Stored Procedure as a Service (SPaaS)

Abhishek Tiwari

Citation: *A. Tiwari*, "Stored Procedure as a Service (SPaaS)", Abhishek Tiwari, 2016. doi:10.59350/bxfbd-ah658

Published on: October 08, 2016

Functions as a service (FaaS) is an emerging pattern to build APIs and microservices at scale. You can use various FaaS implementations such as AWS Lambda, Azure Functions, and Google Cloud Functions to build APIs ecosystem for your organisation. If you remove serverless requirements, stored procedures can be considered as a variation of FaaS because they share similar traits and concerns. Developing APIs against these stored procedures is generally considered as an anti-pattern. But what if you have a substantial investment in stored procedures (8-12 years) and most of your business logic is embedded in stored procedures. In this blog post, I will discuss how you can leverage existing stored procedures to create a fast-track API transformation program on top of your legacy systems.

Functions as a service (FaaS)

In Functions as a service (FaaS) pattern, stateless functions are exposed via API endpoints. A function in FaaS is a self-contained piece of reusable functionality and normally run in a serverless environment. These functions can be executed by events or API endpoints. They can read and write to database or storage backend. Depending on the cloud provider, these functions can be written in JavaScript, Java, Python, C#, etc.

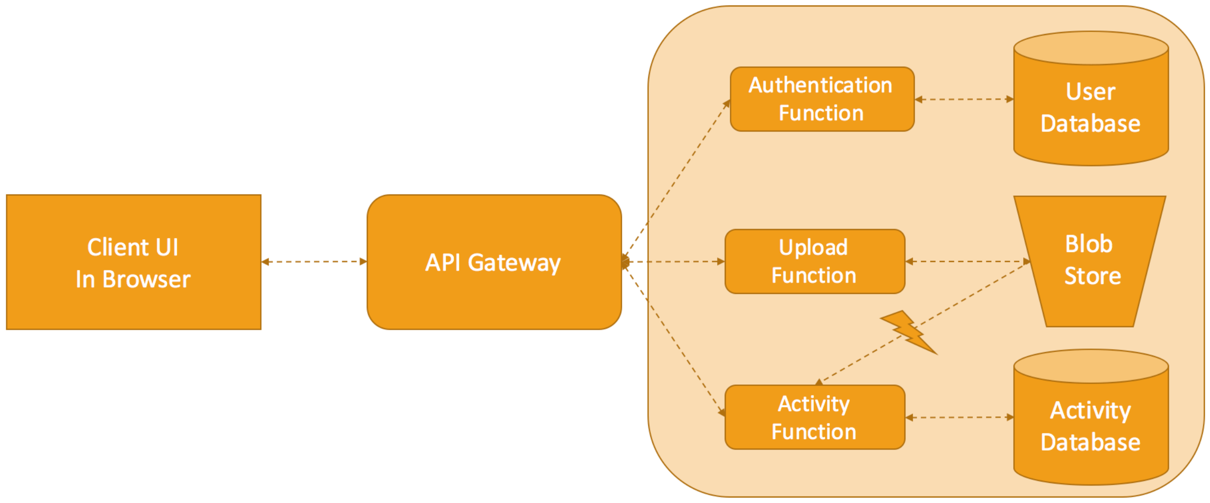


Figure 1: FaaS Pattern

FaaS Concerns

Currently, most of FaaS implementations are vendor-specific. Underlying serverless environments are very different due to different function specifications, runtime constraints, and execution methods adopted by cloud providers. In addition, tooling for FaaS continuous integration and continuous delivery is in very early stage or non-existence. Then we have monitoring, logging, and debugging related issues. I am not going to even think about the lack of testing approaches, service discovery, security gaps and startup latency.

Stored Procedures

Stored procedures are subroutine stored in the relational database data dictionary. These subroutines can be executed by applications accessing the database. Stored procedures are supported by major relational databases such as Microsoft SQL Server, Oracle, PostgresSQL, MySQL, etc.

