Preview Edition

redgate

Database Lifecycle Management

Achieving Continuous Delivery for Databases

By Grant Fritchey & Matthew Skelton

Database Lifecycle Management

Achieving Continuous Delivery for Databases

By Matthew Skelton and Grant Fritchey

First published by Simple Talk Publishing, 2015

ISBN: 978-1-910035-09-2

Cover Image: James Billings
Typeset: Peter Woodhouse and Gower Associates

A WORD ON LEANPUB

We are developing this book using 'Agile' publishing; producing and then refining content in relatively short, iterative cycles. The outline that follows reflects our current thoughts on the topics and ideas that we need to cover, in order to illustrate how to apply Database Lifecycle Management techniques through all critical stages in the life of a database. However, this can and will change and evolve as we progress, and as we receive feedback from readers.

We'd love to hear your thoughts!

If you spot errors, issues, things you think we've missed, or an idea you think is worth exploring further, please do let us know, by contacting DLMBook@red-gate.com

TABLE OF CONTENTS

Sections listed in blue and bold are included in this special preview

Section 2 - Enabling DLM at the Enterprise Level

Section 3 - DLM for relational databases

Section 4 - DLM beyond Development

Section 5 - DLM and the Data Ecosystem

Introduction

In a world without database lifecycle management

It is 9am in the office of InfiniWidgets, Inc., a company that originally produced and shipped widgets for use in the food industry, and that now offers a SaaS-based manufacturing plant tracking system for its clients. The company is successful and has a large database with orders and inventory records dating back to its inception, 17 years ago. Its four software development teams work on various different aspects of the internal Order and Inventory (OI) system, as well as the customer-facing Factory Management (FM) system.

Jess, the lead software developer from the FM team, and acting Scrum Master, is frustrated. The database changes were supposed to go live today but the scripts are still under review by the database administration (DBA) team. She's had no update via email or the ticketing system to tell her when the changes will be ready. She informs the project manager of the delay, who fires off an impatient email to the DBA team.

Bal, in the DBA team, is multi-tasking as fast as he can. He's just finished some urgent schema updates that the OI team requested yesterday, and it took longer than he hoped, as the scripts were a bit of a mess. He's now working on a ticket from the FM team and they are not happy about the delay. His heart sinks when he sees how the numerous

requested index modifications and additions are likely to affect the performance of transactions on the FM system.

He fires off a mail to Jess explaining the problem and attaching the performance chart, wondering aloud why he never gets to see these proposed changes until the last minute. If he'd seen them earlier he could have warned them that this approach wouldn't work.

Jess views Bal's email and sighs in exasperation. "Why couldn't we have had these metrics before? We might have taken a different approach!" When Jess explains to her project manager that the DBA team have rejected the change, he loses his temper. "How many times do we have to put up with these DBA roadblocks?" He fires off an angry email to the Head of Delivery before marching up to Sam, the lead DBA, and insisting that the FM team's changes be deployed that day in order to meet their client commitment.

Under pressure from all sides, Sam feels there is no choice but to deploy the database changes from Jess's team, even though Bal tells him they are likely to cause problems. Sam instructs Bal to run the scripts after 6pm that day.

The next day, the problems begin. The DBA team begins to receive emails from the customer support team saying that five of their main six customers are having problems accessing their factory management portals. The problem escalates; queries are hanging due to severe blocking, users are seeing queries fail due to deadlocks. Eventually the whole site goes down.

A frustrating half-day and hundreds of emails later, the DBA team breaks the bad news that they need to restore the database from a nightly backup and the customer support team spend the next three days re-entering lost customer transactions.

If this tale, fictional as it is, strikes notes of familiarity with you, then you should find all or parts of this book helpful.

Why you need DLM

If you work for an organization that produces database-driven software systems, and the database needs frequent or periodic changes to features, functionality, or data, then this book is for you. It describes how to apply the business and technical approaches of **Database Lifecycle Management** (DLM) to make all database processes more **visible, predictable,** and **measurable,** with the objective of reducing costs and increasing quality.

Modern organizations collect large volumes of data, in a range of formats and from a range of sources. They require that data to be available in a form that allows the business to make fast decisions and achieve its strategic objectives. The DLM approach acknowledges the dual reality that databases are becoming increasingly complex and challenging to develop and maintain, in response to the business's demand for data, and that we must be able to adapt even complex databases in response to changing requirements.

Given these demands, traditional manual approaches to database changes become unsustainable and risky. Instead, we need to increase **automation** whilst retaining and using the **expertise** of people who understand our organization's data. We need team structures that encourage **communication**. We need a framework within which to make **simple, stepwise improvements** to our core database build, test, and delivery processes, and then to apply the **workflow** that stitches together these otherwise disparate processes into a coherent, automated pipeline to database delivery. We need to make databases easier to maintain, using **instrumentation**, and by considering their **support requirements** as part of the design process.

DLM will help you achieve these goals. The scope of DLM is, as the name suggests, the entire useful life of a database. DLM starts with aspects of design and data architecture, encompasses the database development and delivery processes, and extends to the support and maintenance of the database while it is in operation. It aims to ensure that an organization has in place the team, processes, methods, business systems, and tools that will allow it to design, develop, and then progressively refine even the most complex databases, with the minimum difficulty.

This book will explain how to use DLM to:

- Automate many of the critical processes involved in the design, development, delivery, and ongoing operation of a database
- Improve the organization's knowledge of the database and related applications
- Identify opportunities for optimization
- Encourage technical and organizational innovation
- Ensure that the database supports your key business goals, and drives strategic decision making, within the enterprise

The overall goal is to help you evolve databases by a process of continuous, incremental change, in direct response to the changing data requirements of the business, and to improve the delivery, operation, and overall quality of your database systems.

T-R-I-M database processes to reduce risk

Making changes to database-coupled systems can often be difficult enough within organizations, but when those changes need to be made more frequently, more rapidly, and more reliably traditional manual approaches to database changes become unsustainable and risky. Historically, organizations who rely on these manual approaches have delayed or avoided making changes to database-coupled systems due to the perceived risk and complexity.

However, organizations that lag behind in the technical practices that allow change will find themselves at a disadvantage. Their competitors innovate quicker. Funding is moved to organizations that can make more effective use of scarce resources. The speed of change in the technology sector leaves them at the mercy of legacy, unsupported technologies, costly outdated billing models, and even security vulnerabilities.

Throughout the book, we describe a range of critical database processes in the database lifecycle, from initial design, governance, development, testing, release, deployment and through to ongoing operations and maintenance. We identify typical problems associated with ad-hoc, manual approaches to each process, along with the associated business risks. We highlight common technical issues, bad practices and behaviors, and risks, depending on how the team tackles that process, and then describe how stepwise application of DLM methods help tame and then systematically improve that process, making it more Traceable, Repeatable, Improvable and Measurable (T-R-I-M):

- **T**raceable/Visible – processes are visible to all teams, and to the broader business, from very early in the project
- **R**epeatable – processes are automated, tested and therefore predictable and repeatable
- **I**ncrementally Improved – processes are connected by workflows that provide the necessary information to drive continuous short cycles of improvement
- **M**easurable – processes are instrumented and logged so that errors and deviation from predictable behavior are corrected quickly

We start from ad-hoc, manual approaches to a particular database process (such as testing) and move logically towards an approach that is automated, integrated, measured, and standards-compliant. This progression gradually reduces risk, by making the process more visible, predictable, and measurable. The approach is based loosely on the idea of a "Capability Maturity Model" (described in more detail in Chapter 1).

As a by-product of this approach, we arrive at a structured series of problems and associated solutions. For instance, in the chapter on "DLM for ETL systems," we see:

- ***Challenge:*** Constant changes to the ETL data format, by the data supplier, hampers the team's ability to improve the quality of the ETL processes

This challenge has several solutions, including:

- ***Solution:*** Insist on data format/schema standards for ETL

For a given skill or method, it means you can identify where you are now in terms of your current approach, and the associated risks, and then see what to aim for next in terms of process improvement. It's a structure that we hope also provides fast access to solutions to specific problems, without having to cover the preceding material in depth.

You may even find it beneficial to print out the Contents section of a chapter on large-format paper in your team area to help with tracking progress with DLM.

The book sections

The book splits down broadly into five sections.

Section 1 – DLM foundations

Section 1 describes exactly what we mean by *Database Lifecycle Management*.

We start by discussing the key technical, organizational and philosophical 'barriers' to making the database, relational or otherwise, a willing participant in practices such as continuous integration, test-driven development, and minimum viable product, which enable incremental change, and the continuous delivery of new functionality to the users.

Next, we define precisely what we mean by Database Lifecycle Management, and its benefits. We describe the key DLM skills, or working methods, which will support a continuous and incremental approach to database change, establish predictable delivery times and result in higher quality, more maintainable database applications. We also

describe, broadly, the data and system architecture, design approaches, team structures, and tools that can help with the adoption of DLM techniques.

