TAKING CONTROL OF YOUR DATABASE DEVELOPMENT

Nick Ashley

While language-oriented toolsets become more advanced the range of development and deployment tools for databases remains primitive. How often is the DBA left sitting in the corner? Are you scared to change your database because other applications are using it? Do you have difficulty rolling out incremental changes to your databases? Does your database prevent you from releasing your software more frequently? Using real-world experiences - and the principles outlined in Ambler and Sadalage's Refactoring Databases, this paper aims to demonstrate how to take control of the database and make it possible to incrementally evolve and improve its design. With these techniques the database need no longer be the bottleneck in your develop-test-deploy process. This paper features ideas on how to best use open source tools including Ant, CruiseControl and dbdeploy.

The nuances of one particular database refactoring over another are not addressed here. Rather, the intention is to describe the features of a development environment and toolkit that make the creation, testing and onward deployment of database refactorings easier to manage.

SETTING YOURSELF UP FOR SUCCESS

To get the most from the techniques described in this paper, certain working practices are assumed. These assumptions are based on a particular combination of tools and environments; a typical project setup is described below.

TOOLS

Some of the tools discussed here may be over and above what one finds in a DBA's standard toolkit. The assumption is that tools such as text editors, SQL clients, IDE, source control management (SCM) software and DBMS are already in place.

The following table lists some additional and possibly more unusual tools used in this paper.

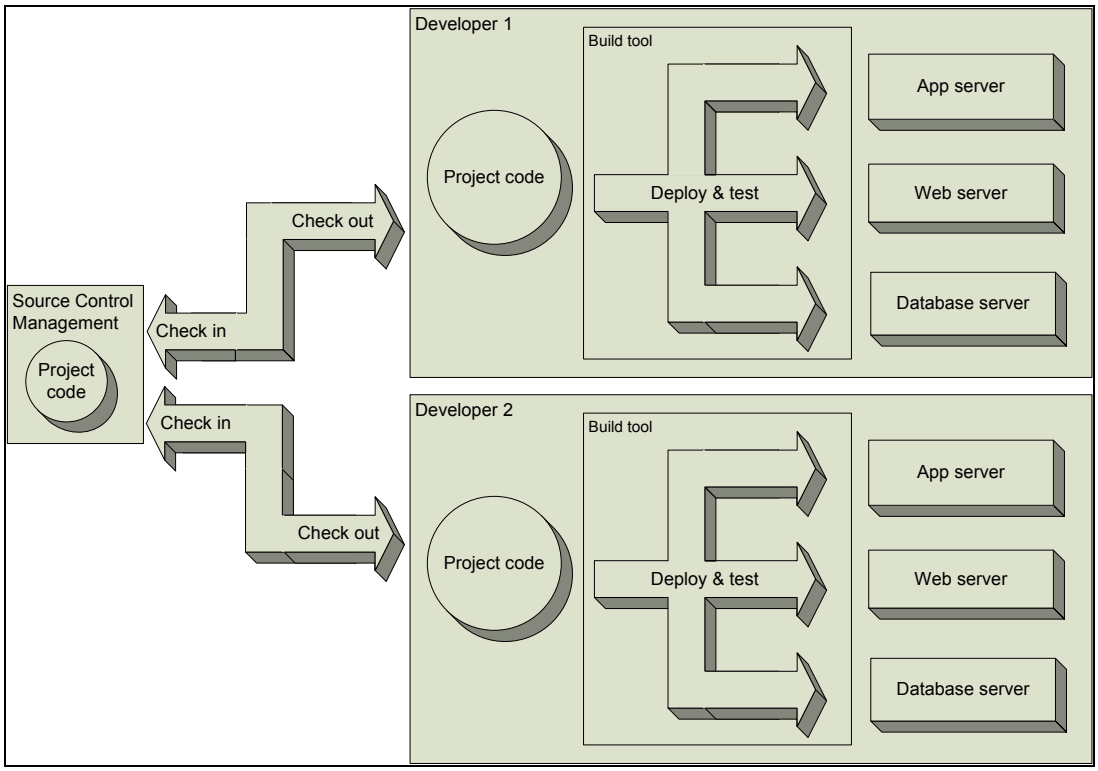
Tool type	*Purpose*	*Specific tool used in this paper*	*Alternatives include*
Build tool	Automates the steps necessary to build and deploy software.	Apache Ant http://ant.apache.org	Nant, Maven, MSBuild
Continuous integration (CI)	Monitors source code management repository for checkins, automatically performs automatic build & test cycles and publishes the results.	CruiseControl http://cruisecontrol.sourceforge.net	CruiseControl.Net, CruiseControl.rb, Bamboo, TeamCity
Database refactoring manager	Automates the process of establishing which database refactorings need to be run against a specific database in order to migrate it to a particular build.	dbdeploy http://dbdeploy.com	MIGRATEdb, Sundog

DEVELOPER SETUP

A developer is anyone whose role requires them to make or modify software artifacts. The definition of software artifacts goes beyond just source code. It includes configuration scripts, test data and, most importantly for the purpose of this paper, database scripts.