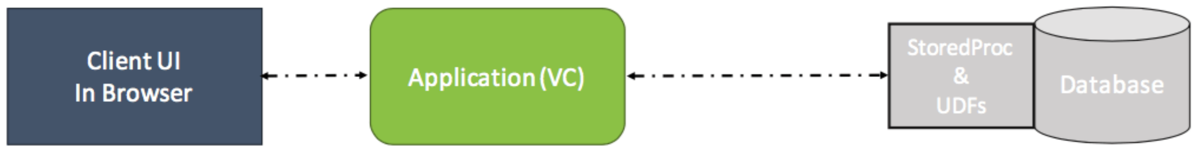


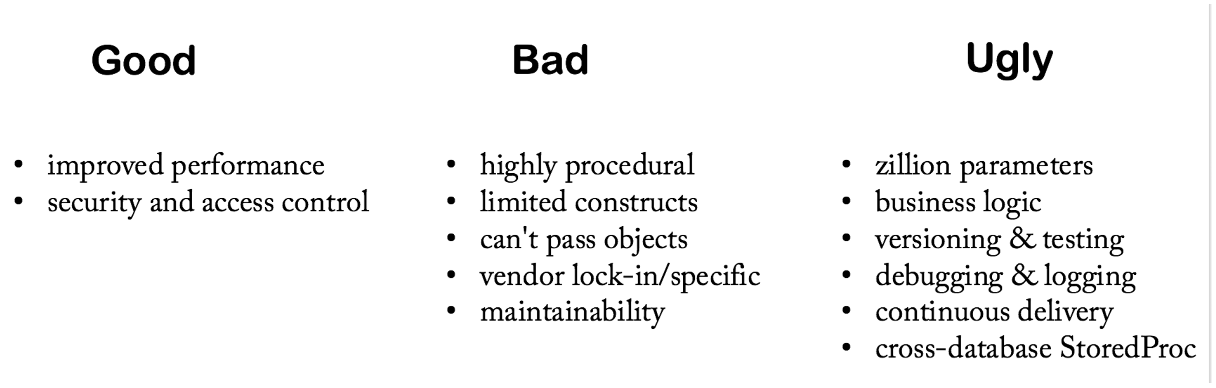
Figure 2: Executing stored procedures via an application to deliver data to a client UI in the browser. Quite similar to MVC application exception the Model layer i.e. M is missing

Ugly

There are numerous articles on the web describing how stored procedures are bad and should be avoided at all costs. I am generally in agreement with these concerns. In fact, I consider stored procedures as an anti-pattern, i.e. a good idea that quickly turned into massive technical debt. Most of these technical debts were accumulated due to poor implementation and lack of continuous delivery models. For instance, it will be difficult to test your applications if all or most of business logic is implemented in stored procedures. Although it is possible to unit test stored procedures, often due to the complexity of data logic unit tests are either very slow or not very effective as unit tests. In addition, stored procedures share quite similar issues as FaaS pattern such as debugging, logging, and versioning.

Bad

Several of these stored procedure concerns are caused by of inherent issues. For instance, Stored procedures are highly procedural, you can't pass objects which mean you need to deal with a huge number of parameters which can be very painful. Stored procedures provide limited constructs when compared to a full-blown programming language. Similarly, vendor lock-in or specific implementation makes maintaining stored procedures even harder.



Good

- improved performance
- security and access control

Bad

- highly procedural
- limited constructs
- can't pass objects
- vendor lock-in/specific
- maintainability

Ugly

- zillion parameters
- business logic
- versioning & testing
- debugging & logging
- continuous delivery
- cross-database StoredProc

Figure 3: Stored procedure - Good, Bad, and Ugly

Good

Moving data is harder than moving logic - stored procedure bring logic close to data a honey trap used often to justify the use of stored procedures. Using the stored procedures for very large database and data warehouses enables one to avoid the network round trip. In addition, because stored procedures are compiled and cached they generally offer better performance than dynamic SQL queries on large data sets. To be honest, benefits of stored procedure are not quite obvious unless you are dealing with a very large database or data warehouse - one with billion rows or 5+ terabytes of data.

Legacy Monolith

Generally speaking, if you have a legacy system with large stored procedure footprint (1000+ stored procedures) or your application is a data warehouse application pretending to be database application - you have very limited options in fact only two options. This is particularly true if your applications are thin (no or minimal business logic), your stored procedures are fat (heavy on business logic), and your databases are very large strongly coupled, monoliths (billion rows and counting). Not to mention, if you are constrained by time and resource - delivery new products and services in a short span which is typically the case in many large organisations.

