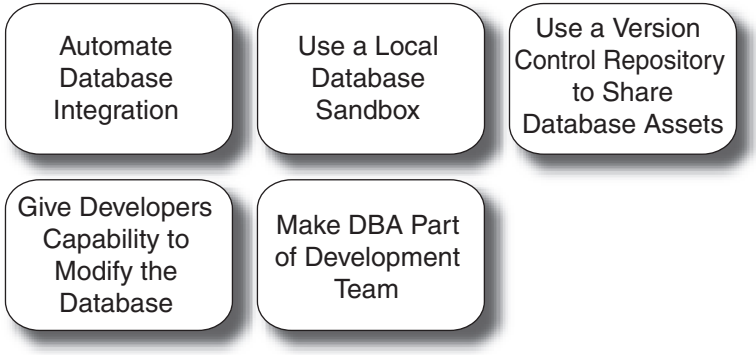
Chapter 5
Continuous Database Integration



Things do not change; we change.
—Henry David Thoreau

Continuous Database Integration (CDBI) is the process of rebuilding your database and test data any time a change is applied to a project's version control repository.

Do you ever feel like your source code and database are operating in different "galaxies" throughout the development lifecycle on projects? As a developer, you may wait several days for a change to the database. You may even be restricted from making minor test data changes, or are afraid to make data changes for fear of ruining the one *shared* database for fellow developers. Situations like these are not unusual, and effectively utilizing CDBI can help alleviate some of these challenges and many others as well.

Revisiting the theme of the book, database integration is one of the parts of the Integrate button (see Figure 5-1), because it is based on the principle that database code (DDL, DML, configuration files, etc.) is, in essence, no different from the rest of the source code in a system. In fact, the artifacts related to database integration:

- Should reside in a version control system
- Can be tested for rigor and inspected for policy compliance
- And can be generated using your build scripts

Therefore, the building of the database can be incorporated into a CI system and can enjoy the same benefits as the rest of the project source code. What's more, changes to database source code can trigger an integration build *just as other source code changes do.*

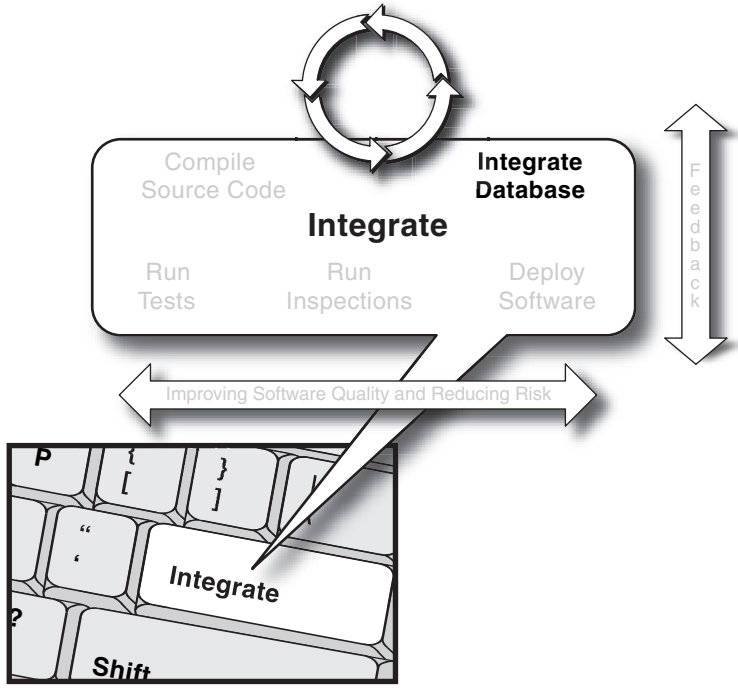


FIGURE 5-1 Database integration in the Integrate button

Not All Data Sources Are Alike

Some projects, or portions of projects, don't use a database exactly the way that we define it in this chapter. However, most projects need to persist data, be it in a flat file, an XML file, a binary file, or an RDBMS. Regardless of your chosen persistent store, the principles of CDBI apply.

As a first step in describing how to automate database integration with CI, we start by describing how to incorporate database integration into a build process. The scripts used to build, configure, and populate a database need to be shared with the rest of the project team, so we discuss which database files are committed to a version control repository. Automating a database integration build process solves only part of the problem, so we go one step further by rebuilding the database and data at *every* software change—making the verification process continuous. If a team is adopting CDBI for the first time, most people on a project will probably need to modify their development practices, so we finish the chapter looking at effective CDBI practices.

