts the RIA services for the Silverlight application and the deployment package is a Web Deploy package. Silverlight includes other options for managing the client configurations, deploying the Silverlight application so it supports Out-of-Browser experience, and supporting calling services across domains. More information about these options can be found in the links below.

Silverlight Client Configuration

Silverlight 4 added web client configuration support, which enables post build configuration. If the Silverlight application requires a service reference to a service, it can use the configuration changes for each environment

[177] http://learn.iis.net/page.aspx/262/configuring-iis-for-silverlight-applications

without having to recompile the application. The article Configuring Web Service Usage in Silverlight Clients[178] describes how to set up client configuration settings for the Silverlight application.

The Silverlight Business Application Template manages the reference between the web application and the Silverlight client without using the client configuration.

Out-of-browser Support

Silverlight now supports running Silverlight application outside of the Browser. These applications can still be installed from the web browser but can be run outside of the browser after it is installed. The article Out-of-Browser Support[179] describes configuring an application for Out-of-browser support.

Calling Cross Domain Services

By default, Silverlight does not allow cross-domain services to be called because of possible security threats, including cross-site forgery exploits. However, there is an opt-in method for enabling this for specific domains. The article Making a Service Available Across Domain Boundaries[180] explains the two options for accomplishing this.

[178] http://msdn.microsoft.com/en-us/library/cc197941(v=vs.95).aspx

[179] http://msdn.microsoft.com/en-us/library/dd550721(v=vs.95).aspx

[180] http://msdn.microsoft.com/en-us/library/cc197955(v=vs.95).aspx

WCF REST Web Service

This section describes how to add a WCF REST Web Service to a solution and how to solve the deployment issues for the web service as well.

With .NET Framework 4, it has become very easy to build REST-ful Web Services. A REST Web Service will typically be hosted in a web application because it should be exposed as a plain HTTP service. Building the service is no different from building a Web Application and the deployment is very similar to Web Application deployment. The key challenge in the deployment of a WCF REST Web Service is handling the service configuration correctly. In the following sections, we will look at how to do that.

To add a WCF REST Web Service to your solution, use the WCF REST Service Application project template available online. You can download the template easily by using the Visual Studio Extension Manager:

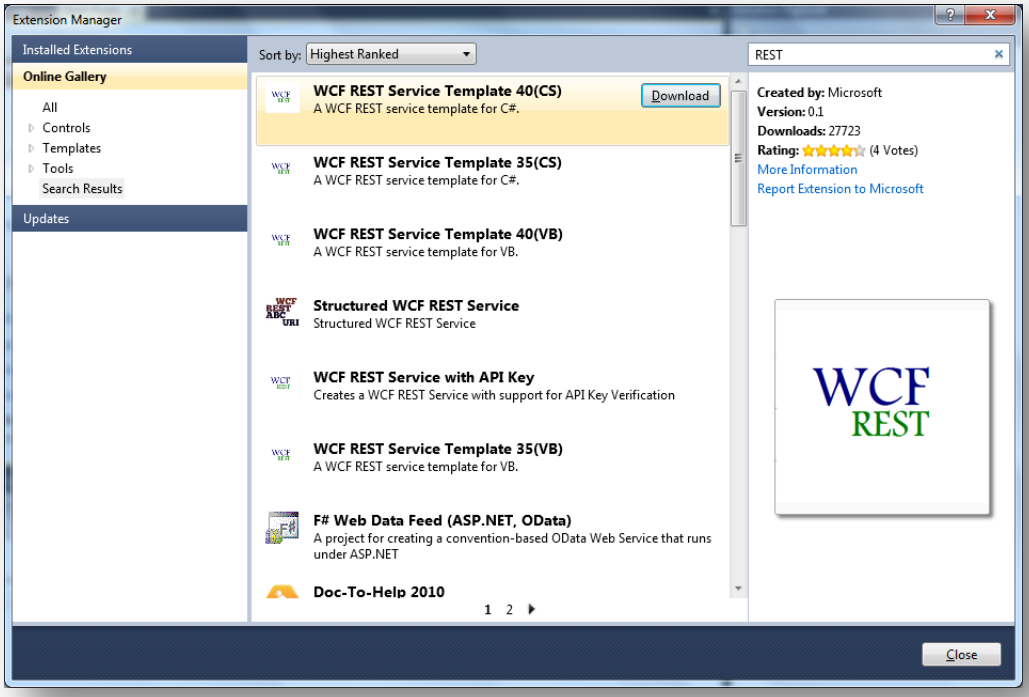


Figure 198 – WCF REST Service Application project template

Then create a new project based on the REST template:

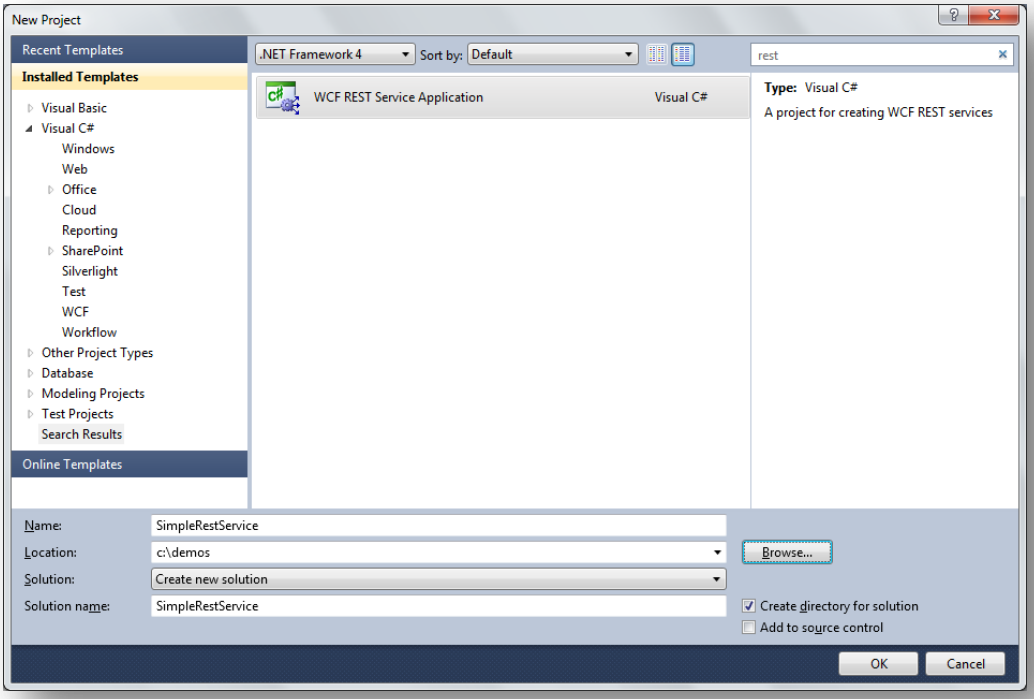


Figure 199 – WCF REST Service Application

The new project will have a very basic structure similar to the one below (note that there is no .svc file for the WCF Web Service anymore!):