We finish the section with a real life DLM case study, from Frazier Kendrick, called Stumbling towards Database Change Management. Frazier worked for 15 years as a DBA and IT Manager in the insurance and financial markets. He describes a world in which database change processes were chaotic at best, where IT improvements were funded only when there was little alternative, rather than as part of a longer-term strategic plan, and teams succeeded through periods of 'heroic endeavor', in response to very short-term priorities and targets. Nevertheless, despite this, he did witness a slow journey towards more managed and optimized systems, where database or application changes could be made relatively quickly, and with less disruption, lower risk and fewer surprises, and a semblance of proper IT governance was a reality, with the development team more generally accountable to the business as a whole, for the changes they wished to make.

Section 2 – Enabling DLM in the enterprise

Section 2 explains what sort of thinking and structures that must be present at the organizational level in order to enable effective DLM. Like any form of significant organizational change, DLM cannot be purely a grassroots, bottom-up process. You'll need support from whoever is the technical decision maker, you'll need to consider the team structures that will support DLM most effectively, the personalities within your team, and how your teams are rewarded and for what.

DLM means arriving at a set of practices that help us to manage the entire lifecycle of our databases, not just the development cycle. We need to think about maintenance and operational support, and the needs of other teams within the organization, such as those responsible for data analysis. We need to consider what an "operations first" approach really means to the design and development of our database applications. It means we need to look beyond the immediate world of SQL scripts and database backups to the approaches required to support data analysis/mining systems, to issues of data strategy and traceability for audit, and much more.

We close section 2 by reviewing, at a high level, some of the core techniques, such as use of version control and well-defined interfaces that decouple application and database, without which any DLM effort will flounder. .

Section 3 – DLM for the relational database

Section 3 explains in deeper, strategic detail how to apply DLM techniques to the design, development, deployment, and maintenance of the relational database which, for many decades, has been at the heart of an organization's data.

It will describe the database design, build, test, and deployment processes that we need to consider, in order to manage to the database schema, data, and metadata for a database, along with the DLM techniques and associated workflow that will allow the team to tackle each process in an automated, integrated, and measurable way.

Section 4 – DLM beyond development

Sections 4 looks beyond the foundational DLM practices that are essential to building a database development and delivery pipeline, to the broader needs of the organization. It will consider the fundamental requirements of access control and configuration management that are essential when deploying to production. It will discuss the need for sustainable automated testing and for traceability, reporting, documentation and so on. In particular, it will consider the topics of governance and compliance, and to establish accountability and visibility for any changes that affect the organization's data.

Section 5 – DLM and the data ecosystem

Section 5 considers the applicability of the practices and methods described in Section 2 to the broader "data ecosystem", to the ETL process that supply the data, BI systems that must analyze it, as well as to the design, development, delivery, and maintenance of non-relational databases and other supporting, non-transactional systems.

We describe relevant DLM techniques within organizations that have experienced a proliferation of different types of databases, such as graph-, column-, and document-oriented databases, alongside structured, relational data. This is often as a result of changes in development practices, such as adoption of application architectures based around containers and microservices, as well as changes in the type and volume of data that must be collected and analyzed.

Section 5 closes by asking the question "where next?" What new data challenges are on the horizon? What are the emerging new technologies will enable our data systems to cope with them?

Who should read this book?

We have tried to write a book that appeals to anyone with an interest in making databases changes smoother, faster, and more reliable. As far as possible, we avoid industry jargon in favor of simple, straightforward terms and descriptions.

This book offers practical but strategic-level advice on how to tackle the various aspects of database change automation, as illustrated by the Problem-Solution approach described previously. It does not contain hands-on examples, extensive code listings or technology-specific walkthroughs. We will not describe, for example, how to go about setting up a CI Server using a specific tool such as TeamCity, although we will present the various alternatives, pros and cons where possible and relevant, and resources for further details.

Database Lifecycle Management encompasses a broad set of practices, requires collaboration and coordination of activities across many different parts of the organization and therefore involves a broad cross-section of an organization's IT staff. As a result, this is a slightly uncommon book in that it is aimed at several different readers, and should probably be read slightly different depending on who you are and what you need.

When writing this book we have kept in mind three broad areas of responsibility, and typical associated job roles:

	**Governance**	CTO Head of Program Management Head of IT
	**Operations/Administration**	Database Administrators Systems Administrators Release Managers
	**Software Development**	Database and Application Developers Testers Project Managers

Throughout the text we use these icons to indicate content of particular relevance to each broad category of job group. However, we expect that people from any of these roles will benefit from reading the entire book because each group understanding the needs of other groups will be beneficial for all.

We have used the established technique of developing **user personas** in order to validate the material and focus of the different parts of the book. Appendix A provides a series of full-page Personas, describing our understanding of the knowledge, motivation, and goals for each of the main job roles.

You may find that the personas help to distinguish better the needs of different groups within your organization, and so adapt your working practices to better meet their needs.

Feedback

We would love your feedback on the book, particularly in its early stages as a 'lean' publication. A core tenet of our approach to DLM is early and rapid feedback on changes, and this is the approach we have adopted when writing the book. All feedback, positive and negative, will help us make the book as relevant and useful as possible.

As chapters become available, we will post them to:

www.leanpub.com/database-lifecycle-management

We will then progressively refine the content based on your feedback, which you can submit using the comment form at the above URL or by sending an email to dlmbook@red-gate.com

Look forward to hearing from you!

Section 1
DLM Foundations

Covering:

1. Databases and change
2. How DLM works
3. Core DLM practices
4. **Stumbling towards database change management: a case study (included in this preview)**

4: STUMBLING TOWARDS DATABASE CHANGE MANAGEMENT: A CASE STUDY

By Frazier Kendrick

I worked for 15 years as a DBA and IT Manager in the insurance and financial markets. The scale of change in the insurance and financial markets is such that there is little time for the application or database developer to sit back and work out ways of improving the delivery process. Over time, however, it is possible to improve the process so that individual heroics are required less and less as release and deployment become more managed and predictable. It can be messy and error-prone at times but the long-term benefits make the struggle worthwhile.

When I first started out, 'chaotic' would be the best word to describe our approach to the design, development and ongoing maintenance of our business applications and data. I remember vividly the "Wild West" days when, with only a database backup as a fallback policy, I'd make ad-hoc database changes directly to production with little to no testing. I learned the hard way that processes that can survive for short periods, with a slice of luck and when working on relatively small systems and in small teams, become untenable as the team, systems, and business ambitions grow.

It would be nice to be able to say that as the companies I worked for grew and matured, so I witnessed a smooth technical and cultural change that allowed us to continually refine and optimize our IT systems in a planned and structured way. In truth, it was a lot messier than that. Some of the sensible models that exist for continual process improvement, such as that enshrined in the CMMI Maturity Model from Carnegie Mellon University, are barely on the radar of many organizations. My experience in the industry

was characterized by periods of 'heroic endeavor', in response to very short-term priorities and targets. IT improvements were funded only when there was little alternative, rather than as part of a longer-term strategic plan.

Nevertheless, in fits and starts, we did move gradually and sometimes painfully towards more managed and optimized systems, where database or application changes could be made relatively quickly, and with less disruption, lower risk, and fewer surprises. By the time I left the finance industry, our testing was automated, deployments were "1-click", and a semblance of proper IT governance was a reality, with the development team more generally accountable to the business as a whole, for the changes they wished to make.

"I've put all the files in this folder"

In my early days as a database developer-cum-reluctant DBA (early 2000), I was working in financial services, developing and managing some in-house trading software for a small hedge fund.

The trading system had an Access front end and a SQL Server database. Change management was rudimentary. There was no version control of any kind. A "release" entailed shouting to the other developer, who was really a trader, to inform him that *"I've put all the files in this folder."* All changes were made live to the production system, after a brief sanity check. It was quite a small operation, but the trading system was still dealing with a few $100 million of people's money.

Later, this hedge fund management firm merged into another larger hedge fund that had far greater assets under management (about $5 billion) but if anything an even more chaotic approach to change management.

Their key business functions were defined in Excel files. The system used SendKeys macros to send instructions to a DOS-based investments management system. We produced basic reports using some Java components to run queries against this DOS database. Nothing was source controlled. All changes were direct to production. There was no formal issue tracking beyond erratic entries into a central spreadsheet.

Every attempt to introduce a change to a report would involve hacking the Java code, and then running it in production to see what the new report looked like, hopefully while nobody else was trying to look at it. The

Java code was badly written and it was very easy to make a mistake and blow everything up, at which point, I simply had to revert to my original file system copy and try again.

My first task was to migrate the database to SQL Server and replace the unreliable SendKeys macros with a COM interface. Along the way, I managed to get the Java code into source control, well **Visual SourceSafe** (VSS) at least. The database code was less of a priority at that point, since making any schema changes wasn't a realistic prospect.

