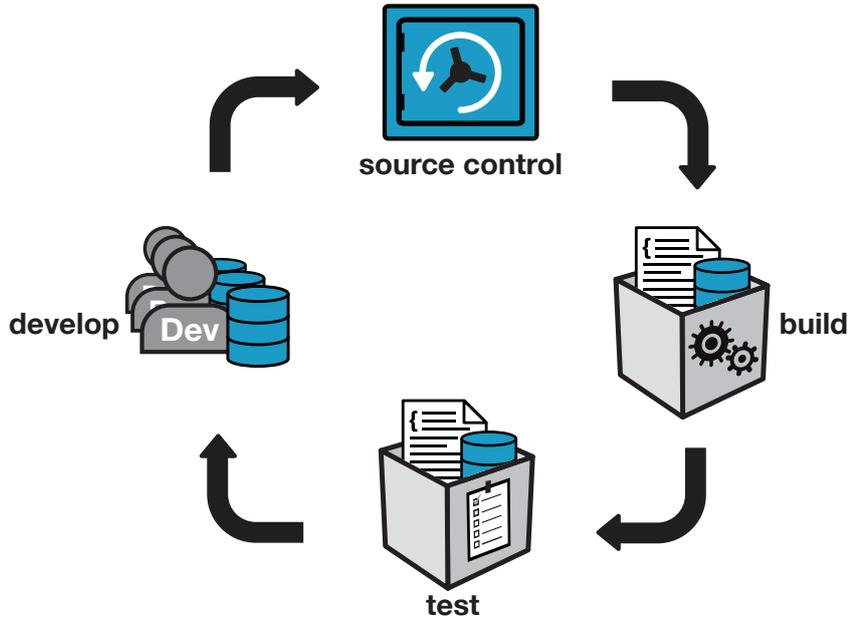


# Continuous integration for databases using Red Gate tools

A technical overview



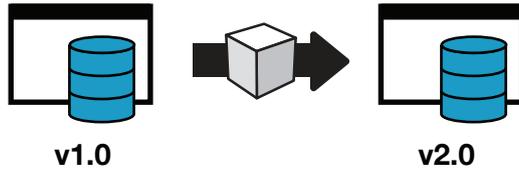
## Continuous Integration



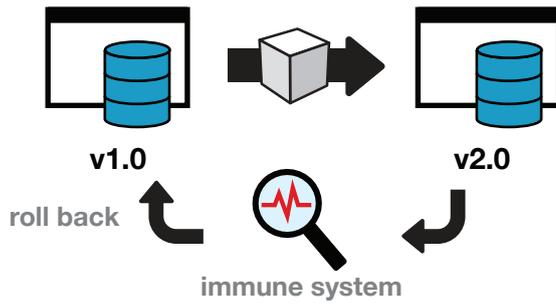
## Automated Deployment



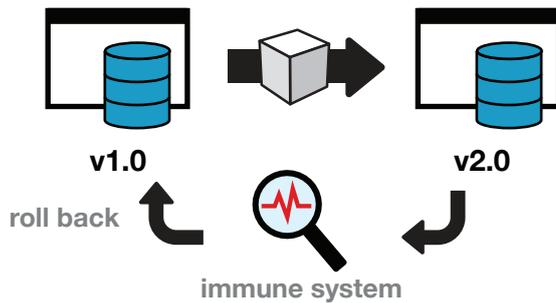
### Testing



### STAGING / UAT



### PRODUCTION



---

# Continuous integration for databases using Red Gate tools

## Introduction

This whitepaper examines the challenge of integrating SQL Server databases into an existing build automation process, such as continuous integration, and describes how Red Gate tools can be used to automate the process.

Included in this paper are simple examples using SQL Source Control and the command line interfaces of Red Gate tools. SQL Source Control is an add-in to SQL Server Management Studio that source controls your database schema. Changes are committed into source control, triggering the continuous integration process, which builds a test database from source control, exercises and validates the deployment or upgrade process, and runs automated regression tests on your database.

