CONTENTS

Tutorial Series: How To Manage Data with MongoDB

1/16 An Introduction to Docume...

2/16 How To Use MongoDB Acc...

// Conceptual Article //

An Introduction to Document-Oriented Databases

Published on July 20, 2021 · Updated on July 20, 2021

Databases NoSQL Conceptual MongoDB

By Mateusz Papiernik
Software Engineer, CTO @Makimo

The author selected the Open Internet/Free Speech Fund to receive a donation as part of the Write for DOnations program.

Introduction

Although they were first invented decades ago, computer-based databases have become ubiquitous on today's internet. More and more commonly, websites and applications involve collecting, storing, and retrieving data from a database. For many years the database landscape was dominated by relational databases, which organize data in tables made up of rows. To break free from the rigid structure imposed by the relational model, though, a number of different database types have emerged in recent years.

These new database models are jointly referred to as *NoSQL databases*, as they usually do not use *Structured Query Language* — also known as *SQL* — which relational databases typically employ to manage and query data. NoSQL databases offer a high level of scalability as well as flexibility in terms of data structure. These features make NoSQL databases useful for handling large volumes of data and fast-paced, agile development.

This conceptual article outlines the key concepts related to document databases as well as the benefits of using them. Examples used in this article reference MongoDB, a widely-used document-oriented database, but most of the concepts highlighted here are applicable for most other document databases as well.

What is a Document Database?

Breaking free from thinking about databases as consisting of rows and columns, as is the case in a table within a relational database, document databases store data as *documents*. You might think of a document as a self-contained data entry containing everything needed to understand its meaning, similar to documents used in the real world.

The following is an example of a document that might appear in a document database like MongoDB. This sample document represents a company contact card, describing an employee called `Sammy`:

Sammy's contact card document

Copy

```
{
    "_id": "sammyshark",
    "firstName": "Sammy",
    "lastName": "Shark",
    "email": "sammy.shark@digitalocean.com",
    "department": "Finance"
}
```

Notice that the document is written as a JSON object. JSON is a human-readable data format that has become quite popular in recent years. While many different formats can be used to represent data within a document database, such as XML or YAML, JSON is one of the most common choices. For example, MongoDB adopted JSON as the primary data format to define and manage data.

All data in JSON documents are represented as field-and-value pairs that take the form of `field: value`. In the previous example, the first line shows an `_id` field with the value `sammyshark`. The example also includes fields for the employee's first and last names, their email address, as well as what department they work in.

Field names allow you to understand what kind of data is held within a document with just a glance. Documents in document databases are *self-describing*, which means they contain both the data values as well as the information on *what kind* of data is being stored. When retrieving a document from the database, you always get the whole picture.

The following is another sample document representing a colleague of Sammy's named `Tom`, who works in multiple departments and also uses a middle name:

Tom's contact card document

Copy

```
{
    "_id": "tomjohnson",
    "firstName": "Tom",
    "middleName": "William",
```

```
    "lastName": "Johnson",
    "email": "tom.johnson@digitalocean.com",
    "department": ["Finance", "Accounting"]
}
```

This second document has a few differences from the first example. For instance, it adds a new field called `middleName`. Also, this document's `department` field stores not a single value, but an array of two values: `"Finance"` and `"Accounting"`.

Because these documents hold different fields of data, they can be said to have different *schemas*. A database's schema is its formal structure, which outlines what kind of data it can hold. In the case of documents, their schemas are reflected in their field names and what kinds of values those fields represent.

In a relational database, you'd be unable to store both of these example contact cards in the same table, as they differ in structure. You would have to adapt the database schema both to allow storing multiple departments as well as middle names, and you would have to provide a middle name for Sammy or else fill the column for that row with a `NULL` value. This is not the case with document databases, which offer you the freedom to save multiple documents with different schemas together with no changes to the database itself.

In document databases, documents are not only self-describing but also their schema is *dynamic*, which means that you don't have to define it before you start saving data. Fields can differ between different documents in the same database, and you can modify the document's structure at will, adding or removing fields as you go. Documents can be also nested — meaning that a field within one document can have a value consisting of another document — making it possible to store complex data within a single document entry.

