Challenges of using a database with DevOps

In this section, we want to talk about the challenges of including your relational databases in a DevOps process. Is there a fundamental reason why database code if different from application code?

The answer is the classic “it depends”, a common phrase used in database work. There are reasons why *some* database code is different and some is the same. In a SQL Server database, we have a variety of different objects. We have tables, views, stored procedures, functions, assemblies, indexes, constraints, and more. Of these, some are code just like your C#, Visual Basic, Java, Python, or other application code. Some aren’t.

The objects that are like application code, the views, stored procedures, functions, and assemblies, can be treated just like application code. Deploying a new version of this code, or restoring an old version, is very similar to deploying a new version of the application code. We can switch back and forth, compiling various versions without impacting our database. Of course, the logical effects of the different versions might be problematic, but this is the same as replacing a method in one of your classes with a different version.

If you only ever need to change your view, stored procedure, and function code, then including your database in a DevOps process is easy. You can even automate the rollback of one version to a previous one. The process would be slightly different as the rollback process needs to recompile an old version in the SQL Server database, as opposed to copying the previous version of a .DLL or .EXE, but this are just implementation details. Fundamentally, the process is the same.

However, for tables and the associated objects, such as indexes or constraints, the code is different. The reason is that these objects need to maintain state between versions. The state is the data, and we cannot just delete and replace the objects with new, or old, versions. We might incur data loss, and that is almost always a problem for an organization

The need to maintain state can cause multiple problems. First, changes must account for the state of the system and cannot cause data loss. This means no destructive changes. We must ensure our alterations of these objects are evolutionary. Second, because we need to ensure our data is intact, even as a deployment might cause a transform, we may have difficulty performing a rollback of changes. We’ll look at an example of this later in the course.

Lastly, the database is often a common access point for multiple applications or systems, which means that changes can impact multiple processes. Either we must provide forward and backward compatibility, which brings up integrity and performance issues, or we must coordinate changes across multiple development teams. This need for tight coupling is fundamentally at odds with the idea of loosely coupled, contract or API based DevOps methodologies.

These are core issues with including a database in a DevOps process, but there are other challenges with databases that are less technical, which we’ll cover in the next module.