Refactoring Databases

The topics covered in this chapter could even be the subject of a separate book.[1] Other materials already make the case for treating your database as just another type of source code that is managed through the version control repository. This chapter gives you the essentials to automate and run database integration processes continuously.

1. In fact, Scott Ambler and Pramod Sadalage have much more in a book called *Refactoring Databases*. Martin Fowler and Pramod Sadalage wrote about similar topics in "Evolutionary Database Design," at www.martinfowler.com/articles/evodb.html.

Automate Database Integration

On many projects, a database administrator (DBA) can often feel like a short-order cook. DBAs typically have analytical skills that took many years to cultivate, but they often spend most of their time performing low-level command tasks. What's more, this job role can also be stressful, because the DBA often becomes a development bottleneck as the team members wait for the DBA to apply one small change to the database after another. Here's a familiar scenario.

> *Nona (Developer):* Hi Julie, will you set up a development database for me on the shared development machine?
>
> *Julie (DBA):* I am in the middle of something. I should be able to set it up later this afternoon. Would you like the data from last week or an export of today's data?
>
> *Nona:* Today's data.
>
> *Julie:* Okay, I can have that for you by tomorrow morning.
>
> *10 minutes later…*
>
> *Scott (Technical Lead):* I am unable to perform testing on the test server because there are no assigned records for the Reviewer role.
>
> *Julie:* Oh, let me create some test records that are assigned this role. I think Nona may have used up all of those records.
>
> *Scott:* Thanks. While you're at it, would you remove the Y/N constraint on the APPROVED columns on the PERSON table? We'd like to use different flags on this column.

It's more of the same on a typical day for the DBA. Not only is this a poor use of the DBA's talents, it causes a significant bottleneck, especially in the continuous approach promoted by CI. If you asked any DBA what they'd rather do on a day-to-day basis, they would probably tell you that they'd rather spend time on data normalization, improving performance, or developing and enforcing standards, not giving people database access or recreating databases and refreshing test data. In this section, you'll see how you can automate these repetitive tasks so both the DBA's and the team's time is spent on improving the efficacy and

TABLE 5-1 Repeatable Database Integration Activities

Activity	*Description*
Drop database	Drop the database and remove the associated data so that you can create a new database with the same name.
Create database	Create a new database using Data Definition Language (DDL).
Insert system data	Insert any initial data (e.g., lookup tables) that your system is expected to contain when delivered.
Insert test data	Insert test data into multiple testing instances.
Migrate database and data	Migrate the database schema and data on a periodic basis (if you are creating a system based on an existing database).
Set up database instances in multiple environments	Establish separate databases to support different versions and environments.
Modify column attributes and constraints	Modify table column attributes and constraints based on requirements and refactoring.
Modify test data	Alter test data as needed for multiple environments.
Modify stored procedures (along with functions and triggers)	Modify and test your stored procedures many times during development (you typically need to do this if you are using stored procedures to provide behavior for your software).
Obtain access to different environments	Log in to different database environments using an ID, password, and database identifier(s).
Back up/restore large data sets	Create specialized functions for especially large data sets or entire databases.

efficiency of the database—not on simple administration. Table 5-1 identifies database integration activities typically performed by a project member that can be automated.

Once you have automated these database-related tasks, you'll find yourself solving problems just by dropping and creating a database followed by inserting test data. This chapter's examples utilize Ant, but the principles apply to any build platform that supports communicating with a database. If your build platform is NAnt, Rake, or Maven, you can do the same things this chapter demonstrates. Listing 5-1 executes a series of SQL statements to create a database including its related tables, comments, constraints, and stored procedures. The script also applies test data for the given environment, such as development or

QA. Using this process, you can simply type `ant db:prepare`[2] from the command line and the build process will perform the tasks outlined in Table 5-1. If you'd like to see this same process using other tools, like NAnt or Maven, we've provided additional examples at the book's associated Web site.[3]

LISTING 5-1 build-database.xml: Automating Database Integration Using Ant

```
> ant -f build-database.xml db:prepare
Buildfile: build-database.xml

db:create:
      [sql] Executing file: data-definition.sql
      [sql] 8 of 8 SQL statements executed successfully

db:insert:
      [sql] Executing file: data-manipulation.sql
      [sql] 60 of 60 SQL statements executed successfully

BUILD SUCCESSFUL
Total time: 20 seconds
```

As you can see, using a single instruction from the command line enables the execution of SQL scripts that define (`db:create`) and manipulate a database (`db:insert`). We describe each of these tasks in more detail in subsequent sections.

