Building with Patterns: The Schema Versioning Pattern

Learn More About MongoDB at MongoDB University

Daniel Coupal and Ken W. Alger
April 19, 2019 | Updated: July 12, 2019

It has been said that the only thing constant in life is change. This holds true to database schemas as well. Information we once thought wouldn't be needed, we now want to capture. Or new services become available and need to be included in a database record. Regardless of the reason behind the change, after a while, we inevitably need to make changes to the underlying schema design in our application. While this often poses challenges, and perhaps at least a few headaches in a legacy tabular database system, in MongoDB we can use the Schema Versioning pattern to make the changes easier.

Updating a database schema in a tabular database can as mentioned, be challenging. Typically the application needs to be stopped, the database migrated to support the new schema and then restarted. This downtime can lead to poor customer experience. Additionally, what happens if the migration wasn't a complete success? Reverting back to the prior state is often an even larger challenge.

The Schema Versioning pattern takes advantage of MongoDB's support for differently shaped documents to exist in the same database collection. This polymorphic aspect of MongoDB is very powerful. It allows documents that have different fields or even different field types for the same field, to peaceful exist side by side.

The Schema Versioning Pattern

The implementation of this pattern is relatively easy. Our applications start with an original schema which eventually needs to be altered. When that occurs we can create and save the new schema to the database with a `schema_version` field. This field will allow our application to know how to handle this particular document. Alternatively, we can have our application deduce the version based on the presence or absence of some given fields, but the former method is preferred. We can assume that documents that *don't* have this field, are version 1. Each new schema version would then increment the `schema_version` field value and could be handled accordingly in the application.

As new information is saved, we use the most current schema version. We could make a determination, depending on the application and use case, as to the need of updating all documents to the new design, updating when a record is accessed, or not at all. Inside the application, we would create handling functions for each schema version.

Sample Use Case

As stated, just about every database needs to be changed at some point during its lifecycle, so this pattern is useful in many situations. Let's take a look at a customer profile use case. We start keeping customer information before there is a wide range of contact methods. They can only be reached at home or at work:

```
{
    "_id": "<ObjectId>",
    "name": "Anakin Skywalker",
    "home": "503-555-0000",
    "work": "503-555-0010"
}
```

As the years go by and more and more customer records are being saved, we notice that mobile numbers are needing to be saved as well. Adding that field in is straight forward.

```
{
    "_id": "<ObjectId>",
    "name": "Darth Vader",
    "home": "503-555-0100",
    "work": "503-555-0110",
    "mobile": "503-555-0120"
}
```

More time goes by and now we're discovering that fewer and fewer people have a home phone, and other contact methods are becoming more important to record. Items like Twitter, Skype, and Google Hangouts are becoming more popular and maybe weren't even available when we first started keeping contact information. We also want to attempt to future proof our application as much as possible and after reading the *Building with Patterns* series we know about the Attribute Pattern and implement that into a `contact_method` array of values. In doing so, we create a new schema version.

```
{
    "_id": "<ObjectId>",
    "schema_version": "2",
    "name": "Anakin Skywalker (Retired)",
    "contact_method": [
        { "work": "503-555-0210" },
        { "mobile": "503-555-0220" },
        { "twitter": "@anakinskywalker" },
        { "skype": "AlwaysWithYou" }
    ]
}
```

The flexibility of the MongoDB document model allows for all of this to occur without downtime of the database. From an application standpoint, it can be designed to read both versions of the schema. This application change in how to handle the schema difference shouldn't require downtime either, assuming there is more than a single app server involved.

Conclusion

The Schema Versioning pattern is great for when application downtime isn't an option, updating the documents may take hours, days, or weeks of time to complete, updating the documents to the new version isn't a requirement, or a combination of any of these. It allows for a new `schema_version` field to easily be added and for the application to adjust to these changes. Additionally, it provides us as developers the opportunity to better decide when and how data migrations will take place. All of these things result in less future technical debt, another big advantage for this pattern.

As with the other patterns mentioned in this series, there are some things to consider with the Schema Versioning pattern too. If you have an index on a field that is not located at the same level in the document, you may need 2 indexes while you are migrating the documents.

One of the main benefits of this pattern is the simplicity involved in the data model itself. All that is required is to add the `schema_version` field. Then allow the application to handle and process the different document versions.

Additionally, as was seen in the use case example, we are able to combine schema design patterns together for extra performance. In this case, using the Schema Versioning and Attribute patterns together. Allowing to make schema upgrades without downtime makes the Schema Versioning pattern particularly powerful in MongoDB and could very well be enough of a reason to use MongoDB's document model versus a legacy tabular database for your next application.

The next post in this series will be a wrap up of all of the patterns we've looked at thus far and provide some additional information about use cases we've found to be particularly well suited for each pattern.

If you have questions, please leave comments below.

Previous Parts of Building with Patterns:

- The Polymorphic pattern
- The Attribute pattern
- The Bucket pattern
- The Outlier pattern
- The Computed pattern
- The Subset pattern
- The Extended Reference pattern
- The Approximation pattern
- The Tree pattern
- The Pre-allocation pattern
- The Document Versioning pattern

← Previous

Hacking For Single Mothers And Kids In Poverty

One of MongoDB's core values is "Build Together," and that value is one that reaches outside the...

April 19, 2019

Next →

Accelerating Stablecoin Innovation in US Banking

The stablecoin market has become a global center of attention, with total market capitalization surging to ove...

September 2, 2025