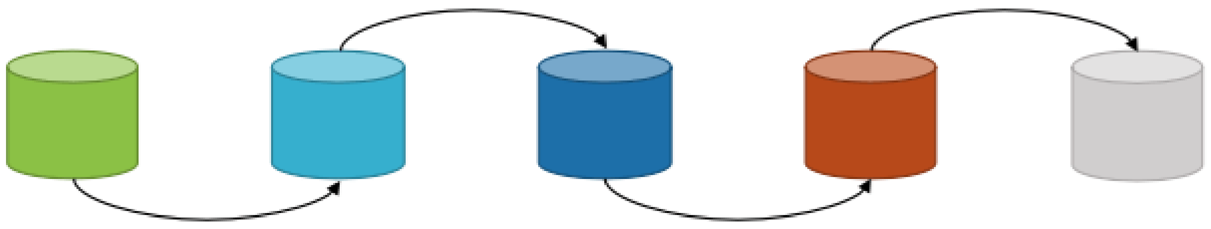


Figure 4: A data pipeline built using stateful, strongly coupled, monolith databases. Database are used as queue, ETL using StoredProc - trigger calling StoredProc calling trigger is not that uncommon

Phased Approach

To modernise your legacy systems irrespective of stored procedures involved or not, I always recommend a phased approach. In this journey, very first step requires decoupling of UI and backend. In next phase, convert your legacy backend layer into an ecosystem of APIs. Finally, your team begins breaking monolith backend into microservices - one at a time while maintaining existing API contracts. For legacy systems, an API-first strategy makes more sense when compared to microservices first approach mainly because your business can realise benefits early and frequently .

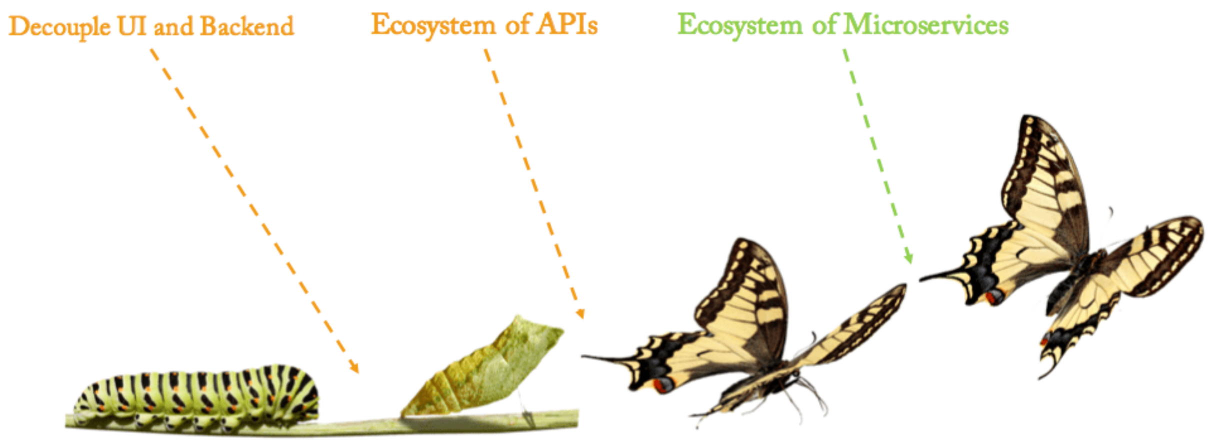


Figure 5: A phased approach for modernizing your legacy systems

Unless your business is considering to re-build existing new applications as part of a multi-year program, I will never suggest microservice first approach. Like it or not, system re-build projects are always overly underestimated due to complex and undocumented business logic developed over the years. Plus, organisation are trapped into bastardise version of agile where requirements are discovered on the fly. This literally means you may end up with a fresh new system which functionality wise either same level (carbon copy) or slightly better (less crappy) than your old legacy monolith. With API first, your business can start moving quickly without risking delay and quality issues of re-build.

Separate UI layer from the backend layer

Develop static stateless UIs - native or hybrid mobile apps or client-side web UI using frameworks such as Angular or React. Before you build any backend APIs, mock API requests & responses using Swagger and provide it as contracts to your UI team. Treat API mocks as contract and in next phase build actual APIs. Serve static Stateless web UI from a content delivery network (CDN) in a client browser. As UI are stateless, data is delivered via the APIs.

