mongoDB®

Working With Sensitive Data While Keeping It Secure: A Guide to Data Masking

JULY 2021

mongoDB.

Introduction

Your database stores the information necessary to run your applications — often, vast quantities of it. But just as important as the quantity of information is the sensitive nature of that data. Often, the information in your database needs to be kept private, for reasons both legal and ethical.

Because a database is designed with security in mind — and rightly so — accessing this data can be a challenge. To protect your data from attack, you may have isolated your database from the rest of your network or limited access to only certain developers.

However, there are circumstances in which you'll need to expose that information, and it can be difficult to do so safely. Sometimes you'll need production data to help develop new features or build tests to replicate user behavior. No matter how well code might appear to run, it often turns out that local logic can't account for real-life behavior. In order to re-create production scenarios in a local development environment, you may need to get the actual data — but without exposing that information publicly.

One solution is to incorporate data masking into your workflow. Data masking is a well-established approach to protecting sensitive data in a database while still allowing the data to be usable. Data masking manipulates the original data so that the result is obfuscated and can't easily be tracked back to its initial value.

mongoDB.

Data masking: A flexible solution for privacy and compliance challenges

Data masking is useful for more than just local testing. Some government regulations, such as HIPAA or GDPR, require you to maintain the anonymity and confidentiality of the source user while you're processing data. Exposing personally identifiable information (PII) can have serious legal consequences, even if the information is leaked in a location you believe to be internal (such as server logs). Data masking can make it possible for you to provide some kind of open database access to your users, perhaps via an API, while obfuscating information they shouldn't be able to access.

Rather than null out values, you can use data masking to return results that resemble, but don't duplicate, the real thing. You can even set up a robust test suite that operates on a clone of real production data rather than on generated test fixtures. Data masking also will ensure that you're not introducing an attack vector against your CI service.

Data masking can be applied in one of two ways:

- Persistent data masking permanently modifies original records
- Dynamic data masking allows you to alter data on an as-needed basis

Masked data still can be useful for analytics, because you can mask the data by changing the original value by a small random percentage. In a large data set, these small variances will cancel each other out, allowing you to derive similar patterns and trends from the masked data as you would from the original.

In this paper, we'll talk about how MongoDB natively supports both forms of data masking, and we'll walk you through the process of protecting data in a variety of use cases.

A DEEP DIVE INTO DATA MASKING

To learn more about data masking, and to see examples of it in use, visit:

Restricted View, from *Practical MongoDB Aggregations*

Mask Sensitive Fields from *Practical MongoDB Aggregations*

Reversible Data Masking a MongoDB technical blog

Aggregation Pipeline Stages MongoDB documentation

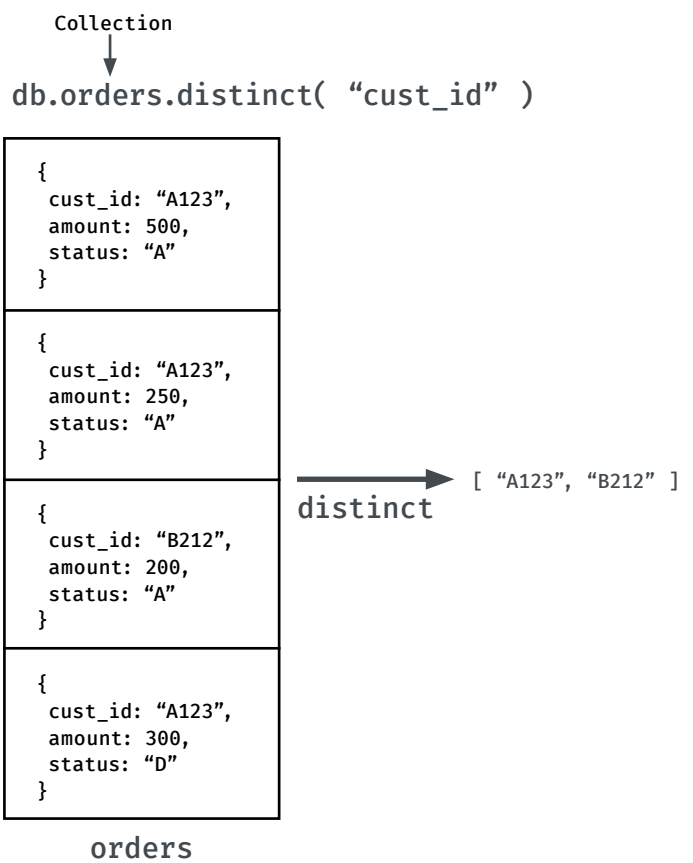
GitHub Repo with data masking examples

Using an aggregation pipeline to mask data

MongoDB supports data masking via the core feature of an aggregation pipeline, as shown below. You can think of an aggregation pipeline as a static function: The input is a MongoDB BSON document, and the output is the result of a defined data transformation sequence. After defining an aggregation pipeline, it is optimized to run efficiently and effectively across all your shards without any additional administration.

An aggregation pipeline is a sequence broken down into two stages:

- A `$match` stage, where you identify which documents to modify
- A `$group` stage, which takes a document field and performs some modification on its value



Aggregation pipeline

This makes the aggregation pipeline a perfect tool for data masking, because it's essentially a one-way (irreversible) pipeline. Documents go in readable and come out masked.

Data masking is not a special concept applicable only to NoSQL databases such as MongoDB. However, for relational database management systems, such as MySQL, data masking is provided by plugins and implemented through SQL queries. This requires additional database configuration and is prone to errors, because you need to remember to request a mask for every query. By making data masking a first-class feature carried out by way of an aggregation pipeline, MongoDB provides a standardized way to implement masking as a layer that responses pass through. Because pipelines are written in your programming language of choice, they also provide a reusable mechanism separate from the actual issued queries.

Aggregation pipelines are extremely performant. After you write one, MongoDB optimizes its performance for massive data sets, such as a cluster with billions of records. Because NoSQL databases are more amenable to partitioning, the aggregation pipeline is built to transparently process matches and groupings in parallel across each shard, reducing the overall completion time.

To better understand all the commands that are available when modifying selections, you may want to familiarize yourself with MongoDB's aggregation pipeline stages.

mongoDB.

Data masking in practice

Suppose you run an ecommerce business and you offer to store credit card information, making checkouts easier for users. You want to create a system that masks this sensitive data, so that your developers can build out tests to validate information provided by the user.

The hardest part of data masking is identifying all of the data that needs to be masked. With a credit card, this might seem like a no-brainer: It's just the 16-digit number that needs to be hidden, right? Well, not exactly. The whole point of data masking—and indeed, of adhering to privacy and compliance regulations such as GDPR — is to remove all personally identifiable information. With that in mind, to properly obfuscate credit card information, you might need to mask:

- The card number
- The card's security code
- The card owner's name
- The card's expiration date

That's just for the card itself. You may need to go further and mask information about purchases, so that they cannot be linked to any specific individual. This could include:

- The transaction date
- The date a payment was settled
- The payment amount

When in doubt, you should always opt to over-mask rather than risk leaking anything.

Once you've assembled such a list, the next step is to determine a consistent pattern for masking the data. Often, this involves taking a look at the data itself and coming up with a change that fits the original pattern while still concealing it effectively. For example:

- Record the first 12 digits of a credit card as X's, and keep the last four unmodified
- Swap the three-digit security code for three random numbers
- Replace the letters in someone's first name with X's, but keep the last name unchanged
- Add or subtract a fixed number of hours or days to transactional date-time values

The actual procedure for masking the data is less important than properly documenting the fields that should be masked.

With that established, let's see what a data processing pipeline could look like.