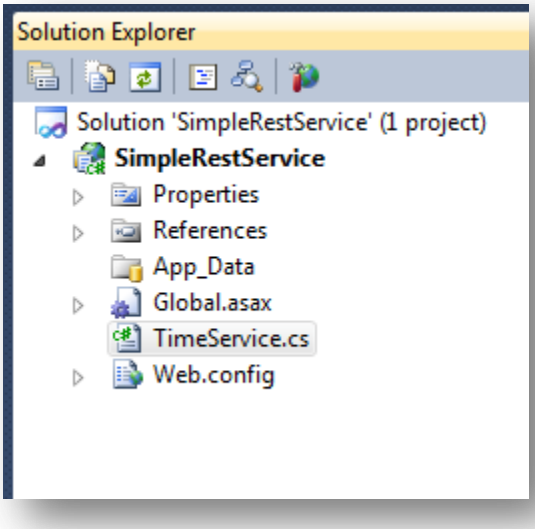


Figure 200 – WCF REST Service Application project structure

The following code (from TimeService.cs) implements a simple REST Web Service that returns the current time:

```
[ServiceContract]
[AspNetCompatibilityRequirements(RequirementsMode =
                                 AspNetCompatibilityRequirementsMode.Allowed)]
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
public class TimeService
{
    [WebGet(UriTemplate = "CurrentTime")]
    public string CurrentTime()
    {
        return DateTime.Now.ToString();
    }
}
```

Deploy Web Service to Web Application

Before you can start deploying your Web Application or Web Service, you need to have a configured IIS Web Application in place. Refer to page 191 for more information.

The deployment of a WCF REST Web Service can be very straightforward. A WCF REST Web Service is typically built as a standard Visual Studio Web Application and can be deployed by simply copying the content of the _publishedWebSite folder to the target environment as a part of the build process. However, for a more flexible solution, use of the Web Deploy tool is preferred. = By using Web Deploy, you can customize what is deployed in a good way.

A very nice improvement for WCF Web Services deployment is that the WCF configuration has been reworked in .NET Framework 4. The WCF configuration files have historically been complex to maintain and this, of course, can cause problems during deployment to different environments as well. With .NET Framework 4, there is no need for explicit service files (.svc) and the Web.Config file does not have to define the services anymore. Instead, the hosting environment (IIS in this case) will automatically create endpoints for each of the contracts implemented by the service. More information about the simplified WCF configuration files can be found in the MSDN article "Simplified Configuration".

The following Web.Config shows how little you need to configure to get a standard WCF REST Web Service working:

```
<?xml version="1.0"?>
<configuration>
  <system.serviceModel>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>
    <standardEndpoints>
      <webHttpEndpoint>
        <standardEndpoint name="" helpEnabled="true"
                          automaticFormatSelectionEnabled="true"/>
      </webHttpEndpoint>
    </standardEndpoints>
  </system.serviceModel>
</configuration>
```

See page 192 for more information about how to use the Web Deploy tool to publish a Web Site during the build and deployment process.

Using Web.Config transforms to configure target environments

Even if the way to handle WCF configuration has been simplified in .NET Framework 4, you might still need to make changes to the configuration settings. In a standard WCF REST Web Service, we can use the simplified configuration that allows us to provide just the non-default settings for our services, which makes much of the configuration generic. If you do need different settings for your target environments, then using configuration specific Web.Config files together with the Web.Config transformation feature in Web Deploy can be used to solve this gracefully. Simply add a configuration file for each environment and add the logic to alter the master configuration in each specific file to add, change or remove items as needed.

Below is a simple Web.Config transform example that removes publishing Web Service metadata (enabled with helpEnabled=”true” in the master config file):

```
<?xml version="1.0"?>
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
  <system.serviceModel>
    <standardEndpoints>
      <webHttpEndpoint>
        <standardEndpoint xdt:Transform="RemoveAttributes(helpEnabled)" />
      </webHttpEndpoint>
    </standardEndpoints>
  </system.serviceModel>
</configuration>
```

See page 194 for more information about how to work with Web.Config transformations in the deployment process.

For more information, refer to:

- Overview of Web.Config Transformation[181]
- Web Deployment Tool Installation[182]

[181] http://msdn.microsoft.com/en-us/library/dd465326.aspx

[182] http://technet.microsoft.com/en-us/library/dd569059(WS.10).aspx

Running automated integration tests during build in the Integration Environment

This section covers the items that are required to execute tests in the integration environment as part of a build. Setup of the integration environment, along with the activities required to deploy the application, are covered in their respective sections.

Background Information / Rationale

Although running Unit Tests as part of a build is quite common, there are limitations imposed by the standalone nature of this environment. The ability to test a project in an environment, which is integrated with the other components that comprise the entire solution, provides significant advantages in creating "real world" scenarios. The ability to integrate this level of testing with continuous integration (CI) or daily builds greatly reduces the amount of manual work required.