This paper only covers databases, and not the building of your application code or any other configuration or setup required for your product.

This whitepaper is organized into the following sections:

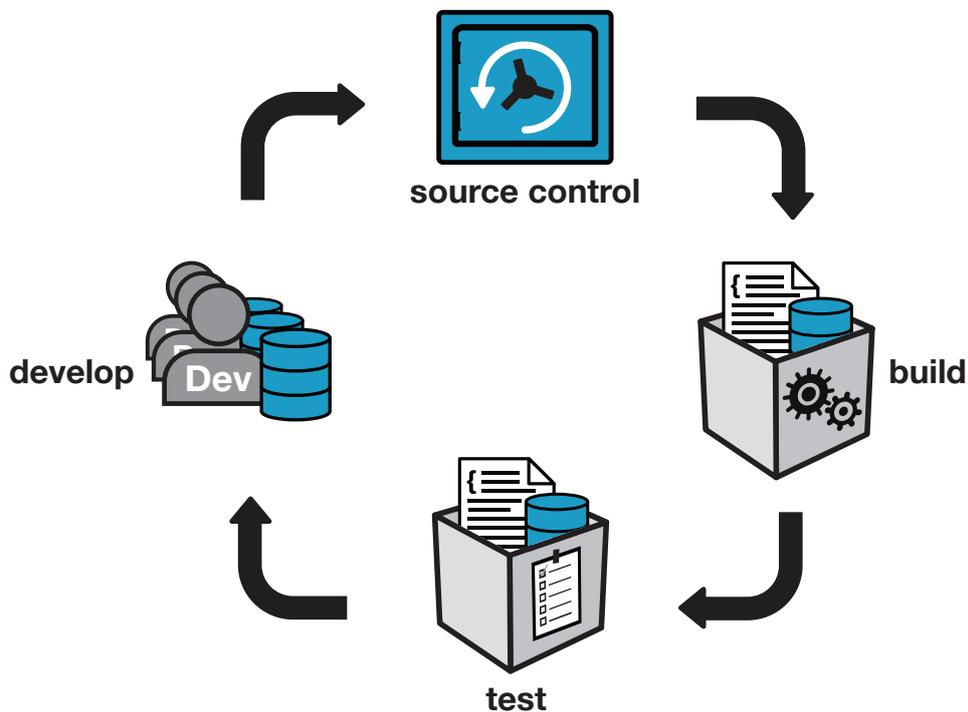
I.	Why continuous integration?	p4
II.	How are databases different?	p6
III.	The database delivery lifecycle	p7
IV.	Database continuous integration	p8
V.	Database deployment	p10
VI.	How Red Gate tools help	p11
VII.	Worked examples	p12
VIII.	Further reading and resources	p19
IX.	Conclusions	p19

## I. Why continuous integration?

Continuous integration (CI) is the process of ensuring that all code and related resources in a development project are integrated regularly and tested by an automated build system. Code changes are checked into source control, triggering an automated build with unit tests and early feedback in the form of errors returned. A stable current build is consistently available, and if a build fails, it can be fixed rapidly and re-tested.

A CI server uses a build script to execute a series of commands that build an application. Generally, these commands clean directories, run a compiler on source code, and execute unit tests. However, for applications that rely on a database back-end, build scripts can be extended to perform additional tasks such as creating, testing, and updating a database.

The following diagram illustrates a typical integration process. The automated continuous integration process begins each time the server detects a change that has been committed to source control by the development team. Continuous integration ensures that if at any stage a process fails, the 'build' is deemed broken and developers are alerted immediately.



CI originated from the Extreme Programming (XP) movement and is now an established development practice.

*“Continuous Integration is a practice designed to ensure that your software is always working, and that you get comprehensive feedback in a few minutes as to whether any given change to your system has broken it.”*

**Jez Humble, ThoughtWorks, co-author of Continuous Delivery**

For many software projects, this will include a database. Author and thought leader, Martin Fowler, recommends that “getting the database schema out of the repository and firing it up in the execution environment” should be part of the automatic build process. However, this is not always simple, which is why this paper seeks to clarify the process of integrating databases into an existing automatic continuous integration process.