The developer setup should be completely self-contained. The developer should be free to experiment as much as possible, safe in the knowledge that the worst that could happen is they destroy only their own environment and not impact the productivity of others. If developers require complete independence then the components represented in the diagram below may all be running on the development workstation. The setup may be adjusted to suit individual needs – for example the developer may be given their own schema on a shared database server.

The following diagram illustrates how developers work in their own self-contained environments by modifying what has been checked out from the SCM repository.



Developer setup

The developer's environment possesses the following key attributes:

- Self-contained
- As representative of production as possible
- Automated tear down and rebuild of all software artifacts – accomplished using the build tool

Working within this environment requires a disciplined approach. Developers should work at the most atomic level possible. Once a feature has been added and is proven to work (using the local build system to test and deploy) it must be checked in immediately. Frequent check-ins play a part in mitigating the need for a manual integration task in your project planning.

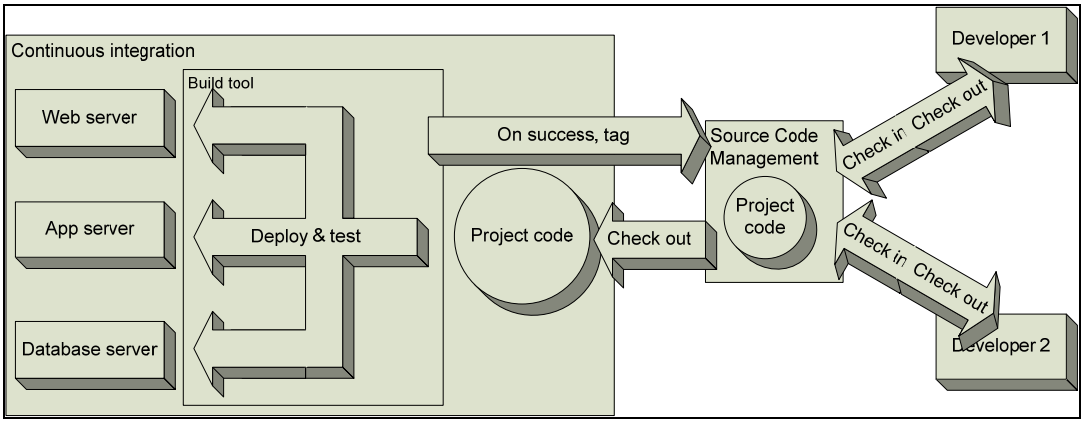
CONTINUOUS INTEGRATION SETUP

While checking in frequently can help to reduce the need for a manual integration task, it's not the whole solution. A developer wants to be certain that when they perform a SCM update they are getting a bundle of code that is working. The developer then performs their own cycle:

- write some code to a point where they think it works
- validate this by running a deploy & test sequence in their own environment

Once the revised code base is working again, it can be checked back into the SCM repository.

The code-test-check-in model starts to break down as developers begin to check in simultaneously. Developer 1 and Developer 2 may have modified their code in such a way that in their own environments the deploy and test passes but when combined in the SCM repository it breaks. This is where continuous integration (CI) helps.



Continuous integration setup

The CI system exists in its own environment and in many ways is very similar to the setup used by individual developers. The role of the CI system is to monitor the SCM system. When CI detects a developer has checked into SCM it triggers a process that broadly does the following:

1. Checks out the head of what's in SCM
2. Runs a deploy and test sequence similar (if not identical) to that which the individual developer runs
3. If the build-deploy -test sequence succeeds, then that version of the code in SCM is tagged with an incremental number, known as the build number
4. If the build-deploy-test sequence fails, then the build itself is said to have failed; the CI server will then drop into a procedure to notify the relevant parties of the failure

In the event of a CI build failure, fixing the build becomes the highest priority of the developer(s) who caused the failure. Keeping the CI build passing prevents a backlog of bugs and integration issues from building up.

DEVELOPING DATABASE REFACTORINGS

Having set the wider context by describing the basic topology of the development environments and procedures it is time to focus in on the creation and management of database refactorings.

The basic process for adding a database refactoring to the code base is as follows:

1. Write the database refactoring
2. Verify the database refactoring in the developer's own environment
3. Check in the database refactoring for verification by the CI build

While the procedure itself is simple enough the ability to implement it has historically been awkward due to a lack of basic tool support. Described here is one potential mix of tools and techniques aimed at making the process as simple and automated as possible.

