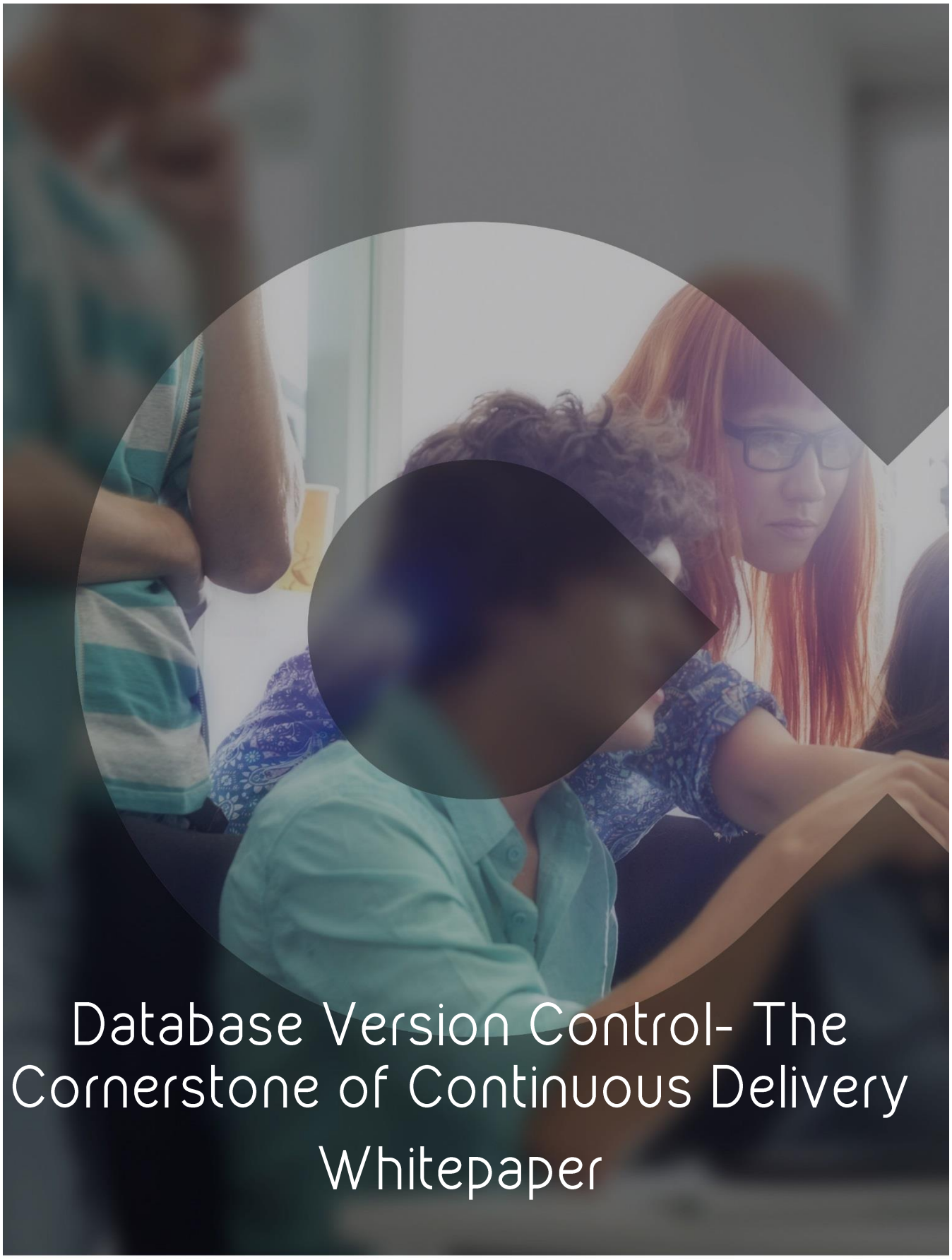


Database Version Control- The Cornerstone of Continuous Delivery

Whitepaper

DBmaestro
DevOps for Database

DBmaestro
DevOps for Database

An ever increasing number of organizations are implementing DevOps using continuous delivery process. They are fueled by reports of the benefits, which include quicker time to market, reduced costs and higher quality products.