Scenarios

Common Features

Each of the scenarios covered in this section make use of a subset of the environment described in "Deploy Environments" on page 176. The key elements are shown in the diagram below:

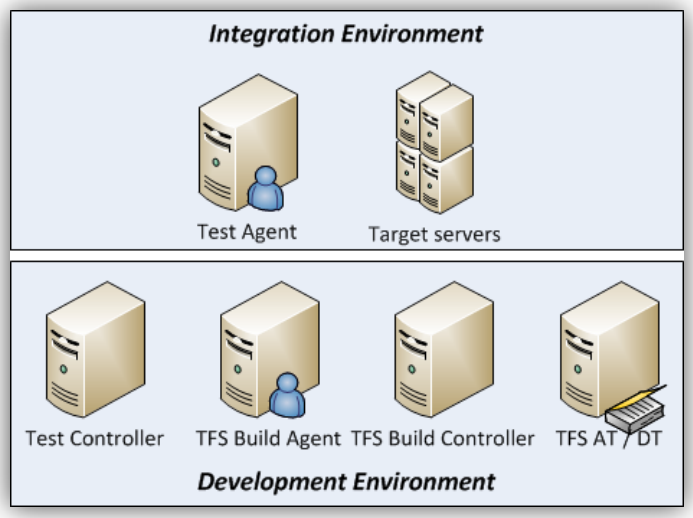


Figure 201 – Common Scenario Environments

When an appropriate build is requested, the Build Controller will select an available Build Agent to perform the build. In addition to the steps normally performed by a build, the result needs to be deployed to the appropriate machines in the Integration Environment (each of which have a Test Agent running) and a test initiated from the Test Controller.

The instructions for deploying the application are contained in Build and Deploy ASP.NET Web Application in Integration and QA Environments on page 192.

Introduction

When you run integration tests, you want to test the complete chain of all the components that your application depends on. Traditionally, we regularly verify the correctness of the complete chain by running a series of tests. This series of tests is called the *regression set*. The goal of this section is to show you how you can automate the execution of this regression set, so you can execute the verification more often. For example, you could verify every night, or even after every check-in.

To be able to automate your tests, you need to perform the following tasks, which will be described in more detail in the next sections:

1. Set up Team Foundation Server test components
2. Set up the environment in the Microsoft Test Manager
3. Set up the test settings in the Microsoft Test Manager
4. Create the regression set
5. Create the build to execute the regression set

We will consider the most interesting scenario: test executed against the user interface (CodedUI); you may leverage existing unit testing technology to execute code-only integration tests.

Set up Team Foundation Server test components

Visual Studio 2010 introduced a new application called the Microsoft Test Manager. This is available to you when you have the Ultimate or Test Professional version. We are using the Microsoft Test Manager to execute our tests. To set up the test infrastructure, there are two components that you need to install: the test controller and the test agent. The test controller is used to orchestrate the test execution, and the test agent is used to run the actual test and do the data collection.

You can install the test controller on any machine. After the installation, you get the option to configure the test controller. During this configuration, you specify the identity on which the controller is running and you can bind the test controller to the project collection. If you want to use the test controller from within the Visual Studio IDE, you need to leave the test controller unbound, but if you want to use it from the Microsoft Test Manager, bind it to your active project collection.

You install the test agent on any machine where the CodedUI tests are executed, thus being the front end for our test. Make sure that you set up to run the test agent as an interactive process so that you can execute a CodedUI test. Install the test agent software on every server that is part of your Integration Environment from which you want to collect the data. We will discuss this in more detail later in this document.

Your infrastructure should look like this:

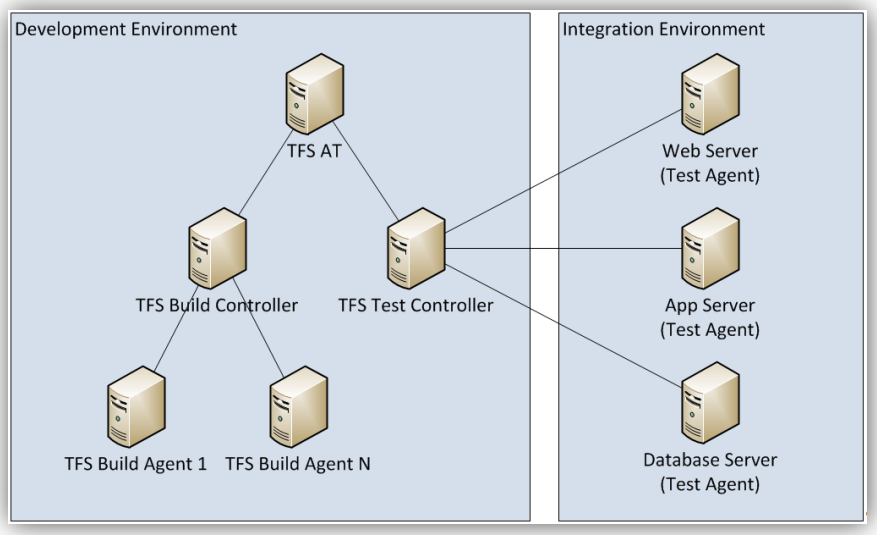


Figure 202 – Sample Development / Integration Infrastructure

Set up the environment in the Microsoft Test Manager

When you have finalized the steps of installing the prerequisite software, you can set up the environment in the Microsoft Test Manager. We use the physical environment option in the Lab Center of Microsoft Test Manager to

build this environment. Go to Lab Center and then create a new physical environment. Walk through the wizard to finalize the creation of your environment. You can find a detailed explanation in the Hands-on Lab.

Create the regression set

The next step in the progress is to create a regression set, which is a test plan in the Microsoft Test Manager. In the build (see next section) you specify which test plan you want to execute. To this test plan, you add all the test cases that you want to add to the regression set. The build executes every automated test case in the test plan. To automate a test case, you need to bind a CodedUI test to the test case.

Figure 203 – Associating a CodedUI Test

To bind a CodedUI test to a test case, open the test case in Visual Studio. Click the ellipsis button (…) and select the CodedUI test that automates the test case work item.

Set up the test settings in the Microsoft Test Manager

By default, the Test Manager has some default test settings for a manual test, but has nothing configured for the test automation. Before you can run the automated tests, you need to set up these settings. Go to the properties of your test plan in the Microsoft Test Manager. Select the created physical environment in the Automated Environment drop-down list.

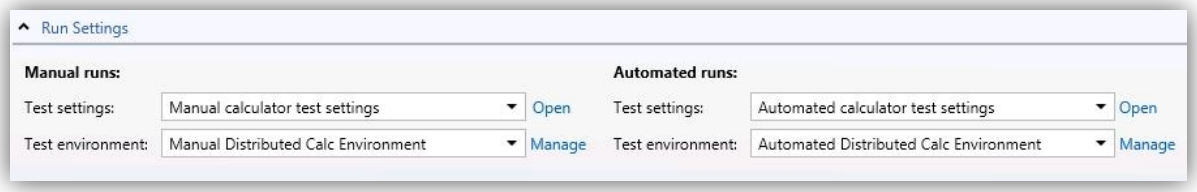


Figure 204 – Test Settings

You can also change the test settings to your own wishes by clicking the **Manage** button.