## II. How are databases different?

They aren't. Database code is code, and should therefore be treated in the same way as your application code. However, the principal difficulty underlying continuous integration for databases is the lack of a simple way to keep a database in source control and deploy it to a target server.

The database is unlike application code in as much as it contains state that needs to be preserved after an upgrade. Where a production database already exists, DML and DDL queries modify the existing state of a database, and unlike for application code, there is no source code to compile. Migration and deployment therefore rely on creating upgrade scripts specifically for that purpose.

The lack of database source code makes it difficult to maintain a current stable version in source control. Creation and migration scripts can be checked into the source control repository, but despite its importance, the disciplined creation and on-going maintenance of these scripts is often not considered to be a core part of the database development cycle.

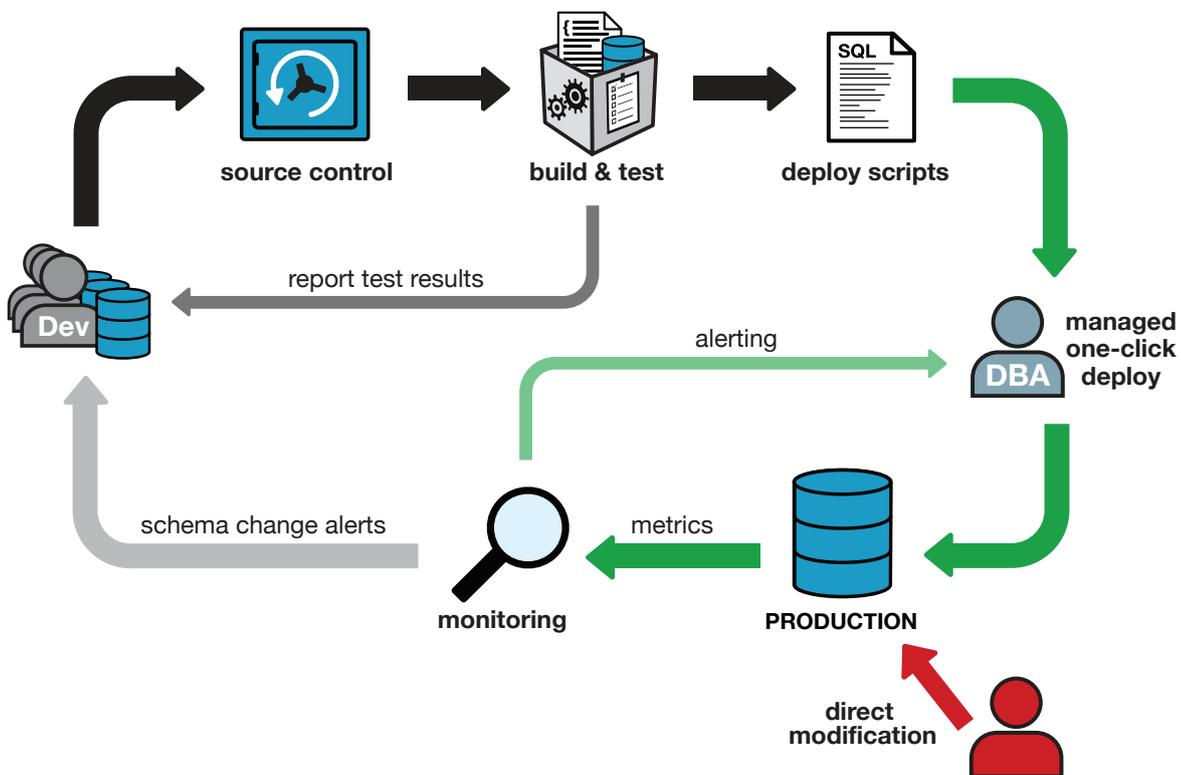
Migration scripts may contain ALTER and UPDATE statements to update the target version of the database with the development version; alternatively, the scripts may create a new database. Where changes are deployed to an existing database, all differences and dependencies must be accounted for. In some production deployments, this involves multiple targets with different schemas and data. In either case, the manual process is time consuming, prone to errors, and one that should not be left unresolved at the end of the project cycle.