Let's imagine the contact card must store information about social media accounts the employee uses and add them as nested objects to the document:

Tom's contact card document with social media accounts information attached

Copy

```
{
    "_id": "tomjohnson",
    "firstName": "Tom",
    "middleName": "William",
    "lastName": "Johnson",
    "email": "tom.johnson@digitalocean.com",
    "department": ["Finance", "Accounting"],
    "socialMediaAccounts": [
        {
            "type": "facebook",
            "username": "tom_william_johnson_23"
```

```
        },
        {
            "type": "twitter",
            "username": "@tomwilliamjohnson23"
        }
    ]
}
```

A new field called `socialMediaAccounts` appears in the document, but instead of a single value, it refers to an array of nested objects describing individual social media accounts. Each of these accounts could be a document on its own, but here they're stored directly within the contact card. Once again, there is no need to change the database structure to accommodate this requirement. You can immediately save the new document to the database.

> **Note:** In MongoDB, it's customary to name fields and collections using a `camelCase` notation, with no spaces between words, the first word written entirely in lowercase, and any additional words having their first letters capitalized. That said, you can also use different notations such as `snake_case`, in which words are all written in lowercase and separated with underscores. Whichever notation you choose, it's considered bast practice to use it consistently across the whole database.

All these attributes make it intuitive to work with document databases from the developer's perspective. The database facilitates storing actual objects describing data within the application, encouraging experimentation and allowing great flexibility when reshaping data as the software grows and evolves.

Benefits of Document Databases

While document-oriented databases may not be the right choice for every use case, there are many benefits of choosing one over a relational database. A few of the most important benefits are:

- **Flexibility and adaptability**: with a high level of control over the data structure, document databases enable experimentation and adaptation to new emerging requirements. New fields can be added right away and existing ones can be changed any time. It's up to the developer to decide whether old documents must be amended or the change can be implemented only going forward.
- **Ability to manage structured and unstructured data**: as mentioned previously, relational databases are well suited for storing data that conforms to a rigid structure. Document databases can be used to handle structured data as well, but they're also

quite useful for storing unstructured data where necessary. You can imagine structured data as the kind of information you would easily represent in a spreadsheet with rows and columns, whereas unstructured data is everything not as straightforward to frame. Examples of unstructured data are rich social media posts with human-generated texts and multimedia, server logs that don't follow unified format, or data coming from a multitude of different sensors in smart homes.

- **Scalability by design**: relational databases are often write constrained, and increasing their performance requires you to *scale vertically* (meaning you must migrate their data to more powerful and performant database servers). Conversely, document databases are designed as distributed systems that instead allow you to *scale horizontally* (meaning that you split a single database up across multiple servers). Because documents are independent units containing both data and schema, it's relatively trivial to distribute them across server nodes. This makes it possible to store large amounts of data with less operational complexity.