DevOps is mostly about organizational culture, while continuous delivery and continuous integrations are mainly about automation and tests, which of course require a trustworthy source control.

This white paper focuses on the unique requirements of database continuous delivery from the source control infrastructure.

***

Let's begin by examining how native code development, continuous integration and continuous delivery processes interact with the source control.

Native Code

Every organization has its own processes for compiling native code. It can be manual or automatic using Jenkins, Bamboo etc., and the build system can be maven, make, or others. But all builds have one thing in common which the build process relies on:

Each build starts with an empty folder and then gets the relevant source code files from the file-based version control repository (SVN, Git, Perforce, Microsoft TFS, IBM RTC etc.). Then it compiles the source code files, and, if the compilation succeeds, the process can continue to the next step which is to deploy to an automated test environment. Some organizations also save the compilation phase output (binary artifacts) result in a binary source control repository (SVN, Git, Perforce, TFS, RTC etc.) so deployment process can retrieve the relevant artifacts from the file-based (binaries) source control repository.

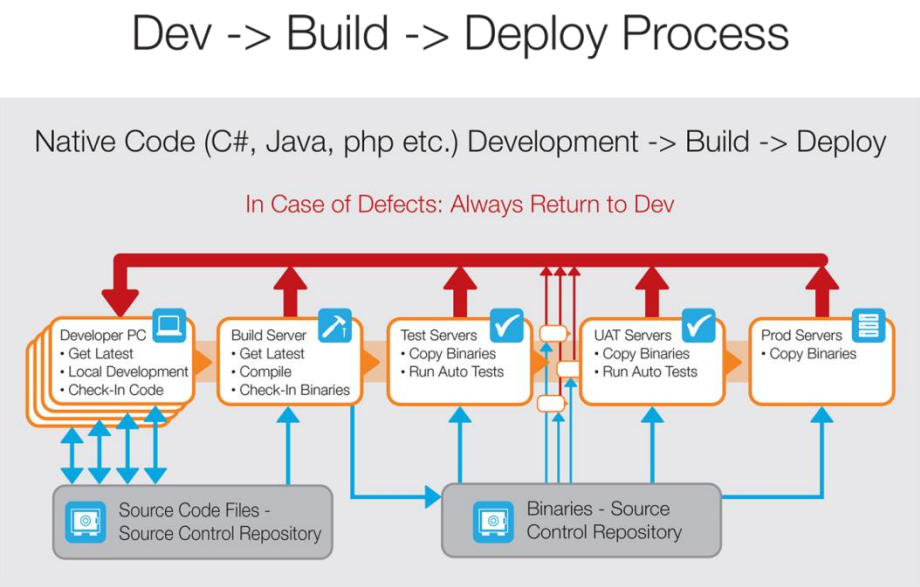
Deployment to an automatic test environment process can be done differently in different organizations. One company will do it manually; a second will run scripts and third will use an application release automation tool, such as IBM UrbanCode Deploy or CA Release Automation etc. The common factor for all deployments is copying the artifact (binary) to the relevant server, while overriding what was done before (as seen in image above).

Every change a developer makes must be documented in the source control repository. If it doesn't exist in the source control repository it is not included in the build process. Furthermore, if a developer copies an artifact generated locally to a test environment, the next deployment will override this out-of-process change.

Occasionally, defects will only recreate in test environments but due to infrastructure limitations (storage, costs, and complex architecture) the developer is required to work in the test environment and not in the development environment. In those cases, the developer may need to copy the locally-generated artifact directly to the test environment. Once the developer checks-in the code changes, the next deploy to the test environment will override the locally-generated artifact with an artifact from the binaries source control. This built in safety net in the process, prevents out-of-process locally-generated artifacts from entering into the production environment.