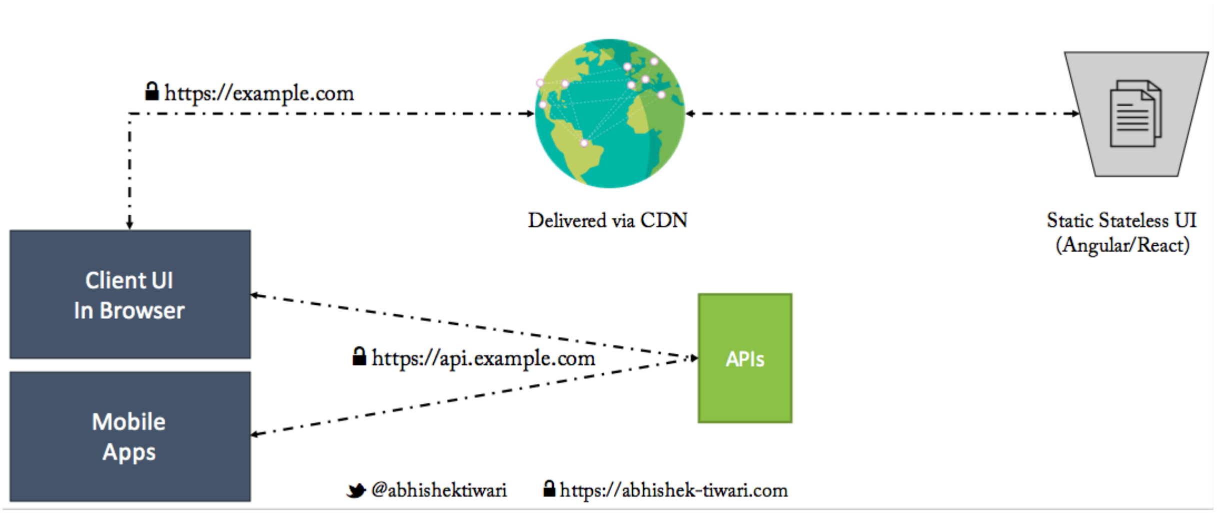


Figure 6: Separate UI layer from the backend layer

Convert legacy backend layer into ecosystem of APIs

To build an ecosystem of APIs on top of legacy systems, one must tap into existing system databases or data sources used by legacy systems. Once you have identified databases, you can either create instance CRUD APIs using frameworks like Loopback and DreamFactory without or minimal coding, or roll your own APIs to have more flexibility and control.

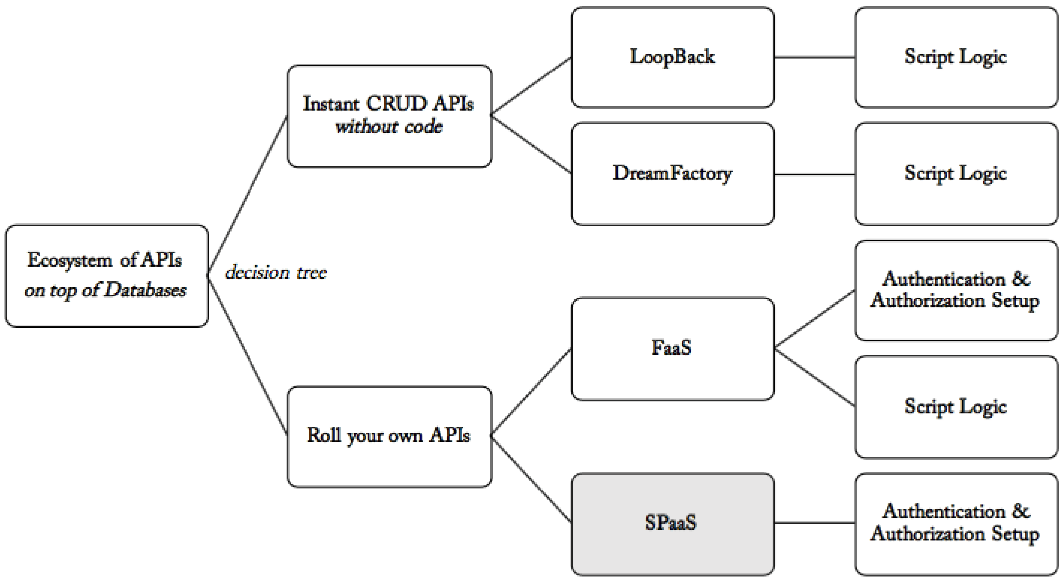

Ecosystem of APIs on top of Databases
decision tree
Instant CRUD APIs without code
Roll your own APIs
LoopBack
DreamFactory
FaaS
SPaaS
Script Logic
Script Logic
Authentication & Authorization Setup
Script Logic
Authentication & Authorization Setup

