# Expert .NET Delivery Using NAnt and CruiseControl.NET

MARC HOLMES

Apress®

**Expert .NET Delivery Using NAnt and CruiseControl.NET**

**Copyright © 2005 by Marc Holmes**

Library of Congress Cataloging-in-Publication Data

Holmes, Marc.
Expert .NET delivery using NAnt and CruiseControl.NET / Marc Holmes.
p. cm.
Includes index.
ISBN 1-59059-485-1 (pbk. : alk. paper)
1. Install programs (Computer programs) 2. Computer software--Development. 3. Computer software--Testing. I. Title.
QA76.76.I55H6 2005
005.1--dc22
2005008428

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 8

# Database Integration

In this chapter we look at two problems that we have ignored until now. The first is the integration of the database, which can cause problems unless managed carefully. The second problem is a related one: the deployment of environments using multiple servers.

We are going to take a look at likely scenarios and requirements for database integration and solutions for the automated and continuous integration of the database environments. We will then look at how this impacts our current framework for delivery and apply some changes to handle these more complex scenarios. Finally, we will tackle issues with automating the deployment of Data Transformation Services (DTS) packages, an area that is easy to overlook.

## The Problems with the Database

So what is the problem with the database? Well, there are a number of issues that by themselves are not so severe, but when combined and then considered against the process we are attempting to install, they demand our attention:

**Lack of source control.** Generally, source control for databases is quite poor, or at least is full of risk. Generally no real automated links exist between databases and source control systems, so it is easy to lose control of database assets that are stored in the source control database. Once confidence in those assets is lost, we are in trouble. VS .NET has made some strides toward improving source control support for SQL Server, and I hope that this improvement continues over time, but it still does not easily provide the required level of automated control.

**Attention to detail.** Because of the lack of automated source control facilities, managing database assets requires diligence and attention to detail. This usually means the handling of many migration scripts and administration scripts. Once again, if this area is not managed correctly and diligently at all times, then the development and migration picture will not be complete. Worse, this may not be obvious for some time until the script is needed.

**Standing data.** The final big issue with the database is the need for development and migration to occur against standing user data. Whereas the code for an application is simply the logical model and should be replaced entirely on each release, the database is both the logical model and the data itself. Management of development and migration scripts for the database needs to take account of this at all times.

We can take several steps to solve these problems within the delivery framework we currently have:

**Provide automated management of source control.** If the scripts are available, then it is quite easy to provide tasks that add the scripts to source control when required. To best accomplish this, the next measure is important.

**Outline a standard for managing DDL and SQL scripts.** As usual, to best manage automated processes on a larger scale, a standard mechanism for storing database scripts aids the implementation of standard techniques.

**Provide automated integration mechanisms.** In the same way that the scripts can be added automatically to source control, as long as they are available we can automate the running of the scripts to provide the migration to the required database.

These measures do not offer a panacea to database integration issues but do provide a framework and standards—and therefore a set of measures/safeguards—to database integration. The developer is still responsible for diligently preparing the scripts for database development, though.

We need to consider how best to apply these potential solutions in our given scenario.

## Database Scenarios

Most code development these days involves using a "local development" model. In other words, the code is created on the developers' local machines, and the shared source control database is the integration point and provides the mechanism for sharing new code. The automated delivery processes then provide the isolated continuous integration process as already specified. Database development is not necessarily handled in the same way, but it can be. The actual model used affects the requirements for handling database integration. Figure 8-1 shows the local database development model.

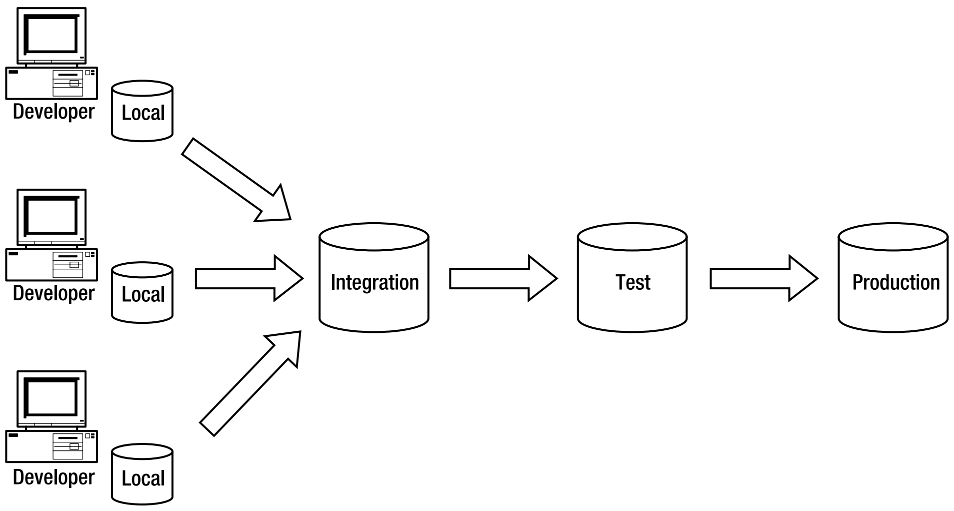


**Figure 8-1.** *Local database development*

This model demonstrates the use of local SQL Server instances on developers' machines. In this model, we need to provide database integration at the software system's build time since the integration will have to occur on a separate instance of the database for testing.

The next model, shown in Figure 8-2, is a shared development database instance.

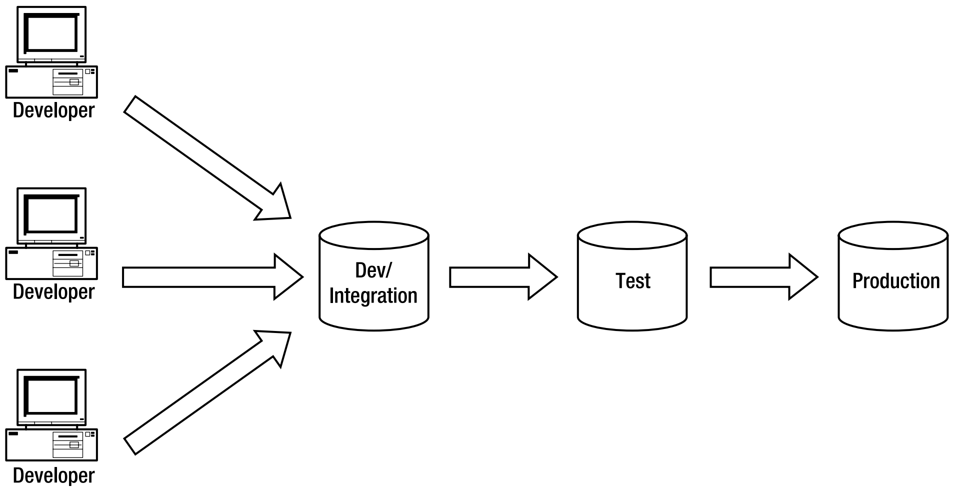


**Figure 8-2.** *Shared database development*

In this model, one database is shared between all developers. This means that regular integration occurs as a matter of course. Therefore, we do not necessarily need to run a database integration at build time since we have the complete database for testing already available. This may be risky, though, and it may be preferable to run an integration on a separate database instance in any case. Otherwise, we may just want to perform an integration when we choose to deploy the system to another environment.

Another factor is the number of database instances that require integration. We may want to perform integration to a database containing only artificial test data specifically designed for unit or system testing and then perform another integration to a database containing a copy, or agreed representative copy, of the production database.

With this in mind, it seems to make sense to ensure that the process we have in place allows for integration and testing at build time since this seems to be the most likely scenario we will come across one way or another.

## Planning the Database Tasks

Based on the above discussion, there are a few tasks we can provide as building blocks for database integration. Table 8-1 lists these tasks.

**Table 8-1.** *Required Database Tasks*

| Name | Purpose |
|---|---|
| Control | Adding new migration scripts to the source control database |
| Analyze | Assessing the database for changes and providing automated migration scripts |
| Integrate | Running migration scripts on a particular database instance |
| Configure | Ensuring that the tested/deployed code runs on the required database instance |

With those tasks (or suites of tasks), we should be able to handle our requirements for database integration. The Analyze step is a special case and goes further than in our original discussion. We had planned to work with scripts provided by developers and integrate using these scripts, but there are also possibilities for the automation of the migration itself. This will be the icing on the cake for database integration. Let us consider this list in a little more detail, and then examine the impact on the delivery process.

### Control Task

This task should provide the facilities to add new migration scripts to source control in an ordered manner to ensure that the database development and migration path is complete and auditable. In fact, this task does not need to actually use source control, as long as it manages the assets in a way acceptable for configuration management, although this may well be under source control. We manage code assets in VSS, but manage the compiled code in zip files stored (and backed up!) on the CCNet server. We could do the same thing with the database assets, bypassing the source control database, though it is preferable to maintain the assets under source control because they cannot be regenerated in the same way that compiled assemblies can.

We need to decide what scripts are required, and what to do with these scripts during an integration cycle. This is not as easy as it might first sound. Do we want to simply maintain ALTER scripts? Perhaps we want to take a full CREATE script following a successful set of tests. We may want to do the same with the test data as well. Do we remove the migration scripts following a successful build (so that they cannot be changed)? If the build fails, do we want to leave them so that they can be changed?

## Analyze Task

Providing automated database integration in terms of the actual migration scripts would be a huge boon; it significantly reduces the risk of developers failing to manage the scripts accurately and thus introducing build problems. Doing this is a complex business, though. There are two potential areas to analyze: the scripts defining the database structure, and the scripts defining the data. The database structure scripts are a good target initially since there are no semantic issues to deal with. Handling data migration and updates automatically could be a real headache and may ultimately be handled best manually. For this task we will utilize a third-party tool.

## Integrate Task

Integration itself should be quite straightforward as long as the scripts are available. This step will involve looking at the scripts available and applying them in the correct order to the correct environment. Ultimately, though, this should just be a case of running multiple script processing tasks.

## Configure Task

This task ensures that unit tests are carried out against the correct database instance and that when the system is deployed, the correct application instance points at the correct database instance. There are a few ways that this kind of work can be achieved that are also applicable to other configuration settings: the generation of configuration files on the fly, replacing individual configuration elements, or pointing the master configuration file to child configuration files from instance to instance and environment to environment.

## The Impact on Continuous Integration

We need to thread the database migration tasks into the regular process. Figure 8-3 demonstrates how the process looks with the new task inserted.

The diagram demonstrates the opportunity for us to include database integration after the compile and testing steps of the regular process. As we will see, the database integration step will consist of some substeps itself, but for now we should recognize that post-testing of the code is the preferred time for the database work. The reasons for this relate to the nature of database integration: it is to a certain extent a one-way process—rollback is at least painful—because of the lack of natural support for this sort of operation on a database platform. We will see this demonstrated later.

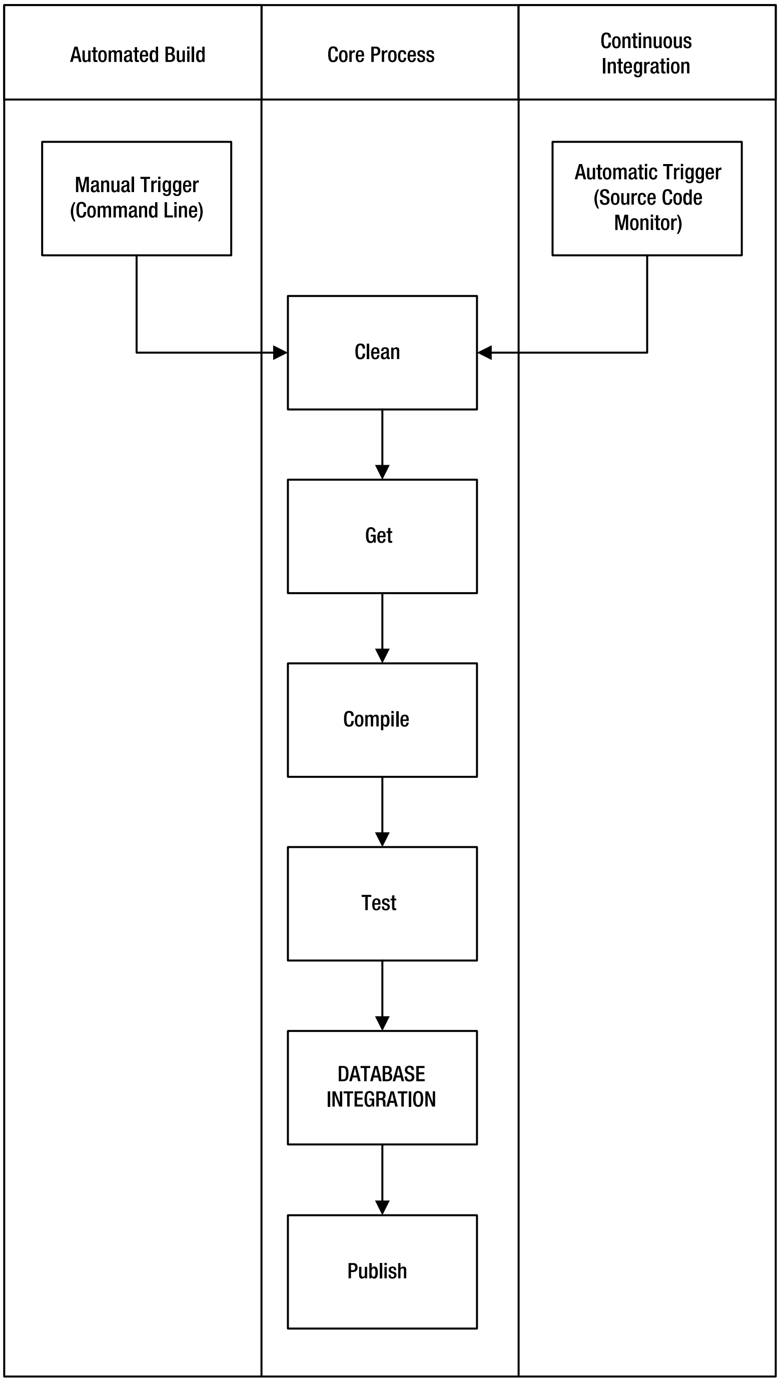


**Figure 8-3.** *CI process with database integration*

## Implementing the Database Tasks

In order to implement the database tasks, we must use a combination of existing NAnt tasks, new custom tasks (or some clever `<exec>` tasks) and the aforementioned third-party tool to handle automated database integration.

We will tackle the Analyze and Integrate tasks initially, then concern ourselves with source control and configuration issues after we have handled the guts of the work. In looking at the Analyze and Integrate tasks, we will effectively be looking at two differing solutions: one for manual scripts and one for automated scripts for the database integration. We will consider the task for processing manually generated SQL scripts first.

### Manual SQL Script Processing Task

So, as discussed earlier, the manual integration task needs to be able to look at a series of migration scripts and execute them in the correct order on a designated target database.

Therefore, in order to use this task, we need something like the following information: the folder where the database scripts are, the correct ordering of the scripts, and the database details (server, database name, user and password credentials).

This means that the following NAnt script could do the job quite nicely:

```
<dbIntegrate
     folder="D:\BookCode\Chapter8\DBTest1\"
     compare="CreationTime"
     server="localhost"
     database="TestDB-Integration"
     uid="sa"
     pwd="w1bbl3"
/>
```

Of course, this task does not currently exist. We need to generate the code for this task, but we are used to this now from our work on the `<fxcop>` task in Chapter 7.