Figure 5-2 shows the steps to automate your database integration.

The following sections present a discussion of each component in Figure 5-2.

Creating Your Database

To automate database integration, you must first create a database. In this script, you typically drop and recreate the database, enforce data integrity through constraints and triggers, and define database behav-

2. To manage other environments from the command line, incorporate a feature into your build script to override the default configuration. For instance, in Ant this would be `ant -Denvironment=devqa <targetname>`.

3. At www.integratebutton.com/.

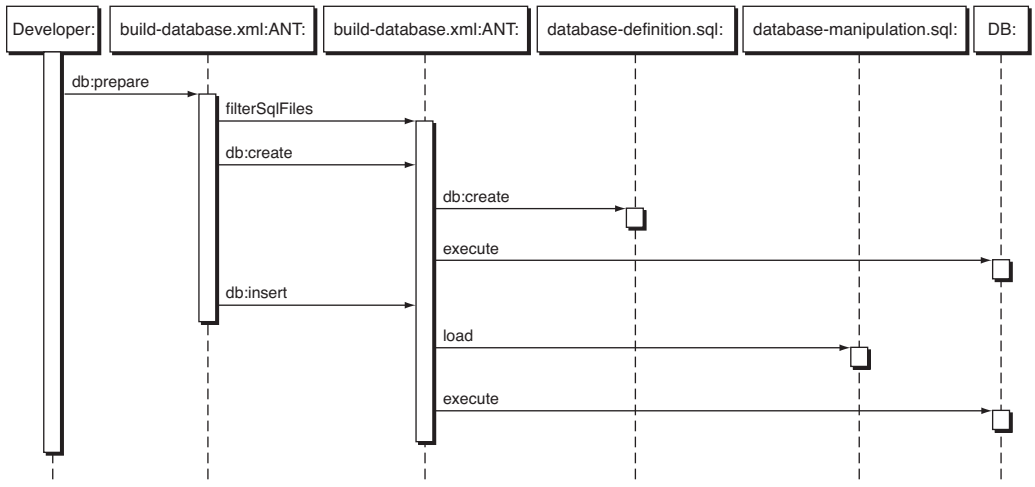


FIGURE 5-2 The sequence of automated database integration

ior through stored procedures or functions. We are using Ant to automate the *execution* of this process in Listing 5-2; however, as mentioned earlier, you can also use make, shell, batch, Rake, Ruby, or any number of tools. Notice that Ant provides a task to execute a SQL script via the `sql` task. Using a build platform like Ant allows you to perform the database integration activities using a sequential approach and enforce dependencies on other targets (a set of tasks) in the script. The example in Listing 5-2 demonstrates the use of Ant's `sql` attributes, such as `driver`, `userid`, and `password`, to connect to the database.

LISTING 5-2 build-database.xml: Defining Your Database Using an Ant Script

```
<target name="db:create" depends="filterSqlFiles" description="Create
  the database definition">
   <sql
      driver="com.mysql.jdbc.Driver"
      url="jdbc:mysql://localhost:3306/"
      userid="root"
      password="root"
      classpathref="db.lib.path"
      src="${filtered.sql.dir}/database-definition.sql"
      delimiter="//"/>
</target>
```

Create Reusable Scripts

When you are writing a script that you plan to reuse, you can define the attributes in a single file so that you only need to define them one time for use in all of your manual and automated scripts, rather than every time you use these attributes.

In Listing 5-3, data-definition.sql is the SQL script that's called by the Ant script in Listing 5-2. We're using a MySQL database in this example, so some of the commands are MySQL-dependent. The data-definition.sql file is responsible for creating the database and its tables, enforcing data integrity, and applying stored procedures. The following is a typical order for this creation process.

1. Database and permissions
2. Tables
3. Sequences
4. Views
5. Stored procedures and functions
6. Triggers

The order of creation within your DDL statements may vary based on database object dependencies. For example, you may have a function that depends on a view, or vice versa, so you may need to list the view first, for example.