The database code has similar characteristics to native code but also has some inherent differences, which require adjustments to the source control infrastructure from perspectives of native code development, build and deploy.

For the simplicity, consider the continuous deliver process as consists of several phases: Develop Build and Deploy. Let's review of this process in reverse (Deploy -> Build -> Develop) because the input of the latter process defines the former's output and so on.



DBmaestro
DevOps for Database

Database Deployment

A database code deployment is done by running SQL scripts (DDL, DCL and DML), which change the current structure and data to the desired version. When comparing database deployment to native code deployment, the differences are crystal clear; native code deployment is done by copying the new binaries (DLL, jar etc.) to the target environment. The previous version of the artifacts is no longer valid and may be saved for a quick rollback. This is the safety net which prevents out-of-process artifacts from reaching the production servers.

This is not the case with database code deployment. The script changes version A to version Z by many DDL, DCL and DML commands, with every command changing the version a little bit. If the current version of the database is not A, there are two possible outcomes:

1. The script will ignore current version and override the structure with whatever exists in the script.

Or:

2. The script will fail. For example: trying to add a column that already exists with a wrong data type .

An error in a deployment process is not usually a desirable outcome. In this case however, getting the error is better than having the script run successfully and then without warning, revert changes made to production as an emergency fix or changes made in the trunk/stash. These changes would have been made by a different team or merged from a different branch.

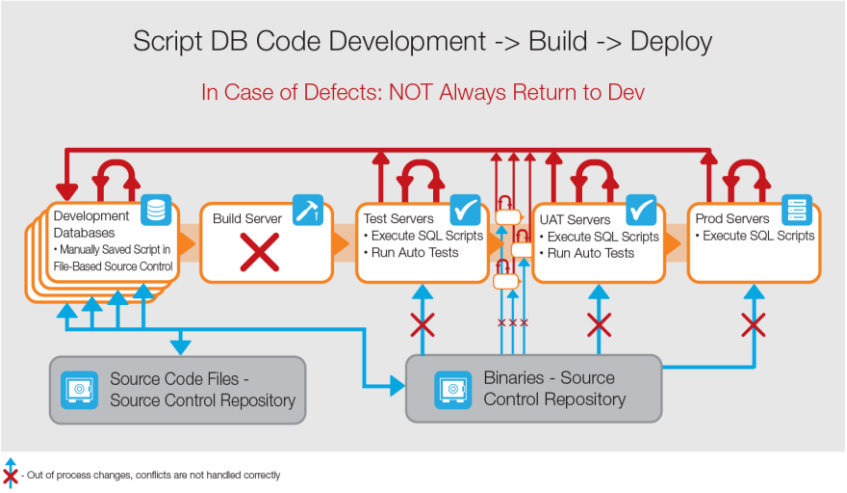
The input of the deployment phase is SQL script(s) which are generated in the build phase.

DBmaestro
DevOps for Database

Database Build

In order to generate the correct database change scripts, the build phase must have information on the current structure and source control structure. But just having the current and source control structure (as is the case with standard compare & sync tools) is not enough .

Dev -> Build -> Deploy Process



Simply comparing two environments does not provide insight regarding the nature of the differences, for example: (a.) a case where the difference conflicts with an emergency fix. (b.) the trunk/stash/QA environment was already updated with other changes from a different branch. (c.) the later environment (trunk/stash/QA) is more up-to-date regarding specific objects - thus the difference should not be part of the delta changes script.

This missing information is only available with baseline aware analysis. The input for the database build phase should absolutely be taken from the source control repository which includes only changes that were checked-in, and does not include changes that are still in work-in-progress mode. This brings us to the starting point of the process – the source control and how to make sure the build process retrieves the relevant changes.

Simple Compare & Sync

Source vs. Target	Action
=	No Action
≠	?