---

■ **Note** As usual, the code for this task can be found in the VSS database.

---

The code for the integration is actually quite simple. The project for this particular task looks like the one shown in Figure 8-4.

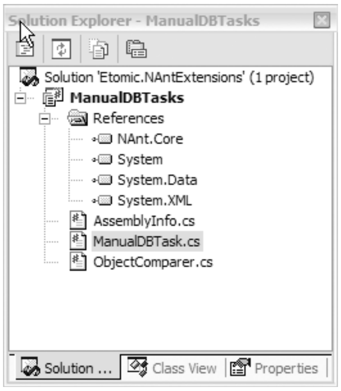


**Figure 8-4.** *ManualDBTasks project*

---

■**Note** The code for this chapter could be included as part of the previous NAntExtensions solution, but I have separated the projects for clarity.

---

The code for all six XML attributes is broadly the same and is quite simple. I could have used a `fileset` for the script folder(s), but this would confuse the ordering of the scripts and is probably not the desired scenario.

```
private DirectoryInfo _folder;
private string _compareOption;

//Database Info
private string _server;
private string _database;
private string _username;
private string _password;

[TaskAttribute("folder", Required=true)]
public DirectoryInfo Folder
{
     get{return _folder;}
     set{_folder = value;}
}

/// <summary>
/// Available options "Name", "LastWriteTime", "CreationTime"
/// </summary>
[TaskAttribute("compare", Required=true)]
public string CompareOption
```

```
{
     get{return _compareOption;}
     set{_compareOption = value;}
}

[TaskAttribute("server", Required=true)]
public string Server
{
     get{return _server;}
     set{_server = value;}
}

[TaskAttribute("database", Required=true)]
public string Database
{
     get{return _database;}
     set{_database = value;}
}

[TaskAttribute("uid", Required=true)]
public string Username
{
     get{return _username;}
     set{_username = value;}
}

[TaskAttribute("pwd", Required=true)]
public string Password
{
     get{return _password;}
     set{_password = value;}
}
```

The key points to notice about the previous code is the use of a `DirectoryInfo` type for the folder, which NAnt can handle automatically, and the available options for the `Compare➥Option`. These are not particularly friendly ways of describing the sorting options for the folder, but are in fact the way that the .NET Framework describes the options. Since we will be using a reflective comparer, it is easier to use the default names. The three options mentioned (`Name`, `LastWriteTime`, `CreationTime`) are not exhaustive but are the most likely ordering to be needed for our purposes (more complete code should limit these options, of course).

---

**■Note** The code for the comparer is held in the `ObjectComparer.cs` file. This is a useful general-purpose comparer and is ideal for the task at hand.

---

The <execute> method then looks like this:

```
protected override void ExecuteTask()
{
   //Get and sort the files
   FileInfo[] files = _folder.GetFiles();
   Array.Sort(files, new ObjectComparer(new String[]{_compareOption}));

   //Execute the SQL into the database
   foreach(FileInfo fi in files)
   {
     Log(Level.Info, fi.Name);
     ExecTask e = new ExecTask();
     e.Project = this.Project;
     e.FileName = @"osql.exe";
     e.CommandLineArguments = String.Format(@"-U {0} -P {1} -S {2} -d {3} -i {4}", ➥
                              this._username, this._password,➥
                              this._server, this._database, fi.FullName);
          e.Execute();
     }
}
```

This method is deceptively simple and makes use of the aforementioned comparer to provide a set of ordered files followed by the internal use of the NAnt <exec> task to complete its work. Once the files are ordered, the task loops through the list and creates a new <exec> task, attaching it to the current project, setting the arguments as required, and then executing the task. The effect is to dynamically generate as many tasks as required to complete the execution of all the database scripts. If written as a regular NAnt task, the <exec> task would look like the following (as an example):

```
<exec
   program="osql.exe"
   commandline="-U sa -P w1bbl3 -S localhost -d TestDB-Integration -i ➥
C:\BookCode\Chapter8\DBTest1\Users.sql"
/>
```

Using the <exec> task facilities saves us from having to handle all of the SQL-DMO bits and pieces that would otherwise be needed through this automation and therefore provides us with a simple and elegant solution that will suffice for our current needs.

We can test this custom task in the usual way: using a NAnt script. We will create a simple database TestDB-Development by using scripts. These scripts can then be used to update the integration database TestDB-Integration. The database scripts are as follows. The first, Users.sql, contains the initial CREATE script for a table Users:

```
CREATE TABLE [dbo].[Users] (
     [ID] [uniqueidentifier] NOT NULL ,
     [Name] [char] (50) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
     [Email] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL
)
GO

ALTER TABLE [dbo].[Users] ADD CONSTRAINT [PK_Users] PRIMARY KEY CLUSTERED  ([ID])
GO
```

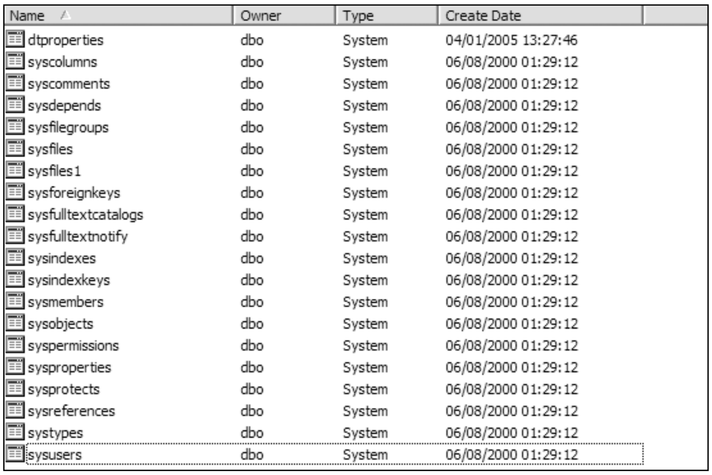
The next script, `Users-AddPostcode.sql`, alters the Users table and adds a new column, PostCode, to the existing table:

```
ALTER TABLE [dbo].[Users] ADD [Postcode] [char] (10) COLLATE
SQL_Latin1_General_CP1_CI_AS NULL
GO
```

The NAnt script to test out the new task is as follows, and can sit in the `debug/bin` folder for the extensions project:

```
<?xml version="1.0"?>
<project>
     <loadtasks assembly="Etomic.ManualDBTasks.dll"/>
     <dbIntegrate
        folder="D:\BookCode\Chapter8\DBTest1\"
        compare="CreationTime"
        server="localhost"
        database="TestDB-Integration"
        uid="sa"
        pwd="w1bbl3"
     />
</project>
```

Prior to the execution of the NAnt script, the database contains no user tables, as can be seen in Figure 8-5.



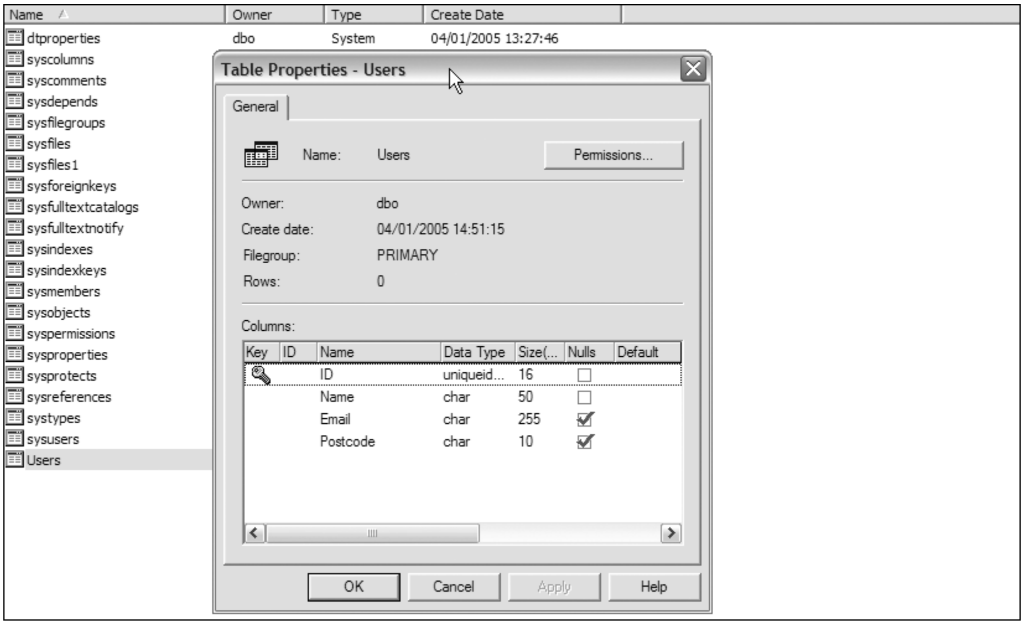
| Name | Owner | Type | Create Date |
|---|---|---|---|
| dtproperties | dbo | System | 04/01/2005 13:27:46 |
| syscolumns | dbo | System | 06/08/2000 01:29:12 |
| syscomments | dbo | System | 06/08/2000 01:29:12 |
| sysdepends | dbo | System | 06/08/2000 01:29:12 |
| sysfilegroups | dbo | System | 06/08/2000 01:29:12 |
| sysfiles | dbo | System | 06/08/2000 01:29:12 |
| sysfiles1 | dbo | System | 06/08/2000 01:29:12 |
| sysforeignkeys | dbo | System | 06/08/2000 01:29:12 |
| sysfulltextcatalogs | dbo | System | 06/08/2000 01:29:12 |
| sysfulltextnotify | dbo | System | 06/08/2000 01:29:12 |
| sysindexes | dbo | System | 06/08/2000 01:29:12 |
| sysindexkeys | dbo | System | 06/08/2000 01:29:12 |
| sysmembers | dbo | System | 06/08/2000 01:29:12 |
| sysobjects | dbo | System | 06/08/2000 01:29:12 |
| syspermissions | dbo | System | 06/08/2000 01:29:12 |
| sysproperties | dbo | System | 06/08/2000 01:29:12 |
| sysprotects | dbo | System | 06/08/2000 01:29:12 |
| sysreferences | dbo | System | 06/08/2000 01:29:12 |
| systypes | dbo | System | 06/08/2000 01:29:12 |
| sysusers | dbo | System | 06/08/2000 01:29:12 |

**Figure 8-5.** *Empty TestDB-Integration*

Executing the test NAnt script produces the following results:

```
---------- NAnt ----------
NAnt 0.85
Copyright (C) 2001-2004 Gerry Shaw
http://nant.sourceforge.net

Buildfile: file:///NAntExtenstions.ManualDBTasks.Debug.xml

[loadtasks] Scanning assembly "Etomic.ManualDBTasks" for extensions.
[dbIntegrate] Users.sql
     [exec] 1> 2> 3> 4> 5> 6> 1> 2> 3> 1>
[dbIntegrate] Users-AddPostcode.sql
     [exec] 1> 2> 1>

BUILD SUCCEEDED
Total time: 0.9 seconds.
Output completed (2 sec consumed) - Normal Termination
```

The results demonstrate that the custom task assembly is loaded, and that the `<dbIntegrate>` task executes the `Users.sql` and `Users-AddPostcode.sql` scripts in the correct order (that is, by `CreationTime`). We can also see the nested `<exec>` tasks being executed for each script. Following this execution, the Users table is included in the database TestDB-Integration, as shown in Figure 8-6.



| Name | Owner | Type | Create Date |
|---|---|---|---|
| dtproperties | dbo | System | 04/01/2005 13:27:46 |
| syscolumns | | | |
| syscomments | | | |
| sysdepends | | | |
| sysfilegroups | | | |
| sysfiles | | | |
| sysfiles1 | | | |
| sysforeignkeys | | | |
| sysfulltextcatalogs | | | |
| sysfulltextnotify | | | |
| sysindexes | | | |
| sysindexkeys | | | |
| sysmembers | | | |
| sysobjects | | | |
| syspermissions | | | |
| sysproperties | | | |
| sysprotects | | | |
| sysreferences | | | |
| systypes | | | |
| sysusers | | | |
| Users | | | |

Table Properties - Users

General

Name: Users

Owner: dbo
Create date: 04/01/2005 14:51:15
Filegroup: PRIMARY
Rows: 0

Columns:

| Key | ID | Name | Data Type | Size(... | Nulls | Default |
|---|---|---|---|---|---|---|
| 🔑 | | ID | uniqueid... | 16 | ☐ | |
| | | Name | char | 50 | ☐ | |
| | | Email | char | 255 | ✓ | |
| | | Postcode | char | 10 | ✓ | |

**Figure 8-6.** *Table Users in database TestDB-Integration*

We could tidy up and bulletproof the code for the integration task, but essentially this task now does what we need it to do as part of the delivery process.

---

■**Note** The same script can also be used to run SQL scripts rather than DDL scripts. We will see this later on.

---

## Automated Integration Task

Next we turn our attention to the automation of the migration scripts themselves. Generating migration scripts is no small task; it is not a simple matter of exporting the database schema. Something like that will perhaps suffice for the generation of a CREATE script for a database, which is useful in itself, but automated migration scripts require the use of ALTER scripts, the removal and addition of constraints, and other tricks and techniques to modify a schema while the database itself contains data. This is something best left to those who specialize in that, and I certainly do not.

Instead, the following tools are a huge boon to the task at hand. Better still, they have fully exposed .NET APIs for programming against, and that can mean only one thing: even more custom NAnt tasks!

### The Red Gate Tools

Red Gate Software Ltd. (`www.red-gate.com`) produces a suite of tools known as the Red Gate SQL Bundle, which contains various tools for SQL Server, as shown in Table 8-2.

**Table 8-2.** *Red Gate SQL Bundle*

| Tool | Description |
|---|---|
| SQL Compare | Provides comparison and synchronization scripts of schemas of SQL Server databases |
| SQL Data Compare | Provides comparison and synchronization scripts of data held on SQL Server databases |
| DTS Compare | Provides comparisons of server and DTS configurations for SQL Server |
| SQL Packager | Packages a SQL Server (data and schema) into either a C# project or an executable for deployment elsewhere |
| SQL Toolkit | Provides access to the API for extension and use of the SQL Compare, SQL Data Compare, and SQL Packager tools |

---

■**Tip** You can download a trial 14-day version from the web site that should allow you to test the concepts and code in this chapter.

---

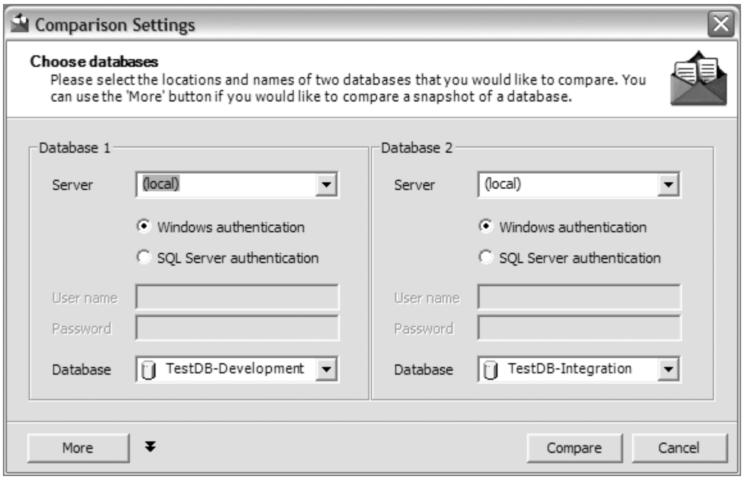
## Using the Tools

Let us take a quick look at the use of the tools as they come out of the box, though the majority of the work we will do uses the APIs in order to work within NAnt.