Create the build to execute the regression set

The last step is to create a build to execute the regression set. Refer to the sample XAML file **BuildDeployTest_Physical.xaml** that is included with this guidance. In addition, there is the Build Process Template, which you need to install on your Team Foundation Server to be able to execute the steps in this section. Installing the template is as easy as adding the XAML file to source control (see “Build process template customization” on page 23).

The prerequisites for this section are that you have a successful build definition that compiles your application and a deployment script, which ensures that the application is installed into the integration environment. This deployment script needs to be unattended.

When you are done with the prerequisites, you can create a new build definition and use the new Build Process Template (see "Deploying the customized template" on page 74 for details on how to activate it).

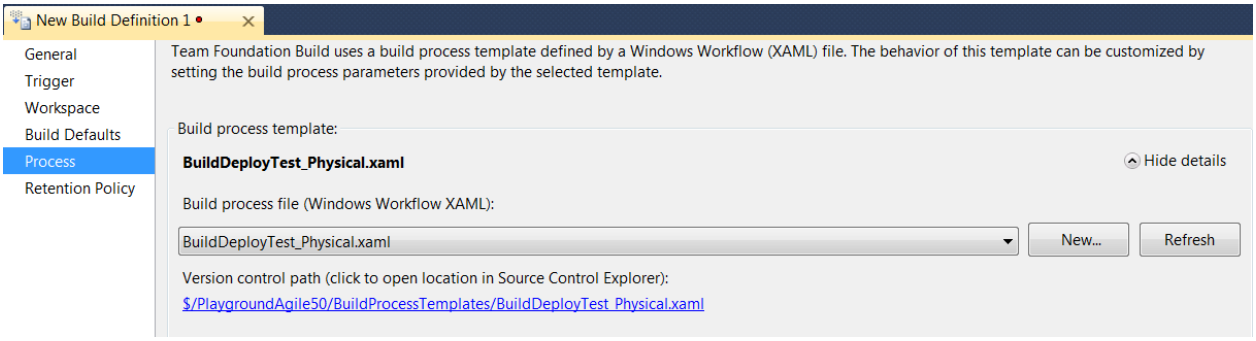


Figure 205 – Physical Build and Deploy Process Template

In the **Process** tab of your build definition, you can specify the three topics for the build. In the section "2. Build: New build" enter the name of the build definition you have created to compile your application.

NOTE

If you choose to queue a new build, make sure that the build controller that you are using has more than one build agent or that the property "Maximum number of concurrently running builds" of the build controller is set to a value greater than 1.

In the section "4. Deployment" enter the deployment scripts you want to execute. Use the format "Build Agent # Path to deployment script # Arguments".

In the section "5. Test" specify the test plan and the test settings you want to use when executing the test.

Supporting References

- Walkthrough: Install Test Controller and Test Agents for Visual Studio Automated Tests[183]
- Managing Test Controllers and Test Agents[184]
- How to Create a Custom Workflow Activity for TFS Build 2010 RTM[185]
- Running Build-Deploy-Test Workflow on physical environments[186]
- How to: Configure and Run Scheduled Tests After Building Your Application[187]

[183] http://msdn.microsoft.com/en-us/library/ff469838.aspx

[184] http://msdn.microsoft.com/en-us/library/dd695837.aspx

[185] http://blogs.msdn.com/b/jimlamb/archive/2009/11/18/how-to-create-a-custom-Workflow-activity-for-tfs-build-2010.aspx

[186] http://blogs.msdn.com/b/lab_management/archive/2011/02/15/running-build-deploy-test-Workflow-on-physical-environments.aspx

[187] http://msdn.microsoft.com/en-us/library/ms182465.aspx

Production Deployments

The ultimate goal of any development team is to successfully deploy their code changes to production. Sound Software Configuration Management (SCM) practices are especially important for production deployments. These practices include documenting, scheduling, coordinating, and approving releases to go to production.

At this point, the development team has controlled build and deployments in Developer, Service, and Integration environments. The infrastructure team has performed the deployment to the QA environment. As much as possible, the development team's deployments should be similar to production deployment. The deployment to QA should be an exact practice run to Production.

Web Deploy

Web Deploy is a key component to providing good SCM practices for web applications. When the development team builds a release, a package is built for each environment. See Build and Deploy ASP.NET Web Application in Integration and QA Environments on page 192 for more information.

Each package contains the particular configuration settings for that environment, managed and merged with the primary web.config through Web Deploy's web config transformation feature.

By using a deployment package such as the Web Deploy package, it helps avoid several pitfalls that are common in deployments. These include:

- Deployment steps that include manually copying config files from the virtual directory before copying the new release to the folder.
- Manually merging or editing the web.config file with updates for production.
- Updating a single web page or assembly that is "supposed to be" the only thing that has changed.

Manual, Scripted, and Automated Deployments to Production

Traditionally, someone outside of application development who is usually part of the infrastructure team performs deployments to production. This means steps need to be clearly described and documented. Manual deployments are error prone. It is human nature to make a mistake or be over confident in the deployment and not read it word for word, possibly missing a step.

Scripted deployments take these manual steps and automatically execute them, usually as part of a scripting language such as PowerShell, Batch, or VBScript. PowerShell is growing in popularity. It combines the benefits of scripting with a programming language. of the following example is a simple PowerShell deployment script based on Web Deploy tool.

```
# script configuration
$webDeploySource = "\\web.test.contoso.com\c$\Program Files\IIS\Microsoft Web Deploy"
$drop = "\\files01.internal.contoso.com\Drop\Main.QA-SimpleMvcApplication"
$package = "\_PublishedWebsites\SimpleMvcApplication_Package\SimpleMvcApplication.zip"

# select build
Write-Host "Choose the build to deploy:"
$choices = @(Get-Item $drop\* | foreach { $_.Name })
for($i=0;$i -lt $choices.Length;$i++) {
    Write-Host "$($i+1) => $($choices[$i])"
}
$i = Read-Host "?"
Write-Debug $i
$packagePath = Join-Path $drop $choices[$i-1]
$packagePath = Join-Path $packagePath $package
Write-Debug $packagePath

# requirement for tempAgent
netsh firewall add portopening TCP 80 "HTTP"
```



```
# copy tempAgent from remote machine
Copy-Item -Path $webDeploySource -Destination "$Env:USERPROFILE\Desktop" -Recurse -Force

# deploy using tempAgent
Set-Location "$Env:USERPROFILE\Desktop\Microsoft Web Deploy"
.\MSDeploy.exe -verb:sync -source:package="$packagePath" -dest:auto`,tempAgent=true

# remove tempAgent
Set-Location "$Env:USERPROFILE\Desktop"
Remove-Item "$Env:USERPROFILE\Desktop\Microsoft Web Deploy" -Recurse -Force

# quitting
Read-Host "Deploy completed, press RETURN to exit..."
```

A Web Deployment package can contain a number of settings and options that would traditionally be manual steps.