Figure 7: Convert legacy backend layer into ecosystem of APIs

Instant CRUD APIs without code

API frameworks such as Loopback and DreamFactory provide a lot of out-of-box functionalities apart from the instance CRUD APIs including but not limited to API authentication, API authorisation, API security, data caching, API management, etc. As soon as you connect Loopback and DreamFactory to your database they discover tables and stored procedures and generate instance APIs. This means your team will spend a majority of your time scripting the business and data logic rather than building the raw APIs. This approach has been greatly successful for both SQL and NOSQL data sources.

Roll your own APIs

If you decided to roll your own APIs then you have to not only code your CRUD APIs but you also need to develop authentication, authorization, security features as well as script the business and data logic. Again you can use FaaS based approach or expose your existing Stored Procedures as a Service (SPaaS). If you have a substantial investment in stored procedures (8-12 years) and most of your business logic is embedded in stored procedures, SPaaS is can deliver your APIs more quickly - as you are not re-writing your logic but just wiring up existing stored procedures into API services.

Wiring up Stored Procedures into APIs

Wiring up your existing stored procedures into APIs should be quite easy. Here I am going to describe wiring approach using MSSQL stored procedures, Node and Express, but a similar approach can be applied to any other database engine, language and framework - Postgres, Python and Flask for that matter. Anyway, a key aspect of wiring is Tabular Data Stream (TDS) protocol - is an application layer protocol used to transfer data between a database server and a client. Node provides several implementations of TDS protocol for MSSQL - some TDS drivers are only supported on Windows runtime but others run on cross-platform including Linux.

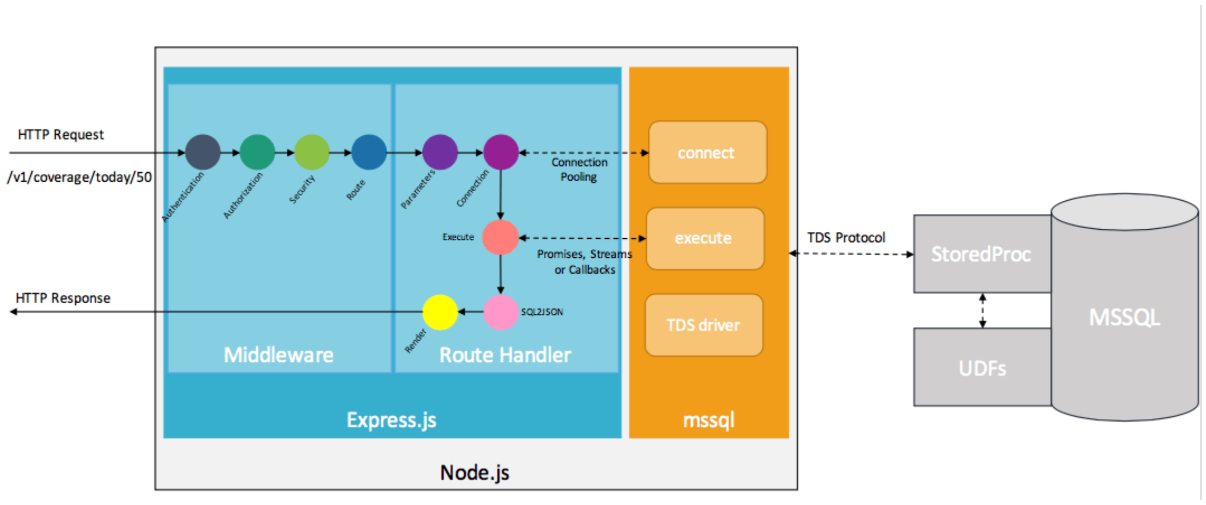


Figure 8: Wiring up MSSQL stored procedures into APIs using Node and Express

As you can in above illustration, an HTTP API request `/v1/coverage/today/50` hits Express middleware which is running inside Node runtime. At this stage, authentication middleware checks if JWT token and session are valid, authorization middleware validates data and API access rules, security middleware filters any security vulnerabilities such SQL or cross-site injection, and event route middleware handles the request to route handler. Route handler, extract parameters, take a database connection from existing connection pool, execute the stored procedures. Stored procedures can be executed synchronous as well as asynchronous manner - here you can use promises or callback or streaming. Results from stored procedure execution is passed to SQL2JSON converter and eventually an HTTP JSON response rendered back to the client.