First, let's assume that we have a collection called *payment*, and we add in a couple of records, like so:

```
db.payments.insertMany([
    {
        'card_num': '1234567890123456',
        'card_sec_code': '123',
        'card_name': 'Mrs. Jane A. Doe',
        'card_expiry': ISODate('2023-08-31T23:59:59Z'),
        'transaction_date': ISODate('2021-01-13T09:32:07Z'),
        'settlement_date': ISODate('2021-01-21T14:03:53Z'),
        'transaction_amount': NumberDecimal('4255.16')
    },
    {
        'card_num': '9876543210987654',
        'card_sec_code': '987',
        'card_name': 'Jim Smith',
        'card_expiry': ISODate('2022-12-31T23:59:59Z'),
        'transaction_date': ISODate('2020-11-24T19:25:57Z'),
        'settlement_date': ISODate('2020-12-04T11:51:48Z'),
        'transaction_amount': NumberDecimal('76.87'),
    },
]);
```

In a real-world scenario, we might have even more fields, such as currency codes or transaction IDs.

Then we begin to construct our pipeline. Given a field name, we will either:

- Perform an operation, such as hiding the first 12 digits of `card_num`; or
- Generate a value that fits the existing pattern, such as generating three random digits for `card_sec_code`

Such a process might look like this:

```
// PART 1 OF PIPELINE
var simpleMasksPt1Stage = {
   // 1. PARTIAL TEXT OBFUSCATION RETAINING LAST NUMBER OF CHARS,
eg: '1234567890123456' -> 'XXXXXXXXXXXX3456'
    'card_num': {'$concat': [
                    'XXXXXXXXXXXX',
                    {'$substrCP': ['$card_num', 12, 4]},
                ]},

   // 2. FULL TEXT REPLACEMENT WITH RANDOM VALUES, eg: '133' -> '472'
    'card_sec_code': {'$concat': [
                        {'$toString': {'$floor': {'$multiply':
[{'$rand': {}}, 10]}}},
                        {'$toString': {'$floor': {'$multiply':
[{'$rand': {}}, 10]}}},
                        {'$toString': {'$floor': {'$multiply':
[{'$rand': {}}, 10]}}},
                    ]},

   // 3a. PARTIAL TEXT OBFUSCATION RETAINING LAST WORD, eg:
'Mrs. Jane A. Doe' -> 'Mx. Xxx Doe'  (needs post-processing in a
subsequent pipeline stage)
    'card_name': {'$regexFind': {'input': '$card_name', 'regex':
/(\S+)$/}},

   // 4. PARTIAL DATE OBFUSCATION BY ADDING OR SUBTRACTING A RANDOM
TIME AMOUNT UP TO ONE HOUR MAX
    'transaction_date': {'$add': [
                            '$transaction_date',
                            {'$floor': {'$multiply': [{'$subtract':
[{'$rand': {}}, 0.5]}, 2*60*60*1000]}},
                        ]},

   // 5. FULL DATE REPLACEMENT BY TAKING AN ARBITRARY DATETIME OF
01-Jan-2021 AND ADDING A RANDOM AMOUNT UP TO ONE YEAR MAX
    'settlement_date': {'$add': [
                            {'$dateFromString': {'dateString':
'2021-01-01T00:00:00.000Z'}},
                            {'$floor': {'$multiply': [{'$rand': {}},
365*24*60*60*1000]}},
                        ]},
```

```
   // 6. FULL DATE REPLACEMENT BY TAKING THE CURRENT DATETIME AND
ADDING A RANDOM AMOUNT UP TO ONE YEAR MAX
    'card_expiry': {'$add': [
                        '$$NOW',
                        {'$floor': {'$multiply': [{'$rand': {}},
365*24*60*60*1000]}},
                    ]},

   // 7. PARTIAL NUMBER OBFUSCATION BY ADDING OR SUBTRACTING A
RANDOM PERCENT OF ITS VALUE, UP TO 10% MAX
    'transaction_amount': {'$add': [
                            '$transaction_amount',
                            {'$multiply': [{'$subtract': [{'$rand': {}},
0.5]}, 0.2, '$transaction_amount']},
                        ]},
};
```

As you can see, this pipeline is just a dictionary, with the field name as a key and the operation as the value. The order of the fields, and when they're modified, doesn't really matter.