There are two flavors of tools in the current version: the regular GUI tools and also a set of command-line tools, which are just as feature-rich and probably more accessible for rapid comparisons and synchronization.

We will concentrate on the SQL Compare tool, which is used to compare and synchronize database schemas. However, the SQL Data Compare and DTS Compare are very similar in terms of operation. The Packager application is a little different—its core responsibility is not comparison and synchronization, but instead it handles the packaging of a database into an executable program or C# project.

Firing up the GUI produces a "nag" screen if you are using an unlicensed copy of the software. After this, a screen asking for connection information will appear. This can be used directly as shown in Figure 8-7, or it can be canceled and a preexisting project can be loaded.


Comparison Settings

Choose databases

Please select the locations and names of two databases that you would like to compare. You can use the 'More' button if you would like to compare a snapshot of a database.

Database 1
Server (local)
[✓ Windows authentication]
SQL Server authentication
User name
Password
Database TestDB-Development

Database 2
Server (local)
[✓ Windows authentication]
SQL Server authentication
User name
Password
Database TestDB-Integration

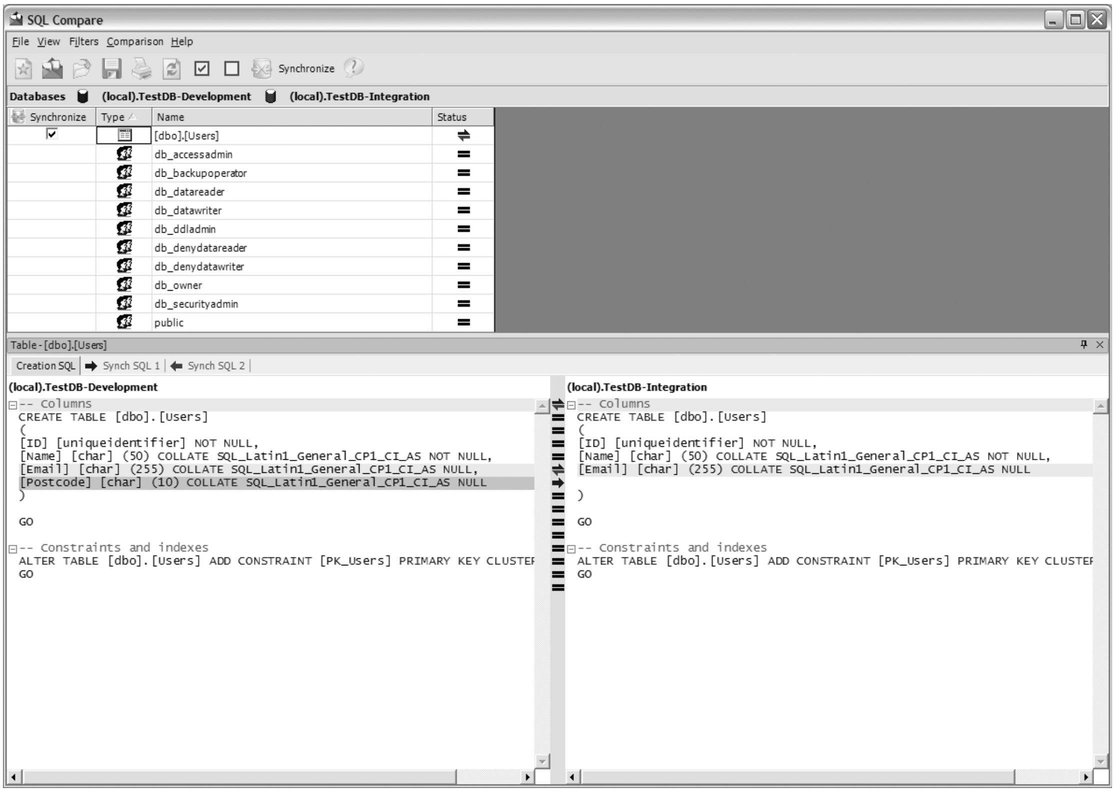
**Figure 8-7.** *The SQL Comparison Settings screen*

---

■ **Tip** The help files that come with the product are more comprehensive than those found with many products, although the tool is relatively straightforward to use.

---

After we enter the information and click OK, SQL Compare does its magic and compares the two databases. We are dealing with very simple databases and so the information provided is minimal. In this instance, I have removed the PostCode column from the TestDB-Integration database; the effects of this can be seen in Figure 8-8.


SQL Compare
File View Filters Comparison Help
Synchronize
Databases
(local).TestDB-Development
(local).TestDB-Integration

| Synchronize | Type | Name | Status |
|---|---|---|---|
| ✓ | | [dbo].[Users] | ⇌ |
| | | db_accessadmin | = |
| | | db_backupoperator | = |
| | | db_datareader | = |
| | | db_datawriter | = |
| | | db_ddladmin | = |
| | | db_denydatareader | = |
| | | db_denydatawriter | = |
| | | db_owner | = |
| | | db_securityadmin | = |
| | | public | = |

(local).TestDB-Development

```
-- Columns
CREATE TABLE [dbo].[Users]
(
[ID] [uniqueidentifier] NOT NULL,
[Name] [char] (50) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
[Email] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
[Postcode] [char] (10) COLLATE SQL_Latin1_General_CP1_CI_AS NULL
)

GO

-- Constraints and indexes
ALTER TABLE [dbo].[Users] ADD CONSTRAINT [PK_Users] PRIMARY KEY CLUSTER
GO
```

(local).TestDB-Integration

```
-- Columns
CREATE TABLE [dbo].[Users]
(
[ID] [uniqueidentifier] NOT NULL,
[Name] [char] (50) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
[Email] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL

)

GO

-- Constraints and indexes
ALTER TABLE [dbo].[Users] ADD CONSTRAINT [PK_Users] PRIMARY KEY CLUSTER
GO
```

**Figure 8-8.** *A simple database comparison*

Figure 8-8 shows that SQL Compare detected the differences between the two databases and produced the CREATE scripts (with highlighted differences) for the two databases. Additionally, the synchronization, or ALTER, scripts can be viewed by clicking the relevant tabs.

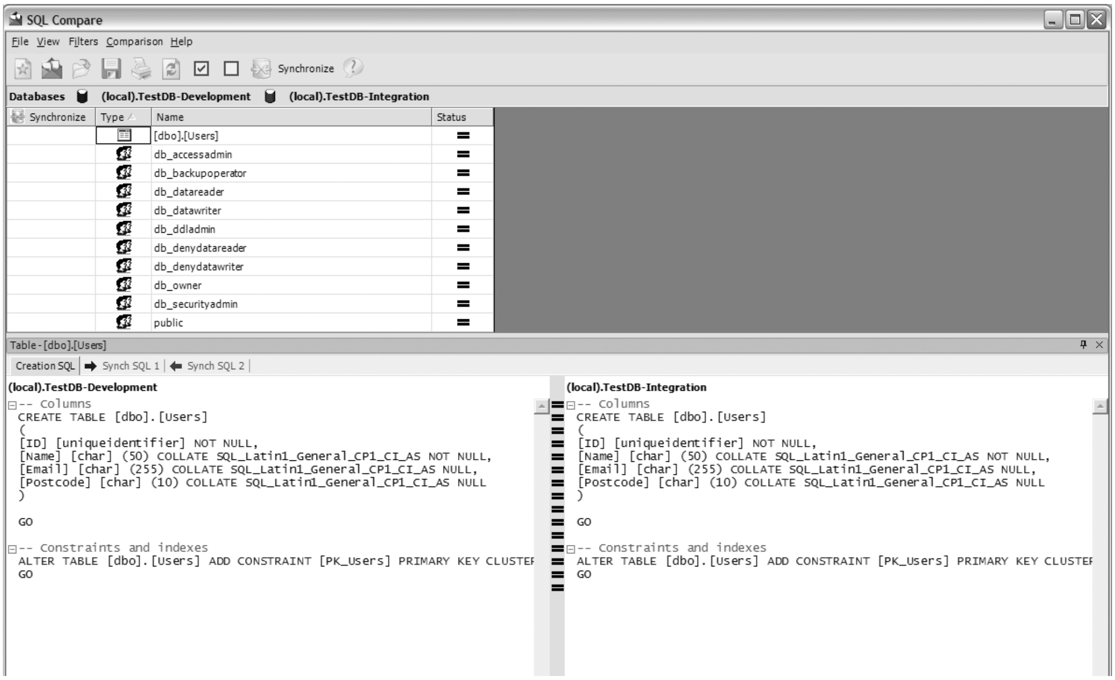
Clicking the Synchronize button then walks the user through a series of screens and results in the execution of the relevant script to provide the database synchronization. Figure 8-9 shows the screen after TestDB-Integration has been updated to the latest development version.

So that is a simple example of the use of the SQL Compare tool. Its real power and utility comes in its continuous use to provide migration capabilities without developers having to think too hard about it, and in its use with more complex databases. Here, for instance, it is not possible to see how SQL Compare handles the addition and removal of constraints in the correct order to ensure error-free schema changes.

---

■ **Tip** If you make wholesale changes to database design—such as the change of a primary key data type—then you may well run into some problems. SQL Compare cannot do everything, but it can do quite a lot. (I would spend more time worrying about the decision to change the data type . . .) The point is that SQL Compare is more likely to be successful when used in small steps in a continuous way—in precisely the same way as unit testing and continuous integration is more successful with small steps.

---

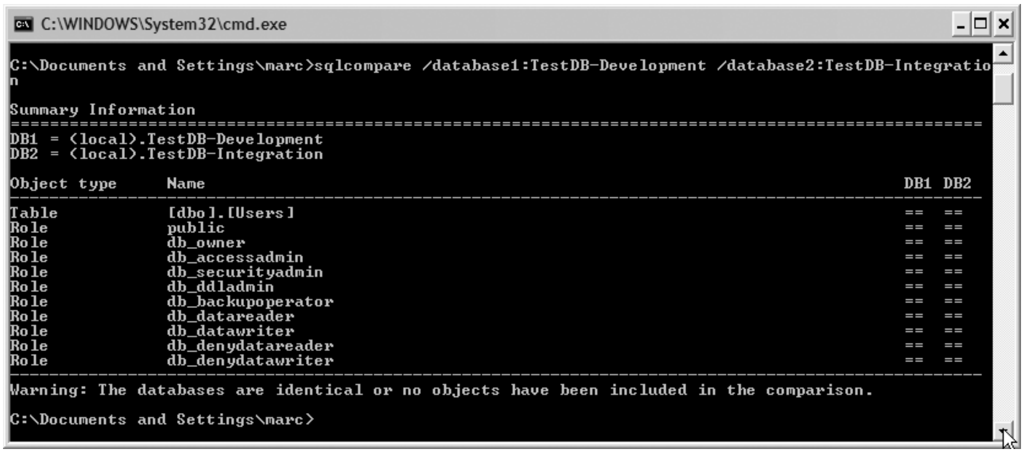

SQL Compare
File View Filters Comparison Help
Synchronize
Databases (local).TestDB-Development (local).TestDB-Integration
Synchronize Type Name Status
[dbo].[Users]
db_accessadmin
db_backupoperator
db_datareader
db_datawriter
db_ddladmin
db_denydatareader
db_denydatawriter
db_owner
db_securityadmin
public
Table - [dbo].[Users]
Creation SQL Synch SQL 1 Synch SQL 2
(local).TestDB-Development
(local).TestDB-Integration

```
-- Columns
CREATE TABLE [dbo].[Users]
(
[ID] [uniqueidentifier] NOT NULL,
[Name] [char] (50) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
[Email] [char] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
[Postcode] [char] (10) COLLATE SQL_Latin1_General_CP1_CI_AS NULL
)

GO

-- Constraints and indexes
ALTER TABLE [dbo].[Users] ADD CONSTRAINT [PK_Users] PRIMARY KEY CLUSTEF
GO
```

**Figure 8-9.** *SQL Compare following synchronization*

We can also use the command line to perform the same operations. The command-line tool comes with an enormous amount of switches and parameter possibilities and so is easiest to use when performing regular activities such as quick comparisons.

We can use the command shown in Figure 8-10 to perform a comparison between the two databases.



```
C:\Documents and Settings\marc>sqlcompare /database1:TestDB-Development /database2:TestDB-Integratio
n

Summary Information
====================================================================================================
DB1 = (local).TestDB-Development
DB2 = (local).TestDB-Integration

Object type      Name                                                                       DB1  DB2
----------------------------------------------------------------------------------------------------
Table            [dbo].[Users]                                                              ==   ==
Role             public                                                                     ==   ==
Role             db_owner                                                                   ==   ==
Role             db_accessadmin                                                             ==   ==
Role             db_securityadmin                                                           ==   ==
Role             db_ddladmin                                                                ==   ==
Role             db_backupoperator                                                          ==   ==
Role             db_datareader                                                              ==   ==
Role             db_datawriter                                                              ==   ==
Role             db_denydatareader                                                          ==   ==
Role             db_denydatawriter                                                          ==   ==
----------------------------------------------------------------------------------------------------
Warning: The databases are identical or no objects have been included in the comparison.

C:\Documents and Settings\marc>
```

**Figure 8-10.** *SQL Compare command line*

As can be seen, the databases are fully synchronized, as we would expect from the efforts using the SQL Compare GUI.

## Automating the Red Gate Bundle

The APIs for SQL Compare and the other tools in the bundle are well documented, and there are also useful code samples that demonstrate how to synchronize databases.

While the previous task is useful, effectively it is simply ordering and executing a series of SQL scripts. It is flexible and can be used as needed in whatever database scenario we have.

This time, since we are expecting SQL Compare to do all of the work in terms of script production as well as synchronization, we need to think a little about what the deliverables from the process should be. In terms of sensible assets to maintain for a build, we should probably maintain a full CREATE script for the database in the current build and then also maintain the migration necessary for a move from the prior build to the current one. With these two assets, we can then synchronize the integration environments or create a new integration environment as desired.

We should also consider that, as discussed previously, we may wish to synchronize multiple database instances, though we may only want to maintain synchronization assets for one of the instances; after all, the databases should be the same.

With these points in mind, the following task would be useful:

```
<dbAutoIntegrate
     folder="D:\BookCode\Chapter8\DBTest2\"
     server="localhost"
     database="TestDB-Development"
     uid="sa"
     pwd="w1bbl3"
     write="true"
     caption="0"
     >
     <databases>
          <database server="localhost" database="TestDB-Integration" ➥
                    uid="sa" pwd="w1bbl3" write="true"/>
          <database server="localhost" database="TestDB-System" ➥
                    uid="sa" pwd="w1bbl3" write="true"/>
     </databases>
</dbAutoIntegrate>
```

The task will have a few features. The attributes in the main element contain the standard four pieces of database information (server, database, username, and password) as well as three other pieces of information: `folder` is the folder in which the scripts should be stored when produced, `write` tells the task whether to produce a script for that particular step (in the case of the main element, whether to produce the CREATE script), and `caption` allows us to pass through information to assist in the naming of the produced scripts (we would ordinarily pass through the build number).