Continuous delivery for SPaaS

Continuous delivery (CD) is key to the success of any API development program. Continuous delivery along with continuous integration (CI) enables faster API builds, frequent API testing, and less risky deployments and releases.

SPaaS API Delivery

API Delivery in SPaaS follows a normal continuous delivery workflow. A developer makes changes in API code and commits it to a local Git source control repository and push it to a centralised Git server. This centralised Git server is hooked into your Jenkins based CD pipeline. A code push triggers artefact build which is followed by artefact testing, artefact upload and artefact deployment steps in you CD pipeline. During testing, you can run a variety of tests but as a bare minimum unit, functional, integration, load tests are advisable.

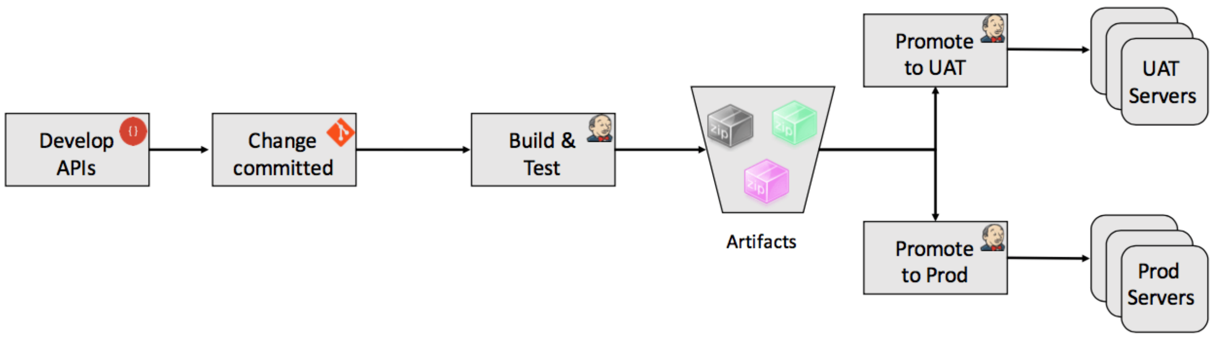


Figure 9: Continous delivery of APIs in SPaaS

SPaaS SQL Delivery

Continuous delivery of database stored procedures and schema changes in a similar type of workflow as APIs is highly desirable. You should start by adopting a tool such as Redgate or ApexSQL which 1) connects your database to your source control system, 2) allows you to compare schemas and deploy differences fast, and 3) integrates with your CI/CD pipeline. Below diagram illustrates a SQL Server delivery pipeline using Redgate's SQL Source Control, SQL Compare, and DLM Automation.

Basically, your developer and DBA will make database schema and stored procedure changes using SQL Server Management Studio with SQL Source Control plugin against the `DEV` database. Once changes are committed to local Git and pushed to a centralised Git server, it triggers the build and test process on Jenkins server which is using Redgate SQL CI plugin along with DLM Automation. After build and test, Jenkins create one or more deployment artefacts (migration scripts) generated by using Redgate SQL Compare or SQL Source Control. Redgate SQL Compare generates migration script by comparing `INT` or integration database local to build process to `UAT` database. These deployment scripts are uploaded to an artefact repository and from there applied to `UAT` and `Prod` databases.