Eventually, I built a dedicated data mart and moved some of the main business functions out of Excel. I had devised a new schema for the data mart, and began making more database changes, so I started scripting out the database schema each evening and saving it to VSS. I also managed to acquire a second server. It wasn't called anything as grand as "staging", but it was at least a place where I could break things with impunity before pushing changes to production. I started using a schema comparison tool (SQL Compare) to push changes between the two environments.

Rapid business expansion, no IT planning

At this time, the firm had some success in winning new business and shifted from being a specialist hedge fund into supporting a broader range of product types. It was a period of rapid business expansion, achieved without giving a great deal of strategic thought to the support functions, including IT.

The business wanted a new trading platform that would drive and support their expanding business interests. They wanted new functionality, and they wanted it delivered quickly. The IT team grew, and we were expected to undertake much more ambitious projects.

Our rudimentary and manual change management processes, which showed evidence of strain even when the development team was relatively small, began to crack. There was no automated workflow around any of our processes. It all relied on people firstly remembering to check their changes into source control, and then getting my "okay" before anything went into production. Inevitably, as the team grew, along with the pressure to deliver, mistakes crept in.

We simply had to make some overdue improvements to our processes. We acquired a third environment, so we now had Development, UAT/QA (depending on what was required at that time), and production. We moved from VSS to Subversion, which was a big leap forward in terms of the ease with which we could make concurrent changes. We also started using SQL Source Control to commit database changes directly from SSMS, as well as CruiseControl as a build server and JetBrains' YouTrack for issue tracking, in place of Excel.

Thanks to these improvements in our development and deployment processes, we managed to deliver some decent bits of functionality for the business, but progress was often frustratingly slow, because we were working against deeper problems.

Deeper data architecture problems

The right tools will help make your application and database change management process more reliable and predictable, but it's still fundamentally hard to deliver changes quickly and efficiently, if the underlying architecture isn't well-designed, flexible or adaptable. These inadequacies in our data architecture, and a lack of time for any forward planning caused us a lot of pain.

In terms of IT budgets, the focus always tends to be on short-term tactical objectives. The business wants a new piece of functionality to support a current objective, and needs it as quickly as possible. This meant there was never time to tackle fundamental problems with the underlying data architecture. There was resistance to the idea that the data architecture was part of the "software stack". Combine this attitude with artificial deadlines (*"I want this by Tuesday"*) and it meant that tactical decision-making was the norm. Over time, this actively subverts good architecture.

Each new application tended to be a 'stovepipe' or 'silo', with its own database and its own definition of a data model. It was hard to adapt the schema for these applications to introduce new features, and even harder to integrate the data from the various silos when we needed a more unified view of the business data.

The core problem was that there was no unified understanding of an underlying data model for the business. In the data model, we had to represent important business entities such as "asset classes" and "risk

categories". If there's no central definition of these entities, what they mean, and how they are defined then you simply can't correlate data from each silo.

At one point we embarked on a project to upgrade our compliance engine, which would test proposed trades against a relatively simple set of rules (*"The exposure of this fund in US dollars cannot exceed 80% of its total value"*). Each customer would have slightly different rules.

The compliance engine had to integrate data from numerous "stovepipe" databases, perform the test and feed the result into the central trading system. We were having to manipulate our schemas a fair bit in order to support these new applications, but without any lead time or forward planning, we ended up creating a lot of cross database dependencies, linked server dependencies, which would make it truly painful to move changes between environments.

We took the pain, and stayed late every night for months, and when things went wrong we fought the fires. We took that pain over and over again for a long time. We knew what the pain was, we knew what the solution was, but we weren't empowered to step back. We needed to overhaul the underlying data model; we needed to introduce an abstraction over the database, but senior management felt that the right way to run a project was to lean on everybody aggressively to "get it over the line", with a promise for time to look into the "other stuff" later. Of course, that time was rarely found.

Lack of testing

Over this period, a lack of automated testing during development became a major problem. When failures happened, the first sight of the problem would be through the front-end, and if it didn't report it very clearly, then you really just had to go through all the connection strings, all the permission sets, and logs, and try to track down the issue.

I recall one project where we requested a more test-driven approach to the development. The outside consultancy presented two prices to the CEO, one with testing and one without. The CEO chose the one without. It seems surprising now, but it wasn't at the time.

We're now well-versed in the case for test driven development but ten years ago it was not the norm. Testing was still someone firing up the application and clicking. That's what people understood it to mean.

The management perspective was to try to hire decent developers who didn't make that many mistakes. It's a different mind-set. If you think application faults are blunders by people who should be better at writing software, then the answer is shout at them more, shout louder, lean on them, fire them if you have to, hire in some new ones then shout at them to make it clear that they're not allowed to make mistakes.

Pain pushing changes between environments

Connection and permission-related failures were a constant pain when promoting changes from one environment to another. We had segregated permission sets for our three environments, but moving anything from one environment to another was still manual and painful. I lost count of the number of time a stored procedure had been introduced, but the associated role hadn't been given the correct permissions, and so it would fail.

By necessity, I spent quite a lot of time smoothing out the process of promoting changes up through the environments. I built up a collection of scripts that I could rerun relatively quickly, but a lot of the stages were still manual and it burnt a lot of my time as the reluctant DBA.

Pushing changes down the environments was even more tedious and error prone. Developers would request a copy of production or a copy of the current UAT for their environments, and I'd stumble constantly into permissions issues, environmental inconsistencies (different numbers of servers, different linked server definitions), and other ad-hoc complications related to a specific request, such as to introduce subsets of changes from UAT that hadn't yet been deployed to production.

Another difficulty we had was imposing the discipline of not making direct changes to the Test/QA environment. We wanted testers to feed bugs back down to development, so they could be fixed, and then the new code tested and promoted back up to UAT/QA from source control. However, it didn't often happen this way. Our change management processes weren't as slick or automated as we needed them to be; they

took time, and the developers were always busy. As a result, testers would demand privileged access to UAT/QA in order to make changes and see whether the change that they thought would make their test pass really did make their test pass. Of course, there was always a risk that without careful oversight, which of course eats up time, direct changes to these environments would not make it back into the "canonical source" in source control.

Data architecture redesign

Eventually, we embarked on a project that more or less forced us to rethink our application and data architecture.

The organization wanted to automate the main middle office operations and functions, and for Foreign Exchange trading one of the main functions is to rebalance. By the nature of the Foreign Exchange, the value of a client's assets in each given currency changes all the time, due to market movement. Rebalancing meant to bring down or up the exposure to market risk, in a particular area. Exposure can be very volatile, so we needed to be able to rebalance quickly, very often between very large foreign exchange trades into the market. At a month's end there would be hundreds of rebalancing trades to be implemented at exactly 4:00 PM on the last day of the month. It was very stressful, and these rebalancing calculations were being done on spreadsheets!

We hired some consultants to help build the main components of that application and at around the same time we also had some consultants in to review our IT infrastructure and processes, of the root cause some of the recurring problems. They came back with a lot of recommendations. Thankfully, one of those recommendations was to implement a new data architecture.

An experienced data architect came in to help us design and build a data architecture that could service the business properly. It was a ground-up, analytical approach. He interviewed users and IT, enumerated systems and business functions, came up with a logical data model for the business, which we then reviewed and tested against real business functions, before coding it into a schema.

With a unified data model in place, we were able to introduce new business functionality at a much-improved rate, adopting a service-oriented approach, based on Windows Presentation Foundation (WPF) with a WCF data service, for the rebalancing project, and all future projects.

With the help of a consultant development team, we also managed for the first time to introduce a proper package of automated tests, running on TeamCity. We deployed from TeamCity into each environment. It meant that the TeamCity projects were relatively complex, with a lot of environment variables determining what was going to happen after a build. However, with the new services architecture, it meant that application developers could build a WCF services and WPF application, hit a button, and run the test suite. If it passed all the tests it was pushed into the test environment, and from there into the UAT environment.

On the downside, despite sorting out the data model, the database was still out of sync with all these positive testing and deployment changes, and the new smooth application processes exposed how relatively backward the database processes were. Database deployments or upgrades were still a relatively manual process, driven by SQL Compare, and we still suffered from a lot of our old problems.

ORMs and the "stateful versus migrations" debate

We gradually built up a more complete layer of reusable services for all of our applications. In many cases, a service required a dedicated database that would store only the data required to support that particular business function.

The application team wanted to adopt a "migrations" approach to managing database changes in sync with application and service changes, automating database migrations through their ORM (Object-Relational Mapping) tool, which was NHibernate.

In this approach, the application team developed the data model in the ORM and a member of the DBA team would then vet the schema to make sure it was consistent in its basic definitions with our underlying data model for the business as a whole.

Once the data was in its final state, the application or service was required to feed completed records into our central, relational data store, where we had in place auditing and constraints to ensure all the data conformed to the rules for that particular business function.

During development, as the team committed each database modification, the ORM generated a migration script describing that change, which was stored in the VCS. The ORM would automatically run the required scripts to generate a new build, or to advance a database from one defined state to another (e.g. build 11 to build 14).