WRITING A DATABASE REFACTORING

In "Agile Database Techniques" Ambler defines a database refactoring as, "*a simple change to a database schema that improves its design while retaining both its behavioral and informational semantics*". The key element with regard to the procedures described in this paper is that database refactorings are, in themselves, small and discreet units of code.

When writing database refactorings, one should aim to work in units that are as small as possible. Put another way, one would never combine database commands that originate from more than one requirement in a single script. Inevitably, the size of the individual scripts will vary.

The smallest script will consist of just one command. An example might originate from a requirement to improve the performance of a particular query in the application. The requirement can be satisfied by adding an index so the refactoring script will contain a single CREATE INDEX command.

There are requirements that necessitate combinations of commands. An example might be a requirement to add a mandatory column to a table. Depending on the precise requirements, the implementation could follow the pattern of adding the column as nullable, defaulting the values of the column in the pre-existing records, and then adding the not null constraint.

Once the script is complete it is saved it to a pre-determined directory in the code base. When saving the script, begin the name of the file with an incrementing number – look at the number of the last refactoring script and add 1 to it. One might also add a comment in the name of the file to make it clear what the script is there to do. An example script name would be "`4 Add mandatory date of birth column to the customer table.sql`".

VERIFYING A DATABASE REFACTORING

Once the script has been written and saved to the database refactoring script directory, the next step is to perform an update from the SCM repository and then run a build-deploy-test cycle in the local environment. Performing this step validates that the change works, does not break the local build and so therefore should not break the CI build.

To achieve this step, use an ant to call dbdeploy. This requires some one-time setup activity.

ANT AND DBDEPLOY

Apache Ant is a mature tool in widespread use for automating common compile-deploy-test operations. Ant works by executing commands in a *build file*. A build file contains *targets* – a unit of work that accomplishes a specific job. At the lowest level of granularity, a target contains one or more *tasks* – these are the individual steps required to complete the specific job.

Ant is extensible in that there is a published API that allows developers to write their own tasks. dbdeploy is an example of a custom Ant task and works as follows:

1. Looks for delta scripts in a specified directory for .sql files and orders them by name. This explains the importance of each refactoring script name beginning with an incremental number.
2. Reads the content of the changelog table in the specified database. The changelog table contains a record of all database refactorings that have been applied to the database.
3. Establishes which database refactorings have not been run against the specified database. With this knowledge it generates a script containing all of the refactorings to be applied.

Note that dbdeploy will not actually execute any of the database refactorings, it will only output a script of the refactorings that are necessary to be run. This is so because we often find that in more controlled database environments (most notably production) DBAs tend to be unwilling to run anything but raw SQL that they can visually inspect beforehand. While there is no logical reason why dbdeploy cannot be run in a production environment, it is the script generated by dbdeploy that one tends to hand to production gatekeepers.

Ant comes with a supplied task that will execute SQL files. It is this task that gets used to execute the refactorings in the dbdeploy-generated script.

INCLUDING DBDEPLOY IN THE BUILD FILE

This example demonstrates how to make a build file call dbdeploy and then execute the script it generates.

```
<target name="gen-and-exec-delta-script">
       <dbdeploy
            driver="oracle.jdbc.OracleDriver"
            url="jdbc:oracle:thin:@localhost:1521:XE"
            userid="dylan"
            password="nalyd"
            dir="./sql/deltas/"
            outputfile="./build_output/db-deltas-hsql.sql"
            dbms="ora"/>

      <sql
            driver=" oracle.jdbc.OracleDriver"
            url=" jdbc:oracle:thin:@localhost:1521:XE "
            userid="dylan"
            password="nalyd"
            src="./build_output/db-deltas.sql"
            onerror="abort"/>

</target>
```

The dbdeploy task is called with the following attributes:

Attribute	*Description*
driver	Specifies the jdbc driver.
url	Specifies the url of the database that the refactorings are to be applied to.
userid	The ID of a dbms user who has permission to select from the changelog table.
password	The password of the user specified by userid
dir	Full or relative path to the directory containing the delta scripts.
outputfile	The name of the script that dbdeploy will output. Include a full or relative path.
dbms	The target dbms. Valid values are: ora -> Oracle syb-ase -> Sybase ASE hsql -> Hypersonic SQL mssql -> MS SQL Server mysql -> MySQL Database

When the Ant target is executed you should expect output similar to the following:

```
gen-and-exec-delta-script:
 [dbdeploy] dbdeploy v2.11
 [dbdeploy] Reading change scripts from directory C:\Projects\dbdeploy-demo\sql\deltas...
 [dbdeploy] Changes currently applied to database:
 [dbdeploy]   1, 2
 [dbdeploy] Scripts available:
 [dbdeploy]   1, 2, 3, 4
 [dbdeploy] To be applied:
 [dbdeploy]   3, 4
      [sql] Executing file: C:\Projects\dbdeploy-demo\build_output\db-deltas.sql
      [sql] 8 of 8 SQL statements executed successfully
```

The file generated by dbdeploy, in this case db-deltas.sql, contains the content of refactoring scripts 3 and 4. Each individual script is included in the file as a fragment. A fragment begins by inserting a record into the changelog table to record that execution has begun. Next comes the content of the refactoring script itself, and this is followed by a footer where the changelog table record is update to record completion of the refactoring.

