17

Deployment Techniques

This chapter covers

* Leaning on continuous integration
* Creating push-button deployments
* Automating remote server deployments

Beyond server environment and hosting scenarios, deploying an application presents an entirely different set of challenges. Manual deployments are wrought with problems, as human errors become more prevalent. Automation eliminates these eleventh-hour problems by removing the human factor from deployments. Each deployment environment is slightly different, because connection strings, configuration settings, and server environments can vary. By introducing change management into our automated deployment process, we can ensure we install the correct application with the correct environment settings.

In this chapter the reader will learn how to simplify deployment through an xcopy deployment strategy, and automate deployment through build automation tools. With these build automation tools, the reader will see how to take advantage of configuration management to automate configuration changes to the various deployment environments. After utilizing these techniques on a local machine, the next logical step is to add remote deployment capabilities. A demonstration of using the Web Deployment Tool to take an existing local deployment and give it remote server capabilities will be presented.

17.1 Automating deployments

On launch night, tensions are high as the smallest mistake could bring your website down. To eliminate the human mistakes that inevitably occur, we would like to automate as much as possible. Ideally, we could simply push a button, and our website would be updated in moments. How this happens depends largely on the deployment environment. Regardless of the deployment environment, any good deployment strategy requires the use of continuous integration.

17.1.1 Employing continuous integration

Working in an environment without an automated integration process can be hectic, and nerve-racking. Because “it works on my machine” does not suffice in a deployment scenario, we need a set of practices to ensure our code always works, and is always ready to deploy. To achieve continuous integration, Martin Fowler laid out a set of practices to adhere to (from <http://www.martinfowler.com/articles/continuousIntegration.html>):

* Maintain a single source repository (use source control)
* Automate the build
* Make your build self-testing
* Make sure everyone commits every day
* Every commit should build the mainline on an integration machine
* Keep the build fast
* Test in a clone of a production environment
* Make it easy for anyone to get the latest executable
* Ensure everyone can see what’s happening
* Automate deployment

We won’t cover all of the continuous integration practices in this book, as entire books have been written on this topic. In addition to adhering to these practices, the “check-in dance” ensures that no one inadvertently breaks the build. The check-in dance steps are:

1. Run the local build
2. Announce to the team you are integrating (for large changes)
3. Pull down the latest version of the mainline. Merge any conflicts
4. Run the local build
5. If successful, commit the changes, providing a descriptive comment
6. Wait for the server build to be successful
7. If the build fails, drop everything and fix it

Depending on the development environment, there are several continuous integration server tools and technologies to employ. One popular continuous integration stack includes:

* Subversion (SVN) for source control
* NAnt for build automation
* NUnit for testing
* CruiseControl.NET for the continuous integration server

Which tool we use does not matter as much as the practices the tools enforce, although we would like our tools to introduce as little friction as possible into the development environment. If we have to wait for a slow or unreliable source control server, our practices are less likely to be followed. Whichever build technology we decide to use, the result of each build should be a single deployment file, checked in to source control at the end of a successful server build. To enable push-button XCOPY deployments, we’ll next look at some key NAnt features.

17.1.2 Enabling push-button XCOPY deployments

In an intranet environment, XCOPY deployments can be as simple as setting up a network share on the deployed machine. In other situations, the deployment file, whether it is an installer or self-contained .zip file, must be copied over manually or pulled down from source control. Regardless, if the files can be pushed from a network share, or pulled manually on the server, our deployment package will include:

* The complete application
* The build tool, if used (NAnt)
* A deployment script
* A batch or Powershell file to kick the process off

Our automated continuous integration build creates and checks in this deployment package. When we have a deployment package in source control, it enables us to deploy any version of our application as needed. With a tool like CruiseControl.net, it is possible to automate the deployment of the latest version of the application as needed.

NAnt, along with the sister project NAntContrib, provides dozens and dozens of tasks out of the box, which can be compiled together to create a single deployment script. These tasks are:

* Source control tasks
* IIS tasks
* File and directory tasks such as creation, deletion, and copying
* Zip tasks
* XML manipulation tasks

With a manual process in place, we can start automating one step at a time with NAnt tasks, until the entire deployment process is automated. Many teams already employ a build process in the form of a Word document or wiki entry, detailing the manual steps. It is only a matter of finding the corresponding NAnt task for each manual task, and the deployment is automated. If no NAnt task exists for a particular operation, NAnt provides the exec task, which can execute anything that can execute in the command-line. The key NAnt tasks for deployments include:

* unzip
* copy
* exec
* xmlpoke

We’ll need the unzip task to unzip the deployment package originally checked in to source control. If this is a manual pull of the deployment package, we can unzip the package manually. The copy task is used to copy the complete application to the correct deployed directory, performing an XCOPY deployment in one automated task. The exec task is used for a variety of scenarios, such as restarting IIS, stopping and starting services, registering assemblies, and so on. The xmlpoke task is used to manage deployment configurations by manipulating key configuration files, such as the Web.config file. In the next section, we’ll examine how to manage multiple deployment configurations with NAnt and xmlpoke.

17.1.3 Managing environment configurations

Development teams often deploy their applications in multiple environments. For any given project, there are at least two environments: production and development. Many teams integrate to one or more test environments before releasing to production. Among these different environments, the deployment must change. Some environments require merely a connection string change, and others require debug flags, configuration values, email addresses, and more. In an automated deployment, the deployment script must take into account the different environment settings. Notably, it must know what environment it is deploying to, and what changes to make to the application to match that environment.

With NAnt, managing all of these environment configurations is straightforward. Deployments are kicked off with a batch file, which merely starts NAnt. The deployment package zip file contains:

* Dev.bat
* CommonDeploy.bat
* deployment.build
* NAnt\
* website\
* database\

The NAnt folder contains the entire runtime distribution of NAnt. We include the distribution to avoid an environmental setup step on every server to which we deploy. The website folder contains the complete application that we XCOPY deploy to the correct folder on the server. The deployment.build is the NAnt build script that contains the complete deployment script. The Dev.bat file is a bootstrapper file that calls CommonDeploy.bat. In listing 17.1, the bootstrapper file Dev.bat call overrides the deploy directory and connection string properties by setting environment variables, and then calls the CommonDeploy.bat script. You would fill in the TODO placeholders when you implement the script for yourself.

Listing 17.1 Setting the environment configuration in Dev.bat

SET driverClass=NHibernate.Driver.SqlClientDriver

SET connectionString=Data Source=.\sqlexpress;Initial Catalog=TODO;uid=sa;pwd=TODO

SET localConnectionString=Data Source=.\sqlexpress;Initial Catalog=TODO;uid=sa;pwd=TODO

SET dialect=NHibernate.Dialect.MsSql2005Dialect

SET websiteTargetDir=\\TODO

SET databaseServer=TODO\sqlexpress

SET databaseName=TODO

SET databaseIntegrated=false

SET databaseUsername=sa

SET databasePassword=TODO

SET shouldReloadDatabase=true #A

CommonDeploy.bat

#A SET command declares variables.

In the Dev.bat file, we set up the environment variables for the environment configuration values (some of which still need to be filled in). With one CommonDeploy.bat batch file that runs off environment variables, we can create additional bootstrapper batch files for each target environment. The end of the Dev.bat batch script calls into the CommonDeploy.bat script, which provides a common bootstrapper file on top of NAnt, shown in Listing 17.2 below.

Listing 17.2 Bootstrapper CommonDeploy.bat file overriding NAnt properties

nant\nant.exe

-buildfile:deployment.build

-D:should.reload.database="%shouldReloadDatabase%" #A

-D:driver.class="%driverClass%"

-D:connection.string="%connectionString%"

-D:local.connection.string="%localConnectionString%"

-D:dialect="%dialect%"

-D:website.target.dir="%websiteTargetDir%"

-D:database.server="%databaseServer%"

-D:database.name="%databaseName%"

-D:database.integrated="%databaseIntegrated%"

-D:database.username="%databaseUsername%"

-D:database.password="%databasePassword%"

-D:test.database.name="%testDatabaseName%"

-D:excel.server.path="%excelServerPath%"

#A Use previously set environment variables

This entire command is in a single CommonDeploy.bat file, calling NAnt, using environment variables set up by a previous environment-specific batch file (Dev.bat in our case). The “-D” command-line switches for NAnt allow us to override properties with the correct deployed values. Because our deployment database will most likely require a different connection string than our local configuration, we need to use NAnt to override this value during deployment. A portion of the deploy.build file is in Listing 17.3 below.

Listing 17.3 Deployment.build NAnt script with the deploy target

<target name="deploy">

<call target="rebuildDatabase" if="${should.reload.database}" /> #A

<xmlpoke #B

file="website/bin/hibernate.cfg.xml"

xpath="${connection.string.path}"

value="${local.connection.string}">

<namespaces>

<namespace prefix="hbm"

uri="urn:nhibernate-configuration-2.2"></namespace>

</namespaces>

</xmlpoke>

<copy todir="${website.target.dir}" overwrite="true"

includeemptydirs="true" > #C

<fileset basedir="website">

<include name="\*\*" />

</fileset>

</copy>

</target>

#A Call another target

#B Change the connection string

#C Copy all website files

The first items to notice in this NAnt script are the XML attribute values in the format ${some.value.here}. These are NAnt properties, whose values were defined earlier through our bootstrapper file. When the CommonDeploy.bat file executes, the command-line switches set these property values with the appropriate environmental settings. Finally, the “deploy” target performs the actual deployment. A NAnt target is a named group of tasks, similar to a method in C#.

17.1.4 Enabling remote server deployments with the Web Deployment Tool

After getting a deployment script that can setup your application and database, the next step is to take on the challenge of pushing deployments to multiple servers. The key takeaway is that by automating the task of deployment, you can eliminate all the manual steps that are error prone. In order to eliminate the need to log on to servers one by one an additional technology is needed. This is where Web Deploy (formerly named MSDeploy) comes into play. You can download it at <http://www.iis.net/expand/webdeploy>. This tool provides a host of features and functions, too numerous to put into this book. The features that are most important related to this approach of deployment are:

* The ability to sync files over HTTP
* The ability to execute a remote command.

These features support both enterprise and hosted environments, and the scripts can be used for both pre-production environments as well as production environments. Typically, for web applications there will be a development server that hosts the web application and database on the same machine. The Quality Assurance (QA) environment may be setup the same. Than in the staging and production environments, more servers come into play. There may be a separate database server, multiple web servers, and even an application server. Automating a deployment to multiple machines can become complex very quickly. In order to reduce the complexity, Web Deploy can be used to fulfill the syncing of files to multiple machines and then executing the deployment script on each server and run remotely, so that they execute the same way that they would in the development environment.

Listing 17.4 shows the command line arguments used to copy deployment files from a build server to a web server and then running the deployment.

Listing 17.4 Using Web Deploy to remotely execute a deployment.

msdeploy.exe -verb:sync -source:dirPath=deploymentFiles |1

-dest:dirPath='c:\installs',computername=192.168.1.34 |1

msdeploy.exe -verb:sync -source:runCommand = 'c:\installs\dev.bat' |2

, -dest:auto,computername=192.168.1.34 |2

First, msdeploy.exe is called with the sync verb specifying a source directory on the local machine (1). This command copies all the files inside the deploymentFiles directory (C:\installs) to the remote server (in this case, the computer with the IP address 192.168.1.34.

Next, msdeploy.exe is again being called with the sync verb but this time the runCommand argument is specified (2). This means that Web Deploy will execute the batch file at c:\installs\dev.bat on the remote server in the same way that you would run it if logged in via remote desktop. Using a technology like Web Deploy can greatly simplify a complex deployment. By running each command locally on each server in the deployment, scripts will run consistently from the development environment through the production environment. The real advantage is that the calls to msdeploy.exe can be scripted which means a multi server deployment can be totally automated and repeatable. Scripting this type of deployment also means that from a single machine you can monitor a deployment and see the results of each script consolidated in your desktop.

Summary

When we configure our environment, we must devise a reliable deployment strategy to ensure the right application is deployed with the correct configuration. At the heart of a solid deployment strategy is continuous integration, which includes practices such as automated deployments and self-testing builds. With free, widely used open source tools such as CruiseControl.NET, NAnt, NUnit, and others, we can build an automated build and deployment server. By packaging NAnt, a build script and a bootstrap batch file, we can harness the flexibility and power of NAnt to deploy and configure our application to multiple environments, up to and including production. By layering on the Web Deployment Tool to reduce the friction of copying and executing the build scripts across multiple servers, we can have a totally automated solution that is repeatable and reliable.