Object creation scripts can be generated relatively simply (for example using Microsoft SQL Server Management Studio), but referential integrity is difficult to maintain. Objects and data must be created and populated in the correct order, and as dependency chains can be complex, third party tools are often required. Data migration and test data creation are tedious and time consuming operations when performed manually.

### III. The database delivery lifecycle

Database CI is an important part of a wider database delivery lifecycle. Continuous integration builds and tests a database, validating the upgrade scripts that will eventually be supplied to the DBA for deployment. Once deployed to the production server, monitoring should be put in place to ensure that not only the performance of the database remains acceptable, but also that the end user is benefiting from the changes.

In the event that a change is made directly to the production database bypassing the rigid development and test processes, monitoring ensures that the DBA and development manager are notified. The new change can either be undone by the DBA, or if approved, copied into the development environment.



## IV. Database continuous integration

### 1. Keeping a database up-to-date

Databases may figure in your CI process simply because the application code requires there to be a database present to function correctly. The database schema version corresponds to an analogous application code version. Any changes to the application code or the database structure could in theory break the system and should consequently trigger the CI process.

Once a database is maintained in source control, Red Gate tools are able to build a clean database from its source files to accompany the application build.

If you already have internal test databases that need to match the development databases, you can keep them up-to-date with the latest version using continuous integration.

### 2. Testing the database creation script

An artifact of the database build process, the database creation script, is one that builds the database from scratch. This is useful not only for when a test database needs to be built, but also if new installations of the application are required, for example, to new customers.

### 3. Testing the database upgrade script

Unlike for application code, upgrading a production database isn't a simple case of replacing it with a fresh copy. Databases have a mission critical state that needs to be preserved.

Safeguarding existing data is the most challenging task to be faced during the upgrade process. This is why it is a highly recommended best practice to repeatedly test the deployment script as part of the CI process and ensure that a working upgrade script can be generated at all times.

In order to test the upgrade, it is necessary to create a database at the version corresponding to the existing production database, and not just create a new database representing the latest version. Using Red Gate tools, the deployment script is generated, applied against the production-level CI database, and subsequently validated against the expected target version. If this validation fails, the failed upgrade process should be regarded as a 'broken build', and measures should be taken to troubleshoot and promptly resolve the issue.

## The need for custom migration scripts

Automated deployment script generation with comparison tools such as Red Gate's SQL Compare is extremely powerful and time-saving. However, the comparison engine that generates the deployment scripts has no way of second guessing developer intent. The most common cause of a broken deployment script is as a result of development changes that include data migrations and object renames. If, for example, a column is renamed, the comparison engine only sees the before and after state, and will interpret this as a DROP and CREATE, leading to disastrous consequences should this script be inadvertently applied to the production database. There are a number of circumstances where developer intent is required to supplement the comparison engine knowledge, some of which are listed below:

- Renaming tables and columns
- Splitting tables and columns
- Merging tables and columns
- Adding a new NOT NULL column to a table without supplying a default value
- Refactorings that include data migrations

Fortunately, Red Gate tools provide the capability to save custom migration scripts to source control, which are then re-used by the comparison engine to generate reliable and repeatable deployment scripts. This capability makes the continuous validation of the upgrade process possible in an automated CI environment.

## 4. Testing database code

Although it is best practice to test application code as part of a CI process, database code and its accompanying business logic is often overlooked. Database code comes in the form of stored procedures, functions, views, triggers, and CLR objects. If it is deemed important to test application code as part of CI, the same must apply to the database code.

Fortunately there are many open source frameworks that can be used for the purposes, some implemented in .NET (e.g. NUnit) and others in SQL (e.g. the popular open source SQL Server unit testing framework, tSQLt). Unit tests can be easily created, run, and managed with Red Gate's SQL Test, a SQL Server Management Studio add-in.