LISTING 5-3 data-definition.sql: Sample Database Definition Script for MySQL

```
DROP DATABASE IF EXISTS brewery//
…
CREATE DATABASE IF NOT EXISTS brewery//

GRANT ALL PRIVILEGES ON *.* TO 'brewery'@'localhost' IDENTIFIED BY
'brewery' WITH GRANT OPTION//
GRANT ALL PRIVILEGES ON *.* TO 'brewery'@'%' IDENTIFIED BY 'brewery'
WITH GRANT OPTION//

USE brewery//
…
CREATE TABLE beer(id BIGINT(20) PRIMARY KEY, beer_name VARCHAR(50),
brewer VARCHAR(50), date_received DATE);
CREATE TABLE state(state CHAR(2), description VARCHAR(50));//
```

```
…
CREATE PROCEDURE beerCount(OUT count INT)
BEGIN
    SELECT count(0) INTO count FROM beer;
END
//
```

Technically Speaking…

You may find it easier to organize your targets and scripts by database definition type (such as a table, view, and function) or by subsystem (e.g., Property and Application).

Manipulating Your Database

Once you've created a database from a build script, you'll need to provide initial data (e.g., lookup tables) and test data for testing code that relies on the database. This is where you supply the test data for your particular environment or testing context. What's more, you may also find yourself needing to use different SQL data files to support different environments, like development, test, QA, and production environments.

The example in Listing 5-4 shows an Ant script pointing to a SQL file, whose contents are inserted as test data into a database.

LISTING 5-4 build-database.xml: Manipulating Your Database Using an Ant Script

```
<target name="db:insert" depends="filterSqlFiles" description="Insert
  data">
   <sql
      driver="com.mysql.jdbc.Driver"
      url="jdbc:mysql://localhost:3306/brewery"
      userid="brewery"
      password="brewery"
      classpathref="db.lib.path"
      src="${filtered.sql.dir}/database-manipulation.sql"
      delimiter=";"/>
</target>
```

The SQL script in Listing 5-5 represents test data. This is the script that is referenced in Listing 5-4. In a typical script, you'll have many more records than the three shown in Listing 5-5. Our intent is to give you an idea of what the SQL scripts often execute. Tools like DbUnit

and NDbUnit[4] can help seed the data that is inserted into and deleted from a database as well.

LISTING 5-5 data-manipulation.sql: Sample Database Manipulation Script for MySQL

```
INSERT INTO beer(id, beer_name, brewer, date_received) VALUES (1,
'Liberty Ale','Anchor Brewing Company','2006-12-09');
INSERT INTO beer(id, beer_name, brewer, date_received) VALUES (2,
'Guinness Stout','St. James Gate Brewery','2006-10-23');
INSERT INTO state (state, description) VALUES('VT','Vermont');
INSERT INTO state (state, description) VALUES('VA','Virginia');
INSERT INTO state (state, description) VALUES('VI','Virgin Islands');
```

To achieve the benefits of automated database integration, you'll need to provide scripts for inserting, updating, and deleting data. These data manipulation scripts execute as part of an overall build process. Next, we discuss how to tie these scripts together with the orchestration script.

Creating a Build Database Orchestration Script

A database integration orchestration script executes the DDL and Data Manipulation Language (DML) statements. Listing 5-6 shows an Ant script that uses the `sql` task to call the data-definition.sql and data-manipulation.sql files we created in Listing 5-3 and Listing 5-5. You'll incorporate this orchestration into your higher-level build and integration processes.

LISTING 5-6 build-database.xml: Database Integration Orchestration Script Using Ant

```
<target name="db:prepare" depends="db:create, db:insert"/>
<target name="db:create">
…
<target name="db:insert" depends="filterSqlFiles">
…
```

4. DbUnit is available at www.dbunit.org/ and NDbUnit is available at www.ndbunit.org/.

Are You on Autopilot?

As you are automating your database integration, a few things may trip you up. It's easy for manual activities to unintentionally accumulate in your database integration process. Try to resist this. As Andrew Hunt and David Thomas mention in *The Pragmatic Programmer*: Don't Repeat Yourself (or DRY, for short), keep your build scripts "DRY." An easy form of "duplication" to miss is when we get acclimated to clicking through the database vendor's GUI application wizard rather than interfacing through the command line where it can run scripted. Another potential problem is the tendency to wait until there are many DDL/DML changes before committing back to the version control repository. Database changes can be pervasive, so try to make and check in small, incremental changes to your database; this will make it easier to test and debug.

Use a Local Database Sandbox

A significant challenge on many software development projects is making changes to the database structure. Many projects I've observed typically use one shared database, so when developers make changes to this shared development database they can adversely affect others on the team—causing each developer's private build to break (if their tests are part of the build). If developers have their own local code "sandbox" to isolate their coding changes from other developers, wouldn't it be great if they had a "database sandbox" too?

Multiple Database Instances

You may not have the resources to get a database for each developer. In this situation, you could assign each developer a separate schema on a central database server or use one of the freely available, lightweight, open source equivalent databases. Furthermore, many of the more widely used RDBMSs provide free developer versions.

Another important capability you gain by automating your database integration is that everyone on the team will be able to create a local instance of the database on their workstations. Every team member can then create a database "sandbox" to make and test database changes without affecting others. If your database integration is scripted, creating a new database instance is a push-button affair; conversely, if you don't automate your database integration, it is more difficult to recreate your database and run tests on your workstation. Figure 5-3 provides an illustration of each developer using a local database instance.

Using automated database integration, you are able to get the latest version of your database scripts along with your application source code. Each developer is able to create a local instance of the database, modify the version of the database on his workstation, test the changes, and commit the changes back to the repository. These changes will be integrated and tested with the rest of the software as part of the CI system. When another developer refreshes her private workspace with changes from the repository, the database changes are

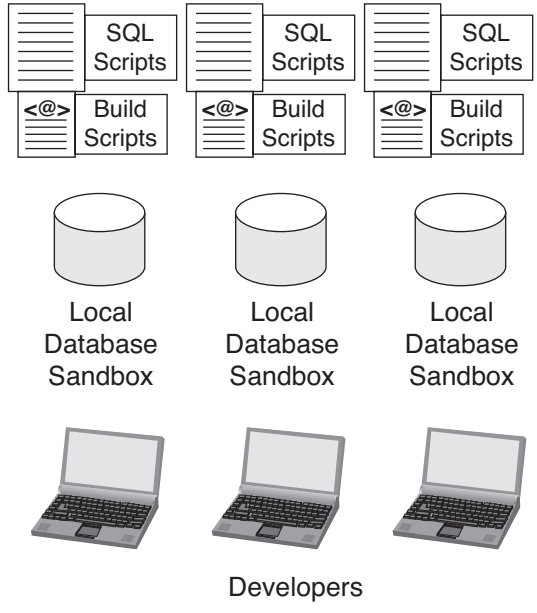


FIGURE 5-3 Each developer uses a local database sandbox

Supporting Multiple Database Environments

The next logical step after creating a local database sandbox is creating different database instances to support multiple database environments. For example, you may need to create a database that contains all of your migrated production data. Assuming there are many records in this database, you probably don't want to include it in your local development database. Usually, this will only be the DML (data changes), *not* the DDL (create, alter, and drop statements to the database). By automating your database integration, you can modify build script parameters to include the data to support these environments. This way, you can execute one command to provide data for different database environments. The same goes for versions. You may want to test new code against a prior version of the database. Use automated database integration to provide this capability with a "push of the Integrate button."

copied down to her workstation along with the other source code changes, and her next private build will incorporate the changes in her local database instance.

The next section identifies the reasons and approach for using a version control repository for database integration.

Use a Version Control Repository to Share Database Assets

Sharing your database integration scripts is a best practice, plain and simple. All software assets need to be in a version control repository, and this includes all database assets. Such assets might include the following:

- DDL to drop and create tables and views, including constraints and triggers
- Stored procedures and functions

- Entity relationship diagrams
- Test data for different environments
- Specific database configurations

For numerous project scenarios, you should be able to recreate your entire database from "scratch" using the scripts in your version control repository (for large data sets, you may store data export scripts rather than row-by-row DML scripts). Once you've applied all your database assets to the version control repository, you'll have a history of all of the database changes, so you can run prior versions of the database with the latest code (or with prior versions of the code as well). This also reduces the gridlock on projects when all the developers need to go to the DBA for everything. Once database assets are in one place, you can make a change to a database column, perform a private build on your machine, commit it to the version control system, and know you will receive feedback after the integration build is run.