In most of these cases, the operations are simple `$concat`s or `$add`s. However, replacing a person's name takes a bit more work. We've made a regex match, but we need an additional stage to perform the actual concat:

```
// PART 2 OF PIPELINE
var simpleMasksPt2Stage = {
    // 3b. PARTIAL TEXT OBFUSCATION RETAINING LAST WORD (post pro-
cessing from previous regex operation to pick out 'match')
    'card_name': {'$concat': ['Mx. Xxx ', {'$ifNull': ['$card_name.
match', 'Anonymous']}]},
};
```

Here, we're combining the text `Mx. Xxx` with the value of `$card_name.match`. It's a bit tricky to do all of this in one step, which is why we've split it into two stages.

A pipeline can have as many stages as necessary. When you've finished building it, you can bring it all together in a single array:

```
// BRING FULL PIPELINE TOGETHER
var pipeline = [
    {'$set': simpleMasksPt1Stage},
    {'$set': simpleMasksPt2Stage}
];
```

How to safely expose masked data

There are several ways to safely expose data. Each has its own performance and security implications.

Option 1: Exposing the data-masked aggregation on demand

This option exposes the masked data on an "as needed" basis. A trusted application can simply call a single method on the database to generate and return the masked versions of all the records:

```
db.payments.aggregate(pipeline);
```

Although this option is the quickest to implement, it can be considered less secure. If you choose this method, ensure that the application providing the data is not exposed to clients that shouldn't be able to access it.

Option 2: Exposing a data-masked read-only view

MongoDB uses the concept of views to define the categories of data a client is allowed to see. If your view is secured by role-based access controls (RBACs), then only users with the appropriate access level will be able to view the data provided. In addition, collection-level access controls allow administrators to grant users privileges scoped to a specific collection in a specific database.

Creating and querying against a view is simple:

```
db.createView('payments_redacted_view', 'payments', pipeline);
db.payments_redacted_view.find();
```

This often is the best option if you're exposing data outside your network.

Option 3: Exposing a data-masked copy of original data

You can choose to produce a brand-new, masked-data-only collection, which can be completely isolated from the raw source data. To set this up, define a new merge pipeline to copy, and modify the data from one source to the other:

```
new_pipeline = [].concat(pipeline);  // COPY THE ORIGINAL PIPELINE
new_pipeline.push(
    {'$merge': {'into': { 'db': 'testdata', 'coll':
'payments_redacted'}, 'on': '_id',  'whenMatched': 'fail',
'whenNotMatched': 'insert'}}
);
db.payments.aggregate(new_pipeline);
db.payments_redacted.find();
```

This option completely separates the concerns of the original data from the redacted version, but at the cost of more storage space.

Option 4: Overwriting the original data with data-masked values

This is the most aggressive choice, because it destroys the unmodified source data. It resembles the earlier pipeline, except that the matched data is replaced rather than inserted:

```
replace_pipeline = [].concat(pipeline);  // COPY THE ORIGINAL
PIPELINE
replace_pipeline.push(
    {'$merge': {'into': { 'db': 'testdata', 'coll': 'payments'},
'on': '_id',  'whenMatched': 'replace', 'whenNotMatched': 'fail'}}
);
db.payments.aggregate(replace_pipeline);
db.payments.find();
```

This can be useful if you absolutely do not need the source data in its original form, but you would also need to make sure that no other processes are changing data as this pipeline runs. Otherwise, it might result in a database in an incomplete state.

mongoDB.

Data masking resources

Data masking is a powerful, flexible technique that allows you to grant access to look-alike data serving as a convenient stand-in for the real thing. With data masking, your development and engineering teams get access to a simulacrum of data that lets them be productive, without running the risk of a leak. As you develop your own data masking techniques, the common obfuscation practices and patterns listed above should serve as helpful guidelines.

If you'd like to see more examples of data masking with MongoDB, our GitHub repo expands on many of the code snippets demonstrated here. With these resources and a careful analysis of the data you need to protect, data masking will provide a flexible tool that enables you and your teams to work more efficiently and securely.

Explore data masking in MongoDB Atlas. Start free.