## V. Database deployment

Once validated in a CI environment, the database (and application) changes need to go through a release process. It is prudent to push these changes through some final test phases using staging and UAT environments.

In many ways continuous integration can be considered a dry run for production deployment. But although the CI environment often mirrors the production environment as closely as possible, it is rarely the case for the database.

Production databases can be huge, and it is therefore impractical to restore a production backup to the CI environment for testing purposes. Test environments rarely benefit from the same storage capacity as for production and pre-production environments, and lengthy restore times make recreating the CI database impractical.

Red Gate's Deployment Manager helps transition code and database changes through the final stages of the release process, by taking the tested output of the CI process encapsulated as 'packages' and deploying these to pre-production test environments. A database package contains the validated deployment scripts along with snapshots of the intended target versions in order to ensure that the actual target is at the expected version and hasn't since 'drifted'. A database package also contains the 'source' database state allowing it to validate the success of the deployment.

## VI. How Red Gate tools help

Red Gate offers the following tools to support the CI and delivery process. Many of these tools feature in the worked examples in the next section.

### SQL Source Control

- Helps maintain database schema and data in a source control system within SQL Server Management Studio
- Allows for the creation of custom migration scripts which are saved to source control

### SQL Connect

- Visual Studio add-in that helps source control database schema by allowing database code to be checked in atomically with application code changes

### SQL Compare and SQL Data Compare command lines

- Creates a database from source files in version control
- Generates schema and data deployment scripts
- Validates that two databases are identical
- Generates pre/post-deployment reports for troubleshooting

### SQL Test

- Allows developers to easily create, run, and manage tSQLt unit tests on databases

### SQL Data Generator command line

- Generates realistic test data based on your existing schema

### SQL Doc command line

- Generates the latest database documentation automatically as part of your CI process

### SQL Virtual Restore

- Creates virtual databases from production backups for more realistic testing

### Deployment Manager

- Makes deployments as easy as possible with automated release management

### SQL Monitor

- Ensures that your production database performance is as expected, and that you are alerted when schema changes are detected

## Licensing

Please contact [sqldev.info@red-gate.com](mailto:sqldev.info@red-gate.com) or visit [www.red-gate.com/CI](http://www.red-gate.com/CI) for information about build agent licensing options for these tools. A free trial is available for all Red Gate tools.

## VII. Worked examples

This section illustrates worked examples for databases as part of a continuous integration process using the Red Gate tools from the command line. These can be easily integrated in your MSBuild, NAnt, or PowerShell scripts, which can in turn be used in most CI tools. CI typically starts with a check-in to the version control system. The CI server copies the latest version of the repository to the Build Agent, which contains both the application code and database scripts.

In these examples the source control repository scripts folder is called `<DatabaseScriptsFolder>` and the database that is created by the CI process is `<RedGateCIDB>` on server `<CIserver>`. The angled brackets are not part of the code and are for illustrative purposes only.

Red Gate tools which can be used via the command line are installed by default in the product installation folders. These must be installed on all of the Build Agents in use by your CI process.

sqlcmd.exe is a command line tool from Microsoft that allows SQL to be applied to a specified database.

### 1. Create a database for CI based on the latest version in source control

To ensure repeatability, drop/recreate the `<RedGateCIDB>` database each time:

```
sqlcmd.exe -E -S <CIserver> -Q "ALTER DATABASE <RedGateCIDB> SET SINGLE_
USER WITH ROLLBACK IMMEDIATE; DROP DATABASE <RedGateCIDB>"
```

Create database `<RedGateCIDB>`:

```
sqlcmd.exe -b -E -S <CIserver> -Q "CREATE DATABASE <RedGateCIDB>"
```

Apply the latest revision in source control to the CI database:

```
sqlcompare.exe /scripts1:<DatabaseScriptsFolder> /server2:<CIserver> /
db2:<RedGateCIDB> /sync
```

We are now left with a clean database with the latest schema from source control that can be unit tested along with our compiled application code.