This worked reasonably well in those cases where there was a clean 1:1 relationship between application and database, and they could change in lockstep. Unfortunately, the application developers were convinced this was the only way ever to do any kind of change control on databases and wanted to apply it everywhere.

However, we also had databases that were shared between multiple applications, and had different reporting responsibilities. We argued that they could not possibly be under the jurisdiction of one particular service or application, and so began an awful lot of bun fights about ORM versus non-ORM, and use of migrations versus simply storing in the VCS the current state of each database object. In the end, we settled on a two-solution approach where application specific databases were built using ORM and managed entirely under that solution, but domain databases, those that supported multiple applications, were treated differently.

A better development model

When dealing with domain databases, multiple projects would request changes to that database. Initially, we had only a single shared database development environment for each domain database and we encountered issues around not being able to freeze an environment without impacting other projects, and not being able to progress an environment, to support a request from one project, without impacting another.

To address these issues, we switched to a "swim lane" model of development with a separate database development server for each project team. We had four development environments, DevA, DevB, DevC, and DevIntegration. In A, B, and C, developers could work on features specific to their project. This gave us the ability to freeze and unfreeze and progress our environments separately.

We tried to avoid branching as far as possible, because of the pain of merging, especially for the databases. Occasionally, we needed to implement a feature that cut across project work, and touched multiple services and apps, forcing us to branch so as not to lose in-flight work. Other than that, we would simply analyze the current set of work and the likelihood of one project's changes "breaking" another project, when we pushed the changes to DevIntegration. Generally, we found the answer was no, and we could take steps to mitigate any likely problems we did foresee.

Each team committed daily during development and we would build that against a copy of production for all the databases and run tests against that, and dev integration every night.

Automated database deployment

By this stage we had four separate development environments, plus test, staging, UAT, and production. We also had seven or eight services, three applications and still had half a dozen databases.

I had a central spreadsheet where I tried to keep track of which service, application, or database was in any given environment at any time, and when a new version had been introduced, but it was becoming time-consuming and error prone. Also, we were still suffering from a lot of the pain described earlier, when trying to promote changes between environments.

This situation improved dramatically once we'd adopted a proper release management tool (Red Gate Deployment Manager, now retired and replaced by Octopus Deploy, with SQL Release) that would help us manage deployments for both the application and the database. Through a single interface, we now had visibility into exactly what had been deployed where and when, and eventually arrived at single-click deployments between development and test environments, and eventually all the way up to production.

We resolved the age-old problem with permission failures, when moving changes from one environment to the next, by separating out the permissions for each environment into a set of script and environment variables. We used a Visual Studio (VS) solution for this such that, for example, on a deployment to Test or UAT, we'd tear down the whole permission set for a database and apply it afresh from the current VS solution, stored in source control.

After a lot of work, I was able to fit multiple development styles (migrations and state-based, for example) into that single deployment interface and we were in a much happier place. We also adopted a scheme where only privileged users could deploy to specific environments. So, for example, in order to deploy to production, you had to login as a special user, whose password was only known to two people, and all the other environments would be greyed out. We would perform these production deployments in front of the people responsible for signing off that change.

Compliance

When I first started out, there was no governance, no real business oversight of what IT did or how. Over time, however, this began to change, especially in the period following the consultant review.

In practice, this meant that there was now a "gated" approval process for each development effort. A candidate piece of work would require a business case and a change control committee would need to approve the project into the current workload.

During development and test, we'd have a small "release gate", for release of a finished feature to UAT. We went through many iterations of the release gate criteria before we arrived at something that was useful to all parties, without causing too much delay or drag. Essentially, it was a wiki page, which we created from a template and which answered questions such as: What's changing and why? What's the risk? Who's affected? How are you communicating with them? How are you performing testing in UAT? What is the worst case scenario? How will you roll back, if necessary? This review process had to involve non-technical people, so for UAT that might just be business analysts and the head of operations.

For a release to production, the gate was higher and the review process involved a compliance officer, the CEO, anyone who you could get into a room, but the more senior the better. For each significant change, such as a software upgrade, or database schema changes to support new functionality, and possibly data changes as well, we'd present along with the wiki page various 'artefacts' to help describe what was changing and why and how we'd tested it. This might include test coverage reports from TeamCity or the SQL comparison report from SQL Compare, and so on.

As a technical manager, it's easy for me to review the tests, the code, and the schema and data changes and say "this is all correct", but I'm only signing off at a certain level. The compliance officer is responsible for making sure we're observing the regulatory rules, as well as the client specific rules that software was designed to implement. They're finally responsible, not the technical team. The technical team would have to do a lot of education of the non-technical team in order to get them to understand what they were approving.

The increased visibility we had provided into our deployment process helped with this. For example the compliance officer could see exactly what had been deployed to UAT and for how long and this gave greater confidence that the change had been tested thoroughly.

We also spent a long time developing a way to communicate simply what was often complex new functionality, such as to that to perform rebalance calculations. This was as much a problem of describing the domain language as the actual software change.

Rather than just describe a change to the "Rebalance" software, or why we needed to introduce a new abstraction to the logical model and therefore a new object to the schema, we had to explain in business terms why these existed and how we intended to drive the required behavior through the software.

Software development for a specific business domain often requires evolving and reinforcing a domain-specific language, so we had to introduce and communicate changes to the business users as we developed the systems.

If all of this hadn't been documented and explained well ahead of time, then the final sign off would be quite difficult.

Summary

When I started out with databases, neither I nor the company I worked for had an idea of the extent of the techniques and processes that contributed to an ordered and repeatable management of database development. In any organization, it takes time and the necessary components have to be introduced step-wise. There is no single product or other silver bullet that allows databases to be developed quickly, and for

features to be added predictably in line with the requirements of the business. Database Lifecycle Management (DLM) requires cultural changes, the learning of techniques, the introduction and integration of a range of tools and a different way of working with the business. For a business that depends for its existence on rapid, robust, and scalable system in a fast-changing industry that is subject to regulatory supervision, the benefits are survival and a good chance of growth.

Section 3

DLM for the relational database

Covering:

9. Version Control
10. **DLM for database builds and migrations (included in this preview)**
11. CI for databases
12. Testing databases
13. Release and deployment
14. Monitoring and feedback
15. Issue tracking

10: DLM FOR DATABASE BUILDS AND MIGRATIONS

In many organizations, the database build or migration process is infrequent, painful, and manual. At its most chaotic, it is characterized by a 'trial-and-error' approach, running successive sets of database change scripts that make, undo, and then redo changes until finally stumbling upon a working database state.

An unreliable database build or migration process will lead inevitably to delayed and unreliable database deployments. Firstly, this means that the database deployment process becomes a bottleneck for all other deployment processes. Each upgrade of an application must wait while the manual steps required for the database deployment are navigated. Secondly, because these processes are manual, they are much more likely to be error prone. This means that the application deployment is waiting on a database deployment that may not even be correct when it eventually gets finished.

Many organizations have been living with this sort of pain for a long time. This chapter will focus on the first step to removing this pain: understanding what comprises a database build, and what comprises a migration, and then how to apply DLM techniques to make these processes automated, predictable, and measurable. Subsequent chapters will tackle testing and continuous integration, and then guiding the build/migration artefact through the release and deployment processes.

Builds, migrations, upgrades, releases, and deployments

Quite frequently, terms such as build, migration, upgrade release, and deployment tend to be used rather interchangeably. It leads inevitably to confusion. You may find subtly or even wildly different definitions elsewhere, but this is how we define the terms in this chapter, and throughout the book.

- **A Database build** – the process of creating a version of a database from its constituent DDL creation scripts. It will create a new database, create its objects, and load any static data. The build process will fail if the database already exists in the target environment.
- **A Database migration** – the process of altering the metadata of a database, as defined by its constituent DDL creation scripts, from one version to another, whilst preserving the data held within it.
    - **Database migration using an automatically-generated differential script** – The source scripts store the current state of each object in the database. Rather than executing the source scripts to build the database, a differential script is created dynamically, and subsequently edited where necessary, to effect the change between the two versions of the database.
    - **Database migration using immutable change scripts** – deploying the same set of immutable scripts in a pre-defined order, starting at the current version of the database.
- **Database upgrade** – a Database migration to a higher version.
- **Release** – making available outside the development environment all the scripts and supporting files necessary to create a version of the database that will support the new version of the software. At a minimum the release will contain the version number of the release, and the scripts from source control required to build the version being released. If an older version of the database exists, it will also contain the necessary

migration scripts. It is likely to also include documentation, deployment packages, and configuration files.

- **Deployment** – the process of taking a database release through the Testing/Compliance/Staging/Production phases, which will include making all necessary changes to supporting systems and services.

A build, according to the strict definition provided above, is a development-centric technique. Builds will be performed many times during the development process. In other environments, such as Production or Staging or User Acceptance Testing (UAT), we sometimes use the term 'the build' more generically to mean "the artefact that will establish the required version of the database in a given environment". In almost all cases, except for the very first, the build will be accomplished via a migration, in order to preserve existing data.

What is a Database Build?

