Visual Studio
ALM Rangers

Visual Studio Build Customization

Windows Azure Build Customization Hands-On Lab

Sunday, July 01, 2012

Visual Studio ALM Rangers
Mark Nichols

Microsoft Corporation

Table of Contents

Introduction 5
  Overview 5
  Visual Studio ALM Rangers 5
  Prerequisites 5
  Suggested Focus and Time Map 6
Exercise 1: Context and Preparation 7
  Preparation 7
Exercise 2: Setup of Team Project Environment 8
Exercise 3: Modification of Utility Applications to use in Build 9
Exercise 4: Update of Build Template to Deploy Windows Azure 14
Exercise 5: Testing the Build Template and Deploying to Windows Azure 20
References 22

Table of Tables

Table 1 - Suggested focus and time map 6
Table 2 - Build Template Arguments 14
Table 3 - InvokeProcess: Copy Files 15
Table 4- InvokeProcess: Upload Files to Blob Storage 17
Table 5- InvokeProcess: Deploy Windows Azure Application 18
Table 6- InvokeProcess: Start Windows Azure Application 19

Table of Figures

Figure 1 – Source Control folder structure 8
Figure 2 – DeploymentApps folder contents 13
Figure 3 – Default Misc section in build definition 20
Figure 4 – Updated Misc section in build definition 20
Figure 5 – Set MSBuild arguments in build definition 21

Introduction

Overview

This Hands-on Lab describes how to create a scalable automated build environment for Windows Azure applications using freely available code and standard Team Build template capabilities. The solution described in this document can be used as-is in your build environment to build and deploy Windows Azure applications or you can modify it to better meet your needs. The steps described below will show you how to create a build solution and will also show you the Windows Azure process and what the environment expects with regards to deployment of Windows Azure applications.

Visual Studio ALM Rangers

The Visual Studio ALM Rangers are 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 is intended for Microsoft "200-300 level" users of Team Foundation Server. They are intermediate to advanced users of Team Foundation Server and have in-depth understanding of the product features in a real-world environment. Parts of this guide might be useful to novices and experts, but they are not the intended audience for this guide.

Prerequisites

To complete the basic and advanced Hands-on Lab walk-through scenarios you need the following environment:

- Team Foundation Server 2010 Source Code Control and Build Environment
- Windows Azure Account (3-Month Trial Account is sufficient)

[1] Rangers Index - 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

Suggested Focus and Time Map

If you are intending to follow the Hands-on Lab (HOL) step by step, use these times as a guideline. If, however, you are intending to investigate each step in detail, do double the times as a bare minimum.

Topic	Duration in minutes	Page
Exercise 1 – Context and Preparation	10	7
Exercise 2 – Setup of Team Project Environment	5-10	8
Exercise 3 – Modification of Utility Applications to use in Build	15	9
Exercise 4 – Update of Build Template to Deploy Windows Azure	30 – 45	16
Exercise 5: Testing the Build Template and Deploying to Windows Azure	10 – 20	23
TOTAL	**70 - 100 min**	

Table 1 - Suggested focus and time map

We wish you a pleasant and interesting journey!

Exercise 1: Context and Preparation

This Hands-on Lab will show you how to create a working build process template that can be used to build, package and deploy Windows Azure applications. A detailed walkthrough is provided to show you how to modify an existing build template to include all of the steps necessary to deploy a Windows Azure application as part of a Team Foundation Build process.

There are several ways of using Team Foundation Build to package, upload and deploy a Windows Azure application. This Hands-on Lab uses two applications downloaded from the Windows Azure Code Sample site[2] to upload and deploy an application to a Windows Azure account.

The two applications used from the code sample site are the Windows Azure Service Management “CSManage”[3] sample and the Windows Azure “ImportExportBlob”[4] sample. Below, the source code of both applications is modified slightly to satisfy some requirements for using them within a Team Foundation Build environment. Both are “console” applications that use app.config files as a default to store Windows Azure Account Data. We modify that so values can be passed through parameters – which make the applications much more scalable within the build environment. That is, you don’t have to update the app.config file every time you run a build. One other point here, because we are using console applications rather than custom activities, the process described below can be directly translated into a scripted build (MSBuild or PowerShell for example) if you don’t want to use Team Foundation Build.

Preparation

1. Obtain a Windows Azure Account[5]. If you do not already have an account, please see the associated guidance document for more detailed information around getting a temporary account.
2. Set up the development management certificate. Again, if you do not have this set up, please see the associated guidance document for detailed instructions for setting up the management certificate. It must be set up properly or the build process will not have the authority to deploy an application to Windows Azure.
3. Download the source code for the 2 applications listed above (CSManage and ImportExportBlob).

