

MongoDB: To Use Schemas or Not?

Published Sep 30, 2015



#SCHEMA #WRITESTUFF #MONGODB

*To schema or not to schema, that is the question that Rob Ludwig takes on in this **Write Stuff** article. Do you have the **Write Stuff** to write about databases here? Why not submit your article ideas to the [Compose Write Stuff program](#).*

In a way, choosing whether or not to use a schema is analogous to choosing whether or not to use a strongly or dynamically-typed programming language: the primary tradeoff is between flexibility and rigor. Just as a dynamically-typed language allows you to quickly iterate and evolve your application, going schemaless allows you to change your data model with very little friction.

However, the strict controls and exactness imposed by schemas and by typed languages allow you to keep large groups of developers on the same page, and can allow you to catch bugs earlier in the development cycle. Every language with a full-featured ODM for MongoDB also has a library for enforcing schemas in application code, so you always have the option to include a schema in your project. To more fully understand the tradeoffs, it's worth it to take a closer look at the pros and cons of schemas.

Wild & Free: MongoDB Without A Schema

Going "schemaless" is a bit of a misnomer as there are very few production use cases where data is entirely unstructured. Most Mongo data usually has an implied structure. In fact MongoDB recently announced version 3.2 which includes some tools for [inferring a schema from your data](#) and validating it. While your data may evolve over time it usually has a common backbone in each collection that you base your queries off of. As much as people argue that going without a schema is a wild-west architecture where anything can be inserted into the database that rarely ends up being the case.

There are, actually, several reasons why you might find it preferable to not use a schema for your Mongo data. For one, Mongo's flexibility when it comes to schemas is a major reason to use MongoDB in the first place. If you're still iterating on how your application uses data, and what data it uses, you might not want to tie yourself down to a single form that data may take. Not using a schema with Mongo allows you to take full advantage of the tools Mongo offers for projecting your data as you please in whatever form you desire.

Additionally, if you have an application that collects data with varied structures into a single collection, attempting to introduce a schema is probably more trouble than it's worth. More than once, I've seen a schema evolve from an ironclad definition of what data should be to a loose collection of objects, where validation was barely possible due to the protean nature of the data being operated on. The real world is often messy: one day you're integrating with a provider who structures their API one way, the next day you've kicked them for a competitor with a completely different structure. As your business needs evolve, your data does too.

The Grand Scheme of Things: Using a Schema With Mongo

Most of the arguments for enforcing a schema on your data are well known: schemas maintain structure, giving a clear idea of what is going into the database, reducing preventable bugs and allowing for cleaner code (no more having to check the type of a field coming out of the database before using it). Schemas are a form of self-documenting code, as they describe exactly what type of data something should be, and let you know what checks will be performed. Many schema libraries allow validation functions beyond simple type checking, so you can even have schemas that ensure, for example, that a string field is a valid email address or a GeoJSON field is within the United States.

On top of these classical reasons to use schemas, there are a few MongoDB-specific reasons to use them. First off, schemas can help with some of the rough edges when querying in Mongo. Querying for subdocuments in Mongo can get frustrating because, as the documentation reminds us, "When performing equality matches on subdocuments, field order matters and the subdocuments must match exactly."

If your data has a complicated structure with multiple levels, making it predictable with a schema can make your life easier. For example, if you have employee records with an array of purchases:

```
{
  _id: 112,
  department: "r & d",
  status: "full time",
  purchases: [{
    authorizedBy: "accounting",
    receiptsSubmitted: true,
    cost: 125.99,
    items: [{
      name: "plane ticket",
      cost: 100,
    }, {
      name: "client lunch",
      provider: "Applebee' s",
      cost: 25.99
    }],
  }],
},
```

...it can quickly become difficult to reason about how to perform complex queries just by looking at the data. A schema provides some guarantees about the data format, and a way to see those

guarantees all in one place. If you're trying to find all purchases greater than \$100 authorized by your department that haven't yet submitted their receipts, it's quite a bit easier to do so with the schema in front of you.

Schemas are also particularly useful when paired with an Object Document Manager (ODM). The mongoose.js ODM for Node, as an example, is able to make intelligent use of its schemas to ensure that your Mongo queries are all set to the correct values. It's able to cast strings to ObjectIds, dates, or numbers as necessary for your query, which is much simpler and faster than managing these types by hand.