The task is a little more complicated by the inclusion of child elements describing multiple target databases for synchronization. In the previous example I have included the regular integration database and also a System Test database server. I have marked both to have the scripts produced, but in fact I would probably only maintain the migration scripts for the integration database. We will make a new type to represent these child elements in the main task. We will call this new type DBInfo. It will be a custom NAnt element that we will create to hold the information for each target database.

Implementing the DBInfo type is straightforward (in fact, thinking of a name for the type was harder since the SQL Compare APIs use Database):

```
[ElementName("database")]
public class DBInfo : Element
{
     private string _server;
     private string _database;
     private string _username;
     private string _password;

     private bool _write;

     [TaskAttribute("server", Required=true)]
     public string Server
     {
          get{return _server;}
          set{_server = value;}
     }

     [TaskAttribute("database", Required=true)]
     public string Database
     {
          get{return _database;}
          set{_database = value;}
     }

     [TaskAttribute("uid", Required=true)]
     public string Username
     {
          get{return _username;}
          set{_username = value;}
     }

     [TaskAttribute("pwd", Required=true)]
     public string Password
     {
          get{return _password;}
          set{_password = value;}
     }

     [TaskAttribute("write", Required=true), BooleanValidator()]
     public bool Write
     {
            get{return _write;}
            set{_write = value;}
        }
}
```

Therefore, the `DBInfo` type is just a property bag with no logic of its own. We will see how this is used in the task implementation that follows.

The task is going to behave as shown in Figure 8-11.

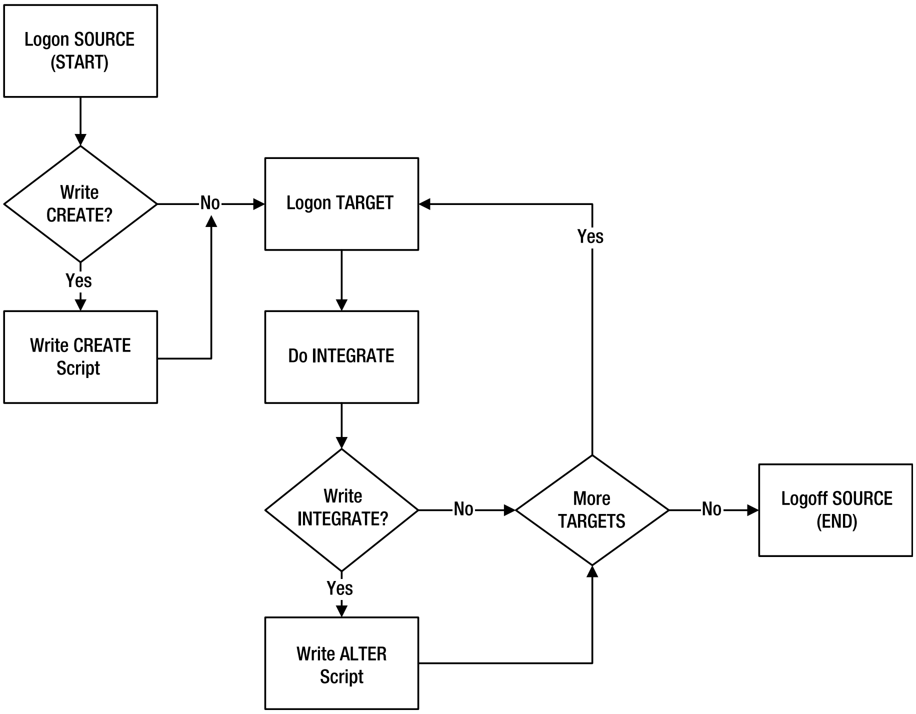


**Figure 8-11.** *<dbAutoIntegrate> task process*

As usual, the first part of the code for this task involves setting up the attributes and elements of the task, as follows:

```
private DirectoryInfo _folder;

//Database Info
private string _server;
private string _database;
private string _username;
private string _password;

private DBInfo[] _dbInfos;
private bool _write;
private string _caption;

private Database _source;

[TaskAttribute("folder", Required=true)]
public DirectoryInfo Folder
{
     get{return _folder;}
     set{_folder = value;}
}

[TaskAttribute("server", Required=true)]
public string Server
{
     get{return _server;}
     set{_server = value;}
}

[TaskAttribute("database", Required=true)]
public string Database
{
     get{return _database;}
     set{_database = value;}
}

[TaskAttribute("uid", Required=true)]
public string Username
{
     get{return _username;}
     set{_username = value;}
}

[TaskAttribute("pwd", Required=true)]
public string Password
{
     get{return _password;}
     set{_password = value;}
}

[BuildElementCollection("databases", "database")]
public DBInfo[] DBInfos
{
     get{return _dbInfos;}
     set{_dbInfos = value;}
}
```

```
[TaskAttribute("write", Required=true), BooleanValidator()]
public bool Write
{
     get{return _write;}
     set{_write = value;}
}

[TaskAttribute("caption", Required=true)]
public string Caption
{
     get{return _caption;}
     set{_caption = value;}
}
```

There are two things to notice in this code. The first is the inclusion of a class-level `Database` type. The source database is used throughout the task and is declared here. The `Database` type is from the SQL Compare APIs. The second is the inclusion of an array of `DBInfo` types, as we described earlier. I could have used this for the source database, too, but seeing as there will never be more than one source database, I have not done so. Here I am using an array since it is all that is needed for the operations the task will perform, but this could equally have been a strongly typed collection.

■ **Tip** Digging around in the NAnt source code will provide many examples of the use of nested elements, too.

To use the collection, a different attribute is required for the property: `BuildElement➥ Collection`. This attribute accepts the names of the element and child elements (notice that it does not have to be named the same as the type) and allows the task to be built as the sample script earlier showed.

The next step is to override the `ExecuteTask` method on the task itself as usual:

```
protected override void ExecuteTask()
{
     _source = new Database();
     _source.Register(new ConnectionProperties(this._server, ➥
                                               this._database, ➥
                                               this._username, ➥
                                               this._password), ➥
                                               Options.Default);

     if(this._write)DoCreateScript();
     DoAlterScripts();

     _source.Dispose();
}
```

This task instantiates and prepares the source database. It then hands off responsibility to other internal methods for the actual work before disposing of the database as per the standards described in the API documentation.

---

■ **Note** As usual, my code does not contain supporting code such as error handling. For production code, I consider it to be imperative to trap and handle errors, particularly with items such as the database registration in the above code.

---

We can take a look at the `DoCreateScript` and `DoAlterScripts` methods next.

```
private void DoCreateScript()
{
   Log(Level.Info, "Handling CREATE Script");
   ExecutionBlock createScript = CalculateScript(null);
   WriteScript(String.Format("CREATE-{0}.sql", _caption), ➥
                               createScript.ToString());
   createScript.Dispose();
}

private void DoAlterScripts()
{
   foreach(DBInfo db in _dbInfos)
   {
    Log(Level.Info, String.Format("Handling migration for {0}", db.Database));

    Database target = new Database();
    target.Register(new ConnectionProperties(db.Server, db.Database, ➥
                                             db.Username, db.Password),➥
                                             Options.Default);

    ExecutionBlock alterScript = CalculateScript(target);
    ExecuteScript(alterScript, target);

    if(db.Write) WriteScript(String.Format("ALTER-{0}-{1}.sql", db.Database, ➥
                                _caption), alterScript.ToString());

          alterScript.Dispose();
          target.Dispose();
     }
}
```

`DoCreateScript` is quite simple, performing a logging action and then handing responsibility to another method for the actual script calculation, and then to another method for the writing of the script. You can see that the script will be named "CREATE-", followed by the required caption as specified in the NAnt script, for example, the project name and/or build number.

DoAlterScripts is a little more involved. It loops through the target databases, creating new Database types for each and then again calculating the scripts and writing them (with the name format ALTER-<database>-<caption>) if necessary. Additionally, once the script is calculated it is executed on the target database, providing the required synchronization.

Both of these methods use the common methods described next for script calculation, execution, and writing. Where DoAlterScripts passes a target database to the CalculateScript method, DoCreateScript passes a null to the method. This feature of the APIs forces a full migration script—effectively a CREATE script—to be produced and so is ideal for scripting the full database schema. Both scripts also dispose of the blocks generated, as per the API standards. If you look at the examples provided with the bundle, you will see that the code is similar in terms of the actual work being done.

The three remaining methods look like this:

```
private ExecutionBlock CalculateScript(Database target)
{
   Differences differences = _source.CompareWith(target, Options.Default);
   foreach (Difference difference in differences)
   {
      difference.Selected=true;
   }

   Work work=new Work();
   work.BuildFromDifferences(differences, Options.Default, true);
   return work.ExecutionBlock;
}

private void ExecuteScript(ExecutionBlock script, Database target)
{
   Utils utils = new Utils(); //This is a RedGate Utils Class
   utils.ExecuteBlock(script, ➥
                         target.ConnectionProperties.ServerName, ➥
                         target.ConnectionProperties.DatabaseName, ➥
                         target.ConnectionProperties.IntegratedSecurity, ➥
                         target.ConnectionProperties.UserName, ➥
                         target.ConnectionProperties.Password);
}

private void WriteScript(string caption, string content)
{
   Log(Level.Info, String.Format("Writing script {0}", caption));
   FileStream script = new FileStream(Path.Combine(this._folder.FullName, ➥
                                          caption), FileMode.Create);
   StreamWriter sw = new StreamWriter(script);
   sw.Write(content);
   sw.Close();
   script.Close();
}
```

---

**Note** Ah, the pleasure of third-party APIs. With my nitpicking hat on, I would say that it would be very useful if there were a few changes such as a method `differences.SelectAll` to save looping through the collection, and it would be nice if `ExecuteBlock` could accept a `Database` type instead of the individual connection information. Grumble, grumble.

---

I have not included much in the way of feedback in the task. It would probably be sensible to include some suitable debugging logging statements, such as writing out the SQL to the NAnt logger and so on.

With that, the job is done. In order to test the task we can create a simple scenario with multiple databases to run the automatic integration. Let us use the same trivial example as before—the useless Users table—and set up the databases as shown in Table 8-3.

**Table 8-3.** *Test Databases*

| Database | Settings |
|---|---|
| TestDB-Development | The full Users table, with the PostCode column |
| TestDB-Integration | The Users table without the PostCode column |
| TestDB-System | The Users table without the PostCode column |

We can verify that the databases are not equal by using SQL Compare.

We will use the NAnt task to generate the full script for TestDB-Development, and then to script and synchronize the two testing databases. This looks as follows:

```
<?xml version="1.0"?>
<project>
     <loadtasks assembly="Etomic.NAntExtensions.RedGateDBTasks.dll"/>
     <dbAutoIntegrate
          folder="D:\BookCode\Chapter8\DBTest2\"
          server="localhost"
          database="TestDB-Development"
          uid="sa"
          pwd="w1bbl3"
          write="true"
          caption="0"
          >
          <databases>
               <database server="localhost" database="TestDB-Integration"
                         uid="sa" pwd="w1bbl3" write="true"/>
               <database server="localhost" database="TestDB-System"
                         uid="sa" pwd="w1bbl3" write="true"/>
          </databases>
</dbAutoIntegrate>
</project>
```

Because this is a debug script, I am passing `0` as the caption. Running the script produces the results shown here.

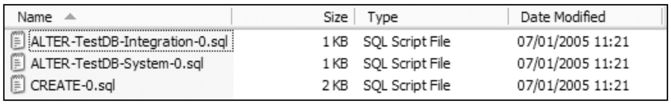
---

**Note** Bear in mind that if you are using the unlicensed version of SQL Compare, then a nag screen will be shown. This precludes the use of the task in an automated environment because of the interactive nagging.

---

```
---------- NAnt (RedGate) ----------
NAnt 0.85
Copyright (C) 2001-2005 Gerry Shaw
http://nant.sourceforge.net

Buildfile: file:/// AntExtensions.RedGateDBTasks.Debug.xml
Target framework: Microsoft .NET Framework 1.1

[loadtasks]
   Scanning assembly "Etomic.NAntExtensions.RedGateDBTasks" for extensions.
[dbAutoIntegrate] Handling CREATE Script
[dbAutoIntegrate] Writing script CREATE-0.sql
[dbAutoIntegrate] Handling migration for TestDB-Integration
[dbAutoIntegrate] Writing script ALTER-TestDB-Integration-0.sql
[dbAutoIntegrate] Handling migration for TestDB-System
[dbAutoIntegrate] Writing script ALTER-TestDB-System-0.sql

BUILD SUCCEEDED
Total time: 6.5 seconds.
Output completed (7 sec consumed) - Normal Termination
```

The output is as we would expect. If we look into the output folder, we will find the scripts shown in Figure 8-12.



| Name | Size | Type | Date Modified |
|---|---|---|---|
| ALTER-TestDB-Integration-0.sql | 1 KB | SQL Script File | 07/01/2005 11:21 |
| ALTER-TestDB-System-0.sql | 1 KB | SQL Script File | 07/01/2005 11:21 |
| CREATE-0.sql | 2 KB | SQL Script File | 07/01/2005 11:21 |

**Figure 8-12.** *Output from the <dbAutoIntegrate> task*

These scripts contain the output we would expect from running SQL Compare manually to perform the same actions. Performing a `diff` on the two synchronization scripts demonstrates that they are the same. Running the CREATE script in a new database and then performing a SQL Compare on the databases will demonstrate that they are the same too.

With that, our database schema synchronization woes are banished on a practical level. We can consider some process implications a little later.

## Thinking about Data Synchronization

The tasks we have created were done so with database schema migration and synchronization in mind, although as stated, the manual task will handle any arbitrary SQL.

We do need to consider how to handle data synchronization. In the scenario earlier, the task synchronizes the schema for an integration database that may be used for unit-testing purposes, and a database used for system testing, which may contain a different set of data—for example, a representation of the production system data. Therefore, the data migration requirements could well be different for each database:

**Integration database.** The integration database may need to be fully refreshed with data for unit testing, or just some standing data such as ISO country codes. Good unit tests should create their own data and then destroy it, but we all know that unit tests come in all varieties.

**System database.** If the system database does contain a representation of the live data, then migration scripts are needed rather than refresh scripts. This is the ideal time to test those scripts for eventual use in the production environment.

So how to do this? The options are certainly the same—the processing of scripts that are manually generated or the use of SQL Data Compare (for instance) to automatically synchronize the data—but the decision may be different. Whereas synchronization is what we were trying to achieve with the database schemas, it is not our goal with data migration. Data migration also tends to be a more semantic affair than schema migration, and thus developer decision-making skills could be called for.

My feeling is that it is probably best to manually create scripts for data migration and manipulation, using the manual task to run these scripts as part of the delivery process. Tools such as SQL Data Compare could then be used to verify these scripts if desired. Martin Fowler's article on agile databases and Scott Ambler's book *Agile Database Techniques* (see "Further Reading") describe toolkits of scripts for data migration to speed up and standardize this work. This works by taking an approach to database manipulation similar to one a developer might take to code refactoring: creating specific mechanics for performing specific tasks under specific contexts. So, for example, in code refactoring we may choose to "Extract Method" and follow a mechanism to perform this refactoring; in database manipulation we may choose to "Introduce Trigger" following a mechanism to perform this work. The database examples are not quite so rounded as the original work on code refactoring, but there are a lot of ideas and examples in both of these sources. This is a useful practice for a development team. Additionally, in order to make the automation easy given the context of the task we have created, it is a good idea to organize the output of these activities. Table 8-4 shows how this might be done.

