t +61 3 9600 2339
w readify.net

Managing database schemas in a Continuous Delivery world

January 2015

Executive Summary

One of the trickier technical problems to address when moving to a continuous delivery development model is managing database schema changes. Unlike software changes, it's hard to roll back or roll forward when database changes have been applied. Typically, organisations address this problem by having DBAs manually apply changes so they can manually correct any problems, but this has the downside of providing a bottleneck to deploying changes to production and also introduces human error as a factor.

A large part of continuous delivery involves the setup of a largely automated deployment pipeline that increases confidence and reduces risk by ensuring that software changes are deployed consistently to each environment (e.g. dev, test, prod).

To fit in with that model it's important to automate database changes so that they are applied automatically and consistently to each environment thus increasing the likelihood of problems being found early and reducing the risk of database changes.

This report outlines an approach to managing database schema changes that is compatible with a continuous delivery development model, the reasons why the approach is important regardless of development model and some of the considerations that need to be made when taking this approach.

The approaches discussed in this document aren't specific to continuous delivery and in fact should be considered regardless of your development model.

The problem with manual schema management

Managing database schemas to be in sync with software development is traditionally a complex task fraught with problems such as:

- Tedious manual processes
- Human error
    - Applying the wrong scripts
    - Applying scripts in the wrong order
    - Forgetting to apply scripts
    - Applying scripts multiple times
- Confusion over the current "version" of the database in particular with regards to different environments (e.g. dev vs test vs production)
- Inefficiencies for the development of software when using local databases due to developers being required to constantly modify their database after pulling in changes from other developers while the database schema is actively in development
    - Sometimes developers will instead use a shared development database to overcome this, but this is not recommended since there is additional pain associated with breaking changes and instability
- DBAs becoming a bottleneck for releasing changes to production (and worse pre-production) and reducing the agility of teams and increasing deployment risk (through batching of changes)

  ABN 13 097 323 781

Database Migrations

The recommended approach of this report to address the problems of manual schema management and that is compatible with a Continuous Delivery development model is to use the database migrations technique.

This technique involves **all schema changes** being applied as a "migration", which is a file that describes that schema change (possibly as SQL, possibly in a Domain Specific Language (DSL) via a programming language) and is committed in source control alongside the software that uses the version of the database that results from applying that migration. Migrations will typically have a timestamp or incrementing integer number associated with them to designate the order in which they should be applied.

These migrations can then be automatically applied to the database in any environment by a "migration tool/library". These tools will automatically determine which migrations that are defined have and haven't been applied to the current database and only apply the ones that haven't already been applied in the correct order. The migrations will typically be applied within a transaction so if the migrations fail then they will be rolled back.

The advantages of this approach are:

- Schema changes are made automatically without human intervention thus reducing the chance of human error.
- Schema changes are consistently applied in each environment, so by the time the migrations are applied to production they would have already been successfully applied to dev, test and any other pre-production environments. This **significantly** reduces the risk of schema changes
- Schema changes are kept in source control next to the source code that uses the schema that is defined at that version. This means that any version of the software can be checked out and run and, assuming a fresh database was also used (before any migrations, or at least before the database version of the code that was checked out), the state of the application at that time can be tested against – this can assist with finding bugs in specific deployments.
- Your database is now "versioned" (just like the software application) and you can inspect the version of your database in any environment and know what changes have and haven't been applied. This is invaluable when looking for environment-specific bugs.
- There is no confusion about what scripts do/don't need to be applied to a database at any time and scripts are never forgotten to be run – they always get automatically applied.
- Developers can easily use separate local databases since any database schema changes they pull in from other developers will automatically be applied.
- Any schema change conflicts will be exposed and thus resolved earlier (development time, or at least on the continuous integration server if they are run as a test), which significantly improves development efficiency and reduces risk.
- Schema changes will always be automatically applied in the correct order.

Database Migrations and Continuous Delivery

Database Migrations can be integrated to a Continuous Delivery pipeline in one of two ways:

- Automatically run on app start
    - Requires the application to have DDL permissions, which could be a security risk
    - Means it's possible to do a deployment that will be broken due to database migrations failing
    - Simpler to set up
- Run as a deployment step before the deployment of the application software
    - If the running of the database migrations fails, then fail the deployment

The other consideration is whether or not the database migrations will result in a schema that is incompatible with the application code. To avoid this problem, it's a good idea to run automated tests against the application code that talk to a migrated schema. This requires an isolated database on the continuous integration server that the tests can talk to, which can be achieved by:

- Run the database migrations once only for the whole test run and use a transaction around each test run that is then rolled back afterwards:
    - This can be done as a cross-cutting concern without modifying test or application code by using something like .NET's TransactionScope (in combination with the MSDTC service)
    - This can also be done by sharing a database connection with an open transaction between the different components (seeding, doing the work and verification)
    - If you are using an ORM then you may want to create new session/context instances for each of seeding/work/verification so that their internal cache doesn't interfere with the test run and just ensure they each share the same database connection
- Drop and recreate the entire database each run; possibly by using in-memory version of the database for speed
- Delete any data from the test run after/before each run
    - This can be done manually in each test (but it's likely this will get missed occasionally)
    - This can also be done using the technique described here http://lostechies.com/jimmybogard/2013/06/18/strategies-for-isolating-the-database-in-tests/ (but you need to be careful to not delete seed data)

Rollbacks

Depending on the migration tool being used, database migrations can include an up definition (apply the migration) and a down definition (rollback the migration). This then allows you to move a database back (or forward) to any specific version. On first thought, this can be used in combination with software rollbacks if an error is deployed to production. However, this is fraught with danger:

- If the corresponding migration that caused the break is modified then every single environment that migration had previously been applied to also needs to be rolled back (that also includes all developer machines with that migration applied).
- Down migrations often result in data loss (by necessity, if the up migration added a table the down migration has to drop it) and that might be undesirable.
- If the down migration fails as well then you are in trouble. Given down migrations are not often applied this is actually quite likely.
- It's complex to implement a proper rollback in most Continuous Delivery pipeline tooling because you need to rollback the database version (which has to be worked out somehow) in the code that was just deployed (since it will have the definition of the migrations that need to be deployed) and then rollback to the previous version of the software.
    - You may need to make this a manual process, or provide a tool of some sort to allow someone to go in and select a version to rollback to (and then manually trigger a deployment of the previous code version).
- Another way to address the problem is by dealing with breaking changes carefully (e.g. column renames, dropping tables, etc.) – if you can ensure that the schema changes are generally not breaking changes (or the breaking change is deferred for a deployment or two – e.g. have the old and new column side-by-side for a while) then you can safely perform rollbacks.
    - It's important to keep in mind that this also introduces some complexity around keeping track of those in-progress changes, what has been deployed where and managing technical debt (i.e. not leaving the in-progress changes behind).

To avoid a lot of these issues and take advantage of the fact that the Continuous Delivery pipeline allows the software team to quickly and safely get changes into production a strategy that is often used is to never roll-back (software or database changes), but to always roll-forward (i.e. fix the bug, put that change through the Continuous Delivery pipeline and apply to all environments).

Regardless of whether a rollback or roll-forward is used, it's important that once the issue is fixed the team stops work and determines how that bad change was able to get to production without being picked up first. Then as part of the continuous improvement focus the team should make a change to the Continuous Delivery pipeline to ensure that particular problem can never make its way to production again. This might involve adding better seed data to a pre-production environment, adding automated tests to pick up problems with the migrations, or something else.

Some migration tools don't bother with down migrations because of this complexity (DbUp is an example) and even if the tool you are using does support them you can choose not to implement them. Down migrations can be handy for debugging a specific version in a pre-production environment or useful in development when a developer is making changes to migrations they are developing. If you decide you need down migrations then the recommendation would be to add an automated test that takes a fully migrated database all the way back down and then back up – that gives some confidence that the down migrations are likely to work (but isn't perfect since sometimes production data violates integrity constraints).

Database Migrations are immutable

For the reasons described in the Rollbacks section above it's important that in a Continuous Delivery environment Database Migrations are considered immutable. This is because the master/trunk/mainline branch can be deployed at any moment and as soon as you change a migration those changes will not get applied in any environment that already has that migration version noted as being applied.

This ties into the granularity of migration changes:

- Continuous Delivery requires that every separate set of schema changes that each developer makes are considered a separate migration
- If a project involves a lot of database changes then this can get potentially unwieldy as the number of migrations grows and you may decide to allow migrations to be changed while they are isolated to development environments
    - This is directly incompatible with Continuous Delivery and can be hard to manage (i.e. ensuring that the migrations are "done" before moving past the development environments)
    - If migrations are allowed to be changed while in development then the developers need an easy to way to get their database to a clean slate and re-apply all migrations (e.g. a .bat file or console application or deleting the database versions and ensuring all migration scripts can be safely re-run)
    - This complicates the development process and can cause some merging pain on the migrations (if there is a lot of churn on specific migrations), but it also results in less migrations for database-heavy applications
- Sometimes it makes sense to try and roll-up migrations to a single file for a release at some defined point along the development cycle (e.g. each sprint), but this further complicates the release process (how do you correlate the rolled up file with the previously deployed migration numbers that are contained in it) and should be avoided unless absolutely necessary