Baseline Aware Deployment

Source vs. Baseline	Target vs. Baseline	Action
=	=	No Action
≠	=	Deploy Change
=	≠	Protect Target
≠	≠	Merge Changes

DBmaestro
DevOps for Database

Develop Using a Reliable Database Source Control

In this phase, developers introduce changes to the database structure, reference lookup content or logic in the database (procedures, function etc.)

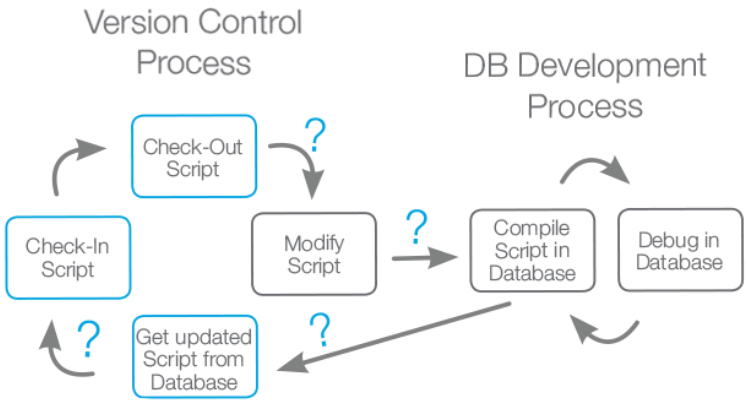
The two common approaches of database development are: (1) using a shared database environment for the team. (2) Using a private database environment for each developer. Both methods have many advantages and challenges. Using a shared database environment reduces the code merges for the database code and also reduces the complexity and cost of updating the database structure based on the source control. Using a private database environment causes many merges of the database code but reduces the potential of code overriding by another colleague. In addition, a private database environment may have other factors to consider such as management overhead, licenses, hardware, and cost .

The primary reason why the private environment method is not commonly used, relates to how developers publish changes from their private (workspace) environment to the integration environment. Publishing changes should not revert changes made by someone else and updating the private environment from the source control repository should not revert work-in-progress.

The same process of building the native code using only changes which are documented in the source control repository should be applied to database code changes. Developers work on the native code in the IDE and then check-in the changes to the source control repository without any additional manual steps. Having a file-based script that a developer is maintaining for his/her changes will create a few challenges that will be difficult to resolve and will require a lot of time:

1. How to guarantee that the version control repository correctly represents the database structure that was tested.

2. Developer A made a number of changes and developer B made other changes to the script, none of the developers can execute his/her entire script, because the script overrides (or reverts) the changes introduced by the other developer.



In addition there are other challenges that occur in deployment phase but originate in previous phases:

1. Controlling the order of the execution of scripts created by several developers.

2. Maintaining the change scripts on a release scope change.

3. Instead of running many small scripts (in the same order they've been executed in QA) which may change the same object several times, execute fewer scripts and change the object only once – this is difficult to practice as it will cost lots of time to generate the script from scratch, test it etc.

Source Control – Single Source of Truth?

Anyone with sufficient database credentials may login to the database, introduce a change, and forget to apply the change in the relevant script of the file-based version control. This is what has reportedly happened in finance, insurance, online travel, algo-trading, gaming and other industries.

Database Deployment logic

Another unique challenge from the database point of view is how deployment is done. Can the database deployment process act as the native code – replacing the existing database/table in production with the new database/table from the development? Or does it have to alter the existing database structure in production from the current state to the target state to preserve the data ?

Deploying native code artifacts – binaries of Java, C#, C++ - is done by copying the new binaries and overriding the existing ones (current state has no effect on the binary content). Deploying database code changes is done by changing the structure of the database or schema from the given state (current state) to the target state (end point). When executing a script in the database, the given state (current point) must be the same as it was when the script was generated otherwise the outcome is not predictable.