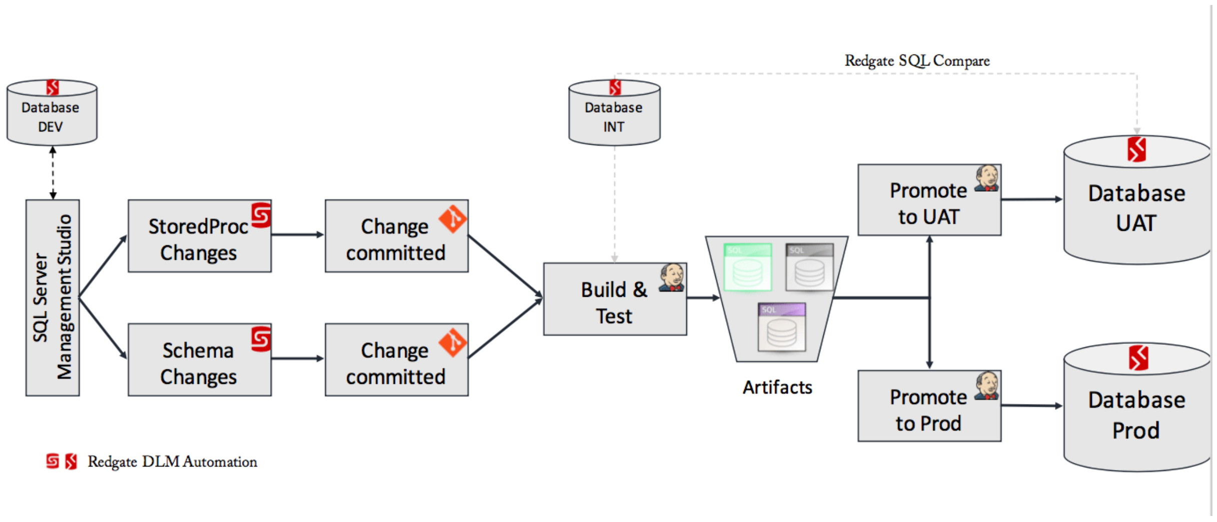


Figure 10: Continuous delivery of database stored procedures and schema changes

Breaking monolith backend into microservices

We are in 2016, hence irrespective of a new shiny API ecosystem wrapped around your existing monolith database - stored procedures are still anti-pattern. Now that you have proved the value of APIs, you must find a way to continue on your journey to modernise your legacy system. Next step is to convert existing stored procedures into microservices one at a time while maintaining existing API contracts. Please note, each microservice must own it's own data store and provide a good separation between business logic and data logic. Before you do so, create a strict rule of no new stored procedures. The role of API gateway is quite critical as we move towards the microservices architecture. Apart from API management features, API gateway act as a proxy which hides API implementation details, for the client API contract is only things that matter.

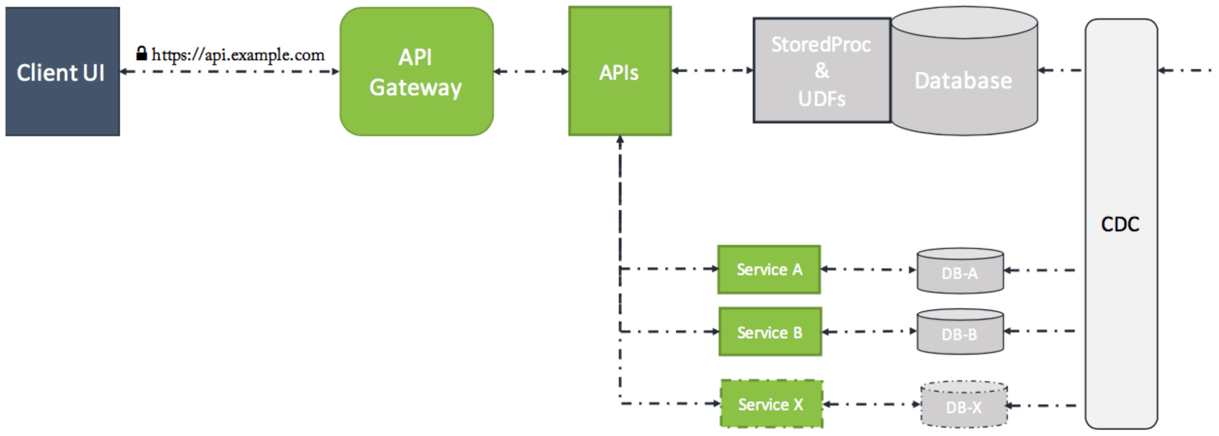


Figure 11: Breaking monolith database into microservices

Conclusion

In this article, we discussed a fast-track approach for creating an API ecosystem on top of your legacy systems with a large footprint of stored procedures. Developing APIs against these stored procedures is only suggested as short to mid-term measure, once API contracts are established replacing stored procedures with microservices - one at a time is highly recommended.