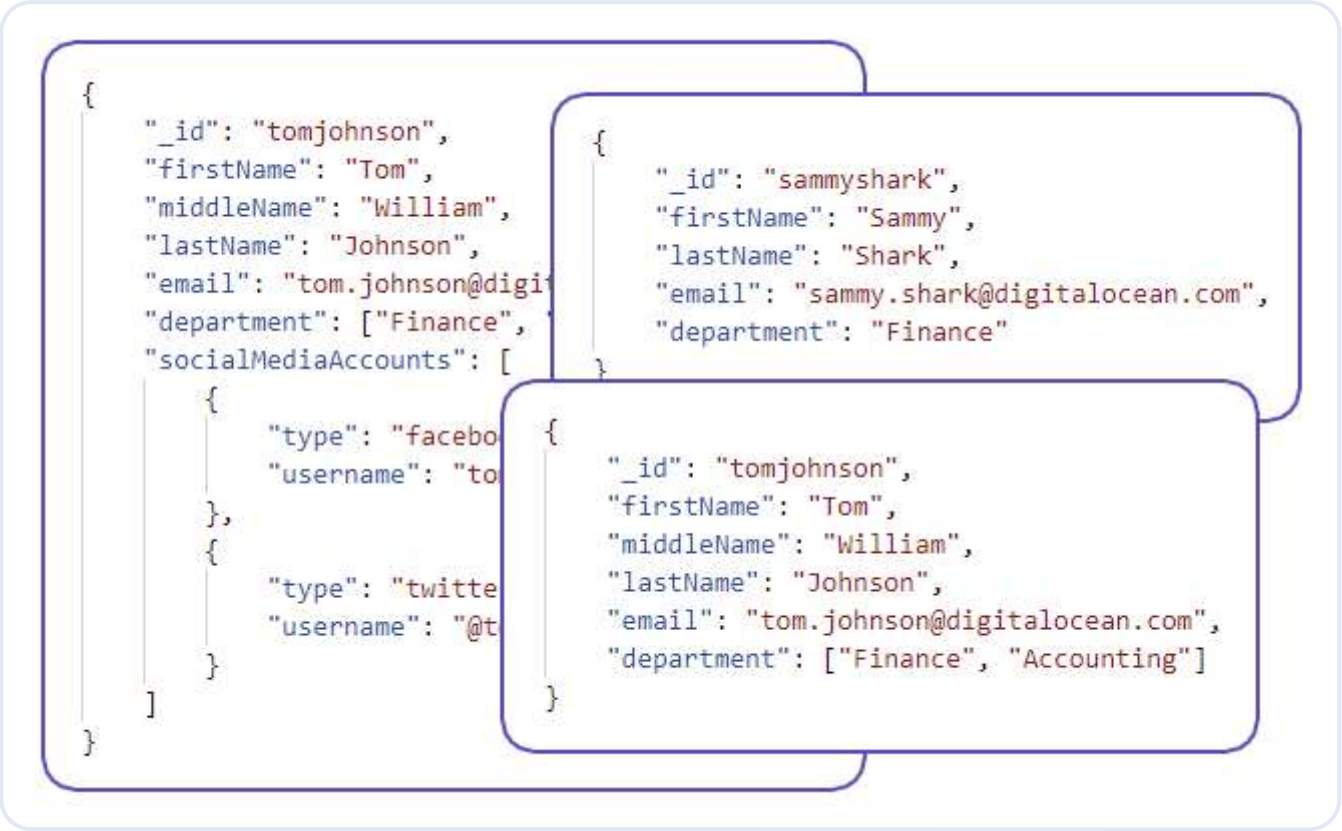
In real-world applications, both document databases and other NoSQL and relational databases are often used together, each responsible for what it's best suited for. This paradigm of mixing various types of databases is known as *polyglot persistence*.

Grouping Documents Into Collections

While document databases allow great flexibility in how the documents are structured, having some means of organizing data into categories sharing similar characteristics is crucial for ensuring that a database is healthy and manageable.

Imagine a database as an individual cabinet in a company archive with many draws. For example, one drawer might keep records of employment contracts, with another keeping agreements with business partners. While it is technically possible to put both kinds of documents into a single drawer, it would be difficult to browse the documents later on.

In a document database, such drawers are often called *collections*, logically similar to *tables* in relational databases. The role of a collection is to group together documents that share a similar logical function, even if individual documents may slightly differ in their schema. For instance, say you have one employment contract for a fixed-term and another that describes a contractor's additional benefits. Both documents are employment contracts and, as such, it could make sense to group them into a single collection:



Note: While it's a popular approach, not all document databases use the concept of collections to organize documents together. Some database systems use tags or tree-like hierarchies, others store documents directly within a database with no further subdivisions. MongoDB is one of the popular document-oriented databases that use collections for document organization.

Having similar characteristics between documents within a collection also allows you to build indexes in order to allow for more performant retrieval of documents based on queries related to certain fields. Indexes are special data structures that store a portion of a collection's data in a way that's faster to traverse and filter.

As an example, you might have a collection of documents in a database that all share a similar field. Because each document shares the same field, it's likely you would often use that field when running queries. Without indexes, any query asking the database to retrieve a particular document requires a collection scan — browsing all documents within a collection one by one to find the requested match. By creating an index, however, the database only needs to browse through indexed fields, thereby improving query performance.

Data Types and Schema Validation

While we mentioned that document-oriented databases can store documents in different formats, such as XML, YAML or JSON, these are often further extended with additional traits that are specific to a given database system, such as additional data types or structure validation features.