A 'build' is a term used in application development for the process of creating a working application from its constituent parts, including compilation, linking, and packaging in the correct order. It will do everything required to create or update the workspace for the application to run in, and manage all baselining and reporting about the build.

Similarly, the essential purpose of a database build, according to our strict definition of the term, is to prove that what you have in the version control system – the canonical source – can successfully build a database from scratch. The build process, fundamentally, is what tells the development team whether or not it's possible to create a working application from the latest version of the committed code, in the source control repository.

During development and testing, builds need to be performed regularly, and therefore need to be automated and instrumented. If a build breaks, or runs abnormally long, then the team need access to detailed diagnostic information and error descriptions that will allow them to identify the exact breaking change and fix it, quickly.

Since we're creating an entirely new database from scratch, a database build will not attempt to retain the existing database or its data. Many build processes will fail, by design, if a database of the same name already exists in the target environment (rather than risk an `IF EXISTS…DROP DATABASE` command running on the wrong server!).

Microsoft database build definitions

Microsoft refers to a build as the mechanism that creates the necessary objects for a database deployment. A build, per their definition, is just the act of creating a script or a DACPAC file. They then refer to the act of applying that script to the database as publishing the build.

A significant build is the one that produces the release candidate, and a deployment involves the same basic build process, but with extra stages, for example to account for the production server settings, and to integrate the production security, and incorporate production architecture that isn't replicated in development (such as SQL Server replication).

Some databases have a more complex release process that requires a separate build for a number of different variants of the database, for a range of customers.

What's in a build?

Firstly, the build will create the database, including all relevant **database objects**, such as tables, stored procedures, functions, database users, and so on. To build a functioning database from scratch, you need to take the current version of the DDL scripts from the Version Control System (VCS) and execute them on the target server in the right order. Firstly, you run a script to create the database itself, then use that database as the context

for creating the individual schemas and the objects that will reside in them, all in the correct dependency order.

A database build may also include a data load. The data could be lookup (or static) data used by the application. It may even include operational data loaded through an automated process.

For a database to work properly, when it comes time to deploy to production, or to production-like environments, the canonical source, in the VCS, will also need to include components that are executed on the server, rather than within the database. These server objects include scheduled jobs, alerts, and server settings. You'll also need scripts to define the interface linking the access-control system of the database (database users and roles) to the server-defined logins, groups, and users so that the appropriate users can access the roles defined within the database. We'll cover server-level objects in a little more detail later in the chapter.

Therefore, if a build succeeds and is validated, it means that all the DDL source code, assemblies, linked servers, interfaces, ETL tasks, scheduled jobs, and other components that are involved in a database have been identified and used in the right order. This makes it more difficult for a subsequent deployment to fail due to a missing component, and all of the people involved in the database lifecycle can see, at any stage, what components are being used.

Pre- and post-build processes

A build process will generally include pre-build and post-build processes. Various automated or semi-automated processes are often included in the build but are not logically part of the build, and so are slotted into one or other of two phases of the build called the 'pre-build process' and the 'post-build process'. A build system will have slots before and after the build to accommodate these.

An important pre-build process, for example, is 'preparing the workspace', which means ensuring that all the necessary requirements for the build are in place. What this entails is very dependent on the nature of the database being built, but might include preparing a Virtual Machine with SQL Server installed on it at the right version, or checking to ensure that the platform is configured properly, and there is sufficient disk space.

Typical post-build processes will include those designed to manage team-based workflow or reporting. For example, we need to log the time of the build, the version being built and the success of the build, along with warning messages. It might involve email alerts and could even list what has changed in the build since the last successful build.

Also, of course, there will need to be a post-build step to validate the build. Specially-devised tests will not only check that the build was successful but that no major part of the system is entirely broken.

Generating the build

The build process can get complicated, and error prone, very quickly if you're putting the scripts together by hand. You may have each script stored and then a mechanism, frequently called a manifest, for calling them in the correct order.

However, there are tools that can automate generation of the build script. Both the DACPAC (SQL Server) and SQL Compare (Oracle, SQL Compare and MySQL) can read a number of scripts and combine them together in the right order to create a 'synchronization' script that can be executed to publish a new database. In effect, a synchronization script that publishes against an entirely blank database, such as `MODEL`, is a build script.

Of course, we can use essentially the same technique to create a synchronization script that we can execute to publish a different version of the same database, i.e. to perform a migration.

What is a database migration?

To build a database from scratch, we `CREATE` all database objects, in dependency order. In other words, we drop the unit of work and re-create it. However, when deploying database changes to production, we must of course take steps to ensure that the changes preserve all existing data. In production, we will generally only perform a database build once, and then all subsequent changes will be applied using a database migration. In other words, we use scripts that will `ALTER` changed objects, while preserving existing data, and `CREATE` new ones, thereby **migrating** the database from one version to another. Likewise, in production-like environments, such as UAT and Staging, it makes little sense to tear down and rebuild a large enterprise database, and then load all data, each time we need to make a minor database change.

In short, a database migration will change a database from one version to the version you require while preserving existing data. The migration process will use as a starting point the build scripts for the individual database objects held in source control.

In this book, we try to make a definite distinction between a build and a migration, but there are inevitably grey areas. For example, Microsoft's DLM solution assumes that every database 'publication' is actually a migration, using a generated script, even if it is a fresh build from scratch. In other words, the database build step is followed by a 'publication' step, which in effect migrates an empty database to the required version.

With any code objects in the database (views, stored procedures, functions, and so on), we can make any change simply by deleting the current object and re-creating the changed object. However, with this approach we need to reset permissions on the object each time. A better approach is to `CREATE` the object only if it doesn't exist, and `ALTER` it otherwise, so that existing permissions on that object are retained (although under the covers, the object is still dropped and recreated).

In the case of changes to tables, child objects such as columns and indexes can be changed without requiring the table to be dropped. Of course, all such changes must be performed in such a way as to preserve any existing data. As described previously, in such cases we

can perform a migration in one of two ways. In either approach, if the migration scripts are treated as unchanging, or immutable, once tested, then it means we only need to sort out a data migration problem once.

State-based approach

The state-based approach is to store in the version control system (VCS) the current state of the database. For every object, we simply store a single `CREATE` script, which we successively modify. We then dynamically generate a differential script to bring any target database to the same state. As with any script that performs a migration, it is good practice to retain each differential script, if it is successful. Once it has been successfully used it should never ever be changed without a great deal of caution and testing.

One problem with automatically-generated differential scripts is that it is not possible to automatically generate the differential script for some table migrations, if existing data has to be preserved. However, if the build tool that you use is canny enough, it can use an existing immutable migration script to 'navigate around' a difficult table-refactoring. Another problem is that comments in and around tables get stripped out of any reverse-engineered script.

Evolutionary change scripts

In this approach, we simply apply to the target database a set of immutable change scripts, in a pre-defined order, which will apply all necessary changes. A new version of a database can be created simply by running these migration scripts to evolve the database from version to version. This potentially makes any build from scratch a lot easier. Like many good IT practices, though, the evolutionary approach can quickly be taken to extreme. Many developers argue that it is unnecessary to hold object-level build scripts in version control at all, beyond the initial `CREATE` scripts for each object. The only requirement is a migration script that takes you between consecutive versions. If these

are executed sequentially in order, the inevitable result is a successful build as long as they remain immutable.

Of course, there are downsides to this approach. Firstly, if every script in the VCS is immutable then the VCS becomes essentially redundant, in terms of its versioning capabilities. Secondly, it means the database is built by a process akin to the theory of 'recapitulation', doomed to repeat past mistakes. Thirdly, it is much harder to determine the current state of an object, and the database as a whole, from a long string of change scripts than it is when each object is defined in the VCS by a single build script.

Finally, as the database evolves through a number of versions, a migration can entail running a very large number of change scripts. Some teams consider performing a 'rebase' at certain versions. For example, if we perform a rebase at version 3, we transform the string of existing change scripts into a single build script for that version. Then, if we are building version 4 of the database, then we can run the build script from new to 3, and then merely create change scripts to get from 3 to 4.

The evolutionary approach suits a development practice where there is one or very few developers working on the database or schema, but it breaks down in a team-based development. For a DLM-based build it is suspect not only because it doesn't scale, but also because it prevents others seeing individual object scripts unless these are also, redundantly, stored in the version control system. These problems can be circumvented with the necessary specialist tooling, but is otherwise impractical when taken to extreme.

Typical Problems with manual database builds and migrations

most of the problems I see with the a team's database build and migration processes stem from the fact that they do not rigorously enforce the discipline of always starting from a known version in source control, and regularly testing to prove that they can establish the required version of the database, either via a build-from-scratch, or a migration.

A fairly typical approach goes something like as follows. The development team make changes to a shared database, either directly or with the help of a database administrator (DBA) or a database developer. When it comes time to release a new version of the application, the development team works with the DBA to come up with a list of just the database changes that are needed to support the new version. They generate a set of scripts, run them against the database in the test environment (although, in the worst cases, this is done directly against a production environment), and then build the application.