To keep a database up-to-date with the latest changes in source control without rebuilding it each time, set it as the deployment target and use the `/sync` switch:

```
sqlcompare.exe /scripts1:<DatabaseScriptsFolder> /server2:<YourServer> /
db2:<DatabaseToUpdate> /ShowWarnings /verbose /sync
```

## 2. Running tSQLt tests

You can use your existing unit testing and test automation frameworks to test the database. The following examples will use tSQLt, an open source framework designed exclusively for SQL Server testing.

The tSQLt framework is implemented via a set of tables and procedures which get installed on your development database and checked into source control. The benefit of this is that each new test checked in by a developer will be automatically made available to all the other developers in the team the next time they do a 'get latest'. SQL Test makes it easy to develop and run tests prior to checking them into source control. tSQLt is part of Red Gate's SQL Test found on [www.SQL-Test.com](http://www.SQL-Test.com) or can be downloaded separately from [www.tsqit.org](http://www.tsqit.org)

To avoid CI failing if the tSQLt framework hasn't yet been added to the development database, we add a check to ensure the tSQLt framework exists before attempting to run the tests:

```
sqlcmd.exe -E -S<CIServer> -d <RedGateCIDB> -I -Q "IF EXISTS (SELECT * FROM  
sys.objects WHERE object_id = OBJECT_ID(N'[tSQLt].[RunAll]') AND type IN  
(N'P',N'PC')) BEGIN EXEC tSQLt.RunAll END "
```

tSQLt requires CLR, so the CI Server will need to be enabled for it. This is achieved by running the following one-off command against the server:

```
EXEC sp_configure 'clr enabled', 1  
  
GO  
RECONFIGURE
```

To ensure the tSQLt framework can be added, we apply the TRUSTWORTHY change to the CI database each time we recreate it:

```
sqlcmd.exe -b -E -S<CIServer> -Q "ALTER DATABASE <RedGateCIDB> SET  
TRUSTWORTHY ON"
```

Outputting the results is an optional step to write out an xml file that can be interpreted by your CI Server if it supports JUnit XML report processing:

```
sqlcmd.exe -E -S<CIServer> -b -d $<RedGateCIDB > -h-1 -y0 -I -i "<GetTestResults.  
sql>" -o "<TestResults.xml>"
```

The <GetTestResults.sql> file should be placed in source control, containing the following code:

```
:XML ON
IF EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[tSQLt].[XmlResultFormatter]')
AND type IN (N'P',N'PC'))
BEGIN
    EXEC [tSQLt].[XmlResultFormatter];
END
```

This ensures that if tests fail, the build script acknowledges the failure and aborts the CI process. If the tSQLt framework is not installed, -1 will be returned. The number of failed tests will be returned as the exit code:

```
sqlcmd.exe -E -S<CIServer> -d <RedGateCIDB> -h-1 -y0 -I -Q ":EXIT(IF EXISTS
(SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[tSQLt].[TestResult]')
AND type IN (N'U')) SELECT COUNT(*) FROM tSQLt.TestResult WHERE Result !=
'Success' ELSE SELECT -1)"
```

We've now adapted our CI process to not only build a database from the latest in source control, but also to run some database unit tests.

### 3. Generating and validating a database creation script

A database creation script is one that builds the database from scratch. This is useful not only in case a test database needs to be built, but also if new installations of the application are required.

To generate and test the database creation script, instead of using the script from the first example where <RedGateCIDB> is synchronized with the latest <DatabaseScriptsFolder>, we generate the creation script, apply it using sqlcmd.exe, and validate it with sqlcompare.exe.