For example, MongoDB internally uses a binary format called BSON (short for Binary JSON) instead of a pure JSON. This not only allows for better performance, but it also extends the format with data types that JSON does not support natively. Thanks to this, we can reliably store different kinds of data in MongoDB documents without being restricted to standard JSON types and use filtering, sorting, and aggregation features specific to individual data types.

The following sample document uses several different data types supported by MongoDB:

Copy

```
{
    "_id": ObjectId("5a934e000102030405000000"),
    "code": NumberLong(2090845886852),
    "image": BinData(0, "TGVhcm5pbmcgTW9uZ29EQg=="),
    "lastPurchased": ISODate("2021-01-19T06:01:17.171Z"),
    "name": "Document database sticker",
    "price": NumberDecimal("13.23"),
    "quantity": 317,
    "tags": [
        "stickers",
        "accessories"
    ]
}
```

Notice that some of these data types not typical to JSON, such as decimal numbers with exact precision or dates which are represented as objects, such as `NumberDecimal` or `ISODate`. This ensures that these fields will always be interpreted properly and not mistakenly cast to another similar data type, like a decimal number being cast to a regular double.

This variety of supported data types, combined with schema validation features, makes it possible to implement a set of rules and validity requirements to provide your document database structure. This allows you to model not only unstructured data, but to also create collections of documents following more rigid and precise requirements.

Conclusion

Thanks to their flexibility, scalability, and ease of use, document databases are becoming an increasingly popular choice of database for application developers. They are well suited

to different applications and work well on their own or as a part of bigger, multi-database ecosystems. The wide array of document-oriented databases has distinct advantages and use cases, making it possible to choose the best database for any given task.

You can learn more about document-oriented databases and other NoSQL databases from DigitalOcean's community articles on that topic.

To learn more about MongoDB in particular, we encourage you to follow this tutorial series covering many topics on using and administering MongoDB and to check the official MongoDB documentation, a vast source of knowledge about MongoDB as well as document databases in general.

Thanks for learning with the DigitalOcean Community. Check out our offerings for compute, storage, networking, and managed databases.

Learn more about us →

Next in series: How To Use MongoDB Access Control →

Tutorial Series: How To Manage Data with MongoDB

MongoDB is a document-oriented NoSQL database management system (DBMS). Unlike traditional relational DBMSs, which store data in tables consisting of rows and columns, MongoDB stores data in JSON-like structures referred to as *documents*.

This series provides an overview of MongoDB's features and how you can use them to manage and interact with your data.

Subscribe

Databases NoSQL Conceptual MongoDB

Browse Series: 16 articles

Expand to view all

About the authors

Mateusz Papiernik Author

Software Engineer, CTO @Makimo

Creating bespoke software ◦ CTO & co-founder at Makimo. I'm a software enginner & a geek. I like making impossible things possible. And I need tea.

Mark Drake Editor

Manager, Developer Education

Technical Writer @ DigitalOcean

Still looking for an answer? Ask a question

Search for more help

Was this helpful? Yes No

Comments

1 Comments

Leave a comment...

This textbox defaults to using `Markdown` to format your answer.

You can type `!ref` in this text area to quickly search our full set of tutorials, documentation & marketplace offerings and insert the link!

Sign In or Sign Up to Comment

LargYacht • March 16, 2022

Thank you friend

Reply

This work is licensed under a Creative Commons Attribution-NonCommercial- ShareAlike 4.0 International License.

Try DigitalOcean for free

Click below to sign up and get **$200 of credit** to try our products over 60 days!

Sign up

Popular Topics

Ubuntu

Linux Basics

JavaScript

Python

MySQL

Docker

Kubernetes

All tutorials →

Talk to an expert →

Get our biweekly newsletter

Sign up for Infrastructure as a Newsletter.

Sign up →

Hollie's Hub for Good

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.

Learn more →

Become a contributor

You get paid; we donate to tech nonprofits.