Testing reveals that some of the necessary database changes aren't there, so additional scripts are generated and run. Next, they discover that some scripts were run that implement database changes that the application is not yet ready to support. They generate another set of scripts to roll back the unwanted changes.

This chaotic process continues until, more by perseverance than planning, the application works. The team now have a whole new set of scripts that have been created, manually, and that will have to be run in the correct order in order to deploy to the next environment.

This causes all sorts of problems. Development and testing are slowed down because of all the additional time needed to identify the changes needed. The manual process of generating scripts is time-consuming and very error prone. It leads to delayed functionality, and often to bugs and performance issues. In the worst cases, it can lead to errors being introduced into the data, or even loss of data.

To avoid this sort of pain, database builds and migrations should be automated, controlled, repeatable, measurable, and visible.

What is different about a DLM build or migration?

A build-from-scratch is a regular health-check for the database. If all is well, then automated builds will run regularly and smoothly. Assuming all necessary database and server-level objects have been accounted for in the build, the team will be confident that little can go awry as it is shepherded through the release process to deployment. Likewise, a reliable, automated, and tested migration process will very significantly reduce the likelihood of introducing data integrity issues into the production database.

Of course, if the database builds and migrations break frequently, then all bets are off. A serious problem with a database deployment will knock the confidence of the affected organization, perhaps to the point that they request that the team perform less frequent releases, until the database build issues are resolved. Unreliable database build and migration processes lead to slow delivery of the functionality needed by the business. Anything that increases the number of errors can also directly impact the business with bad or missing data, or in extreme cases, even data loss. Getting your database deployments under control within the development environment is an extremely important aspect of Database Lifecycle Management.

Throughout the remaining sub-sections, and mainly to avoid the tedium of writing and reading "build or migration" a hundred times, we use the term "build" more generically to mean the process that establishes the required version of the database. If the advice is specific to the build-from-scratch process or the migration process, I make that explicit in the text. Most of the advice is relevant to either process.

DLM database builds are automated

To make a database build easy, individual scripts should create a single object as defined by SQL Server (a column or key, for example, is not an object). Where this is the case, and the scripts are named systematically by the schema and object names (object names are

only guaranteed unique within a schema), then it is possible to generate automatically an accurate list of scripts, in the order in which they should be executed. Unfortunately, this ideal world is seldom found. To allow a database to be built that has scripts with a number of different object in them, you either need to create, and subsequently maintain, a 'manifest' file that lists the files in the order of execution, or you should use a tool such as SQL Compare or DacFx to do the build.

- If using a manifest, keep that in source control with the scripts.
- Label and branch the manifest with the scripts
- If using a differential tool, such as SQL Compare or DacFx, save the generated script in source control with the appropriate label or branch

A big part of automation is having the ability to both replicate what you've done and go back and review what you've done. If you keep the manifest or generated scripts in source control, you'll always be able to do this.

Don't reverse engineer the source code

It is a bad idea to reverse-engineer the source code of an object from your development server. We all sometimes use the visual aids in SSMS to create objects. The database diagramming tool in particular is useful for creating the first cut of a database, and there are tools for creating tables, views, and expressions that can speed the process up. However, the job you are doing must end with the script that was generated by the design tool being edited so as to add comments and then save to the VCS.

Although a build tool or proprietary build server is often used to build databases, it is not necessary. You merely need an automated script that executes each script serially in order, and SQLCMD.exe can be used easily to execute a SQL file against a target server.

If you are using a shared development server with the database in it, and it is kept entirely up to date with source control (all current work is checked in) then you can calculate the build order with a SQL routine that does a topological sort of the dependencies between objects to list first the objects that have no dependencies and then successively, all the objects that are depending only on already listed objects until all objects are accounted for. You can, of course, create a PowerShell routine that calls SMO to do the task for you. If all else fails, you can use SSMS to generate an entire build script for you from the live database and use that creation order for your manifest.

DLM database builds are frequent

During development, the build process should be done regularly, because it tells you whether it is possible to do this with the latest version of the committed code, and allows you to keep all the team up to date. The canonical database that results from a successful build can also be checked against other development versions to make sure that they are up to date, and it can be used for a variety of routine integration and performance tests.

Frequent or overnight builds can catch problems early on. A build can be scheduled daily, once all developer changes have been committed and if necessary merged, or after every significant commit, when any component parts change. Testing is performed on the most recent build to validate it. Usually, builds are done to the latest version, but with version control, ad-hoc builds can occasionally be done when necessary on previous versions of the code to track down errors, or to revive work that was subsequently deleted.

DLM database builds are consistent across environments

Do you need a different build process for development, test, integration, release, and deployment? Essentially, the answer is no. The same build process should be used, with any necessary modification, for development, test, and production. It should be

automated, and the modifications for the different contexts should be in the manifest rather than in the scripted process itself. All these manifests can be saved either in development source control or in the CMS.

The only data that should be held in source control is that of 'enumerations', 'lookup-lists' or static data, error explanations, and such that are necessary for the database to work.

All other data is loaded in a post-build script as dictated by the manifest, unless the build takes place against a version that already has the required data. Test data should be generated to the same distribution, characteristics, and datatype as the potential or actual production data, and each type of test is likely to require different test data sets. Performance and scalability testing will require a range of large data sets whereas integration tests are likely to require standard 'before' and 'after' sets of data that includes all the likely outliers. Test data is best loaded from 'native' BCP format using bulk load.

A build process within development will generally use only local resources. This means that up-stream ETL processes will be 'mocked' sufficiently for them to be tested. Allied databases, middleware services and message queues will either be skeletal or mocked. Likewise 'downstream' processes that are 'customers' of the data from the database are generally also 'mocked'.

A development build process will also have a security and access control system (e.g. GRANTs and user accounts) that allows the developers complete access. If it is role-based, then the production access control can easily be used for a production build during the deployment process. Typically, operations will want to assign users via Active Directory, so the production build will have a different set of Windows groups created as database users, and assigned database roles as befits the security model for the database.

In order to work this magic, all DLM solutions give a high degree of freedom to what is included, or not, in a migration script. Templates for such problems as references to external systems, linked databases, file paths, log file paths, placement on a partition scheme, the path to full-text index files, filegroup placements or filesizes can be used that would produce different values for each environment.

DLM database builds are tested

Of course, we need to validate not just that the build or migration succeeded, but that it built the database exactly as intended. Is your database and data intact? Do all parts of your application and important code still function as expected?

By incorporating automated testing into your build process you add additional protections that ensure that, ultimately, you'll deploy higher quality code that contains far fewer errors. Chapter 12 of the book addresses testing in more detail.

DLM database builds are instrumented

Once you have the build process automated, the next step is to set up measurements. How long does a build take? How many builds fail? What is the primary cause of build failure?

You need to track all this information in order to continuously improve the process. You want to deliver more code, faster, in support of the business. To do this, you need to get it right. If you're experiencing the same problems over and over, it's best to be able to quantify and measure those so you know where to spend your time fixing things.

As the databases increase in number, size, and functionality within an organization, the complexity and fragility of the build process will often increase, unless serious thought has been given to structuring and modularizing the system via, for example, the use of schemas and interfaces.

A common blight of the database build is, for example, the existence of cross-database dependencies. At some point, the build chokes on a reference from one object to a dependent object in another database, which it can't resolve for one reason or another. At this point, it's very important that the build process is well instrumented, providing all the details the developer needs, including the full stack trace if necessary, to pinpoint with surgical precision the exact cause of the problem.

In poorly instrumented builds, it's more likely the developer will see a vague "can't find this column" error, and will be left to sift through the remaining few thousand objects that didn't load to find the one that caused the problem. No wonder, in such circumstances, database builds and releases are infrequent and often delayed, and therefore potential problems spotted much later in the cycle.

DLM database builds are repeatable, measureable, and visible

If a build is successfully automated, it is much more likely to be repeatable. The governance process will be more confident in the time that must be allowed for a build process and validation, and will be much more confident in a successful outcome. This makes the entire end-to-end delivery process more predictable.

An automated build can be more wide-ranging, building not only the database, but also any automatically-generated build notes and other documentation such as help pages.

It is easier to report on an automated build, as well as simple facts such as the time it took and whether it succeeded, the number of warnings, and even an overall code-quality metric, or code policy check, if necessary. This allows the governance process to do its audit and accountability tasks, and firm-up on planning estimates.

Finally, a DLM build is designed to be flexible enough that external dependencies such as ETL processes and downstream BI can be 'mocked', by creating interface-compatible stubs, to enhance the range of testing that can be done and allow development to run independently.

Database builds and migrations in more detail

On the face of it, the database build process seems simple; simply build the tables and other objects in the right order. However, a surprising number of objects can comprise a successful build, in enterprise databases, and there are many choices to be made as to how the build processed in case of various eventualities.

Database-specific migration considerations

There seem to be an infinite variety in databases and this is reflected in the number of options that have to be considered in the build or migration, and subsequent deployment process.