Generate the creation script which builds the database from scratch:

```
sqlcompare.exe /scripts1:<DatabaseScriptsFolder> /server2:<CIServer> /
db2:<RedGateCIDB> /ShowWarnings /verbose /ScriptFile:<DatabaseCreationScript_Schema.sql> /force
```

Apply the creation script to the CI database:

```
sqlcmd.exe -b -E -S<CIServer> -d<RedGateCIDB> -i <DatabaseCreationScript_Schema.sql>
```

If the creation script has done its job, the resulting <RedGateCIDB> should be equivalent to the latest in source control as we confirm with the following check:

```
sqlcompare.exe /server1:<CIServer> /db1:<RedGateCIDB > /scripts2:<DatabaseScriptsFolder> /verbose /assertidentical
```

**NOTE:** Be aware the /assertidentical switch is used to ensure that sqlcompare.exe returns an exit code of 0 when the source and target data sources are identical instead of its default (if data sources are identical) exit code 63.

#### 4. Generating, testing, and validating a database upgrade script

For a new project, the deployment script is simply a database creation script that builds the database from scratch. For an incremental release of an established product, this will involve upgrading a previously deployed database version to a newer version. This is a crucial process that is all too often left until the end of the project. With tools from Red Gate there is no longer any reason why this should not be tested continuously.

The following process should be repeated for each upgrade path that your application supports.

To test the upgrade deployment script, we need to start with a database that corresponds to the currently deployed version, not with the latest version in source control. In the following example, my source control repository is located at <http://my.source.control/my/repository>, the revision number corresponding to my Production release is 123, and <ProdDBScriptsFolder> is the name of the folder I am checking out to.

The following example will recreate the production database version from source control.

**NOTE:** A different approach to creating a production-level database from source control would be to use a restored backup of Production instead, although depending on the length of the restore time this could be a time-consuming operation that is generally not suited for a regular CI process. Consider restoring a database using SQL Virtual Restore to avoid the need for additional storage requirements.

You can use your source control system's command line tool to check out the scripts folder, or you can use the Red Gate command lines. The following example compares revision 123 with your CI database:

```
sqlcompare.exe /sourcecontrol1 /versionusername1:<username> /  
versionpassword1:<password> /revision1:123 /scriptsfolderxml:<repo_location_  
xmlfile> /s2:<yourserver> /db2:<CIDatabase>
```

/versionusername1 and /versionpassword1 are the credentials required to connect to your source control system. You can optionally use /migrationfolder if you are adding your own custom migration scripts to source control and wish for these to be pulled into your deployment scripts.

The <repo\_location\_xmlfile> is a text file that you must create to accompany the command line. It contains the information required to connect to the source control repository. Create a new text file and paste the contents of the SQLSourceControl Scripts Location extended property of your source controlled database. To find this, right click on the database in the Object Explorer, select Properties, Extended Properties page, and copy the Value text from the SQLSourceControl Scripts Location property to your text file.

It should look something like this:

```
<?xml version="1.0" encoding="utf-16" standalone="yes"?>
<ISOCCompareLocation version="1" type="SvnLocation">
  <RepositoryUrl>
    http://my.source.control/my/repository
  </RepositoryUrl>
</ISOCCompareLocation>
```

Now we need to generate the upgrade script, <DeploymentScript.sql>.

To ensure the production database is not littered with development artifacts, measures should be taken to ensure the test framework doesn't accidentally feature in the deployment script. To exclude the tests and tSQLt schema you can specify the SQL Compare option:

```
/options:default,IgnoretSQLt
```

In this example we generate <PreDeploymentDiffReport.html> which helps troubleshooting if CI aborts, as this represents a log of all changes between the Production revision and the latest in source control:

```
sqlcompare.exe /scripts1:<DatabaseScriptsFolder>/scripts2:<ProdDBScriptsFolder>
/options:default,IgnoretSQLt /ShowWarnings /verbose /Report:
"<PreDeploymentDiffReport.html>" /ReportType:Interactive /ScriptFile:
"<DeploymentScript.sql>" /force"
```