Automated deployments should leverage the same scripts and add the ability for these scripts to be launched from a central location and executed remotely on the destination machines.

There are several challenges to creating automated deployments into production. These include:

- Production Servers should not have “deployment” plumbing applications installed. This means Visual Studio Agents or Web Deployment Agents should not be installed.
- Team Foundation Builds are centralized and convenient for application-development project teams. Infrastructure teams that are performing the deployments would not usually use Team Foundation Server. In addition, the Team Foundation Server administrators who are often members of the development team would have rights, by default, to start these deployments.

Using Web Deploy for Production Deployments

For the other environments, as described in other sections of the document, Web Deploy deployments connect to the remote service to control the installations. Because one of the recommendations for deploying to production environments is that there are no agents, the deployments would need to be deployed by some other manner, without agents. Fortunately, Web Deploy includes an option called Web Deploy on Demand that does not require the Web Deployment tool be installed. When you use Web Deploy with this option, it temporarily installs a tempAgent on demand to install the package. Although this is necessary for production deployments and there are advantages to this, the following conditions need to be met before you use the tempAgent option:

1. The user initiating the Web Deploy command must be a local administrator on any remote source or destination machine being used.
2. TCP Port 80 must be open on the destination machine. Typically, this is already enabled on servers with IIS.
3. Windows Management Instrumentation (WMI) must be configured so the service is running on any remote source or destination machine and the firewall allows WMI traffic.
4. The Web Deployment Agent Service must not be installed on the remote destination machine.

NOTE

For additional details about these conditions and the steps to change the settings on the machines to meet these conditions, see Web Deploy on Demand in TechNet at http://technet.microsoft.com/en-us/library/ee517345(WS.10).aspx.

To use Web Deploy on Demand with the tempAgent, the Infrastructure team member who performs the deployments does not need to install anything to call the command. The command can also be run locally from a machine or executed from a network share. These two options only require several files be copied from an existing installation of Web Deploy or extracted from the MSI.

NOTE

For the details of how to acquire the files required to use Web Deploy, see Web Deploy on Demand[188] on TechNet.

To call Web Deploy on Demand with the tempAgent, use the following command. This example deploys the production deployment from deployment packages folder to the production server.

```
msdeploy -verb:sync -source:package="simplewebapp.zip" -
dest:auto,computername=EPS45WR01,tempAgent=true,username=administrator,password=password
```

Web Deploy and PowerShell

Web Deploy and PowerShell are both powerful tools for deployments and scripting. However, using these two technologies together can be challenging. Web Deploy and PowerShell can be used together two different ways. PowerShell can execute Web Deploy by calling the MSDeploy.exe directly like a scripting language. It can also be called through the Web Deploy API. A package created by Visual Studio includes the manifest that includes other providers that will be executed as part of the deployment. Because of the manifest, the only option for –dest is auto and therefore calling the MSDeploy.exe directly is simpler. If you want to take advantage of some of the more advanced features and have more control, here is more information about the Web Deploy API.

Web Deploy API

If the Web Deployment Tool is installed on the machine in can be referenced by:

```
[System.Reflection.Assembly]::LoadWithPartialName("Microsoft.Web.Deployment")
```

Otherwise, specify the file location of the Microsoft.Web.Deployment.dll and use the LoadFrom method.

```
[System.Reflection.Assembly]::LoadFrom("d:\webdeploy\Microsoft.Web.Deployment.dll")
```

For some examples of how to use the Web Deploy API, see the following Part I[189] and Part 2[190] articles:

Argument	Notes
-verb:sync -source:package="\\DeployServer\DeploymentPackages\SimpleWebApp\Latest\Production\simplewebapp.zip"	The sync operation synchronizes data between a source and a destination. The source:package option specifies that the source is a Web Deployment Tool package.
-dest:auto	The –dest:auto option specifies that the destination on the target server will be the same as the source.
computername= EPS45WR01	Specifies the name/IP Address of a remote computer.
tempAgent=true	Using tempAgent enables synchronization operations to run from a temporary, “on

[188] http://technet.microsoft.com/en-us/library/ee517345(WS.10).aspx

[189] http://blogs.iis.net/jamescoo/archive/2009/09/09/cool-msdeploy-powershell-scripts.aspx

[190] http://blogs.iis.net/jamescoo/archive/2009/10/24/msdeploy-powershell-scripts-part-ii-exceptions-and-remote-server-syncs.aspx

	demand" installation.
username=administrator,password=password	If the current credentials are not a local administrator on the remote source and destination machines, credentials can be passed in.

Table 33 – Web Deploy API Arguments

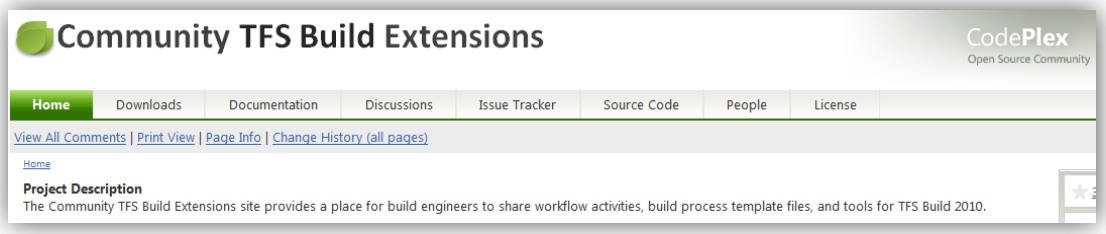
Web Farm Deployments

Many production topologies for web applications include multiple servers that make up a web farm. This is for high availability and load balancing. There are two options when using Web Deploy to deploy to a web farm. One option is to target the Web Deploy deployment to each server in the web farm. The other option is to utilize a new tool called Web Farm Framework. At time of this writing, the Web Farm Framework is in beta. Like Web Deploy, the Web Farm Framework is part of the Web Platform Installer. For more information the Web Farm Framework, see http://www.iis.net/download/WebFarmFramework. Essentially, with the Web Farm Framework, servers are configured for the farm with one of the servers specified as the primary. Then, the Web Deployment Tool only needs to be deployed to the primary server. The other servers in the farm will automatically get the updates. The following example shows a Web Deploy deployment script for deploying the web application to all of the servers in a load-balanced topology.

Reference build template embracing the guidance (BRD Lite)

Introduction

The **B**uild **R**elease and **D**eploy (BRD) Lite is a build process reference template that allows you quickly implement a real-world build process in your environment. Embracing the build customization guidance and community build activities from "Community TFS Build Extensions"[191], it assists you with implementing capabilities such as automatic compile, build version number customization, build packaging, code signing, basic deployment functionality, and environment configuration file management into your Team Foundation builds.



Where do I start?

The following chart helps you decide where to start with the BRD Lite reference build template.

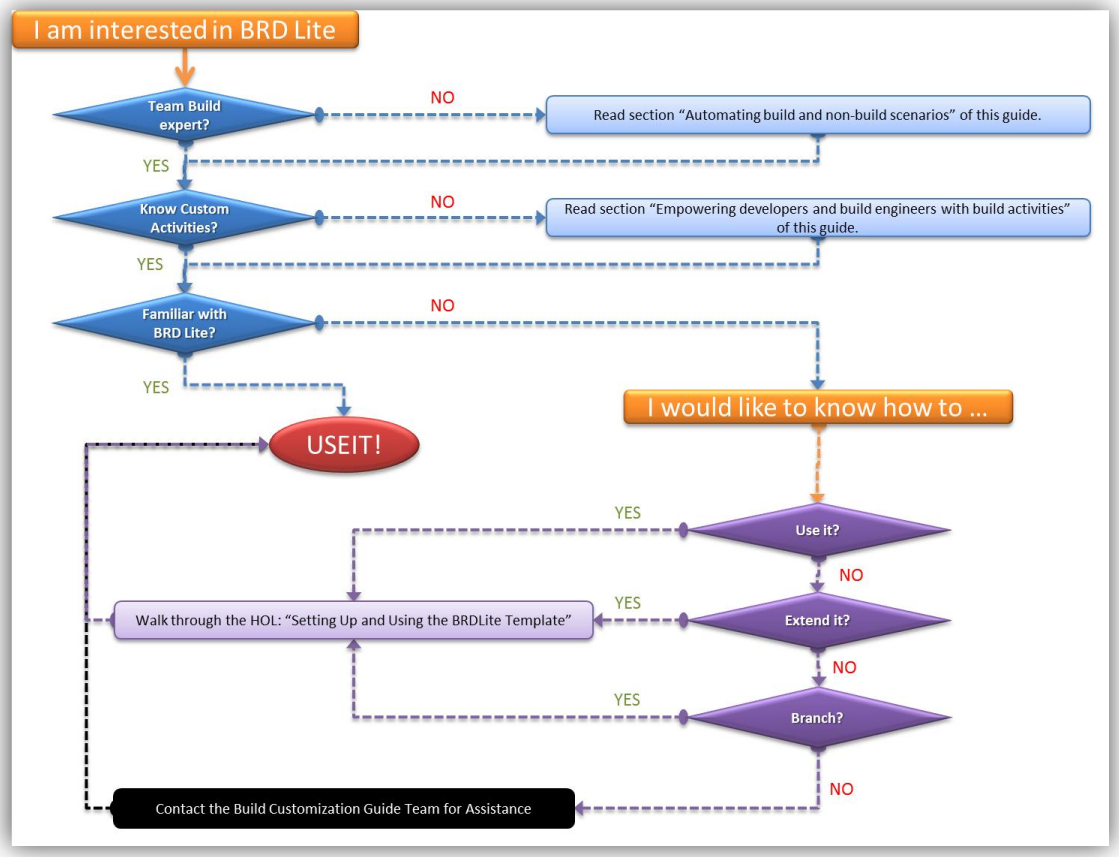


Figure 206 – BRD Lite Decision Chart

[191] http;//tfsbuildextensions.codeplex.com

Features

The scope of the current BRD Lite includes parameterized options at Build Queuing time for:

- Zipping
- Deploy (MSBuild)
- Email
- Digital signing
- StyleCop
- Custom Versioning
- Web Deployment Tool using MSI
- Release Notes

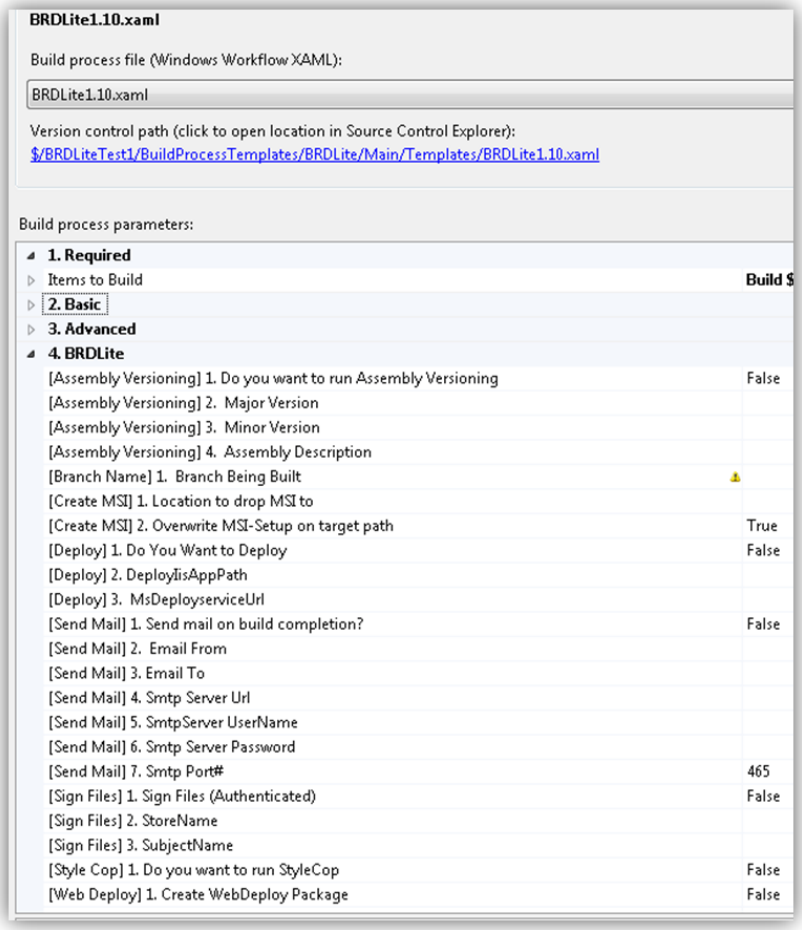


Figure 207 – BRD Lite Build Process Parameters Example

Related Hands-on Labs

- HOL - Setting Up and Using the BRDLite Template

Frequently Asked Questions

Software & Features

Which features are natively supported by Team Foundation Build in Visual Studio?

	Visual Studio 2010	
Automated Feature	**Team Foundation Build Natively Supports**	**What Software to Install to Enable Support**
.NET unit tests	✓	
Code coverage		Premium
Coded UI tests		Premium
Generic tests		Premium
Load tests		Ultimate
Ordered tests	✓	
Web performance tests		Ultimate
Test impact analysis	✓	
Architectural validation	✓	
Database project build/deploy	✓	
VSDBCMD.exe	✓	
Database test data generation	✓	
Database unit tests	✓	
Database code analysis	✓	
Code analysis	✓	
Code metrics		[192]
Publishing test results to Team Foundation Server	✓	

Table 34 – Team Foundation Build Natively Supported Features

[192] Refer **to Visual Studio Code Metrics PowerTool 10.0** for a Power Tool that enables this feature: http://www.microsoft.com/downloads/en/details.aspx?FamilyID=edd1dfb0-b9fe-4e90-b6a6-5ed6f6f6e615

Setup and Configuration

Should we use a common TFSBUILD account for different servers/controllers/machines serving different project collections?

Context: An organization has three team project collections (Frodo, Bilbo, and Sam). Therefore, the organization has (at least) three Team Foundation Build servers/controllers/machines. Rather than run each under a common "TFSBUILD" account, should we be creating a FrodoSvc, BilboSvc, and SamSvc each having their own read/write permissions to the various servers and environment? This would keep team Frodo from writing and running a "malicious" build that accesses team Bilbo's stuff.

Answer: It depends on the non-technical, "political" environment. If there's little trust between groups and it's possible that someone would misconfigure build definitions, agents, etc., to use the wrong resources, then it's worth it, otherwise it's an additional overhead (ACLs, password changes, confusion, etc.). For most operations Team Foundation Build cannot cross the TPC boundary unless the agent is configured to connect to the wrong TPC. Drop folders are a different story.

How can we configure MSBuild to run on multiple cores?

The build service runs a separate app domain for each configured controller/agent. We do not force affinity with a particular processor core; we rely on the operating system. MSBuild, by default, will only use a single processor core. You can override this default by passing the /maxcpucount switch on the command line via the "MSBuild Arguments" parameter of your build definition. For example, to direct MSBuild to use a maximum of two processor cores, you would specify "/m:2" (without the double quotes) in the MSBuild Argument parameter.

How can I split Team Foundation Build Outputs into Folders

See "Splitting Team Build Outputs Into Folders"[193] blog post, by William Bartholomew for more information.

How can I take advantage of source and symbol support when using the Upgrade Template?

1. Add a new argument of type Microsoft.TeamFoundation.Build.Workflow.Activities.SourceAndSymbolServerSettings
   <x:Property Name="SourceAndSymbolServerSettings"
   Type="InArgument(mtbwa:SourceAndSymbolServerSettings)" />
2. As the last thing inside the AgentScope (that is, after all other activities in the AgentScope) add the following (this was taken verbatim from DefaultTemplate.xaml):

```
<If Condition="[SourceAndSymbolServerSettings.IndexSources Or
                SourceAndSymbolServerSettings.HasSymbolStorePath]"
    DisplayName="If SourceAndSymbolServerSettings.IndexSources Or
                 SourceAndSymbolServerSettings.HasSymbolStorePath"
                 mtbwt:BuildTrackingParticipant.Importance="Low">
  <If.Then>
    <mtbwa:InvokeForReason
          DisplayName="Index Sources and Publish Symbols for Triggered Builds"
          Reason="Triggered">
      <mtbwa:InvokeForReason.Variables>
        <Variable x:TypeArguments="scg:IEnumerable(x:String)" Name="symbolFiles" />
      </mtbwa:InvokeForReason.Variables>
      <mtbwa:FindMatchingFiles DisplayName="Find Symbol Files"
            mtbwt:BuildTrackingParticipant.Importance="Low"
            MatchPattern="[String.Format(&quot;{0}\**\*.pdb&quot;,
            If(String.IsNullOrEmpty(BinariesSubdirectory), buildDirectory,
               System.IO.Path.Combine(buildDirectory, BinariesSubdirectory)))]"
            Result="[symbolFiles]" />
      <If Condition="[SourceAndSymbolServerSettings.IndexSources]"
          DisplayName="If SourceAndSymbolServerSettings.IndexSources"
          mtbwt:BuildTrackingParticipant.Importance="Low">
        <If.Then>
          <TryCatch DisplayName="Try Index Sources"
                    mtbwt:BuildTrackingParticipant.Importance="Low">
            <TryCatch.Try>
              <mtbwa:IndexSources DisplayName="Index Sources" FileList="[symbolFiles]" />
```