**Table 8-4.** *Organizing Database Scripts*

| Name | Content |
|---|---|
| Schema | Contains the scripts produced by the automated SQL Compare task |
| Reference | Contains the scripts for the insertion and refresh of standing data such as lookup |
| Test | Contains scripts for the insertion and refresh of test data if required |
| Migrate | Contains scripts for the migration (updates, inserts, deletes) of existing production (or production representation) data |

With the scripts organized in this way, the task we have created can be used to run the sets of scripts required given the particular database instance. Additionally, we can leverage reusability in the build scripts on a large scale.

## A Process for Database Integration

Having worked out the practical details for the actual work to be done, let us consider how we can implement database integration into the CI process. First, we return to the Control and Configure versioning tasks we identified at the outset of this chapter.

### Control and Configure Tasks

Neither task requires us to build new code for NAnt, but they do require some thought as to how the database assets should be controlled.

#### Control

As previously mentioned, the core decision for control is whether to use the source control database. If source control is used, then the files will need to be GETted from the source control database prior to any execution, and any generated files will need to be added to source control when they are produced. In itself, this is not such a problem, but the management of the files may be more awkward than simply adding the files to source control. The issue stems from the coupling of code changes and database changes in the CI process: when should the files be stored in source control? Consider the following scenarios and the possibilities they present:

> If no separate database server is used for handling unit testing, the database integration work can wait until the code has been successfully compiled and tested. Only then are the migration scripts generated and used on the integration server. At the end of the cycle, these scripts are then a specific version and migration step on the database life cycle, and so the existing scripts should be removed and stored safely (both the automatic and the manual scripts) because they have already been applied.
>
> If a separate database server is used for unit testing, then the integration (or at least some of it) needs to occur before testing of the code is complete. If a code-based error occurs, it may not be desirable to have the database integrated, but at that point it is too late. This is not necessarily a big deal here, but it might be in your own scenario.
>
> If no separate database server is used for integration of any kind (perhaps migration scripts are generated during comparison to the production server but not actually executed), then the scripts need to be removed and stored since they are still a specific version at that point in the build. Although this sounds like a poor scenario, consider the next scenario.

If there is always integration to the integration server, then the migration scripts *must* be applied incrementally and sequentially to the production server when the system is finally deployed. If a system undergoes 20 build cycles before an actual release, all 20 database integrations must be applied since the changes are made only once to the static database server. In the previous scenario, the migration scripts are formed against the currently released platform and are not incremental. The advantage is that only one set of migration scripts is required. The disadvantage is that the increments become larger, and therefore carry more risk since they are not in the spirit of CI.

If you are going to implement database integration in a CI way, the final scenario is preferable in terms of the best process, although clearly there is some overhead to the actual deployment. With some cunning file manipulation, this could be made fairly seamless: if the deployment to the production server triggered the removal of the migration scripts rather than the build process, the correct scripts could be maintained together from release to release rather than build to build, though this will result in duplication of scripts in the code zip files (in the context of the current process). We could deal with this in a couple of ways, as we will see in a moment.

With that in mind, publishing of database assets can be handled in the same way as the web assets: through the use of zip files and an HTTP-accessible folder for GETting prior to deployment.

### Configuration

With regard to configuration, if unit tests are to be carried out on one of the integrated databases, then the configuration file for the unit tests needs to be changed, or just made available prior to the tests being run. Fortunately, there is an attribute on the `<nunit2>` task that allows us to include an application configuration file. This can easily be parameterized to ensure that the correct configuration file is used by the automated build process:

```
<nunit2>
     <formatter type="Xml" usefile="true" extension=".xml"
               outputdir="${core.reports}\" />
     <test appconfig="myconfig.config">
          <assemblies basedir="${core.output}\">
               <include name="*Tests.dll"/>
          </assemblies>
     </test>
</nunit2>
```

Therefore, this is a relatively straightforward configuration issue, though it does add another parameter to the build file, which could be standardized to the form `<solution.name>.➥ Tests.Config` or something similar. In the next section, we address some additional configuration concerns.

## Deployment Considerations

Both of the deployment issues we examine here may seem relatively minor at first glance, but they can cause a headache if not handled effectively. Let us consider three deployment scenarios; they may be applicable in any combination:

**Automatic deploy following build.** When the build is complete, we could deploy the latest version automatically. Perhaps the latest version would point at the integration database instance. In this scenario, the integration version is "just for show" and serves no purpose in the delivery pipeline (e.g., quality assurance, or QA). On the other hand, the system test database might be used for deployment, so that part of the QA pipeline is automated too. The problem here is that the build action under the CI process can occur at any time, which might not be ideal if system testing is already taking place.

**Deploy new instance.** If the deployment process is separate from the CI process, this is the first of two possibilities. In this scenario, we deploy a clean build of a specific version of the system. This is the simpler of the two scenarios.

**Integrate existing instance.** In this deployment scenario, we deploy against an existing version of the system. This is no problem in terms of code—we simply replace all code assets—though in fact we may have to consider the management of noncode assets (e.g., uploads from users) during deployment. For the database, that means applying the incremental migration scripts.

We will build scripts allowing both the deployment of a new system/database instance, and the integration of an existing instance.

## Using the Correct Database Instance

Once the site is deployed, we need to ensure that the application points to the correct database instance. This configuration issue is actually a subset of the overall configuration issues for deployment. It can be easily handled when we do not have many configuration settings to change, but the process can become unwieldy when many changes are involved. Possible strategies include the following:

**`<xmlpoke>`.** Because configuration files are XML-based, the `<xmlpoke>` task can be used to change values, add keys, and so on. On a small scale, this is a very useful task, but making several changes to files can become unwieldy and a maintenance headache.

**Configuration file linking.** General configuration settings for a system can be held in companion config files and referred to from the main config file. For example, `web.config` might refer to settings held in `dev.config`, `test.config`, and so on. Using a single `<xmlpoke>` and a few cleanup tasks could handle a bulk configuration change. The benefit of this scenario is that the data for change is held outside the actual process.

**Configuration service.** Similarly, using the Microsoft Configuration Management Block (or some other configuration service, perhaps Nini, which you can find at `http://nini.➥ sourceforge.net/`) typically means that the configuration change can be handled in the same minimal way. The benefit is the same, and in addition all configuration data across all applications is maintained in one location.

We will use the simplest scenario, the `<xmlpoke>` task, to handle the configuration changes we need. In fact, with the use of the other two scenarios, the impact to the delivery scripts is minimal: there will still need to be some kind of `<xmlpoke>` to point the configuration file to the appropriate companion or service.

### Asset Management

As mentioned earlier, the deployment scenario could entail multiple migrations to move the database version on by several builds. The deployment script should take account of this. This issue could be handled simply by passing the current and desired build numbers of the database and then looping through the database deployment routines. Or it may be worthwhile to build a custom task to handle these issues separately. We will see an implementation of this in our deployment scripts.

## Implementing Database Integration

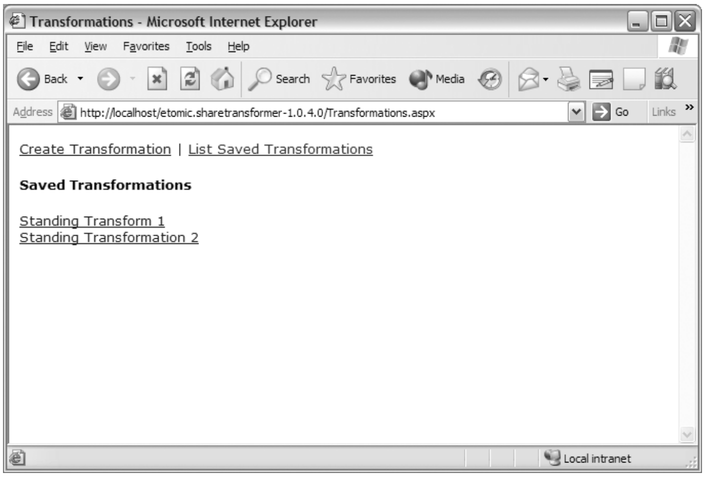
Once again, the wizards at Etomic have produced another work of software art. This time they have extended the web-based Transformer application so that it can use a database to store the XML and XSLT snippets. These can then be viewed and used by other users of the application.

---

■**Note** In fact, this would be a natural extension of the previous application from Chapters 4, 5, and 6 but for clarity, and purposes of the previous examples, I have created a new application called Etomic.Share➥Transformer. The code is in the "usual" place in the VSS database. Additionally, it does not rely on the separate engine assembly for the same reason, though I imagine it would if this was a real application.

---

Figure 8-13 shows the new screen for the application that has generated all the excitement at Etomic.


Transformations - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/etomic.sharetransformer-1.0.4.0/Transformations.aspx
Go Links
Local intranet

Create Transformation | List Saved Transformations

**Saved Transformations**

Standing Transform 1
Standing Transformation 2

**Figure 8-13.** *Etomic.ShareTransformer application*

To implement an automated build, and then continuous integration, for this application we will follow the standard steps from the previous Design to Deliver work. Once this is satisfactory, we will add in the database integration tasks, at which point we will finally decide which process to use and then add the CI scripts for CCNet. Finally, we will need to carry out some fairly significant work on our simple deployment scripts to handle the various deployment scenarios satisfactorily.

## The Build Script

In this version of our build scripts, I have amended the `Build.Core.Xml` file to load all noncore assemblies separately, which we discussed earlier as a possible best practice. It looks like this:

```
<loadtasks
   assembly="D:\dotNetDelivery\Tools\NAntContrib\0.85rc2\bin\➥
             NAnt.Contrib.Tasks.dll"/>
<loadtasks
   assembly="D:\dotNetDelivery\Tools\NUnit2Report\1.2.2\bin\➥
             NAnt.NUnit2ReportTasks.dll"/>
<loadtasks
   assembly="D:\dotNetDelivery\Tools\Etomic.NAntExtensions\ ➥
             Etomic.NAntExtensions.GeneralTasks.dll"/>
```

I have loaded the Red Gate task assembly from the debug directory, but of course this should form part of the Etomic.NAntExtensions project we have already constructed.

The build script for the application (without considering database issues) is quite simple:

```
<?xml version="1.0" encoding="utf-8" ?>
<project name="Etomic.ShareTransformer" default="help">
     <description>
        Build file for the Etomic.ShareTransformer application.
     </description>

     <property name="project.name.1" value="${solution.name}.UI" />
     <property name="project.name.2" value="${solution.name}.Engine" />
     <property name="solution.isweb" value="true"/>

     <target name="go" depends="build, test, document, publish, notify"/>

     <target name="build" description="Compile the application.">
          <solution solutionfile="${core.source}\${solution.name}.sln"
             configuration="Debug" outputdir="${core.output}\"/>
     </target>

     <target name="test" description="Apply the unit tests.">
          <property name="nant.onfailure" value="fail.test"/>
          <nunit2>
               <formatter type="Xml" usefile="true" extension=".xml"
                          outputdir="${core.reports}\" />
               <test>
                    <assemblies basedir="${core.output}\">
                         <include name="*Tests.dll"/>
```

```
                </assemblies>
            </test>
        </nunit2>

        <nunit2report out="${core.reports}\NUnit.html">
            <fileset>
                <include name="${core.reports}\*.Tests.dll-results.xml" />
            </fileset>
        </nunit2report>

        <nant buildfile="Build.Common.xml" target="report.fxcop"
              inheritall="true"/>

        <style
           style="D:\Tools\FxCop\1.30\Xml\FxCopReport.xsl"
           in="${core.reports}\fxcop.xml" out="${core.reports}\fxcop.html"/>

        <property name="nant.onfailure" value="fail"/>
    </target>

    <target name="document" description="Generate documentation and reports.">
        <ndoc>
            <assemblies basedir="${core.output}\">
                <include name="${project.name.1}.dll" />
                <include name="${project.name.2}.dll" />
            </assemblies>
            <summaries basedir="${core.output}\">
                <include name="${project.name.1}.xml" />
                <include name="${project.name.2}.xml" />
            </summaries>
            <documenters>
                <documenter name="MSDN">
                  <property name="OutputDirectory" value="${core.docs}\" />
                  <property name="HtmlHelpName" value="${solution.name}" />
                  <property name="HtmlHelpCompilerFilename" value="hhc.exe" />
                  <property name="IncludeFavorites" value="False" />
                  <property name="Title" value="${solution.name} (NDoc)" />
                  <property name="SplitTOCs" value="False" />
                  <property name="DefaulTOC" value="" />
                  <property name="ShowVisualBasic" value="False" />
                  <property name="ShowMissingSummaries" value="True" />
                  <property name="ShowMissingRemarks" value="False" />
                  <property name="ShowMissingParams" value="True" />
                  <property name="ShowMissingReturns" value="True" />
                  <property name="ShowMissingValues" value="True" />
```

```
                    <property name="DocumentInternals" value="True" />
                    <property name="DocumentProtected" value="True" />
                    <property name="DocumentPrivates" value="False" />
                    <property name="DocumentEmptyNamespaces" value="False" />
                    <property name="IncludeAssemblyVersion" value="True" />
                    <property name="CopyrightText"
                              value="${company.name} Ltd., 2005" />
                    <property name="CopyrightHref" value="" />
                  </documenter>
             </documenters>
        </ndoc>
   </target>

   <target name="publish"
     description="Place the compiled assets, reports etc. in agreed location.">
        <nant buildfile="Build.Common.xml" target="publish" inheritall="true"/>
   </target>

   <target name="notify"
      description="Tell everyone of the success or failure.">
        <echo message="Notifying you of the build process success."/>
   </target>

   <target name="fail">
        <echo message="Notifying you of a failure in the build process."/>
   </target>

   <target name="fail.test">
        <nunit2report out="${core.reports}\NUnit.html">
           <fileset>
              <include name="${core.reports}\Etomic.*.Tests.dll-results.xml" />
           </fileset>
        </nunit2report>
   </target>

   <target name="help">
        <echo message="This file should not be executed. Use Build.Core.xml"/>
   </target>

</project>
```

There is nothing that we have not seen before in this script. One of the more interesting areas is the `test` target. If you look at the source code, the developer has included a config file that is used by NUnit when the unit-testing task is called. This can be seen in the source code shown in Figure 8-14 and then in the output directory shown in Figure 8-15.