One final reason to prefer schemas with MongoDB is that they prevent you from changing the shape of your documents too often for a given collection. Updates to a Mongo document which convert data between types or sizes can be costly because Mongo has to reallocate space for the newly-resized record, resulting in a slow update. Even relatively minor examples can lead to issues. If, say, you originally stored names on the user documents, as strings:

```
{
  _id : 1,
  hostName : "Jay Leno",
  timeSlot : "20:00",
  show: "The Tonight Show"
}
```

... and you convert in a later version of the application to storing the name as a subdocument when a name is updated:

```
{
  _id : 1,
  hostName : {
    first: "Conan",
    last: "O'Brien"
  }
  timeSlot: "20:00"
  show: "The Tonight Show"
}
```

... this can cause slow writes while Mongo shifts documents to make space for the newly-resized documents. If you find that frequent changes to document size in a collection is slowing down your application, a schema can help you by forcing you to think about the changes you're making to your data.

Deciding

As you assess your options, take some time to think about the road ahead. If you already have a good sense of the data you'll be using (for example, if you're interacting with data in a fixed format, if you're implementing a standard), then a schema makes a lot of sense. Other times, the decision is not so easy. Ask yourself how agile you're planning on being as a company: Will new customers require changes to the data formats? Will you need to change partners and service providers as your needs change? Are you still fleshing out what data you're interested in keeping and what isn't

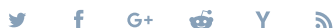
as helpful? If you answer any of those questions affirmatively, there's a strong case to be made for forgoing a schema at the present.

Finally, one important thing to consider when forgoing a schema is that it's not a permanent decision. If you decline to impose a schema in the present, the option still remains open to you in the future. At any point if your data becomes more orderly, or you find the lack of schema hinders your ability to coordinate among multiple developers, you can adopt a schema fairly trivially. Likewise, you can always open the floodgates on an application that previously had a schema and allow any type of data to come in.

Additional Resources

Most languages have a library for enforcing a schema at the application level in Mongo. Here is a selection:

- Python: There are a few schema libraries that integrate with PyMongo. Humongulus provides quite a few validation tools: <https://github.com/entone/Humongulus>
- Node: Mongoose.js is the premier Node ODM and includes extensive validation: <http://mongoosejs.com/docs/guide.html>
- Java: Spring Data's Mongo driver enforces validation by mapping documents to classes: <http://projects.spring.io/spring-data-mongodb/>
- Ruby: The excellent Mongoid project supports a whole host of schema features, from typecasting to validation: <http://docs.mongodb.org/ecosystem/tutorial/ruby-mongoid-tutorial/#fields>



Robert is a backend engineer with a background in building large cloud applications. He currently works at [RideAmigos](#), a Transportation Demand Management software platform company.



This article is licensed with [CC-BY-NC-SA 4.0](#) by Compose.



Conquer the Data Layer

Spend your time developing apps, not managing databases.


Try Compose for Free for 30 Days

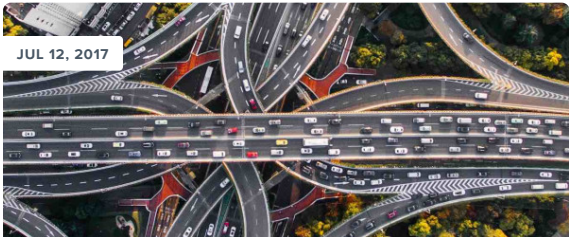
RELATED ARTICLES



Push Notifications With MongoDB


Push notifications are a staple of mobile and Internet of Things applications, and in this Write Stuff contribution Don Omond...

 Guest Author



Integration Testing Against Real Databases


Integration testing can be challenging, and adding a database to the mix makes it even more so. In this Write Stuff contribu...

 Guest Author



Simple OAuth With MongoDB & MySQL

Don Omondi, Campus Discounts' founder and CTO, discusses securing applications with OAuth and shows you how to securely store...

 Guest Author

Products

- Databases
- Pricing
- Add-Ons
- Datacenters
- Enterprise

Company

- About
- Privacy Policy

Learn

- Why Compose
- Articles
- Write Stuff
- Customer Stories
- Webinars

Support

- Support
- Contact Us

[Terms of Service](#)

[Documentation](#)
[System Status](#)

[Security](#)



© 2018 Compose, an IBM Company