[2] http://code.msdn.microsoft.com/windowsazure

[3] http://code.msdn.microsoft.com/windowsazure/Windows-Azure-CSManage-e3f1882c

[4] http://code.msdn.microsoft.com/windowsazure/Windows-Azure-ImportExportB-9d30ddd5

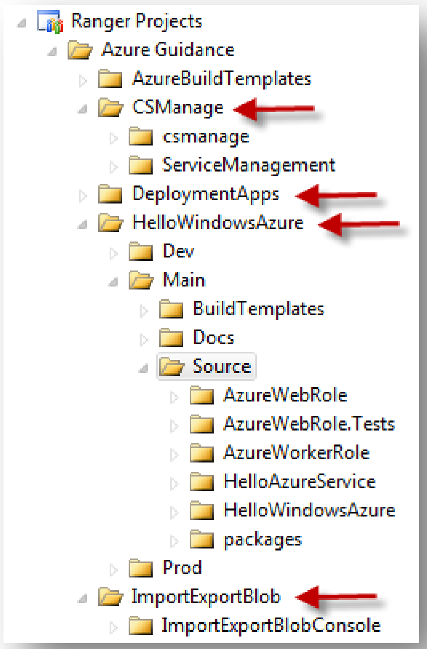
[5] http://www.windowsazure.com

Exercise 2: Setup of Team Project Environment

NOTE

In all cases it is assumed that you will update the "DisplayName" of the build activities with appropriate names. This is totally up to you what you want to name them.

Create a location in Team Foundation Server source control for the application to reside. Below is the folder structure used in the examples of this hands-on lab. You can see below there are folders for the HelloWindowsAzure application that we will be building and deploying. There are also folders for the CSManage and ImportExportBlob applications where we will make the changes to allow the extra parameter values. Lastly, there is another folder called "DeploymentApps". This folder will contain the updated CSManage and ImportExportBlob applications. The instructions below define the changes that need to be made to make the utility applications more scalable in an automated build environment.



```
Ranger Projects
  Azure Guidance
    AzureBuildTemplates
    CSManage
      csmanage
      ServiceManagement
    DeploymentApps
    HelloWindowsAzure
      Dev
      Main
        BuildTemplates
        Docs
        Source
          AzureWebRole
          AzureWebRole.Tests
          AzureWorkerRole
          HelloAzureService
          HelloWindowsAzure
          packages
      Prod
    ImportExportBlob
      ImportExportBlobConsole
```

Figure 1 – Source Control folder structure

Exercise 3: Modification of Utility Applications to use in Build

If you would like to follow along and make your own modifications to the utility applications then proceed as described below. If you would rather skip this part and only work on the build template customization, you can use the "End" code provided in the package as it has all of the required modifications. If you do this, you may jump to "Exercise 4: Update of Build Template to Deploy Windows Azure". Either way, you will need to build the code and create the two console applications.

1. Once you have everything in source control, open the **ImportExportBlob** application in Visual Studio. There are a few fairly simple changes to be made.
    a. Edit the ExportDirectoryToContainer method. This method pushes files within a local directory up to a blob storage location in Windows Azure. This is how we get the application up to the cloud. It doesn't actually deploy the application but it does get a copy up to the cloud so it can be deployed.
        i. Add two more parameters to the method: "`string containerNameIn, string connectionString`" right after the existing "`dirPath`" parameter
        ii. Modify the statement that creates the storageAccount variable to use the connectionString if it contains a value, otherwise use the value in the configuration file:

```
CloudStorageAccount storageAccount = string.IsNullOrEmpty(connectionString) ?
CloudStorageAccount.Parse(ConfigurationManager.AppSettings["StorageAccountCon
nectionString"]) :
CloudStorageAccount.Parse(connectionString);
```

        iii. Change the creation of the "`containerName`" variable to use the containerNameIn parameter if it has a value:

```
string containerName = !string.IsNullOrWhiteSpace(containerNameIn) ?
containerNameIn : MakeContainerName(Path.GetFileName(dirPath));
```

```
    /// <summary>
    /// Exports the user's directory to a container of the same name in their storage account.
    /// </summary>
    /// <param name="dirPath">The full path to the directory to export.</param>
    /// <param name="containerNameIn">Name of the blob storage container</param>
    /// <param name="connectionString">account name and key for blob storage</param>
    private static void ExportDirectoryToContainer(string dirPath, string containerNameIn, string
    connectionString)
    {
        // Retrieve storage account information from app.config file (uses dev storage by default).
        CloudStorageAccount storageAccount = string.IsNullOrEmpty(connectionString) ?
            CloudStorageAccount.Parse(ConfigurationManager.AppSettings["StorageAccountConnectionString"])
    :
            CloudStorageAccount.Parse(connectionString);

        // Create client object for Blob service.
        CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

        // Get container name corresponding to top-level folder name
        string containerName = !string.IsNullOrWhiteSpace(containerNameIn) ? containerNameIn :
            MakeContainerName(Path.GetFileName(dirPath));
        Console.WriteLine("Uploading blobs to container " + containerName);

        // Get named container or create if it does not exist.
        CloudBlobContainer container = GetContainer(blobClient, containerName, true);

        // Create a blob for each file in the top-level directory and all subdirectories.
        foreach (string fileName in Directory.GetFiles(dirPath, "*", SearchOption.AllDirectories))
        {
            // Exclude hidden, system, and temp files
            if (0 == (new FileInfo(fileName).Attributes &
                (FileAttributes.Hidden | FileAttributes.System | FileAttributes.Temporary)))
            {
                // Construct the blob reference from the top level folder path and file name.
                CloudBlob blob = container.GetBlobReference(fileName.Substring(dirPath.Length + 1));

                // Upload the file to the blob.
                try
                {
                    blob.UploadFile(fileName);
                    Console.WriteLine("Uploaded new blob " + blob.Uri);
                }
                catch (StorageClientException e)
                {
                    Console.WriteLine("Could not upload blob named " + blob.Uri);
                    Console.WriteLine("Error message: " + e.Message);
                }
            }
        }
    }
```

b. Now, edit the "`Main`" method to call the "`ExportDirectoryToContainer`" method with the extra parameters. These extra parameters will be provided via the command line rather than through the app.config file (as was mentioned above). We are only changing the "export" or upload section of the code.