This section has concentrated on a snippet of a typical Ant build file. This step to apply database refactorings is likely to be just one of many required in a complete developer build. Other jobs performed by the Ant build commonly include code compilation, artifact deployment, unit tests and functional tests. Discussion of these areas is out of the scope of this paper.

For a more comprehensive description of dbdeploy setup with Ant visit http://dbdeploy.com.

CHECKING IN THE DATABASE RAFACTORING

Assuming that the local developer build has passed the next step is to check the changes into the SCM repository. This will serve two purposes. The first is to make your own modification available to the rest of the team, the second is to initiate a CI build.

We know that dbdeploy works by checking to see if a particular delta script has been run against a particular database. A consequence of this is that once a refactoring script has been checked into the SCM repository, it should be considered to be immutable. The example used previously described a requirement to add a date of birth column to the customer table. Assume that this requirement has now been met and the refactoring script has been added to the SCM repository.

After a few days it becomes apparent that holding a customer's date of birth is no longer a requirement. If one were to simply remove the file "`4 Add mandatory date of birth column to the customer table.sql`" from SCM, it is likely that the databases (developer, QA, UAT etc) would start to become out of step. Any database created before the script was deleted would have this column and any database created afterwards would not. To avoid this, treat *every* database refactoring as a brand new script so in this instance one adds a further script called "`9 Drop mandatory date of birth column from the customer table.sql`". Using this approach ensures that all databases are kept in step with one another.

It is also expected that database refactorings, like all other code changes, will be checked in frequently. This practice is beneficial for a number of reasons, including:

- It encourages work to be split into small, manageable chunks. In the words of Martin Fowler, "this helps track progress and provides a sense of progress".
- It avoids the phenomenon of "check-in Friday" where developers all commit a large number of changes at once. This working pattern often leads to multiple code integration problems that take time to unpick and resolve.
- A steady flow of completed requirements enable downstream consumers of the software (QA engineers, production support engineers and even end users) to get early visibility of what the development team is producing.

The use of a CI tool is core to this method of database refactoring. Building functionality by small steps and rigorously building, deploying and testing the output at every step helps to raise bugs and other integration issues early and as so provide more time for them to be fixed in a more controllable manner.

A THOUGHT FOR THE FUTURE – WOULDN'T IT BE NICE IF…

The procedures outlined in this paper provide an approach to making database refactoring more manageable by making use of tools that are currently available. Scripts are manually written in a text editor or maybe a SQL client, the SCM repository is accessed through the command line or maybe through an addin to file explorer. One executes Ant from the command line or maybe writes scripts to automate this. Whatever the specifics, it is necessary to be continually switching between tools and applications.

Given that the basic goal was to write, verify and check in a simple database refactoring, it is apparent that there is far too much effort involved. Here is the author's Nirvana…

> In my SQL client GUI I right click on a column and select the "Add index…" refactoring option. I am then presented with a dialog that allows me to name the index and then maybe adjust some of the DBMS-specific physical properties. I add a comment to summarize the refactoring and then click the "Save refactoring" button on the dialog. Save refactoring knows where my refactoring scripts live, it calculates the next available script number and writes it to the file system. I then click the "Run database build" button in my SQL Client – this performs the verification for me and ensures that my refactoring is fit for addition to the SCM repository. My final step is to click "Check in".

Developers using languages like Java and the .Net family have had these types of tools for years. It's time for DBA's to demand equal treatment.

REFERENCES

Ambler, "Agile database techniques", Wiley, 2003
Fowler, "Continuous integration", http://www.martinfowler.com/articles/continuousIntegration.html

RESOURCES

A working example of dbdeploy using hsql can be downloaded here: http://dbdeploy.com/software/download
The complete set of Ant documentation can be found here: http://ant.apache.org/manual

THE AUTHOR

Nick Ashley has 10 years consulting experience and has spent that time supporting large development teams as a Data Architect and DBA. His experience ranges from custom application and platform development through to large-scale product development and implementation. Experience with technologies includes Oracle, MS SQL Server, Sybase, J2EE and .Net.

Most recently Nick has spent time working on dbdeploy - an open source database change management tool that's in tune with Agile development methodologies.

You can contact Nick on nick.c.ashley@gmail.com