Is the server version (platform) compatible with the database code?

Different versions and editions of the RDBMS will have different capabilities. It is easy to produce working code for one version of SQL Server that will fail entirely, work in a subtlety different way, or with degraded performance, on a different version. A database is always coded for a particular version of the RDBMS. What should happen if the build is done on a server instance with a different version of the product?

When updating CLR assemblies, does the process drop blocking assemblies?

SQL Server allows functionality to be written in .NET languages such as C# or VB. By default, any blocking/referencing assemblies will block an assembly update if the referencing assembly needs to be dropped. How should the build process react if this happens?

Should the database be backed up before the update is made? (Migration only)

If a build is done as part of a deployment process, then a number of precautions need to be taken. A database backup is an obvious backstop precaution, though if a deployment goes wrong, the restore time can be prohibitive.

Should the build be aborted and the changes rolled back if there is a possibility of data loss? (Migration only)

This is only relevant if an existing database is altered to update it to a new version, using a synchronization script. Sometimes, if a table is heavily refactored, it isn't obvious to any automated process how the existing data of the previous version of the table is to be reallocated, then a manual migration script has to be used instead. This requires the build process to be aborted and existing changes rolled back. Then a manual migration script has to be created and tested before being used to migrate a database between two versions. Subsequently, the migration script is used instead of the automated synchronization script to do the build when the target database is on the version before the refactoring.

Should the build be aborted and the changes rolled back if there has been drift? (Migration only)

If the target database is not really at the version it is supposed to be, then someone, somewhere has done a change. If the build deletes this change then work may be irretrievably be lost. If this is a risk, then this is time for the anomaly to be investigated: otherwise the build should go ahead and obliterate the drift.

Should the collation of the target database be changed to the source database being built?

A collation will affect the way that a database behaves. Sometimes, the effect is remarkable, as when you change from a case-sensitive to a case-insensitive collation or the reverse. It will also affect the way that data is sorted or the way that wildcard expressions are evaluated. This will produce subtle bugs that could take a long time to notice and correct.

Are we updating or re-creating the database?

A clean build of a database from scratch is much simpler to do, because there is no risk of over-writing existing configurations or changes that have been done in a manner that is invisible to the build components. A re-creation is always cleaner, but it doesn't preserve existing data. It is always better to import data from file in a post-build process but this is likely to take more time, and if a table is refactored, the existing file-based data will need to be split and manipulated to fit in the new table structure.

Can database-level properties be changed in the build?

Code that works under one database configuration can easily be stopped by changing a database-level configuration item. When a server creates a database, it uses the database-level configuration that is set in the MODEL database. This may be quite different from the database configuration on developer servers, and if these differences are allowed the result will be havoc.

Should the database being built then be registered as a particular version on the server?

SQL Server allows databases to be registered, meaning that an XML snapshot of a database's metadata is stored on the server and can be used to check that no unauthorized changes have been made. It is generally a good precaution to register a database if you are using DacFx, but very large databases could require noticeable space.

Can we stop users accessing the database during the upgrade? (Migration only)

This is only relevant if doing a build using a production database as a target as part of a deployment process. We may require non-breaking online deployments, but will risk locking and blocking on large tables (a table column ALTER is likely to lock the entire table).

Should all constraints be checked after all the data is inserted into the build? (Build-from-scratch only)

This only applies if data is imported into the database as part of the build. In order to guarantee a fast data import, constraints are disabled for the course of the import and then re-enabled after the task is finished. By re-enabling the constraints, all the data is retrospectively checked at once. It is the best time to do it but it could be that more data has to be inserted later, and so this should, perhaps, be delayed.

Should we disable DDL triggers that are auditing DDL changes, during the build?

Many databases track changes to the database definition as part of an audit. This catches intruders who are attempting to promote their permissions to gain access to tables. Unfortunately, these generally need to be disabled for a build as DDL triggers will slow the build and produce invalid audit entries.

Do we alter Change Data Capture objects when we update the database? (Migration only)

Change Data Capture involves capturing any changes to the data in a table by using an asynchronous process that reads the transaction log and has a low impact on the system. Any object that has Change Data Capture requires sysadmin role to alter it via a DDL operation.

Should we alter replicated objects in the database?

Replicated tables will contain special artefacts that have been added to the table when it became a publisher or subscriber.

Do we drop existing constraints, DDL Triggers, indexes, permissions, roles or extended properties in the target database if they aren't defined in the build? (Migration only)

Where operations teams are maintaining a production system, they may make changes that are not put into development source control. This can happen legitimately. Normally, changes to indexes or extended properties should filter back to development version control, but in a financial, government or healthcare system, or a trading system with personal information, there must be separation of duties, meaning that database permissions, for example have to be administered by a separate IT role, and the source stored separately in a configuration management system. To apply a build to a target database that has separation of duties, permissions and role membership has to be ignored. DDL Triggers that are used for audit purposes are sometimes used to monitor unauthorized alterations of DDL, and so obviously cannot be part of development source control.

How do we deal with NOT NULL columns for which no default value is supplied? (Migration only)

If the build involves updating a table that contains data with a column that does not allow `NULL` values, and you specify no column default, should the build provide a default to make existing null data valid?

Developers sometimes forget that, if they are changing a `NULL`-able column to a `NOT NULL`-able constraint, all existing rows with a `NULL` in that column will be invalid and cause the build to fail. Sometimes a build that succeeds with development data then fails when the build is used to deploy the version to production. Of course, the code needs to be altered to add a `DEFAULT` to a non-null value to accommodate any rogue rows. However, some developers want the build to do this for them!

Should the build ignore differences in file paths, column collation, and other settings?

Many paths and database or server settings may vary between the source and target environments. For example, there will likely be different filepaths for data and log files, perhaps different filegroup placements, a different path to full-text index files or for a cryptographic provider.

There may be other differences too, such as in the order of DDL triggers, the default schema, extended properties, fill factors, the seed or increment to identity columns, keyword casing, lock hints, not-for-replication settings, placement on a partition scheme, semicolons, and other syntactic sugar such as whitespace.

Migration scripts that take an existing database and change it to the new version have a particular difficulty. It is not an all-or-nothing task. There are shades of grey. Some settings, such as file paths, log file paths, placement on a partition scheme, the path to full-text index files, filegroup placements or filesizes have nothing to do with development and a lot to do with operations and server administration. They depend on the

way that the server is configured. It can be argued that the same is true of FillFactor. There are dangers in over-writing such things as the seed or increment to identity columns, or collation. The most curious requirement that has been made is that differences between semicolons or even whitespace is enough to require an update to the code in a routine such as a stored procedure or function.

How should the build handle security options, security identifiers, and differences in permissions, user settings or the role memberships of logins?

Should these be ignored or updated? This very much depends on the context of the build. It is, for many databases, probably illegal to allow the same team of people to both develop the database and to determine, if the database is holding live personal, financial or healthcare data, the access rights or security settings for database objects. Best practice is for these to be held separately in a central CMS archive by operations, and determined according to whether the database holds real data or spoofed test data. Within development the problem is greatly simplified as long as live data is never allowed. The development and test builds can run unhindered. As soon as live data is used, as in staging, forensic analysis, or production, then all security and access control settings need to comply with whatever legislative framework is in force.

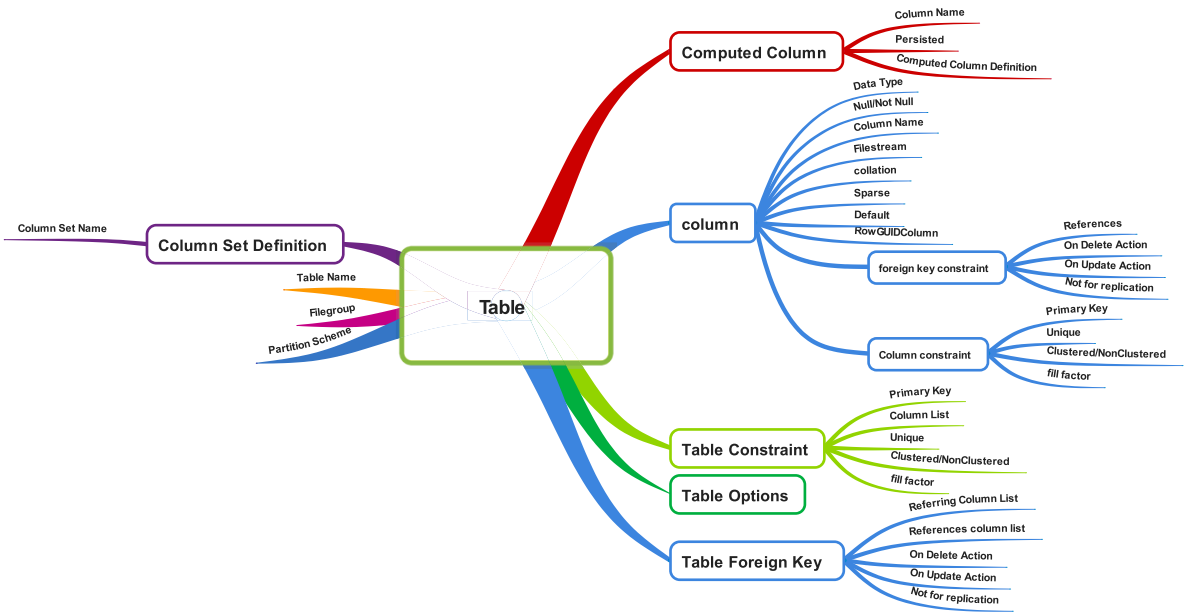
How should the build handle differences in whether CHECK constraints or FOREIGN KEY constraints have been disabled?