   i. Comment out the checking of the "-c" argument from the command line. Make sure you add the end parentheses. So, the code should be:

```
if (argCollection["-d"] == null)
```

   ii. Create a method variable called connectionString and set it to "Empty"

```
var connectionString = string.Empty;
```

   iii. Make sure the account name and key are provided via the command line and if so, create a connection string so the application can authenticate properly.

```
if ((argCollection["-a"] != null) && (argCollection["-k"] != null))
{connectionString = string.Format("DefaultEndpointsProtocol=https;AccountName
={0};AccountKey={1}", argCollection["-a"], argCollection["-k"]); }
```

   iv. Create a new variable "`containerName`" and use the value passed in through the command line argument using the "-c" switch.

```
string containerName = (argCollection["-c"] ??
string.Empty).ToLower();
```

   v. Modify the call to the `ExportDirectoryToContainer` method to include the containerName parameter

```
ExportDirectoryToContainer(Path.GetFullPath(argCollection["-
d"]).TrimEnd('\\'), containerName, connectionString);
```

```
/// <summary>
/// Console application entry point.
/// </summary>
/// <param name="args">The command line argument list.</param>
private static void Main(string[] args)
{
    try
    {
        // Check for the usual help conditions
        if (args.Length == 0 ||
            args.Contains("-h") ||
            args.Contains("/h") ||
            args.Contains("-?") ||
            args.Contains("/?"))
        {
            OutputHelp();
            Environment.Exit(1);
        }

        string error = "Error: command-line arguments incorrectly specified. Type " +
            "ImportExportBlob -h for guidelines.";

        // Ensure an even number of arguments
        if (args.Length % 2 != 0)
        {
            Console.WriteLine(error);
            Environment.Exit(1);
        }

        // Return name-value pairs containing command-line values
        NameValueCollection argCollection = ProcessCommandLineArgs(args);

        // Perform requested operation
        string order = (argCollection["-o"] ?? string.Empty).ToLower();

        if (order == "export")
        {
            if (argCollection["-d"] == null) // || argCollection["-c"] != null)
            {
                Console.WriteLine(error);
                Environment.Exit(1);
            }

            var connectionString = string.Empty;
            // Was account name and account key provided?  If so then create a connection string
            if ((argCollection["-a"] != null) && (argCollection["-k"] != null))
            {
                connectionString =
string.Format("DefaultEndpointsProtocol=https;AccountName={0};AccountKey={1}",
                                    argCollection["-a"], argCollection["-k"]);
            }

            // Was the container name provided?  If so, use it.
            string containerName = (argCollection["-c"] ?? string.Empty).ToLower();
            ExportDirectoryToContainer(Path.GetFullPath(argCollection["-d"]).TrimEnd('\\'),
containerName, connectionString);
        }
        else if (order == "import")
        {
            if (argCollection["-d"] == null || argCollection["-c"] == null)
            {
                Console.WriteLine(error);
                Environment.Exit(1);
            }

            ImportContainerToDirectory(
                argCollection["-c"].ToLower(),
                Path.GetFullPath(argCollection["-d"]).TrimEnd('\\'));
        }
        else
        {
            Console.WriteLine(error);
            Environment.Exit(1);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine("Exception: " + e.ToString());
        Console.WriteLine("Press any key to exit.");
        Console.ReadLine();
    }
}
```

c. Save all your changes and compile. We will eventually copy the compiled application over to the "DeploymentApps" folder so it can be used by the build template.

2. Now, on to the **CSManage** application. There are a few more edits in this application and they are in two different source files.
    a. The first edits are in the Program.cs file
        i. Within the "ExecuteActions" method, the author set up a try/catch block to capture errors but did not set the "hasErrors" flag when an exception is thrown. Within the catch block add: `hasErrors = true;` after it logs the error.

```
public bool ExecuteActions(string[] args)
{
    try
    {
        ParseArguments(args);
        if (hasErrors)
        {
            LogMessage("Use /? for command line syntax.");
        }
    }
    catch (Exception e)
    {
        LogError("Encountered and unexpected error {0} {1}.", e.Message, e.StackTrace);
        hasErrors = true;
    }
    return hasErrors;
}
```

        ii. Next, within the ParseArguments method, we need to allow passing in the Windows Azure subscription ID and the management certificate "thumbprint". This information will authenticate the account attempting to deploy an application. Windows Azure requires this authentication before it will allow changes to the application environment. Add the following code to the end of the first switch statement within "ParseArguments" – right after "role-instance-name". The Code below is just the changes to be made since the method is quite long.

```
case "/subscription-id":
    {
        CSManageCommand.SubscriptionId = tok.args[0];
        break;
    }
case "/certificate-thumbprint":
    {
        CSManageCommand.CertificateThumbprint = tok.args[0];
        break;
    }
```

        iii. The last edit within the Program.cs file is within the "Main" method. Here, the author set the return value to 1 when it succeeds and 0 when it fails. This is the reverse of what the Team Foundation Build activity is expecting when calling a command line application. If we don't change this, Team Build will mistakenly interpret a successful deployment as a failure in the build. All we need to do here is change the return statement to this:

```
static int Main(string[] args)
```

```
{
    ProcessCheckServerCertificate();

    Program prgm = new Program();

    return prgm.ExecuteActions(args) ? 1 : 0;
}
```