[193] http://blogs.msdn.com/b/willbar/archive/2011/02/05/splitting-team-build-outputs-into-folders.aspx


```
                </TryCatch.Try>
                <TryCatch.Catches>
                  <Catch x:TypeArguments="s:Exception">
                    <ActivityAction x:TypeArguments="s:Exception">
                      <ActivityAction.Argument>
                        <DelegateInArgument x:TypeArguments="s:Exception" Name="exception" />
                      </ActivityAction.Argument>
                      <mtbwa:WriteBuildError Message="[exception.Message]" />
                    </ActivityAction>
                  </Catch>
                </TryCatch.Catches>
              </TryCatch>
            </If.Then>
          </If>
          <If Condition="[SourceAndSymbolServerSettings.HasSymbolStorePath]"
              DisplayName="If SourceAndSymbolServerSettings.HasSymbolStorePath"
              mtbwt:BuildTrackingParticipant.Importance="Low">
            <If.Then>
              <TryCatch DisplayName="Try Publish Symbols"
                        mtbwt:BuildTrackingParticipant.Importance="Low">
                <TryCatch.Try>
                  <mtbwa:SharedResourceScope DisplayName="Synchronize Access to Symbol Store"
                         mtbwt:BuildTrackingParticipant.Importance="Low"
                         MaxExecutionTime="[TimeSpan.Zero]"
                         MaxWaitTime="[New TimeSpan(1, 0, 0)]"
                         ResourceName="[SourceAndSymbolServerSettings.SymbolStorePath]">
                    <mtbwa:PublishSymbols DisplayName="Publish Symbols"
                           FileList="[symbolFiles]"
                           ProductName="[BuildDetail.BuildDefinition.Name]"
                           StorePath="[SourceAndSymbolServerSettings.SymbolStorePath]"
                           Version="[BuildDetail.BuildNumber]" />
                  </mtbwa:SharedResourceScope>
                </TryCatch.Try>
                <TryCatch.Catches>
                  <Catch x:TypeArguments="s:Exception">
                    <ActivityAction x:TypeArguments="s:Exception">
                      <ActivityAction.Argument>
                        <DelegateInArgument x:TypeArguments="s:Exception" Name="exception" />
                      </ActivityAction.Argument>
                      <mtbwa:WriteBuildError Message="[exception.Message]" />
                    </ActivityAction>
                  </Catch>
                </TryCatch.Catches>
              </TryCatch>
            </If.Then>
          </If>
        </mtbwa:InvokeForReason>
      </If.Then>
    </If>
```

Installation & Deployment

WiX 3.0 has not been updated to be officially "released" with Visual Studio 2010 RTM support

You need to use WiX 3.5 or later.

Are there custom build activities that support building MSIs via WiX?

Currently, there are none. Even though you can use the InvokeProcess or MSBuild activities to execute candle, light, or the other command line tools, the suggested approach is to use WiX projects (.wixproj) which are fully integrated with Visual Studio and MSBuild; this article http://msdn.microsoft.com/en-us/magazine/cc163456.aspx describes how to write an MSBuild script for WiX.

Is there anything special to do to build WiX projects?

Yes, the Build service account needs local admin privileges to generate the MSI package.

What is the best approach to build MSI generated Installers with Team Foundation Build in Visual Studio 2010: Visual Studio Setup Projects, WiX, or Install Shield?

The following table aims at comparing the pros and cons of the three approaches.

Product	Company	Included in Visual Studio	Visual Studio Integration	Team Foundation Build
InstallShield	Flexera Software	No	Yes	Yes / Standalone Build
InstallShield LE	Flexera Software	Yes	Yes	No
vdproj‡	Microsoft	Yes	Yes	Somewhat
WiX	Microsoft	No, free	Yes	Yes

† Deprecated in Visual Studio 2010

See also Choosing a Windows Installer Deployment Tool[194].

There are also other products that creates MSI but are aimed at IT professionals, not developers, like Wise Package Studio Quest Software's or Scriptlogic MSI Studio.

Are there any custom Team Foundation Build activities for Install Shield Limited edition?

Please see http://www.flexerasoftware.com/pl/standalone-build.htm.

Is there guidance for having an automated build that generates an MSI and then automatically deploys that MSI to a remote server and installs (uninstalls previous version if necessary) the application?

The best and suggested approach is to adopt Visual Studio Lab Management. When this is not feasible, you might get some results using WMI. The following script automates the install of an MSI package.

```
$msiPath = "C:\Packages\Sample.msi"
$p = Get-WmiObject -Namespace "root\cimv2" -Query "SELECT * FROM Meta_Class WHERE __Class = 'Win32_Product'" -Impersonation 3 -ComputerName localhost
$rc = $p.Install($msiPath, $null, $True).ReturnValue
echo $rc
```

More information about Win32_ProductWMI class is available on MSDN.

[194] http://msdn.microsoft.com/en-us/library/ee721500.aspx

How can we pass the build number into the MSI as a product version so that we can know what version of the MSI we have built when we install and uninstall?

One solution is to use WiX pre-processor variables. See http://www.ageektrapped.com/blog/setting-properties-for-wix-in-msbuild/.

Another way is to use a bind variable to get the version number from a source assembly, as shown in the following WiX snippet.

```
<Product Id="some-guid-here"
    Name="your-product-name"
    Language="1033"
    Version="!(bind.FileVersion.$(var.name-of-you-VS-project.TargetFileName))"
    Manufacturer=" your-company-name"
    UpgradeCode="some-guid-here"/>
```

References

Quick Reference Posters

The following Quick Reference Posters are available as part of this guidance:

- **Build Customization Quick Reference Guide – Teams and Personas**
- **Build Customization Quick Reference Guide – Default Build Process**
- **Build Customization Quick Reference Guide – Teams and Personas**
- **Build Customization Quick Reference Guide – Upgrade Build Process**
- **Visual Studio 2012 Team Build Quick Reference Guide**

Hands-on Labs (HOL)

The following Hands-on Lab (HOL) material is available as part of this guidance:

- **HOL – What's new in Team Foundation Build 2012**
- **HOL – Windows Azure Build Customization**
- **HOL – Developing Custom Activities**
- **HOL – Build and Deploy an ASP.NET web site**
- **HOL – Setting Up and Using the BRDLite Template**
- **HOL – Working with Hyper-V**

General Links

Visual Studio ALM Rangers Site

http://msdn.microsoft.com/teamsystem/ee358786.aspx
http://www.tinyurl.com/almrangers

Team System Widgets

http://www.teamsystemwidgets.com

Videos for Team System

http://msdn.microsoft.com/vsts2008/bb507749.aspx?wt.slv=topsectionsee

MSDN Site

http://msdn.microsoft.com/default.aspx