Constraints can be temporarily disabled for ETL processes and they are sometimes mistakenly left disabled. It is always a mistake to leave them disabled. If the intention is to not use them they should be dropped, otherwise anyone inspecting the code might get the false impression that the database is protected from bad data.

What database objects need to be built?

There are over fifty potential types of database object in SQL Server. Only base tables hold data. Many database stray not much further than schemas, database roles, tables, views, procedures, functions, and triggers.

In SQL Server, tables are objects but the schema that it resides in isn't. Its constituent parts are child objects or properties in a slightly confusing hierarchy:



Without a doubt, tables are best handled at table-level rather than treating the child objects in their own right. In scripts generated from DacPacs, Microsoft are rather inclined to detach foreign keys and extended properties from their parent table for a reason that isn't at all obvious.

In SQL Server, in alphabetic order, the database objects include

Aggregates	ApplicationRoles	Assemblies	AsymmetricKeys	BrokerPriorities
Certificates	Contracts	DatabaseRoles	DatabaseTriggers	Defaults
ExtendedProperties	Filegroups	FileTables	FullTextCatalogs	FullTextStoplists
MessageTypes	PartitionFunctions	PartitionSchemes	Permissions	Queues
RemoteServiceBindings	RoleMembership	Rules	ScalarValuedFunctions	SearchPropertyLists
Sequences	Services	Signatures	StoredProcedures	SymmetricKeys
Synonyms	Tables	TableValuedFunctions	UserDefinedDataTypes	UserDefinedTableTypes
ClrUserDefinedTypes	Users	Views	XmlSchemaCollections	Audits
Credentials	CryptographicProviders	DatabaseAuditSpecifications	Endpoints	ErrorMessages
EventNotifications	EventSessions	LinkedServerLogins	Routes	ServerAuditSpecifications
ServerRoleMembership	ServerRoles and ServerTriggers			

Server-based objects that need to be built with the database

It is a mistake to think that by storing the database objects in source control and building those that you have the entire database. Some database application objects are stored at server level and if they are tightly bound to the database logic, then they must be held as SQL scripts, in development VCS and built with the database. Some server-based objects are associated with the database application but need entirely different scripts for each server environment. Operators, the contact details of the person who is alerted when a database goes wrong, is a simple example of this type of script.

As we prepare to deploy the database, or database changes, to production, the same build process must be able to define the appropriate operator in each environment, whether it be development, test, integration, staging or production.

We need to store in source control the T-SQL scripts to create the latest versions of all of the required Agent and other server objects. SMO exposes these objects through the Server and JobServer class, and we can create a PowerShell script, for example, to iterate over all of the various collections of this class, containing the jobs and alerts and other server objects, along with associated operators, schedules and so on.

Not every object will be relevant for the application so after running the script for the first time, you'll have some tidying up to do, weeding out irrelevant objects. You'll need a reliable way to 'flag' which job, alert, endpoint, or whatever, is part of which database. There are a few ways to do, one of which is to use the permissions on the resource. However, using the name is probably the easiest because it's a property common to all the objects. In some cases, such as with job steps, things are easy since SQL Server allows you to assign them explicitly to a database, but this isn't generally the case. Using the name is ugly, but it meets the purpose.

Agent Jobs and Job Steps

DBAs use SQL Server Agent Jobs for all manner of tasks, such as running background processes, maintenance tasks, backups, and ETL. Databases of any size will have Agent Jobs that are entirely part of the database functionality. Because of this, they should be stored in the development VCS rather than be 'owned' by Ops and stored in the CMS archive. Although there could be Agent Job Steps that are little to do with individual databases, many of them are involved with ETL, such as replication jobs, analysis services jobs, integration services jobs.

Agent Jobs are generally, but by no means always, T-SQL batches. They can even be PowerShell scripts, ActiveX Scripting jobs or executables. SQL Server stores jobs for all databases on the SQL Server instance, all together, in the SQL Server Agent, in MSDB.

It is best to start with these jobs being in source control. It is not easy to unpick the scheduled jobs on a server with more than one database on it to script out the ones that are appropriate to a particular database application

It isn't a good idea to involve several logically separate databases with the one job unless it is an administrative server job such as backups or maintenance. For application tasks it is one job, one database. Otherwise, what happens when the database versions get 'out-of-sync'?

Agent Jobs often reference scripts in file locations. Different servers may have these in other locations so these need to be parameterized in the build script so that the build process can assign the correct filepath.

It is wise to document the source code of Agent Jobs with the name and description of the job which database(s) it is accessing. Use the description to provide special detail, such as a PowerShell script on the server that is called from a PowerShell-based job and which should also be saved.

SSIS tasks that are called by SQL Agent must be in source control too, along with batch files that are called from job steps. Is that PowerShell script executing a file as a script block? (That means that the file must be saved as well.) Are server-based executable files involved? Someone needs to ensure all of this is in source control and then check they have not changed since the last deployment (the description field of the job step may help).

Agent Alerts

Agent Alerts are generally more relevant to the server than the database, but they are part of the build process. For the application, it is a good practice to set up special alerts for severe database errors. Operations people will have definite opinions about this. Alerts should, for example, be fired on the occurrence of message 825 (tried to read and failed), and a separate alert for each severity 19, 20, 21, 22, 23, 24, and 25. It is also wise to

alert on severity levels 10 – 16, indicating faults in the database application code, usually programming errors and input errors. Usually, the end user hears about these before the developers, or Ops people. There will also be various performance alerts, set up in order to alert the operator when a certain performance condition happens.

Alert notifications will change according to the type of installations in order to specify the right recipient. For integration or UA testing, the database developers need to know about it, whereas in production, the production DBA will need to know about these. The build needs to take these alerts from the appropriate place, the development VCS or the operations central CMS archive.

Wherever they are held, we need to script all these alerts and, like jobs, we should use the name to signal which alerts belong to which database.

Operators

These are the recipients of the alerts, the addresses notified by email of corruption, errors in code, job failures, or major problems. The operator is likely to be different in development, test, integration and production. This means that the script will be different for each environment. The build process must be flexible enough to deal with settings and scripts like this that are different for every server environment

Proxy accounts

Proxy accounts, or proxies, are useful when some principals need to access external resources, but when it is unwise, for security reasons, to grant those necessary extra permissions to those principals. To avoid having to do this in order to, for example, run a PowerShell script, we can map, purely for that task, the agent account to another account that already has these rights. This is generally going to be a database-level security device, so we need to install those proxies appropriately for the server on which the database is to

be deployed. You may even need different proxies in development, test, integration, and production.

Server triggers

Server-scoped triggers track server-wide DDL changes, such as a database creation or drop, and login attempts to the current server. If specified, the trigger fires whenever the event occurs anywhere in the current server. Their main use is for server security and audit, but an application might use them for security purposes.

Linked servers

An application might use linked servers for ETL, distributed database layering or data transfer. They are sometimes used for archiving old data. We need to install these, with the application, and it makes things neater for the build process if we obey the following rules:

- One linked server, one database
- Keep the source of the linked server in source control
- Ensure that the creation or update is repeatable.

Conclusion

When developing a database application, it is a mistake to believe that database build is a simple task. It isn't. It has to be planned, scripted, and automated. The build must always start from a known state, in the version control system. You can then use various means to generate the scripts necessary for a build or migration. You need testing in order to validate the build, and ensure that you're building the correct items in a safe fashion. You need instrumentation and measurements that will log the time of the build, the version being built and the success of the build, along with warning messages. Warnings and errors need to provide detailed diagnostics, if necessary, to pinpoint the exact cause of the failure quickly.

As a result, your database builds and releases will be frequent, punctual, and you'll spot and fix potential problems early in the cycle, and you'll have the measurements that will prove just how much your processes are improving and how much higher the quality and speed of your development processes has become.

Feedback

We would love your feedback on the book, particularly in its early stages as a 'lean' publication. A core tenet of our approach to DLM is early and rapid feedback on changes, and this is the approach we have adopted when writing the book. All feedback, positive and negative, will help us make the book as relevant and useful as possible.

As chapters become available, we will post them to:

www.leanpub.com/database-lifecycle-management

We will then progressively refine the content based on your feedback, which you can submit using the comment form at the above URL or by sending an email to dlmbook@red-gate.com

Look forward to hearing from you!