The last bit of code changes is in the CSManageCommand.cs file within the CSManageCommands folder. This change at the top of the "Run()" method is intended to better capture error conditions resulting from invalid certificate, subscription ID or certificate thumbprint information. The first "if" statement needs a little extra validation. Again, because this method is quite long, only the changed code is provided below. It replaces the code at the very beginning of the method.

```
public void Run()
{
    if ((string.IsNullOrEmpty(SubscriptionId)) || (String.IsNullOrEmpty(CertificateThumbprint)))
    {
        if (!ReadFromConfigFile() || !ValidateSubscriptionId() || !ValidateCertificate() ||
!this.Validate())
        {
            Console.WriteLine("There was an error processing this command!");
            throw new ArgumentException("Argument validation failed.");
        }
    }
    else
    {
        if (!ValidateCertificate())
        {
            Console.WriteLine("There was an error processing this command!");
            throw new ArgumentException("Argument validation failed.");
        }
    }
```

b. Save all your changes and compile.

c. Copy both compiled applications over to the "DeploymentApps" source control folder so it can be used by the build template. The file list should look like this:

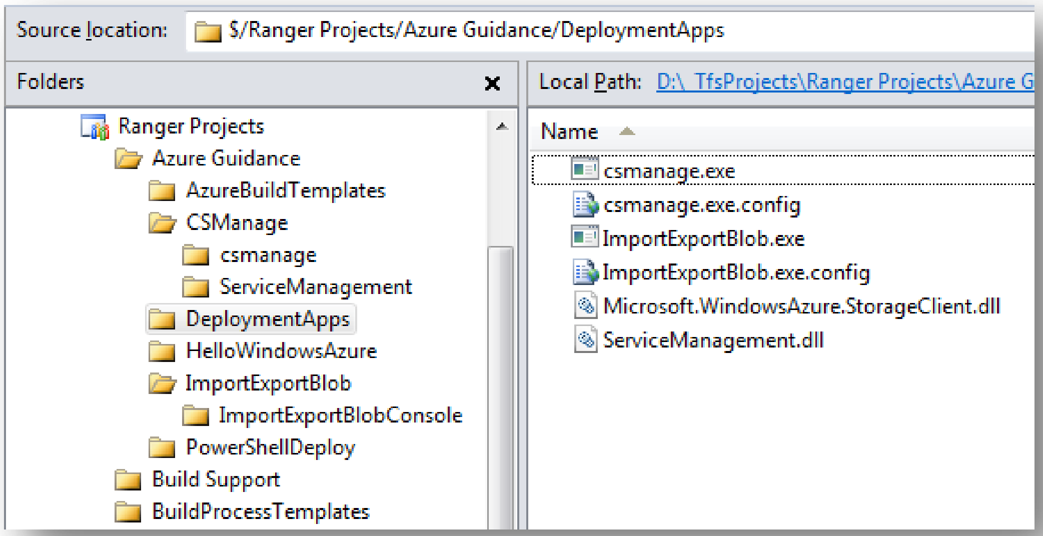


Figure 2 – DeploymentApps folder contents

Exercise 4: Update of Build Template to Deploy Windows Azure

1. First, make a copy of the “DefaultTemplate.xaml” build template within source control and name it “AzureBuildTemplate.xaml”
2. Edit the file by double clicking on it. This should open it up in the Windows Workflow editor rather than an XML text editor
3. Create the list of workflow arguments that we will need in the build. The naming convention used here is:
    a. All arguments used to extend the default template with the Windows Azure build and deploy are prefixed with “Azure”. For example, we need to create a Boolean flag argument that we can pass in to indicate if we want the Windows Azure package/deploy process to execute. The name of this argument is “AzureDoCSPack”. The full list of arguments is below. Go ahead and create all of these arguments by clicking on “Arguments” at the bottom of the workflow edit window and then clicking on “Create Argument” to start the process for each argument.

Name	Argument type	Default value
AzureDoCSPack	Boolean	True
AzureDeployContainer	String	“deploycontainer”
AzureDoCSPkgUpload	Boolean	True
AzureDoDeploy	Boolean	False
AzureDeploySlot	String	“Staging”
AzureDnsPrefix	String	No Default Value Here
AzureStorageAccount	String	No Default Value Here
AzureSubscriptionId	String	No Default Value Here
AzureCertificateThumbprint	String	No Default Value Here
AzureUtilityAppFolder	String	“Your app folder location here”
AzureBlobAccessKey	String	No Default Value Here