Other considerations when using Database Migrations

There are a number of things to watch out for when using Database Migrations:

- While database migrations mean that a DBA isn't needed to manually apply the changes that doesn't mean there isn't a need for DBA skillsets
    - DBAs can still provide advice to development teams around the best way to model the schema for their application and can collaborate with the developers to create the migrations
    - DBAs can be on hand to help with any problems that do occur when a deployment is made – the need for this should reduce over time as more confidence is gained by the deployment pipeline, but if this concept is new then there is likely to be more involvement needed
    - DBAs can be kept up to date with schema changes to provide advice or raise any potential issues by sending an automated diff of the schema differences contained in the migrations for a particular deployment as part of the deployment pipeline (this is a relatively simple thing to write code for)
- Ideally, a new developer or a tester testing against an ad hoc software revision should be able to quickly and easily spin up a fresh copy of a database before any migrations have been applied
    - This is easy for an application that has used migrations from the start, but slightly more difficult for a legacy application to which migrations have been added
- Seed data should be also be automatically added; depending on the Migration Tool being used, this can be achieved by a set of SQL statements, or something more programmatic like loading a CSV of data
- Testing data is slightly more tricky since you want to apply it to some environments and not others
    - If you want to apply the testing data once only, on all environments apart from a set of specific environments (e.g. production) then it makes sense to apply the data in a migration that has guard clauses to prevent execution in those environments
    - If you want the ability to apply the testing data in a more ad hoc fashion then you should use some other means to apply the data – e.g. a custom console application, or a set of SQL scripts that can be manually applied
- Ideally, **all** schema changes should be applied using migrations otherwise the following disadvantages apply:
    - Not all schema changes are automatic leaving human error as a problem

- Not all schema changes are consistently applied in all environments leaving a risk that some of the manual changes or indeed the automatic changes might not work when applied to production
    - There is no clear way to tell the complete "version" of the database
    - There is confusion about which scripts have / haven't been run
    - Local development databases are trickier to facilitate
    - Not all schema change conflicts will be exposed early
    - Schema changes might not be applied in the correct order (e.g. if the automatic migrations rely on the schema changes from manual changes or vice versa)
- Stored procedures and functions are software rather than schema changes and often it is thus easier to manage them differently
    - One way to do this is to keep them in source control and automatically apply the latest version to the database every time the schema migrations have been applied
    - This ensures that the version in source control is always the version applied to the database
- Some ORMs provide the ability to automatically change the schema to match the object model defined in code
    - This is really handy at development time, particularly when spiking changes or working on a proof-of-concept application
    - This is dangerous and risky for a production application and should be avoided – it can result in unpredictable schemas, poor performance schemas and data loss
- If the database engine you are targeting doesn't allow transactions around DDL statements then applying migrations is more risky because a problem that occurs can leave the database in a semi-modified state – an example of this is Oracle database
- Techniques like blue-green deployments are tricky to perform unless there is a way to have a hot copy of the database that can also be switched at the same time (not easy to achieve)
- Sometimes, migrations can fail in production because they time out and are too slow to be applied (an example might be adding a new column with a default value to a table with millions of records)
    - Getting production-like data earlier in the pipeline is important in these scenarios to ensure that the problem is picked up earlier
    - Increasing the timeout of the migrations might be necessary, but if you do that it's important to ensure migrations are applied separately to the deployment of the application rather than alongside

- Separating out slow changes to be applied when there is lower load on the database may help, but it complicates the release process
- Identifying the migration modifications that are slow to run is important so they can be identified early and some sort of appropriate action taken (e.g. in the above example, making the new column nullable and adding the default value in the application code instead may be an option)

Database Migrations and .NET applications

There are three main migration tools that we use at Readify for applying migrations to .NET applications:

- DbUp – An Open Source library that runs SQL scripts (up migration only) against SQL Server (including SQL CE and Azure SQL Database). http://dbup.github.io/
- FluentMigrator – An Open Source library that includes a DSL for defining migrations (up and down) in code against most database engines. https://github.com/schambers/fluentmigrator/wiki
- EntityFramework Migrations – Part of the EntityFramework 6 library that includes a DSL for defining migrations (up and down) in code which can be automatically generated from your model against most database engines. http://msdn.microsoft.com/en-au/data/ef.aspx