As in the first example, we ensure that <RedGateCIDB> database is dropped and recreated with the TRUSTWORTHY property for tSQLt:

```
sqlcmd.exe -E -S <CIServer> -Q "ALTER DATABASE <RedGateCIDB> SET SINGLE_
USER WITH ROLLBACK IMMEDIATE; DROP DATABASE <RedGateCIDB>"
```

```
sqlcmd.exe -b -E -S <CIServer> -Q "CREATE DATABASE <RedGateCIDB>"
```

```
sqlcmd.exe -b -E -S <CIServer> -Q "ALTER DATABASE <RedGateCIDB> SET
TRUSTWORTHY ON"
```

As we are testing the upgrade process, we need to set our CI database to the production version we retrieved from source control previously:

```
Sqlcompare.exe scripts1:<ProdDBScriptsFolder> /server2:<CIServer> /
db2:<RedGateCIDB> /sync
```

Populate the database with data to make the upgrade more realistic. There are some upgrade processes that will appear to pass successfully when run on a data-free database, such as adding NOT NULL columns. Ensuring that your tables contain data will catch these issues early.

In this example we're loading a previously saved SQL Data Generator project file, <WidgetDevDataGen.sqlgen>:

```
sqldatagenerator.exe /project:<WidgetDevDataGen.sqlgen>
```

Now we apply the deployment script we generated earlier to the CI database:

```
sqlcmd.exe -b -E -S<CIServer> -d<RedGateCIDB> -i <DeploymentScript.sql>
```

To validate the resulting database, we use sqlcompare.exe to ensure that the end result is what we expect it to be, namely the latest version in source control.

Here we also output a second comparison report, <PostDeploymentDiffReport.html>, which upon a CI failure helps us understand what hasn't deployed correctly:

```
sqlcompare.exe /server1:<CIServer> /db1:<RedGateCIDB > /scripts2:<DatabaseScrip
tsFolder> /options:default,IgnoretSQLt /verbose /Report:<PostDeploymentDiffReport.
html> /ReportType:Interactive /assertidentical
```

Should the above comparison return exit code 0, the deployment script has been validated as correct, and we are now in a position to run further build processes, such as the unit tests.

## 5. Documentation

The SQL Doc command line can be used to auto-generate and publish documentation based on your latest validated source controlled database:

```
sqldoc.exe /project:"<SQLDOCCI.sqldoc>" /outputfolder:.myDocFolder /force
```

If you have implemented the above examples, you have a CI process that continuously tests the upgrade process, runs database-level unit tests, and supplies an appropriately versioned database for your application-code-level tests.

This means that throughout your project cycle you will find out immediately whether you have application code, database code, and a database upgrade script that is working.

## 6. Using database packages and Deployment Manager

Red Gate database packages are a convenient way to package up all the artifacts necessary to deploy a database safely.

Database packages contain resources to ensure successful upgrades:

1. **Database upgrade scripts.**
2. **The database creation script.**
3. **Snapshots of the 'before' and 'after' versions. This allows for drift detection, as well as validation that the deployment has been successful.**

Database packages can also dynamically generate upgrade scripts should an appropriate one not exist in the package.

Create the package:

```
SQLPackager.exe /ScriptsFolder:<scriptsfolder> /PackageName:<packagename>
```

Other switches include:

/upgradePackageVersion (a package version to upgrade from)

/packageFeed (NuGet feed URI – required if /upgradePackageVersion used)

Once created, packages can be made available to Deployment Manager by publishing them to a package repository.

## VIII. Further reading and resources

Please visit [www.red-gate.com/CI](http://www.red-gate.com/CI) for the latest resources and further reading. Information on Deployment Manager can be found by visiting [www.red-gate.com/DM](http://www.red-gate.com/DM)



## IX. Conclusions

This article has outlined some best practices and worked examples for implementing databases as part of your CI development process. As with application code, database code is managed in version control using SQL Source Control, and automatically deployed to a CI environment. Red Gate command line tools handle the scripting and deployment process, removing the hurdle that has previously obstructed CI and source control for databases. A database package can be created as part of the CI process and, with minimal effort, deployed through your dev, staging, and production environments using Red Gate's Deployment Manager.