Table 2 - Build Template Arguments

4. We will be creating variables as well. They will be prefixed in a similar way except the full prefix will be “AzureV”. The extra letter V makes it easier to tell the difference between arguments and variables when looking at the workflow code.
5. Within the build workflow, locate the **“Try Compile, Test, and Associate Changesets and Work Items”** activity. It is at the bottom of the “Run On Agent” parent activity. Below the “Try Compile…” activity **insert an “If” activity** from the Toolbox.
    a. Within this “If” activity, we will create our entire set of tasks for packaging and deploying the Windows Azure application
    b. For the “Condition”, just type in “AzureDoCSPack”. This will indicate if our new process should be executed
6. **Drop a “Sequence” activity** in the “Then” side of the “If” activity
    a. Add the following variables within the sequence:

Name	Variable type
AzureVPackages	System.Collections.Generic.IEnumerable<String>
AzureVXCopyFrom	String
AzureVXCopyTo	String

7. **Drop a “FindMatchingFiles” activity in the sequence**. Packaging of the Windows Azure application happens automatically within MSBuild when you add the Windows Azure SDK to the build machine. However, it does

not copy the resulting package to the build drop location. We are going to do that copy so the package is brought over to the drop.

a. In “MatchPattern” enter: String.Format("{0}***.cspkg", SourcesDirectory)
b. In “Result” enter the variable: AzureVPackages

8. **Drop an “If” activity** below “FindMatchingFiles” and enter the following condition: AzureVPackages.Count > 0 The process should only run if it has a packaged application to work with.
9. Within the “Else” side, **drop a “Throw” activity** and enter this in the “Exception”: New Exception("No Azure Packages To Process")
10. On the “Then” side, **add a “Sequence”** and create the following variables:

Name	Variable type	Default Value
AzureVInvokeProcessResult	Int32	
AzureVDeploymentApp	String	“csmanage.exe”
AzureVUploadApp	String	"ImportExportBlob.exe"

The variable “AzureVInvokeProcessResult” will be used each time the InvokeProcess activity is called (4 times) to capture the result of the executed application. We are using InvokeProcess to call XCopy, CSManage.exe (called twice), and ImportExportBlob.exe. The other two variables carry the names of the applications that will do the upload and deployment work within Windows Azure.

11. Within that sequence **add two (2) “Assign” activities**. This is to create the copy from and copy to locations for copying the Windows Azure package to the drop location. Make the following assignments:

Variable Name	Assignment
AzureVXCopyFrom	String.Format("{0}*", System.IO.Path.GetDirectoryName(AzureVPackages(0)))
AzureVXCopyTo	String.Format("{0}\AzureVPackage{1}", BuildDetail.DropLocation, System.IO.Path.GetDirectoryName(AzureVPackages(0)).Split(New String() {"bin"}, StringSplitOptions.RemoveEmptyEntries)(1))

12. Right below the assignments, **add an “InvokeProcess” activity** from the list of Team Foundation Build Activities:

a. In all cases of using the InvokeProcess activity, leave the Standard and Error output settings (“stdOutput” and “errOutput”) as they are by default.
b. Add the following to the properties – for the properties not listed, leave them with their default values:

Property Name	Value
Arguments	String.Format("/I /Y ""{0}"" ""{1}""", AzureVXCopyFrom, AzureVXCopyTo)
FileName	"xcopy.exe"
Result	AzureVInvokeProcessResult

Table 3 - InvokeProcess: Copy Files

c. This will execute XCopy.exe to perform the file copy. XCopy is by default is part of windows so no need to add any functionality for this process.
d. For build logging purposes, **add a “WriteBuildMessage” activity** below the Handle Standard Output and Handle Error Output where it says “Drop activity here” and add the respective messages:

```
String.Format("Copy Azure Package: {0}", stdOutput)
String.Format("Copy Azure Package: {0}", errOutput)
```

13. Just in case the XCopy fails, **add an “If” activity** that will check the AzureVInvokeProcessResult value.

a. If the value is not zero (0) then there was an error. Condition: `AzureVInvokeProcessResult <> 0`

b. In the “Then”, **insert a “Throw”** that includes this exception: `New Exception("Azure XCopy Failed")`

14. Next **add another “If” activity** below the one you just created. This will contain the upload and deployment process and the first check is to see if we want to perform the upload. Set the condition to:

    ```
    AzureDoCSPkgUpload
    ```

    If we don’t want to upload then we can’t deploy so that also depends on the value of this argument which we pass in from the build definition.

15. In the “Then” side of the “If” activity, **drop in a “Sequence” activity**.

    a. Add these variables to the activity:

Name	Variable type
AzureVDeployExeFilename	String
AzureVUploadExeFilename	String
AzureVUtilityTempFolder	String

16. If we are going to do the upload and possibly deploy, we need to know where the utility (upload and csmanage) applications reside. They can be “installed” on the build machine, in a known folder – the down side here is that you would need to “touch” the build machine to get them there. In some environments that is not allowed. Or, the applications can stored in source control, in a central location and the build process can get them in a “just in time” method so they are available for the current build.

    a. First, we will check to see if the files are in source control. We do this by **adding an “If” activity** and checking to see if the AzureUtilityAppFolder starts with the source control designator so add the following to the condition:

    ```
    AzureUtilityAppFolder.StartsWith("$/")
    ```

    b. In the “Then” section, **add a “Sequence” activity** because there are a few steps to perform to get the files from source control

    c. In the sequence, **drop in an “Assign” activity**. Update the value of AzureVUtilityTempFolder with this:

    ```
    System.IO.Path.Combine(SourcesDirectory, "AzureUtilTmp")
    ```

    “AzureUtilTmp” is a folder that will be created on the fly to contain all of the utility application files for use in the process. This folder is created as a sub-folder of “SourcesDirectory”.

    d. Next **add a “DownloadFiles” activity**. This will do the work of pulling the files from source control. Update each of the activities properties with the following values. Note: this activity is temperamental so make sure you fill in each property.

Property Name	Value
DeletionId	0
LocalPath	AzureVUtilityTempFolder
Recursion	Microsoft.TeamFoundation.VersionControl.Client.RecursionType.Full
ServerPath	AzureUtilityAppFolder
Version	Microsoft.TeamFoundation.VersionControl.Client.VersionSpec.Latest.DisplayString

    e. Just below the “DownloadFiles” activity, **add two (2) “Assign” activities**. Here, we will use the folder location from downloading the files from source control. Set them with the following values:

Variable Name	Assignment

Variable Name	Assignment
AzureVUploadExeFilename	System.IO.Path.Combine(AzureVUtilityTempFolder, AzureVUploadApp)
AzureVDeployExeFilename	System.IO.Path.Combine(AzureVUtilityTempFolder, AzureVDeploymentApp)

f. Now, in the "Else" side of the "If" created above, **drop in a "Sequence" activity**

g. In the sequence **add two (2) Assign activities**. They will be similar to the ones we just created above but the locations will be different since (in this case) the utility applications will already exist where the build process can find them. Set the assignments like this:

Variable Name	Assignment
AzureVUploadExeFilename	System.IO.Path.Combine(AzureUtilityAppFolder, AzureVUploadApp)
AzureVDeployExeFilename	System.IO.Path.Combine(AzureUtilityAppFolder, AzureVDeploymentApp)

17. Ok, we now know where the utility applications are, it is time to use the ImportExportBlob application to upload the application package to blob storage so that it can be deployed. **Drop another "InvokeProcess" activity** just below the "If" we just created

    a. Add the following to the properties – for the properties not listed, leave them with their default values:

Property Name	Value
Arguments	String.Format("-o export -d ""{0}"" -c {1} -a {2} -k {3}", AzureVXCopyTo, AzureDeployContainer, AzureStorageAccount, AzureBlobAccessKey)
FileName	AzureVUploadExeFilename
Result	AzureVInvokeProcessResult

Table 4- InvokeProcess: Upload Files to Blob Storage

i. This will execute ImportExportBlob.exe application to start the deploy process by passing in all of the parameters necessary to complete the task including the files to copy, where to copy them to, the Windows Azure storage account and the secret key to use.

ii. For build logging purposes, **add "WriteBuildMessage" activities** below the Handle Standard Output and Handle Error Output where it says "Drop activity here" and add the respective messages:

```
String.Format("Copy Package to Blob Storage: {0}", stdOutput)
String.Format("Copy Package to Blob Storage: {0}", errOutput)
```

18. We need to check the execution result so **add an "If" activity** right after the InvokeProcess you just created.

    a. The condition will be the same with the value of:

```
AzureVInvokeProcessResult > 0
```

    b. In the "Then" side **add a "Throw" activity** like before and throw this exception:

```
New Exception("Azure Upload Failed")
```

19. Now it is time to deploy. Below the "If" activity we just created, **add another "If" activity**. This is to check the final Boolean argument passed in through the build definition to make sure that we indeed want to deploy the Windows Azure application to one of the operational VM "Slots".

    a. Set the condition to: `AzureDoDeploy`

    b. Within the "Then" side of the "If", **add a "Sequence" activity**.

20. In the new Sequence add two (2) variables:

Name	Variable type
AzureVConfigurationFiles	IEnumerable<String>

Name	Variable type
AzureVDeploymentName	String

21. **Insert a "FindMatchingFiles" activity** at the beginning of the sequence.
    a. In "MatchPattern" enter: `String.Format("{0}\*.cscfg", AzureVXCopyTo)`
    b. In "Result" enter the variable you created above: `AzureVConfigurationFiles`
22. After that, **add an "Assign" activity**.

Variable Name	Assignment
AzureVDeploymentName	String.Format("{0}{1}", AzureDnsPrefix, DateTime.Now.ToString("yyyyMMddhhmm"))

23. **Add an "InvokeProcess" activity** below FindMatchingFiles
    a. Add the following to the properties – for the properties not listed, leave them with their default values:

Property Name	Value
Arguments	String.Format("/create-deployment /name:{0} /label:""{2}"" /hosted-service:{1} /status:running /package:http://{3}.blob.core.windows.net/{4}/{5} /config:""{6}"" /slot:{7} /subscription-id:{8} /certificate-thumbprint:{9} /warnings-as-error", AzureVDeploymentName, AzureDnsPrefix, BuildDetail.BuildNumber, AzureStorageAccount, AzureDeployContainer, System.IO.Path.GetFileName(AzureVPackages(0)), AzureVConfigurationFiles(0), AzureDeploySlot, AzureSubscriptionId, AzureCertificateThumbprint)
FileName	AzureVDeployExeFilename
Result	AzureVInvokeProcessResult

Table 5- InvokeProcess: Deploy Windows Azure Application

    b. This will execute CSmanage.exe application to start the deploy process
    c. For build logging purposes, **add "WriteBuildMessage" activities** below the Handle Standard Output and Handle Error Output where it says "Drop activity here" and add the respective messages:

```
String.Format("Deploy Azure Package: {0}", stdOutput)
String.Format("Deploy Azure Package: {0}", errOutput)
```

24. Once again, we need to check the execution result so **add an "If" activity** similar to the one we added after the first InvokeProcess activity.
    a. The condition will be the same with the value of:

```
AzureVInvokeProcessResult <> 0
```

    b. In the "Then" side **add a "Throw" activity** like before and throw this exception:

```
New Exception("Azure Deploy Failed")
```

Almost done!

We need one more call to CSManage.exe to start the application inside of Windows Azure. The first call deployed the application to a Windows Azure slot but at this point it is just sitting there. So…

25. **Add an "InvokeProcess" activity** just below the "If" you just created.
    a. Add the following to the properties – for the properties not listed, leave them with their default values:

Property Name	Value
Arguments	String.Format("/update-deployment /hosted-service:{0} /status:running /name:{1} /subscription-id:{2} /certificate-thumbprint:{3} /warnings-as-error", AzureDnsPrefix, AzureVDeploymentName, AzureSubscriptionId, AzureCertificateThumbprint)
FileName	AzureVDeployExeFilename
Result	AzureVInvokeProcessResult

Table 6- InvokeProcess: Start Windows Azure Application

    b. This will execute CSmanage.exe application to start the deployed application
    c. For build logging purposes, **add "WriteBuildMessage" activities** below the Handle Standard Output and Handle Error Output where it says "Drop activity here" and add the respective messages:

```
String.Format("Start Azure Application: {0}", stdOutput)
String.Format("Start Azure Application: {0}", errOutput)
```

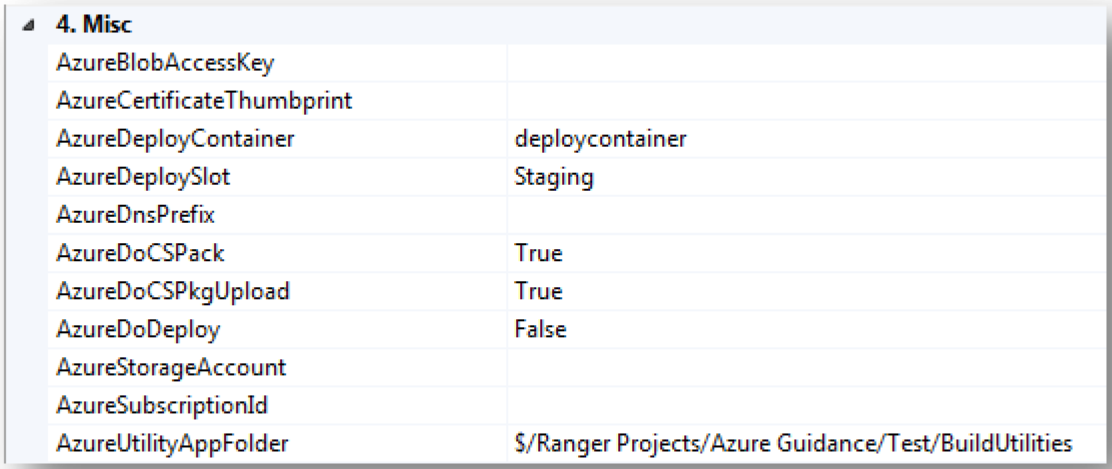
26. Of course we need to check the execution result so **add an "If" activity** similar to the one we added after the first InvokeProcess activity.
    a. The condition will be the same with the value of: `AzureVInvokeProcessResult <> 0`
    b. In the "Then" side **add a "Throw" activity** like before and throw this exception:

```
New Exception("Azure Update to 'Running' Failed")
```

Exercise 5: Testing the Build Template and Deploying to Windows Azure

That's it. Save the build template and then give it a try by pointing a new build definition to the build template that you just created. Please refer to the Windows Azure section of the Build Guidance document for detailed use instructions such as how to set up the build machine with a management certificate, where to get the subscription ID, blob storage, etc.

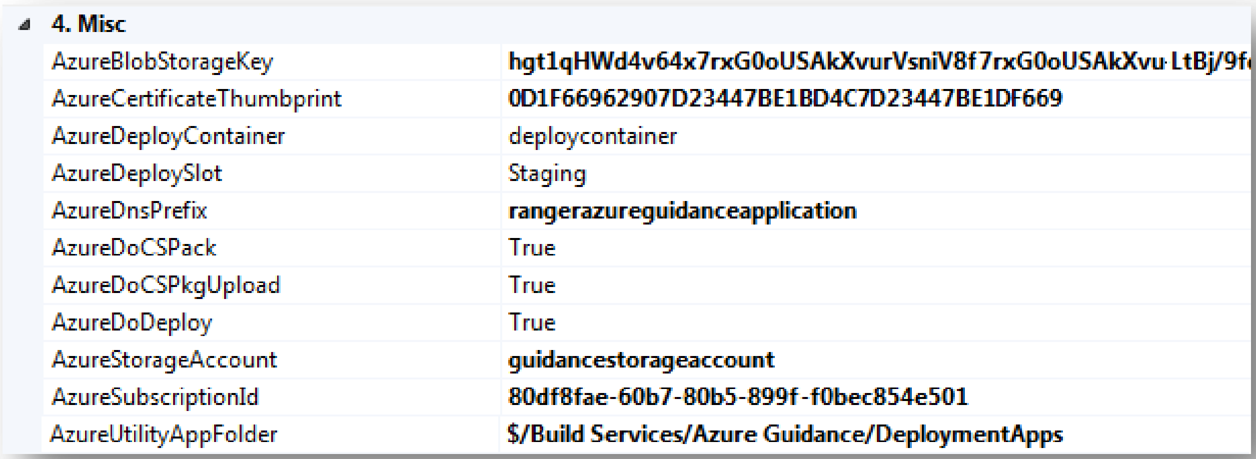
The addition to the build definition will look similar to the following:



4. Misc	
AzureBlobAccessKey	
AzureCertificateThumbprint	
AzureDeployContainer	deploycontainer
AzureDeploySlot	Staging
AzureDnsPrefix	
AzureDoCSPack	True
AzureDoCSPkgUpload	True
AzureDoDeploy	False
AzureStorageAccount	
AzureSubscriptionId	
AzureUtilityAppFolder	$/Ranger Projects/Azure Guidance/Test/BuildUtilities

Figure 3 – Default Misc section in build definition

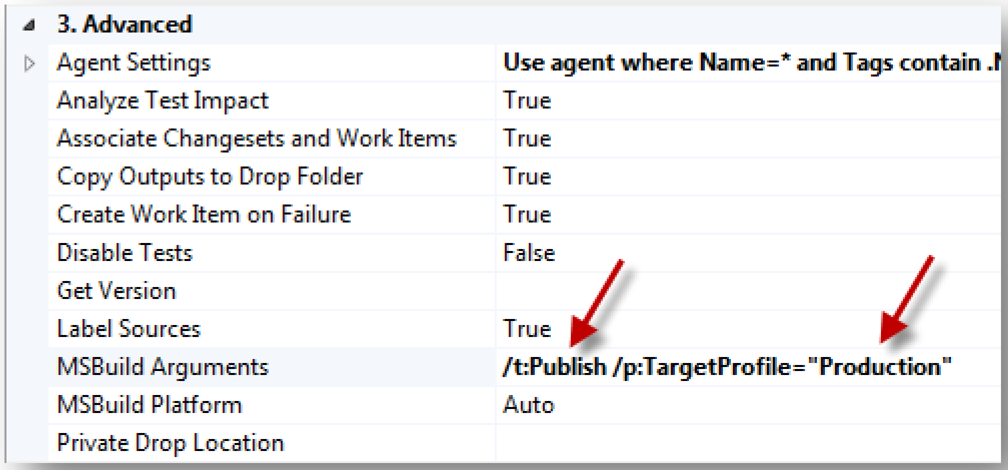
You will need to add the parameter values specific to your Windows Azure account and the location of your "utility" applications that do the upload and deploy.



4. Misc	
AzureBlobStorageKey	**hgt1qHWd4v64x7rxG0oUSAkXvurVsniV8f7rxG0oUSAkXvu LtBj/9f**
AzureCertificateThumbprint	**0D1F66962907D23447BE1BD4C7D23447BE1DF669**
AzureDeployContainer	deploycontainer
AzureDeploySlot	Staging
AzureDnsPrefix	**rangerazureguidanceapplication**
AzureDoCSPack	True
AzureDoCSPkgUpload	True
AzureDoDeploy	True
AzureStorageAccount	**guidancestorageaccount**
AzureSubscriptionId	**80df8fae-60b7-80b5-899f-f0bec854e501**
AzureUtilityAppFolder	**$/Build Services/Azure Guidance/DeploymentApps**

Figure 4 – Updated Misc section in build definition

One thing to remember when setting up your build definition: make sure you include the MSBuild arguments in the “Advanced” section. Detail around these values is included in the guidance document but you need these values to inform the build process to properly package the Windows Azure application (**/t:Publish**) and also which configuration files to use when deploying (**/p:TargetProfile=”Production”**).



3. Advanced	
Agent Settings	**Use agent where Name=* and Tags contain .**
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	**/t:Publish /p:TargetProfile="Production"**
MSBuild Platform	Auto
Private Drop Location	

Figure 5 – Set MSBuild arguments in build definition

You can test with your own application to deploy or you can use the application provided with the Hands-on Lab. In that ASP.NET MVC3 application there is a web-role and a worker role that talk to each other to perform some simple tasks. There is a default view that will appear when you access the site after it is loaded and running in Windows Azure.

LAB COMPLETE

References

Windows Azure Samples: learn through code: http://code.msdn.microsoft.com/windowsazure

Windows Azure ImportExportBlob Sample: http://code.msdn.microsoft.com/windowsazure/Windows-Azure-ImportExportB-9d30ddd5

Windows Azure ServiceManagement Sample: http://code.msdn.microsoft.com/windowsazure/Windows-Azure-CSManage-e3f1882c

Windows Azure 3 Month free trial: http://www.windowsazure.com/en-us/pricing/free-trial/

Windows Azure Management Portal: http://windows.azure.com/