Sometimes during development the database will need to undergo large-scale changes. In most cases, these changes will require the expertise of several people on the team and a longer duration to complete. When such situations arise, it is best to create a task branch[5] to commit the changes back into the version control repository rather than break the mainline and slow the activity of the rest of the team. Without CDBI, often the DBA will be making these large-scale database alterations, and he may be less suited to make all the changes at once to the database, dependent application source code, associated test code, and shared scripts because he may lack the knowledge of the source code that developers are writing.

Just as you have a consistent directory structure for your source code, you'll want to do the same for your database. Define the location of database assets—probably somewhere in the implementation/construction directory where your source code is located. In your database directory, define subdirectories for each of the database entity types and environments. Listing 5-7 shows a directory structure for an implementation directory (using a MySQL database).

5. In *Software Configuration Management Patterns,* Stephen P. Berczuk and Brad Appleton describe a *task branch* as having "part of your team perform a disruptive task without forcing the rest of the team to work around them. . . ."

LISTING 5-7 Sample Implementation Directory

```
implementation
    bin
    build
        filtered-sql
    config
        properties
        xml
    database
        migration
    lib
        mysql
    src
    tests
tools
    mysql
```

Just as with your source code, choose a directory structure that works well for you, one that clearly defines the entities while making it adaptable to changes.

Directory Structure and Script Maintenance

In the beginning, you may find that the directory structure is less important, but beware of making frequent directory structure changes, as you'll spend additional time updating your scripts to account for these changes.

Now that you've automated your database integration activities and are checking them into the version control repository to share with others on the team, let's make the process continuous so that it is run with every change to the software.

Continuous Database Integration

This is where the "rubber meets the road." The reason to automate, share, and build the database integration processes is so you can make these processes continuous. Using CDBI, your database and your source code are synchronized many times a day. Once you commit

your database changes to your version control repository, the CI system proceeds like this: It gets a complete copy of the system source code, including your database data definition and manipulation scripts; recreates your database from the source; integrates your other source code; and then runs through your automated tests and inspections to ensure that the change(s) didn't introduce defects into your system's code base. Figure 5-4 demonstrates how the changes made by each developer are synchronized with the integration build based on the mainline in the version control repository.

Figure 5-4 shows that the changes that were made at 10 AM (by Mike) *and* the changes that were made at 10:15 AM (by Sandy) are