Learn more →

Featured on Community

Kubernetes Course Learn Python 3 Machine Learning in Python

Getting started with Go Intro to Kubernetes

DigitalOcean Products

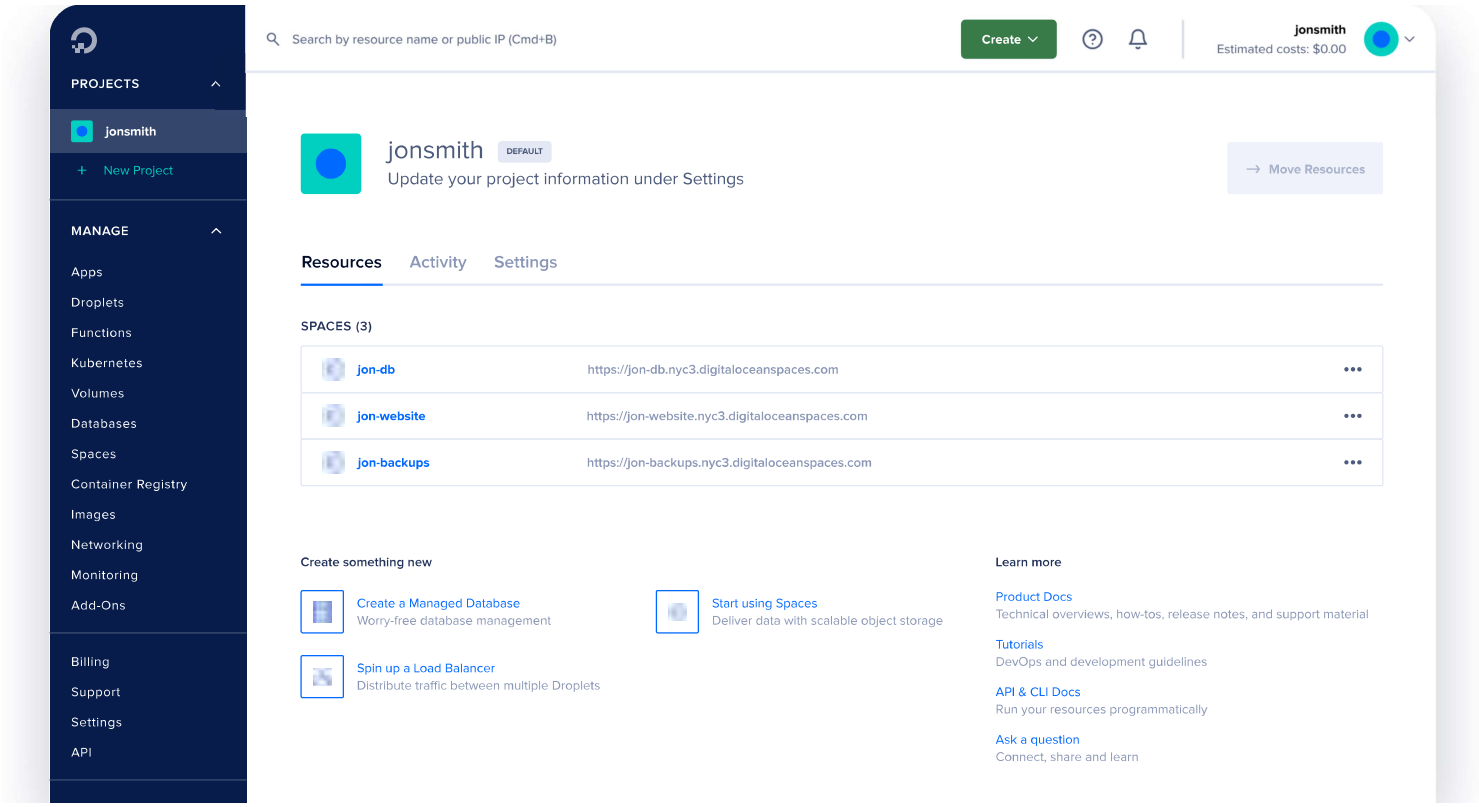
Cloudways Virtual Machines Managed Databases Managed Kubernetes

Block Storage Object Storage Marketplace VPC Load Balancers

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow – whether you're running one virtual machine or ten thousand.

Learn more →



Get started for free

Enter your email to get $200 in credit for your first 60 days with DigitalOcean.

Email address

Send My Promo

New accounts only. By submitting your email you agree to our Privacy Policy.

Company

Products

Community

Solutions

Contact

© 2023 DigitalOcean, LLC.