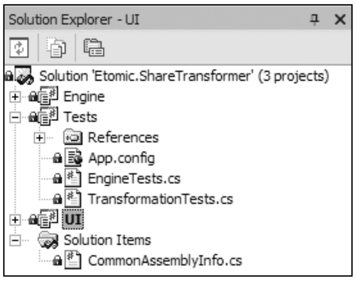
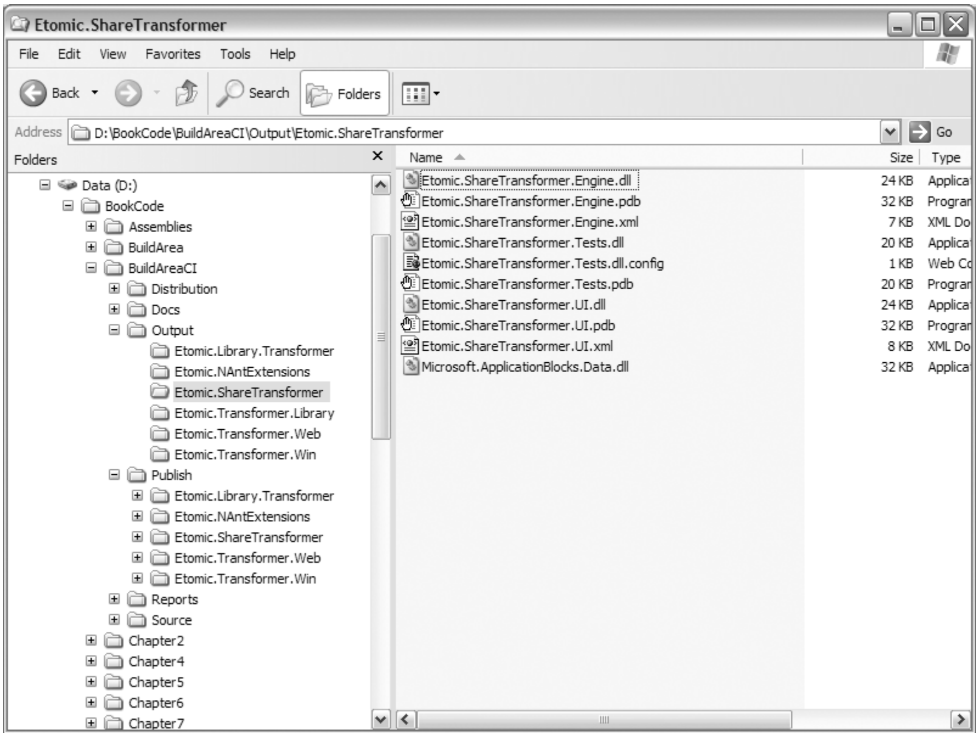


**Figure 8-14.** *App.config in Etomic.ShareTransformer.Tests assembly*



| Name | Size | Type |
|---|---|---|
| Etomic.ShareTransformer.Engine.dll | 24 KB | Applica |
| Etomic.ShareTransformer.Engine.pdb | 32 KB | Prograr |
| Etomic.ShareTransformer.Engine.xml | 7 KB | XML Do |
| Etomic.ShareTransformer.Tests.dll | 20 KB | Applica |
| Etomic.ShareTransformer.Tests.dll.config | 1 KB | Web Co |
| Etomic.ShareTransformer.Tests.pdb | 20 KB | Prograr |
| Etomic.ShareTransformer.UI.dll | 24 KB | Applica |
| Etomic.ShareTransformer.UI.pdb | 32 KB | Prograr |
| Etomic.ShareTransformer.UI.xml | 8 KB | XML Do |
| Microsoft.ApplicationBlocks.Data.dll | 32 KB | Applica |

**Figure 8-15.** *Etomic.ShareTransformer.Tests.dll.config in the output directory*

The developer of this application does not require a specific data load in the database for the unit tests since they generate the required data for testing in the unit tests themselves and then remove it. This can be seen in the following code from `TransformationTests.cs`:

```
[TearDown]
public void TearDown()
{
    DeleteAllTransformations();
}
```

```
[Test]
public void TestGetAll()
{
     DeleteAllTransformations();

     Transformation t = new Transformation();
     t.Output = "aaa";
     t.Title  = "aaa";
     t.Xml   = "aaa";
     t.Xslt   = "aaa";
     t.Save();
     t.Save();
     t.Save();

     ApplicationEngine engine = new ApplicationEngine();
     IList transforms = engine.GetAllTransformations();

     Assert.IsTrue(transforms.Count == 3);
}

private void DeleteAllTransformations()
{
     using(SqlCommand cmd = new SqlCommand("delete transformations", ➥
                                          new SqlConnection(_dbConn)))
     {
          cmd.Connection.Open();
          cmd.ExecuteNonQuery();
          cmd.Connection.Close();
     }
}
```

Given this, we can choose to leave the unit tests to run against the development database and then integrate the database following successful unit testing. Or we can integrate the database prior to the unit testing and then run the unit tests against the integration database instance. Alternatively, we can run the tests against the development database instance, then perform the database integration and run the unit tests against the integration database instance as a "gold-plated" solution. In fact, this is what we will do in our own integration.

To implement the database integration step, we need to add a new target called `database` and then add this to the dependencies in the build file as follows:

```
<target name="go" description="The main target"
        depends="build, test, database, document, publish, notify"/>

<target name="database" description="Handle database integration">
</target>
```

That was the easy part. We can now fill it up with the required steps for integration. To recap, we will do the following:

Compile and run unit tests across the code of the application to ensure a certain level of build success.

Produce a CREATE script for the current development database.

Run an integration cycle on the integration database instance and write these migration files, too.

Point the configuration file for the unit tests to the integration database instance and then run the unit tests again.

Publish the database assets in the same way as the code assets and then remove the "one-off" artifacts, such as the schema script.

We can accomplish this with the following set of scripts in the database target. First up is the integration step for the CREATE and migration scripts:

```
<dbAutoIntegrate
     folder="D:\BookCode\Chapter8\${solution.name}.dbscripts\schema"
     server="localhost"
     database="${solution.name}-dev"
     uid="sa"
     pwd="w1bbl3"
     write="true"
     caption="${sys.version}"
     >
     <databases>
       <database server="localhost" database="etomic.sharetransformer-integrate"
                 uid="sa" pwd="w1bbl3" write="true"/>
     </databases>
</dbAutoIntegrate>
```

The integration scripts are written to a specific working area with folders arranged as described earlier.

We then flag the CREATE script to be written, and specify the database instance for migration.

---

**■Note** The migration instance must exist to begin with, though it does not need to contain any information.

---

Once this work is done, we can point the unit test configuration to the integration test instance as follows:

```
<attrib file="${core.output}\${solution.name}.Tests.dll.config" readonly="false" />
<xmlpoke
    file="${core.output}\${solution.name}.Tests.dll.config"
    xpath="/configuration/appSettings/add[@key = 'DbConnectionString']/@value"
    value="server=localhost;database=etomic.sharetransformer-integrate; ➥
                          uid=transformer;pwd=transform3r" />
<attrib file="${core.output}\${solution.name}.Tests.dll.config" readonly="true" />
```

This code is similar to the versioning code in terms of ensuring the relevant file can be written to. The `<xmlpoke>` task accepts an XPath query to change the configuration file.

■ **Note** I have included the data for the poke directly in the build file (along with other information). This should of course be factored out at the earliest opportunity. If all of your database integration scenarios are the same, then this task can be moved into the `Build.Common.xml` file since it can be completely parameterized.

Finally, we can re-call the `test` target and rerun the unit tests (and the analysis that is a part of the `test` target) before publishing the assets to the publish folder, thereby making them accessible for deployment.

■ **Note** The original test files will be overwritten by the new test call. Whether this matters is up to you. This can easily be handled through the parameterization of the test filenames and a few changes to the `test` target.

```
<call target="test"/>

<zip zipfile="${core.publish}\${solution.name}-DB-${sys.version}.zip">
     <fileset basedir="D:\BookCode\Chapter8\${solution.name}.dbscripts\">
          <include name="**"/>
     </fileset>
</zip>

<delete includeemptydirs="false">
     <fileset basedir="D:\BookCode\Chapter8\${solution.name}.dbscripts\schema\">
          <include name="*" />
     </fileset>
</delete>
```

The delete step in this example only includes the generated files in the schema folder since I am not using any others. In fact, you would probably want to include the removal of scripts in the migrate folder too, because it is likely that these scripts refer to a specific database migration. The scripts in test and reference are more likely to be generally applicable, or constantly maintained across versions. For this build, so that we can test it in deployment, I have included a script in reference called transformations.xml. This file consists of three entries to be inserted into a new database instance and will be included in the published package.

That is that. Running the build script produces the following significant output for the database target:

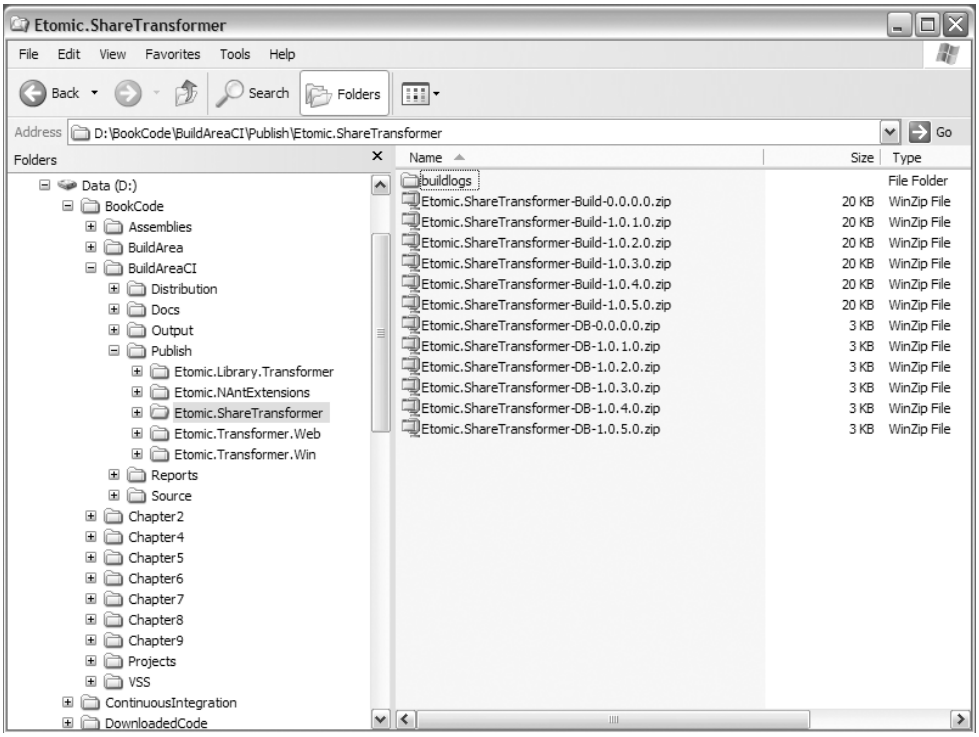
```
database:

[dbAutoIntegrate] Handling CREATE Script
[dbAutoIntegrate] Writing script CREATE-0.0.0.0.sql
[dbAutoIntegrate] Handling migration for etomic.sharetransformer-integrate
[dbAutoIntegrate]
   Writing script ALTER-etomic.sharetransformer-integrate-0.0.0.0.sql

[attrib] Setting file attributes for 1 files to Normal.
[xmlpoke] Found '1' nodes matching XPath expression
   '/configuration/appSettings/add[@key = 'DbConnectionString']/@value'.
[attrib] Setting file attributes for 1 files to ReadOnly.

test:
   <testing output snipped>

[zip] Zipping 3 files to
   'D:\BookCode\BuildAreaCI\Publish\Etomic.ShareTransformer\ ➥
   Etomic.ShareTransformer-DB-0.0.0.0.zip'.
[delete] Deleting 2 files
```

You can use the SQL Bundle tools to confirm that the development and integration instances are identical if you like.

With the background work completed and the build scenario determined, the actual implementation in the build task was not so onerous. Of course, your own scenario may be different.

The next step is to deploy the assets generated. To test out the agreed steps, we should place the system under CI and run a few iterations so that we have some assets to work with. The ccnet.config file for this is included with the source code, and contains nothing of any consequence to the database integration—it is the same ccnet.config script we always use.

If you like, you could make some changes to the database before each iteration, just for fun. Running a few iterations will result in the some assets being available in the publish location for Etomic.ShareTransformer, as shown in Figure 8-16.



D:\BookCode\BuildAreaCI\Publish\Etomic.ShareTransformer

```
Data (D:)
  BookCode
    Assemblies
    BuildArea
    BuildAreaCI
      Distribution
      Docs
      Output
      Publish
        Etomic.Library.Transformer
        Etomic.NAntExtensions
        Etomic.ShareTransformer
        Etomic.Transformer.Web
        Etomic.Transformer.Win
      Reports
      Source
    Chapter2
    Chapter4
    Chapter5
    Chapter6
    Chapter7
    Chapter8
    Chapter9
    Projects
    VSS
  ContinuousIntegration
  DownloadedCode
```

| Name | Size | Type |
|---|---|---|
| buildlogs | | File Folder |
| Etomic.ShareTransformer-Build-0.0.0.0.zip | 20 KB | WinZip File |
| Etomic.ShareTransformer-Build-1.0.1.0.zip | 20 KB | WinZip File |
| Etomic.ShareTransformer-Build-1.0.2.0.zip | 20 KB | WinZip File |
| Etomic.ShareTransformer-Build-1.0.3.0.zip | 20 KB | WinZip File |
| Etomic.ShareTransformer-Build-1.0.4.0.zip | 20 KB | WinZip File |
| Etomic.ShareTransformer-Build-1.0.5.0.zip | 20 KB | WinZip File |
| Etomic.ShareTransformer-DB-0.0.0.0.zip | 3 KB | WinZip File |
| Etomic.ShareTransformer-DB-1.0.1.0.zip | 3 KB | WinZip File |
| Etomic.ShareTransformer-DB-1.0.2.0.zip | 3 KB | WinZip File |
| Etomic.ShareTransformer-DB-1.0.3.0.zip | 3 KB | WinZip File |
| Etomic.ShareTransformer-DB-1.0.4.0.zip | 3 KB | WinZip File |
| Etomic.ShareTransformer-DB-1.0.5.0.zip | 3 KB | WinZip File |

**Figure 8-16.** *Published Etomic.ShareTransformer assets*

At this point, we can move on to construct the database deploy scripts.

## The Deploy Script with Database Deployment

We will construct a deployment script for a fresh database deployment and also a script to handle incremental deployment. First, let us quickly review the standard web deployment script we have previously used:

```
<?xml version="1.0" encoding="utf-8" ?>
<project name="Etomic.ShareTransformer" default="help">
     <description>
        Deploy file for the Etomic.Transformer.Web application
     </description>

     <property name="nant.onfailure" value="fail"/>
     <property name="company.name" value="Etomic"/>
     <property name="solution.name" value="${company.name}.ShareTransformer"/>
     <property name="core.publish"
```

```
          value="http://localhost/ccnet/files/${solution.name}"/>
<property name="core.deploy" value="D:\dotNetDelivery\TempDeploy"/>
<property name="core.environment"
          value="D:\dotNetDeliveryWebs\${solution.name}"/>

<loadtasks
   assembly="D:\dotNetDelivery\Tools\NAntContrib\0.85rc2\➥
                                     bin\NAnt.Contrib.Tasks.dll"
/>

<target name="go"
  depends="selectversion, get, createenvironments, position, database, ➥
                configure, notify"/>

<target name="selectversion"
        description="Selects the version of the system.">
     <if test="${debug}">
         <property name="sys.version" value="0.0.0.0"/>
     </if>
</target>

<target name="get" description="Grab the correct assets.">
  <delete dir="${core.deploy}\" failonerror="false"/>
  <mkdir dir="${core.deploy}\${sys.version}\"/>
    <get
      src="${core.publish}/${solution.name}-Build-${sys.version}.zip"
      dest="${core.deploy}\${solution.name}-Build-${sys.version}.zip"
      />
    <unzip
      zipfile="${core.deploy}\${solution.name}-Build-${sys.version}.zip"
      todir="${core.deploy}\${sys.version}\"/>
</target>

<target name="createenvironments"
        description="Create the environments required">
   <mkdir dir="${core.environment}\${sys.version}\" failonerror="false"/>
   <mkiisdir dirpath="${core.environment}\${sys.version}\"
             vdirname="${solution.name}-${sys.version}"/>
</target>

<target name="position" description="Place required assets">
     <copy todir="${core.environment}\${sys.version}\">
          <fileset basedir="${core.deploy}\${sys.version}">
               <include name="**"/>
          </fileset>
     </copy>
</target>
```

```
     <target name="database" description="Deploy the database changes">
     </target>

     <target name="configure"
             description="Amend configuration settings as necessary">
     </target>

     <target name="notify"
             description="Tell everyone of the success or failure.">
          <echo message="Notifying you of the deploy process success."/>
     </target>

     <target name="fail">
          <echo message="Notifying you of a failure in the deploy process."/>
     </target>

</project>
```

I have amended the dependencies to include the `database` target—the `configure` target has always been there. Both of these targets are empty currently. This deploy script will operate as expected and deploy the web assets as we have previously seen. This is not a lot of help, though, since they will be pointing to the incorrect database: the development instance. The details of the database step can be implemented as follows:

```
<target name="database" description="Deploy the database changes">
     <get
         src="${core.publish}/${solution.name}-DB-${sys.version}.zip"
         dest="${core.deploy}\${solution.name}-DB-${sys.version}.zip"
          />

     <unzip zipfile="${core.deploy}\${solution.name}-DB-${sys.version}.zip"
        todir="${core.deploy}\DB-${sys.version}\"/>

     <delete>
          <fileset basedir="${core.deploy}\DB-${sys.version}\schema">
               <include name="ALTER*" />
          </fileset>
     </delete>

     <dbIntegrate
            folder="${core.deploy}\DB-${sys.version}\schema"
            compare="CreationTime"
            server="localhost"
            database="${solution.name}-Test"
            uid="sa"
            pwd="w1bbl3"
          />
     <dbIntegrate
            folder="${core.deploy}\DB-${sys.version}\reference"
            compare="CreationTime"
            server="localhost"
            database="${solution.name}-Test"
            uid="sa"
            pwd="w1bbl3"
       />
</target>
```

The use of the `<dbIntegrate>` task requires the inclusion of the `ManualDBTasks` assembly and so the following `<loadtasks>` is required (again, referencing the debug assembly):

```
<loadtasks
   assembly="D:\dotNetDelivery\Tools\Etomic.NAntExtensions\ ➥
             Etomic.NAntExtensions.GeneralTasks.dll"/>
```

The `database` target gets and unzips the assets in the same way as the regular code deployment target. After this, it removes the ALTER script from the `schema` folder as we are interested only in the CREATE script, before executing the CREATE script using the `<dbIntegrate>` custom task, and then executing the input of the reference data with another task.

■**Note** This works entirely satisfactorily, though the manual database task is now looking a little long in the tooth. I should add to my notebook that this task could be refactored to include multiple script types to be executed and perhaps revisit the `fileset` issue to save on duplication of information and effort. But that is for another time.

The configuration step is identical to the build script, and therefore suffers from the same data/process mix issue:

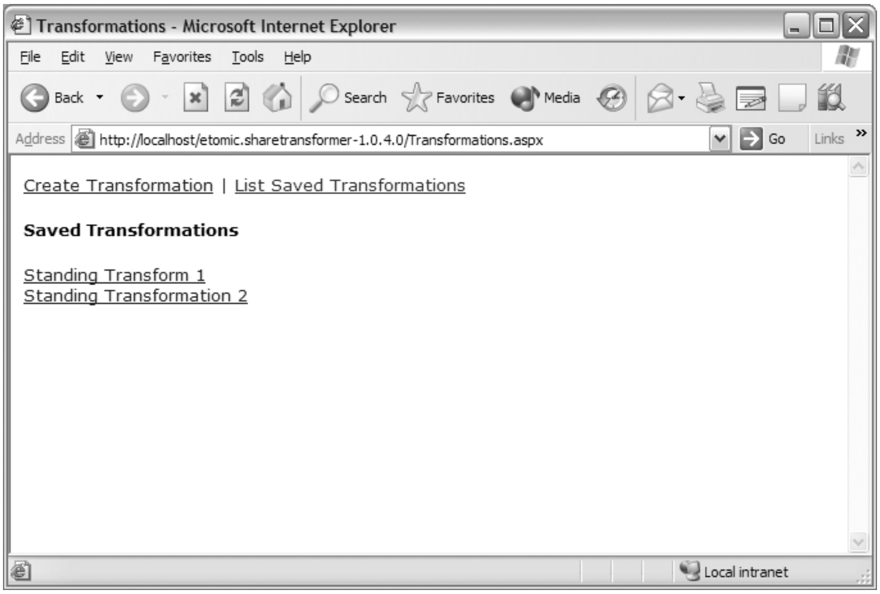
```
<target name="configure" description="Amend configuration settings as necessary">
     <attrib file="${core.environment}\${sys.version}\web.config"
             readonly="false" />
     <xmlpoke
           file="${core.environment}\${sys.version}\web.config"
           xpath="/configuration/appSettings/add[@key = ➥
                                             'DbConnectionString']/@value"
           value="server=localhost;database=etomic.sharetransformer-test; ➥
                                    uid=transformer;pwd=transform3r" />
     <attrib file="${core.environment}\${sys.version}\web.config"
             readonly="true" />
</target>
```

Once again, that is that. Running this deployment script in the usual way will result in a fresh deployment of the chosen application version. Here is the output from the execution of the script against version 1.0.4.0 of the application:

```
---------- NAnt ----------
NAnt 0.85
Copyright (C) 2001-2004 Gerry Shaw
http://nant.sourceforge.net

Buildfile: file:///Etomic.ShareTransformer.Deploy-CREATE.xml
Target(s) specified: go

[loadtasks] Scanning assembly "NAnt.Contrib.Tasks" for extensions.
[loadtasks] Scanning assembly "Etomic.NAntExtensions.ManualDBTasks" for extensions.

selectversion:

get:

   [delete] Deleting directory 'D:\TempDeploy\'.
    [mkdir] Creating directory 'D:\TempDeploy\1.0.4.0\'.
      [get] Retrieving
'http://localhost/ccnet/files/Etomic.ShareTransformer/Etomic.ShareTransformer-
Build-1.0.4.0.zip' to 'D:\TempDeploy\Etomic.ShareTransformer-Build-1.0.4.0.zip'.
    [unzip] Unzipping 'D:\TempDeploy\Etomic.ShareTransformer-Build-1.0.4.0.zip'
to 'D:\TempDeploy\1.0.4.0\' (19571 bytes).

createenvironments:

 [mkiisdir]
Creating/modifying virtual directory 'Etomic.ShareTransformer-1.0.4.0' on
'localhost:80'.

position:

     [copy] Copying 2 files to 'D:\Webs\Etomic.ShareTransformer\1.0.4.0\'.

database:

      [get] Retrieving
'http://localhost/ccnet/files/Etomic.ShareTransformer/Etomic.ShareTransformer-DB-
1.0.4.0.zip' to 'D:\TempDeploy\Etomic.ShareTransformer-DB-1.0.4.0.zip'.
    [unzip] Unzipping 'D:\TempDeploy\Etomic.ShareTransformer-DB-1.0.4.0.zip' to
'D:\TempDeploy\DB-1.0.4.0\' (2229 bytes).
   [delete] Deleting 1 files.

[dbIntegrate] CREATE-1.0.4.0.sql
     [exec] 1> 2> 1> 2> 1> 2> 1> 2> 1> 2> 1> 2> 1> 2> 3> 1> 2> Msg 15023, Level
16, State 1, Server ALIENVM, Procedure sp_grantdbaccess, Line 126
     [exec] User or role 'Transformer' already exists in the current database.
     [exec] 1> 2> Creating role db_owner
     [exec] 1> 2> 'Transformer' added to role 'db_owner'.
     [exec] 1> 2> 1> 2> Creating [dbo].[Transformations]
     [exec] 1> 2> 3> 4> 5> 6> 7> 8> 9> 10> Msg 2714, Level 16, State 6, Server
ALIENVM, Line 1
     [exec] There is already an object named 'Transformations' in the database.
     [exec] Warning: The table 'Transformations' has been created but its maximum
row size
     [exec] (24533) exceeds the maximum number of bytes per row (8060). INSERT or
UPDATE
     [exec] of a row in this table will fail if the resulting row length exceeds
8060
     [exec] bytes.
     [exec] 1> 2> 1> 2> (1 row affected)
     [exec] 1> 2> Creating primary key [PK_Transformations] on
[dbo].[Transformations]
     [exec] 1> 2> Msg 1779, Level 16, State 1, Server ALIENVM, Line 1
     [exec] Table 'Transformations' already has a primary key defined on it.
     [exec] Msg 1750, Level 16, State 1, Server ALIENVM, Line 1
     [exec] Could not create constraint. See previous errors.
     [exec] Warning: The table 'Transformations' has been created but its maximum
row size
     [exec] (24533) exceeds the maximum number of bytes per row (8060). INSERT or
UPDATE
     [exec] of a row in this table will fail if the resulting row length exceeds
8060
     [exec] bytes.
     [exec] 1> 2> 1> 2> (1 row affected)
     [exec] 1> 2> 1> 2> 3> 4> 5> 6> The database update failed
     [exec] 1> 2> 1>

[dbIntegrate] Transformations.sql
     [exec] 1> 2> 1> 2> 1> 2> 3> 4> 5> 6> 7> 8> 9> 10> 11> 12> 13> 14> 15> 16>
17> 18> Msg 2627, Level 14, State 1, Server ALIENVM, Line 1
     [exec] Violation of PRIMARY KEY constraint 'PK_Transformations'. Cannot
insert
     [exec] duplicate key in object 'Transformations'.

configure:

   [attrib] Setting file attributes for 1 files to Normal.
  [xmlpoke] Found '1' nodes matching XPath expression
'/configuration/appSettings/add[@key = 'DbConnectionString']/@value'.
   [attrib] Setting file attributes for 1 files to ReadOnly.
```

```
notify:

     [echo] Notifying you of the deploy process success.

go:

BUILD SUCCEEDED
Total time: 3.7 seconds.
Output completed (5 sec consumed) - Normal Termination
```

The output clearly shows the execution of the relevant database scripts (and some warnings), and indeed the application has been correctly deployed and configured, as can be seen in Figure 8-17.


Transformations - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/etomic.sharetransformer-1.0.4.0/Transformations.aspx Go Links
Local intranet

Create Transformation | List Saved Transformations

**Saved Transformations**

Standing Transform 1
Standing Transformation 2

**Figure 8-17.** *Etomic.ShareTransformer v1.0.4.0*

Checking the `web.config` and other files will also demonstrate that all is well.

So that is fine for a clean build, but what about a migrated build? This is slightly more involved than the clean build for the reasons discussed before. To recap, because the database has its migration scripts produced during each code build but not every code build is deployed, when the deployment finally occurs, the application of the migration scripts from the previously deployed version must be applied sequentially to move the database to the current version.

In practice this means that we need to provide some looping functionality for the database step in a way that assesses the database versions.

To actually implement this, we leave the previous deployment script generally unaltered, but we move the content of the `database` target into a new target called `databaseincrement`. This target is not part of the master dependencies, but will be called as needed by the `database` target. The `databaseincrement` target looks like this:

```
<target name="databaseincrement">
     <get
        src="${core.publish}/${solution.name}-DB-${db.version}.zip"
        dest="${core.deploy}\${solution.name}-DB-${db.version}.zip"
      />
     <unzip zipfile="${core.deploy}\${solution.name}-DB-${db.version}.zip"
         todir="${core.deploy}\DB-${db.version}\"/>

     <delete>
         <fileset basedir="${core.deploy}\DB-${db.version}\schema">
               <include name="CREATE*" />
         </fileset>
     </delete>

     <dbIntegrate
          folder="${core.deploy}\DB-${db.version}\schema"
          compare="CreationTime"
          server="localhost"
          database="${solution.name}-Test"
          uid="sa"
          pwd="w1bbl3"
      />
</target>
```

Note that we are now deleting the CREATE script since we are interested in the ALTER scripts. Additionally, we have removed the step that adds the reference data so that we concentrate on the schema.

■**Note** In a real-life scenario, you would likely want to apply each set of migration scripts, too, and then load in the final set of reference data. This final step could be handled in the regular `database` target.

The database step therefore contains the looping function:

```
<target name="database" description="Deploy the database changes">
     <foreach item="String"
             in="${etomic::get-db-version-list(old.db.version , sys.version)}"
             delim=" ,"
             property="db.version">
          <call target="databaseincrement"/>
     </foreach>
</target>
```

Complexity alert! I have stayed away from introducing scripted functions into the delivery scripts where possible to avoid a move into even more “options” (read: confusion). In this instance, though, I need an adequate way of providing a list to loop through, from the old database version to the desired deployment version. This means that the execution of this script relies on a new command-line input: the `old.db.version` property.

The list I am looking for is something like the following:

```
1.0.2.0, 1.0.3.0, 1.0.4.0
```

This assumes I am migrating from version 1.0.2.0 to 1.0.4.0. To provide this list, the following script function will do the trick:

```
<script language="C#" prefix="etomic" >
  <code><![CDATA[
  [Function("get-db-version-list")]
  public static string GetDBVersionList(string firstVersion, string lastVersion)
  {
    int start, end, major, minor;
    Match match;

    Regex versionRegEx = ➥
        new Regex (@"(?<major>\d*)\.(?<minor>\d*)\.(?<build>\d*)\.\d*",➥
                   RegexOptions.Compiled);

    match = versionRegEx.Match(firstVersion);
    major = Int32.Parse(match.Groups["major"].Value);
    minor = Int32.Parse(match.Groups["minor"].Value);
    start = Int32.Parse(match.Groups["build"].Value);

    match = versionRegEx.Match(lastVersion);
    end = Int32.Parse(match.Groups["build"].Value);

    StringBuilder dbList = new StringBuilder();
    for(int i=start+1; i<end+1; i++)
    {
       dbList.Append(String.Format("{0}.{1}.{2}.0,", major.ToString(), ➥
                                              minor.ToString(), i.ToString()));
    }

    if (dbList.ToString().Length > 0)
       return dbList.ToString().Substring(0, dbList.ToString().Length-1);
    else
       return "";
    }
   ]]></code>
</script>
```

The code is another straightforward piece of C#, using a regular expression to pull apart the version number, create a list, and then put the version numbers back together again. Once again, thanks to the flexibility of NAnt, we can tackle a problem quickly and effectively.

---

**■Note** Sadly, we cannot tackle my own oversight so effectively. Because I have wrapped the version number logic in the process, I am stuck with working with the full version number here without some significant refactoring of the scripts. (Actually, it is not that big a deal, but I like to moan at myself now and then to keep myself in check.) The bigger problem here is that the previous function does not handle a move, for example, from 1.0.0.0 to 1.1.0.0, since we are dealing with only the third digit. Oh well—another entry in the notebook.