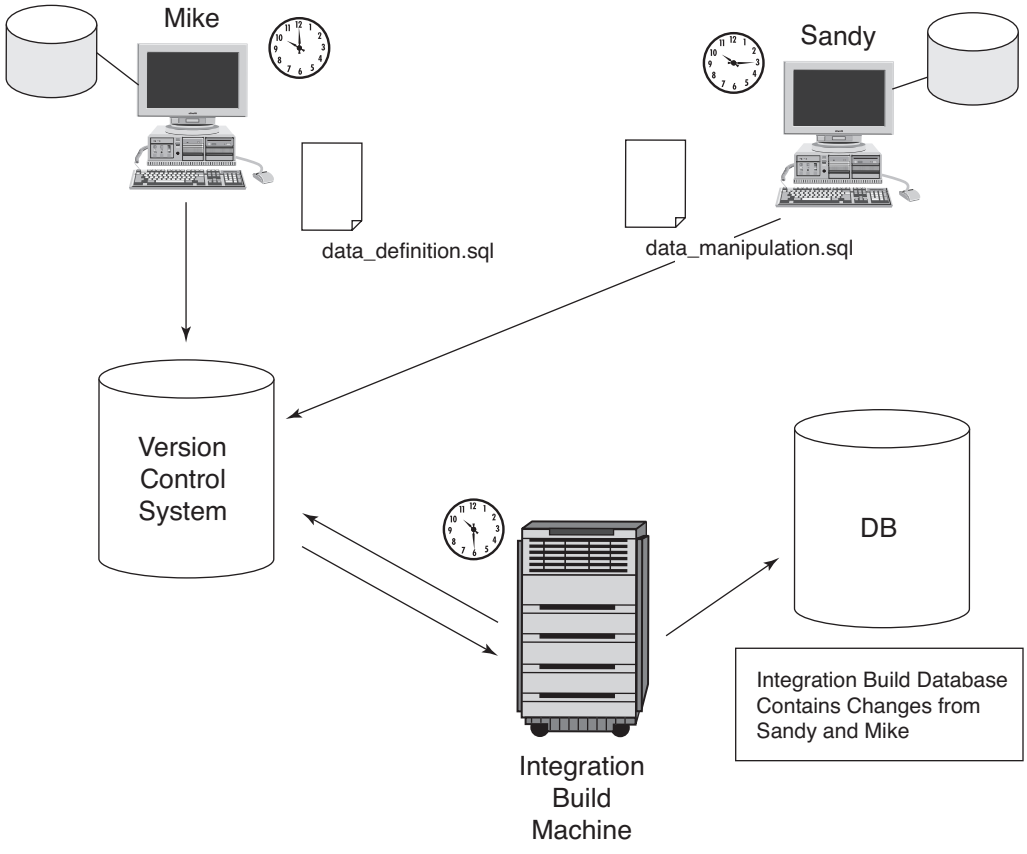


FIGURE 5-4 Single source for database changes

included in the integration build that occurred at 10:30 AM. The integration build machine uses a single source point, provided by the version control repository, to synchronize and test changes as a part of the integration build.

Once you have automated your database integration and incorporated it into your build scripts, making it run *continuously* is simple. Your database integration tasks, along with the rest of your build, should be executed using one command (such as an Ant/NAnt `target`). To run your database integration tasks continuously, you only need to make sure these database integration build task commands are executed as a part of the automated build.

Give Developers the Capability to Modify the Database

Each developer should have the capability to modify any of the database scripts. This doesn't mean that every developer *will* modify these database scripts, because not every developer will have the necessary database expertise. Because each developer will have his own database sandbox, each can modify the local database and then commit the changes to the version control repository. This will reduce the DBA bottleneck and empower developers to make necessary changes. The DBA can evaluate the new changes to the repository by reviewing the integration builds or working with the developers if the build breaks.

As the adage goes, with this additional authority comes additional responsibility. Changes to the underlying database structure can have far-reaching impacts on the system. The developer who makes changes to the database structure must assume the responsibility for thorough testing before committing these changes. We feel it is far more likely in today's industry for a developer to have a knowledge of databases and database scripting—and the DBA is still there to "oversee" what changes, if any, move into the system.

The Team Focuses Together on Fixing Broken Builds

Since you treat the database the same as the other source code, you may experience broken builds because of a database error. Of course, errors may occur in any part of your build: source code, deployment, tests, inspections, as well as the database. When using CDBI, database integration is just another part of the build, so the playing field is leveled: Whatever breaks the build, the priority is to fix it. The payoff comes after this; the fix is now integrated, and that particular issue is prevented from recurring.

Make the DBA Part of the Development Team

Break down barriers and make members of your database team a part of the development team. You may already be doing this, but all too often there is a "wall" between the DBA and the software developers. As mentioned earlier, treat your database code and your other source code in the same manner. The same goes for the people on your team. This is probably the most controversial of the CDBI practices. We've worked on teams that have used CDBI with the DBA on the development team, and we've also seen the more traditional approach with the DBA on another team, the database team. CDBI worked in both environments, but it worked significantly better when the DBA was a part of the team.

Some people ask, "If the DBA is no longer dropping and recreating tables, creating test environments, and granting access, then what is she doing?" The simple answer is, "Now she can do her job!"—spending more time on higher-level tasks such as improving database performance, improving SQL performance, data normalization, and other value-added improvements.

Database Integration and the Integrate Button

The rest of this book covers topics concerning the additional parts of the Integrate button: continuous testing, inspection, deployment, and feedback. This section covers some specific issues concerning these practices when it comes to database integration.

Testing

Just as with source code, you'll want to test your database. We cover testing in detail in Chapter 6. There are tools you can use for database-specific testing such as PL/Unit, OUnit for Oracle, and SQLUnit. Your database may contain behavior in stored procedures or functions that needs to be tested and executed as a part of the build script, just like the behavior of your other source code. You may also want to test the interactions of constraints, triggers, and transactional boundaries by performing application security data tests.

Inspection

As with your other source code, you should be running inspections on your data source. This includes not just your DDL, but reference and testing data as well. There are tools you can incorporate and run in your automated build process so that you do not need to run these inspections manually. Here are a few ideas for inspections on your database.

- Ensure efficient data performance by running `set explain` against your project's rules to target optimizations for your SQL queries.
- Analyze data to ensure data integrity.
- Use a SQL recorder tool to determine which queries are being run the most. These queries might be candidates for stored procedures.
- Ensure adherence to data naming conventions and standards.

Deployment

As we have indicated, the goal of CDBI is to treat your database source code and other source code in the same manner. The Continuous Deployment process will deploy your database to your development and test database instances just as it deploys your other code to its different environments (e.g., application servers). If you need to migrate from one database to another, you will be able to better test the migration process by running through the process on a continuous or scheduled basis.

Feedback and Documentation

When you incorporate continuous feedback and CDBI into your CI system, you will find out if your build failed because of the latest database changes. By default, most CI systems send the build status to the people who last applied changes to the version control repository. Just like with the source code, the CI system notifies those who made database changes quickly so that they can make the necessary fixes to the database.

Documentation is about communication, and there is much about the database you'll want to communicate to other project members or your customer. Your Entity Relationship Diagram (ERD) and data dictionary are excellent candidates for generating as a part of your continuous build process, perhaps as a secondary build (described in Chapter 4).

❑ ❑ ❑ ❑ ❑ ❑ ❑ ❑ ❑ ❑

Summary

This chapter demonstrated that database assets are the same as other source code. Therefore, the same principles apply.

- Automate your database integration using orchestrated build scripts that are run continuously, after any change to your database or its source code.
- Ensure a single source for database assets by placing them in a version control repository.

- Test and inspect your database scripts and code.
- Change database development practices by ensuring that all database integration is managed through the build scripts, that all database assets are checked into version control, and that all developers (who interact with the database) have a database sandbox.

Table 5-2 summarizes the practices covered in this chapter.

TABLE 5-2 CI Practices Discussed in This Chapter

Practice	*Description*
Automate database integration	Rebuild your database and insert test data as part of your automated build.
Use a local database sandbox	All developers should have their own copy of the database that can be generated via SQL scripts. This can be on their workstations or even shared on a development server—as long as all developers have their own copy on this shared server.
Use a version control repository to share database assets	Commit your DDL and DML scripts to your version control system so that other developers can run the same scripts to rebuild the database and test data.
Give developers the capability to modify the database	Avoid the DBA bottleneck that occurs when database changes are restricted to just one or two people. Give developers the capability to modify the DDL and DML scripts and commit them to the version control repository.
Make the DBA part of the development team	Be sure the DBA can run the same automated build—which includes a database rebuild that other developers run—to ensure consistency. By making the DBA a part of the development team, the shared experiences can benefit both the database and the development teams.

Let's see how Julie, Scott, and Nona are doing now that they're using CDBI.

Nona (Developer): I need to refresh my test data. What do I need to do?

Scott (Technical Lead): Just run `ant db:refresh` from the command line. Before you do that, get the latest changes out of Subversion by typing `ant scm:update`, because I made a few changes to the USER database table and the source code that uses this change.

Julie (DBA): Do you guys need any help?

Scott: Yeah, we are having a performance problem on one of the queries. Do you have time to look at it? Also, I think we need to denormalize the PRODUCT table. Can you model the table changes, prototype the DDL changes, and set up a code branch so Nona can modify her code for your changes? When you two are satisfied with the changes, merge the branch and commit it to Subversion so that they run as part of the integration build. Thanks, Julie.

Nona: . . . Sure, Scott. Should we use the test database rather than the development database?

Scott: Yeah, just run `ant -Denvironment=test db:refresh`.

The developers and DBAs, who often perform roles that seem opposing or distant, are now continually working toward the same goal, and both are accomplishing more of their tasks that require analysis or design.

Questions

These questions can help you determine your level of automation and continuous database integration.

- Are you capable of recreating your database from your automated build process? Can you rebuild your database at the "push of a button?"
- Are the scripts (build and SQL) to your database integration automation committed to your version control repository?
- Is *everyone* on your project capable of recreating the database using the automated build process?
- During development, are you able to go back to prior versions of the database using your version control repository?
- Is your database integration process continuous? Are your software code changes integrated and tested with the latest database whenever you apply those changes to the version control repository?
- Are you running tests to verify the behavior of your database stored procedures and triggers?
- Is your automated database integration process configurable? Are you able to modify the userid, password, unique database identifier, tablespace size, and so on using a single configuration file?