DBmaestro
DevOps for Database

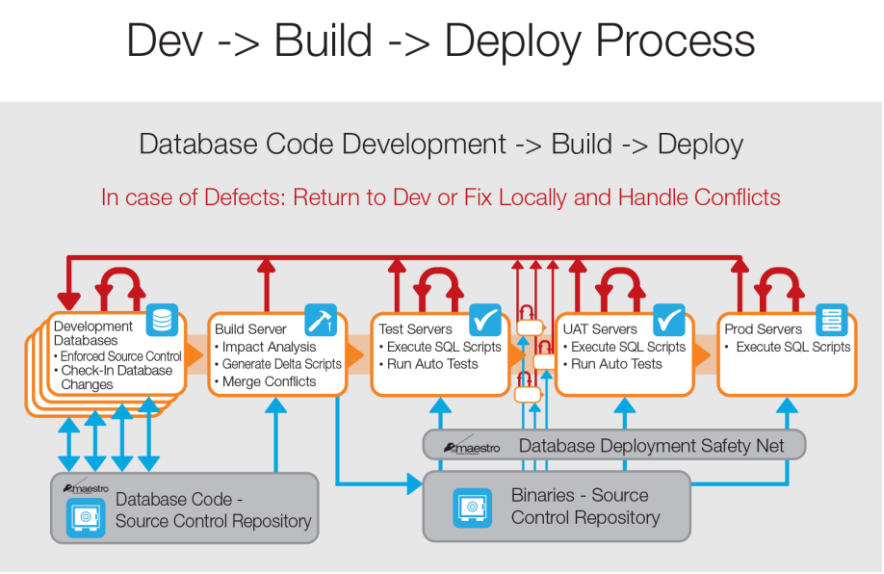
Continuous Delivery for Databases

Continuous Delivery for database changes is implemented by answering the unique challenges of database change management and using the same principles of Continuous Delivery practiced for native code.

Continuous Delivery is all about automation - automating everything. In order to automate we must have confidence in the automation process regardless if it's through scripts, a CI tool or by an ARA (Application Release Automation) tool. If you can't be confident that in case of error (or a suspected error) the automation will raise a red flag, than after the first or second failure, people will stop using that specific automation.

What can be automated in database continuous delivery? Almost all the steps, from the **build** (generating the SQL scripts to), to **deploy** (executing the SQL scripts), to **test** (verifying that the script were executed correctly and that after executing the script the structure is the desired one).

Automated Database Changes Build



Having an automation process means that the SQL scripts generated in the Build phase are executed in the Deploy phase automatically (without any human intervention). If the scripts are generated incorrectly, for example, the order of the commands does not consider database dependencies - or it promotes changes from a specific development environment and ignores changes made by different source environment, (such as a different branch, emergency fix, pre-prod, UAT) - the end result can cause downtime to the organization.

Building the script is done by comparing the object structure (or content) between the source control repository, which has the desired structure, and the current state of the object in the higher environment of the process (QA, SIT, DIT, Pre-Production etc.). When doing this analysis, some questions should be asked in order to decide if the change should be included in the script or not.

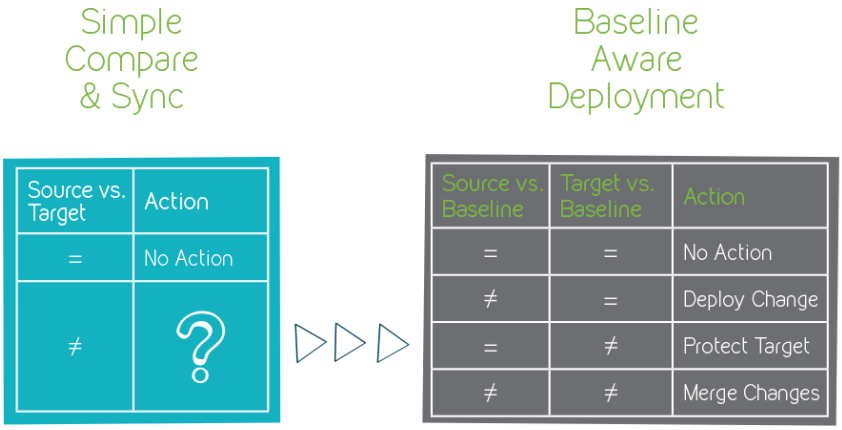
1. Should the entire schema/database structure be analyzed or just objects which were changed based on tasks, user-stories, requirements, change requests etc.?