---

With the addition of the function, this deploy script is complete. It can be run on an existing database version with the following command line (assuming you are moving from version 1.0.2.0 to 1.0.4.0):

```
nant -f:Etomic.ShareTransformer.Deploy.ALTER.xml go -D:sys.version=1.0.4.0 ➥
      -D:old.db.version=1.0.2.0 -D:debug=false
```

The most relevant output here is as follows:

```
database:

databaseincrement:

      [get] Retrieving
'http://localhost/ccnet/files/Etomic.ShareTransformer/Etomic.ShareTransformer-DB-
1.0.3.0.zip' to 'D:\TempDeploy\Etomic.ShareTransformer-DB-1.0.3.0.zip'.
    [unzip] Unzipping 'D:\TempDeploy\Etomic.ShareTransformer-DB-1.0.3.0.zip' to
'D:\TempDeploy\DB-1.0.3.0\' (2228 bytes).
   [delete] Deleting 1 files.
[dbIntegrate] ALTER-etomic.sharetransformer-integrate-1.0.3.0.sql
     [exec] 1> 2> 3> 4> 5> 6> 7> 1> 2> 1> 2> 1> 2> 1> 2> 1> 2> 1> 2> 1> 2> 1> 2>
3> 4> 5> 6> The database update succeeded
     [exec] 1> 2> 1>

databaseincrement:

[get] Retrieving
   'http://localhost/ccnet/files/Etomic.ShareTransformer/
   <remainder>
   <etc>
1.0.4.0.zip' to 'D:\TempDeploy\Etomic.ShareTransformer-DB-1.0.4.0.zip'.
    [unzip] Unzipping 'D:\TempDeploy\Etomic.ShareTransformer-DB-1.0.4.0.zip' to
'D:\TempDeploy\DB-1.0.4.0\' (2229 bytes).
```

```
   [delete] Deleting 1 files.
[dbIntegrate] ALTER-etomic.sharetransformer-integrate-1.0.4.0.sql
     [exec] 1> 2> 3> 4> 5> 6> 7> 1> 2> 1> 2> 1> 2> 1> 2> 1> 2> 1> 2> 1> 2> 1> 2>
3> 4> 5> 6> The database update succeeded
     [exec] 1> 2> 1>

configure:

   [attrib] Setting file attributes for 1 files to Normal.
  [xmlpoke] Found '1' nodes matching XPath expression
'/configuration/appSettings/add[@key = 'DbConnectionString']/@value'.
   [attrib] Setting file attributes for 1 files to ReadOnly.
```

We can see that two executions of the `databaseincrement` target occur, moving the old version (1.0.2.0) through 1.0.3.0 to version 1.0.4.0. Superb.

**■Note** You have probably already realized that this script can be used to perform a “clean” build by simply applying all migration scripts from the very first build.

Adding the old version of the database at the command line is effective but might be seen as a bit kludgy. My suggestion for alternative solutions is to use `<xmlpoke>` and `<xmlpeek>` to maintain a state or version file, or try to use the `<version>` task to do the same. I spent a little time messing around with this idea, and found it satisfactory.

At this point, we are now able to analyze, integrate, and deploy a system with a database as we wish. Now for something completely different…

## Considering DTS Packages

DTS (Data Transformation Services) packages are a very useful feature of SQL Server, and I have seen heavy use of DTS features for handling a variety of scheduled tasks. Unfortunately, they can be a pain to deploy effectively because they are prone to poor standards for configuration and also because it can be awkward to ensure quality.

The DTS Compare tool from the SQL Bundle does an excellent job of analyzing differences between server instances, which means the physical porting of a DTS package from server to server is less painful than manual inspections. However, there is another problem. Usually a DTS package will need to be configured differently from server to server—different FTP server information, for example. The only real option is to access the package on the production server and make changes to variables throughout the tasks within the package. This is possible, but in my experience leads to heated debates between developers and operations personnel as to who forgot to set the variables or who set them incorrectly. This reduces the confidence in DTS package deployment and can affect the success of a project.

The effort and risk can be reduced to some extent through the introduction of some standard tasks in DTS packages and, as usual, a bit of organization.

**■Note** DTS is of course a little bit obscure and archaic. The code that follows works with a minimum of effort but could be accomplished, probably to a higher quality, with a specific custom task implemented in VB 6.0, or C# if you are daring. I hope that the next version of SQL Server has improved configuration and deployment capabilities for DTS packages.

Figure 8-18 shows a package with two steps at the beginning: Load XML File and Set Variables. The first is a VBScript task, and the second is a Dynamic Properties task. We will use these to simplify package configuration.

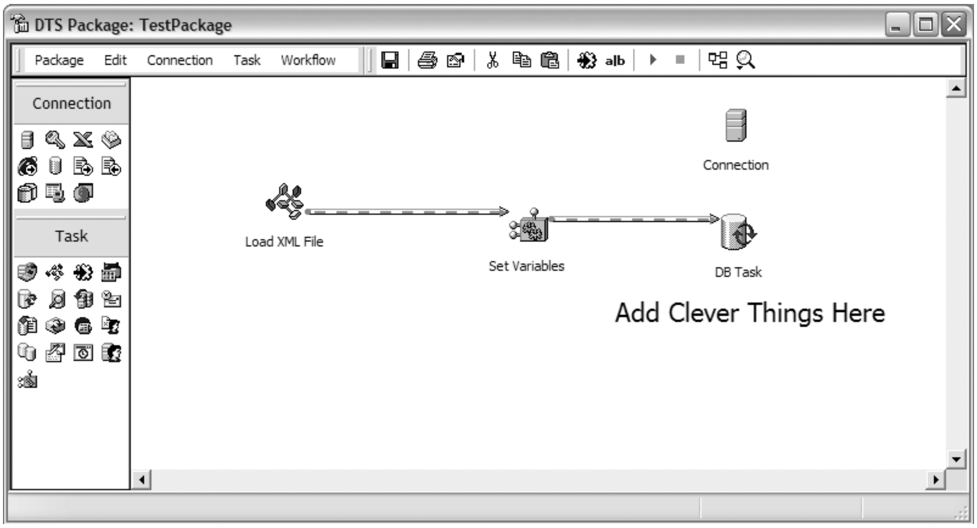


**Figure 8-18.** *Standard DTS package configuration*

## Organization

Generally speaking, DTS packages have a tendency to generate artifacts since they are used to push around data feeds in various formats. These artifacts can be organized sensibly, server to server, by maintaining standard shares in the same way on each server in the form `\\<server name>\<dts share name>\<project name>`. The artifacts for DTS packages can then be maintained neatly and cleaned up from this area.

Additionally, we will use this space to maintain an XML file containing the configuration for the DTS package.

## Load XML File Task

The XML files to be stored in the project folder have the following trivial format:

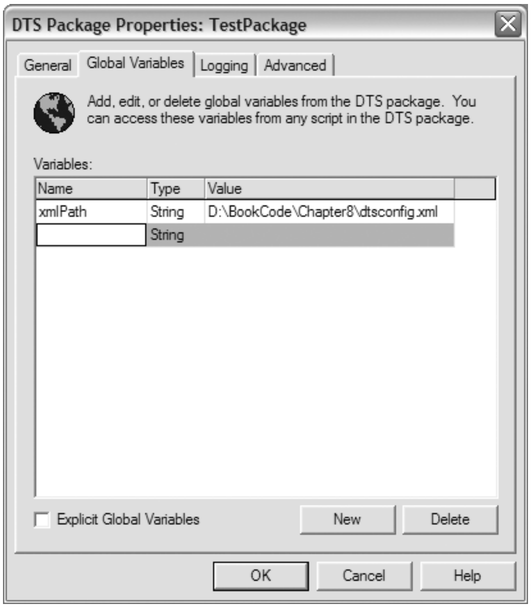
```
<?xml version="1.0" encoding="UTF-8"?>
<parameters>
     <parameter name="ConnectionString">My connection string</parameter>
     <parameter name="Another Name">Another Value</parameter>
</parameters>
```

The Load XML File task should then contain the following (VBScript) code:

```
Function Main()
     Dim oXML
     Set oXML = CreateObject("Microsoft.XMLDOM")
     oXML.Async = False
     oXML.ValidateOnParse = False
     oXML.Load(DTSGlobalVariables("xmlPath").value)
     If oXML.XML = "" Then
          Main = DTSTaskExecResult_Failure
     Else
          Set oRootElement = oXML.documentElement
          for i = 0 to oRootElement.childNodes.length -1
               strName = oRootElement.childNodes.item(i).getAttribute("name")
               strValue = oRootElement.childNodes.item(i).text
               DTSGlobalVariables(CStr(strName)).value = strValue
          next
          Main = DTSTaskExecResult_Success
     End If
End Function
```

■ **Note** Ugh! At least you only have to type it once; you can copy and paste from that point forward.

Love VBScript or hate it, this task loads up the simple XML file and sets global variables for the package according to all of the entries in the XML file. In order for the task to work, there needs to be one preexisting global variable, which should be called `xmlPath` and which should point to the XML configuration file, as shown in Figure 8-19.


DTS Package Properties: TestPackage

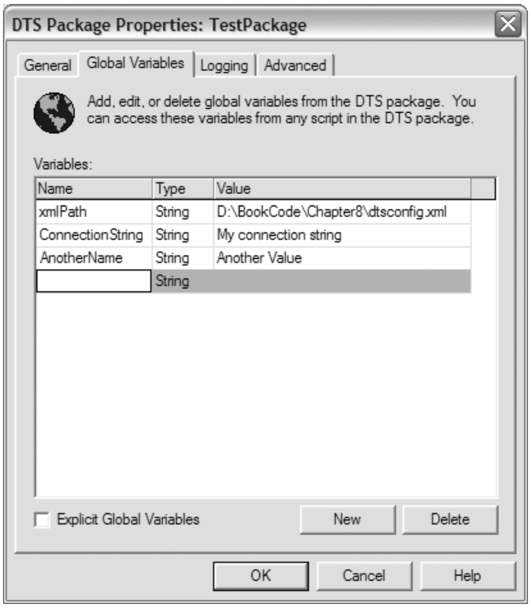
Add, edit, or delete global variables from the DTS package. You can access these variables from any script in the DTS package.

Variables:

| Name | Type | Value |
|---|---|---|
| xmlPath | String | D:\BookCode\Chapter8\dtsconfig.xml |
| | String | |

**Figure 8-19.** *Initial global variables settings*

After this step is executed successfully, the new global variables will also be present, as can be seen in Figure 8-20.


DTS Package Properties: TestPackage

Add, edit, or delete global variables from the DTS package. You can access these variables from any script in the DTS package.

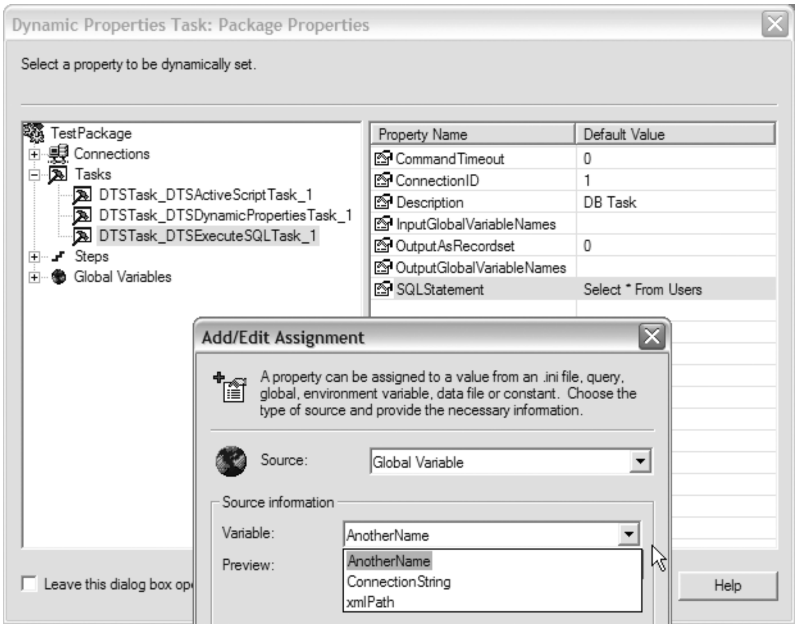
Variables:

| Name | Type | Value |
|---|---|---|
| xmlPath | String | D:\BookCode\Chapter8\dtsconfig.xml |
| ConnectionString | String | My connection string |
| AnotherName | String | Another Value |
| | String | |

**Figure 8-20.** *Global variables after Load XML File task*

## Set Variables Task

With all of the settings available as global variables, the Dynamic Properties task Set Variables can be used to push the variables into the settings of the other tasks for the package, as shown in Figure 8-21.


Dynamic Properties Task: Package Properties

Select a property to be dynamically set.

```
TestPackage
  Connections
  Tasks
    DTSTask_DTSActiveScriptTask_1
    DTSTask_DTSDynamicPropertiesTask_1
    DTSTask_DTSExecuteSQLTask_1
  Steps
  Global Variables
```

| Property Name | Default Value |
|---|---|
| CommandTimeout | 0 |
| ConnectionID | 1 |
| Description | DB Task |
| InputGlobalVariableNames | |
| OutputAsRecordset | 0 |
| OutputGlobalVariableNames | |
| SQLStatement | Select * From Users |

Leave this dialog box op

Add/Edit Assignment

A property can be assigned to a value from an .ini file, query, global, environment variable, data file or constant. Choose the type of source and provide the necessary information.

Source: Global Variable

Source information

Variable: AnotherName

Preview:

AnotherName
ConnectionString
xmlPath

**Figure 8-21.** *Configuring a Dynamic Properties task*

Generally speaking, global variables can be used to set just about any aspect of a DTS package, including properties that are exposed on custom tasks that you may use. You may find that these standards work well as they are.

The result of this work for each DTS package is that when a package is deployed to another environment, the only configuration change that needs to occur is the correct setting of the `xmlPath` global variable—a minimum of manual effort and risk.

■ **Note** There are other options as well. You could use a SOAP call or similar approach to dynamically load the settings instead.

## Summary

There has been a lot to learn in this chapter, but I hope you will agree that the results in terms of describing the beginnings of a database integration framework are satisfactory. The work highlights the fragility and lack of utility for such processes at this time, but with a toolset like SQL Bundle and NAnt and a little thought and creativity, these processes can work. Discipline is required to actually carry through these concepts to a production scenario, and if you review the processes, you will identify risks that cannot easily be covered—such as "How do I roll back in the event of a failure?" The scripts provide an easy means to handle such issues, but they do not avoid the original problem. On the other hand, we suffer these problems daily in any case.

In the next chapter, we will look at generating further efficiency to the delivery process and the scripts through the use of code-generation techniques.

## Further Reading

There are several Internet papers on the subject of agile database management and development (as opposed to straightforward integration). These two are very worthwhile:

*Evolutionary Database Design (*`http://martinfowler.com/articles/evodb.html`*)*

*XP and Databases (*`www.extremeprogramming.org/stories/testdb.html`*)*

This book contains a great deal of techniques and suggestions on the same subject: *Agile Database Techniques: Effective Strategies for the Agile Software Developer* by Scott Ambler (Wiley, 2003).