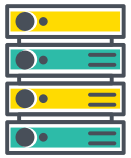
Using Microservices? Don’t Forget the Database

By Brady Byrd, Senior Solution Architect

DBmaestro
DevOps for Database

The microservices pattern is a hot item on the agenda for almost every development shop. The promise of isolating specific pieces of an application into services simplifies development constraints, release cycles, and design.

The question here is, what does that mean for databases? Bold statements like, “with microservices and containers, there won’t be a need for formal deployment tools” are commonplace. Is this accurate? Should this inspire a whole bunch of us to switch careers?



Breaking it down, typically a service will need some sort of persistent storage. There are some cases where a service will simply broker information from other services as a pass-thru, but this is less common. Services might share a database, but according to the strict microservices paradigm, each service should be the owner of its data.

With that synopsis in mind, let's look at what a release means from a database perspective. The simple sales model from https://microservices.io, which includes customers, orders, and product services all driven by a single web app, is a great example and will be the backdrop for this discussion.

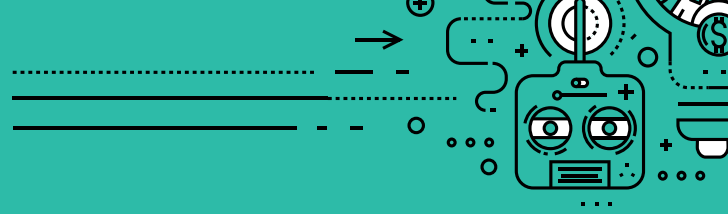
Let's look back to old monolithic applications which might have had one or two schema with thousands of objects. There were constraints, joins, triggers, and code objects like views and stored procedures to enforce relational integrity. In microservices, all of that has been moved into the API and application code so that one team can push updates to customer service and not have to simultaneously update orders, pricing, and the product catalog. If you've ever lived through an integrated release at something like a large insurance company, this is an attractive prospect.

The simplest case would be a release of just one service. Because the API is versioned, the net change for any consumer would be zero, unless they are subscribing to new functionality in the release and thus attach to the new API. That means deploying changes in only the database used by the service, great – that's simple.

To add a degree of complexity, let's suppose that the order and product services were both being enhanced, and that orders needs to go out immediately after the product release. Now there are two databases to update which also have to be deployed in a specific order.

Because these transitions are not complete, and because dependencies in a large enterprise go on and on, the chances are that the release involves changes to both microservices and legacy applications. With regard to database release, this means business as usual, you have to coordinate changes between schema and applications are dependent on each other. So, unless you have made it all the way to the land of milk and honey the database component can be complicated.

Consider a database change: a new table, a column for the relation, and several indexes for performance. The change is first designed in a sandbox and then deployed to a DEV environment. Let's assume that there is a script that will make the change which is also run in DEV. Typically, in a container strategy, the data can perhaps live in the container in a DEV environment. Most data sets however, require persistent storage. In a DevOps setting, the app and database engine can be deployed together. The strategy for the database needs to change to incrementally upgrade the structure and data.



Perhaps the environment model is a straightforward DEV, QA, STAGE, PROD progression and the release requires the upgrade of several services. From a management perspective, all the connection information for each service's databases must be present in order to queue the changes to be applied to each database. Even if the database is instantiated from a container, the latest development changes need to be applied. Moreover, it is critical to assess the database version that the container is presenting. In short, the technical needs and gotchas are numerous, so there is strong need for a tool to manage database changes and detect versions across environments.

Now your fully-fledged microservices architecture is supporting your app and version 1.0 is sailing fine – you've hit the big time. If you work in a large enterprise, all sorts of changes have happened. "Operationalization" has occurred and there is a whole support infrastructure with monitoring, training environments, and incident-response protocols. You probably have a separate release team managing deployments which may not understand your development culture. They might be managing dozens (or even hundreds) of applications in which your µService ecosystem is just one more item on the schedule. Maybe the CI/CD pipeline reaches STAGE, but now, PROD is more formalized. Perhaps segregation of duties requires that the dev team cannot touch the environment. The ability to communicate what changes are needed and the order in which they should be run, as well as portable automation that can operate in any environment (even locked down PROD), become a critical requirement.

For now, fussy databases are still part of the landscape. They contain metadata, structure, and underlying data, and are large, unwieldy, and do not tolerate little mistakes very well. Modern application architectures have simplified some of the processes and reduced dependency nightmares, which empowers teams to release changes more frequently. So, the situation is better overall, but in no way eliminates the need for a managed approach to database change. In fact, distributed architectures rely more heavily on coordination and orchestration software when the business schedule for feature delivery is overlaid.

An essential part of this puzzle is database DevOps. Because microservices architecture is characterized by enabling continuous delivery, employing DevOps for the database is a natural step for any microservices-based operation. Want to discover how? Contact DBmaestro for a free demo and discover the advantages of database DevOps.