2. Does the change between the objects' environment originate from the development environment, and should therefore be promoted?

3. Does the change between the objects' environment originate from a different environment and should therefore be preserved (or should it be skipped and ignored?)

4. Does the change between the objects' environment create a conflict between the object and the code and should it be merged (for example: line 2 was changed in the target environment and line 4 was changed in the source environment)?

The last thing we want is to have a script that contains commands to wrongly change an object. For example: if the script was generated using the DIT environment, then the object structure in the DIT is the correct state while in the developer private environment the object structure is an old version (see question #2). Or if the script was generated using the UAT environment, maybe the object was changed from the emergency fix branch as well as from the development branch, and now there is a conflict (see question #3).



Once the list of objects (which have changes that should be included in the script) are defined, the second step is to build the modification script which will transform the object from state A to state Z. This step involves logic that analyzes the database dependencies and generates the DDL, DCL, DML command in the correct order. Same logic exists in building native code (compiling) as developers manually maintain the dependencies between binaries and the compiler knows the order to compile and generate the binaries.

How to know which DB change relates to which task?

By correlating the check-in to the task.

Once the information regarding the reason for this change is saved in the repository, this information can be used to retrieve only the relevant changes & objects when building the list of objects.

How to know if the script should perform the change?

By running a baseline-aware analysis.

The baseline is a label created beforehand in the version control repository, which reflects the expected structure of the object, schema, and database, and highlights the nature of the change. If there is difference between source & baseline, it means that the change of the object should be in the script. If there is a difference between target & baseline, it would mean the change of the object should not be in the script as the object was not changed in the source environment.

The same principle exists when several developers work on the same Java, C# or C++ code. When a developer checks-in his changes, the version control compares the latest revision of the object, the developer copy and the object baseline. The object's baseline is the object's revision which the developer checked-out (or started modifying). If there is a change between the latest revision and the object's baseline, that would mean that someone else already modified the object and the developer cannot check-in his changes. The developer first needs to resolve the conflict in his local copy and then do a check-in. A baseline is defined for every object and every check-in.

Automated Database Changes Execution

Executing the SQL script is quite simple; it can be done by command line utilities, by scripts, and by ARA (Application Release Automation) tools that know how to execute SQL scripts. However, the important thing in this step is to validate if the script can be executed safely. By not controlling who can make changes in which environment and by not documenting the changes, scripts that were created few minutes ago may not be valid at the execution time.

The database code can be modified in all environments. This requires the database in all environments to be under version control, which enforces the documentation of changes. Otherwise, someone can make a change (not using the normal process and without documenting it) in the QA or any test environment. The application will pass the tests but when the scripts will be executed in the production (without the out-of-process change) the application will break.

Automated Database Changes Test

The last but most important step is to test the script execution. It can be done by checking the log file and searching for database errors, by running unit tests developed especially by developers/DBA, or by running the impact analysis again to get an empty script.

DBmaestro
DevOps for Database

Summary

Using file based source control for the database doesn't hold up due to errors in executing the scripts and because not everything is in the source control. If the scripts are generated incorrectly the end result can cause downtime to the organization .

Anyone can login to the database, introduce a change and forget to apply it in the relevant script of the file-based version control. We need to use a real database source control which will serve as the foundation for database continuous delivery.

When using the correct method, all steps of automated database Continuous Delivery are possible, from the build to deploy, to test.

To read more about Continuous Delivery for the Database, download our free eBook – In database automation we trust.

Headquarters 300 Baker Avenue, suite 300, Concord, MA 01742, USA +1 978.405.3368
EMEA Headquarters 21 Yagiya Kapaim st. Petach Tikva 4900101, Israel +972.3.9248558

Follow Us: