



MongoDB - Administration

Ailton Moreira
ailton.moreira@openup.pt

VN Gaia 2024



We Work With Data

We Work With
Big Data & Data Science

We are Data
Scientists



Consulting



Support



Services



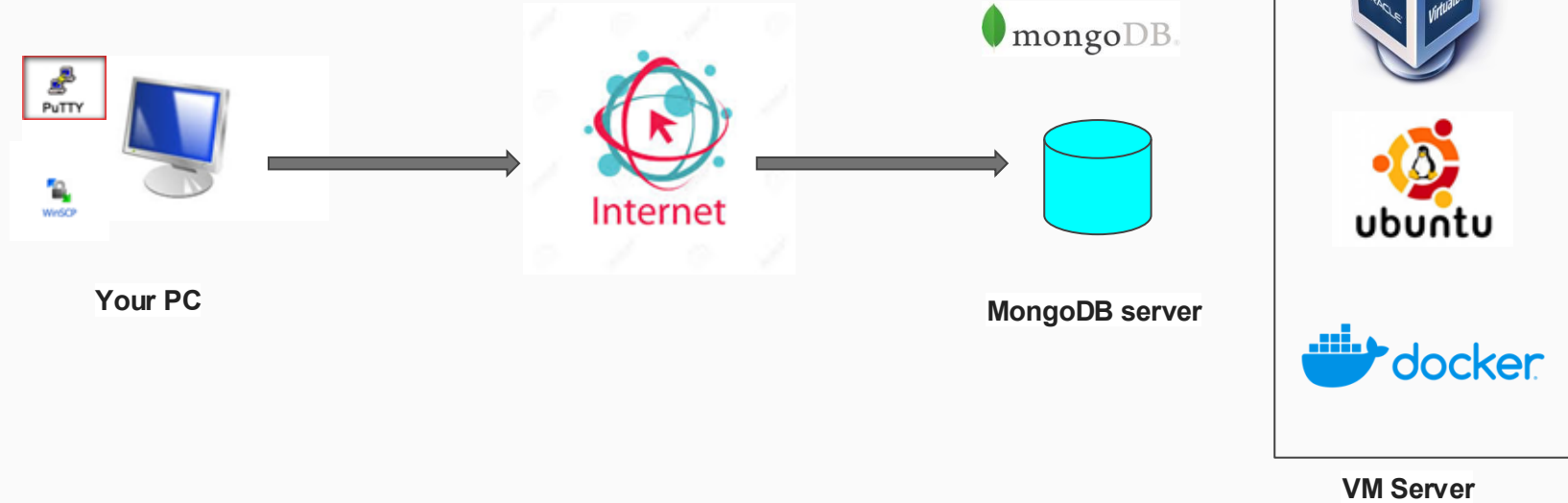
Training

ORACLE
DATABASE



- PhD Candidate in Technology and Information Systems;
- Invited Assistant Professor at the University of Minho;
- Worked with several database systems like OracleDB, MySQL, PostgreSQL, MS SQL Server, Cassandra, Redis...
- Today I work with SQL and NoSQL databases, like MongoDB, Cassandra, Neo4j...
- Work on many projects in the healthcare domain;
- Worked at Bosch Car Media Portugal on project Factory of Future (FoF – Real-time data processing).

- Briefly introduce yourself:
 - Name
 - Title or position
 - Company
 - Experience with database systems
 - Expectations

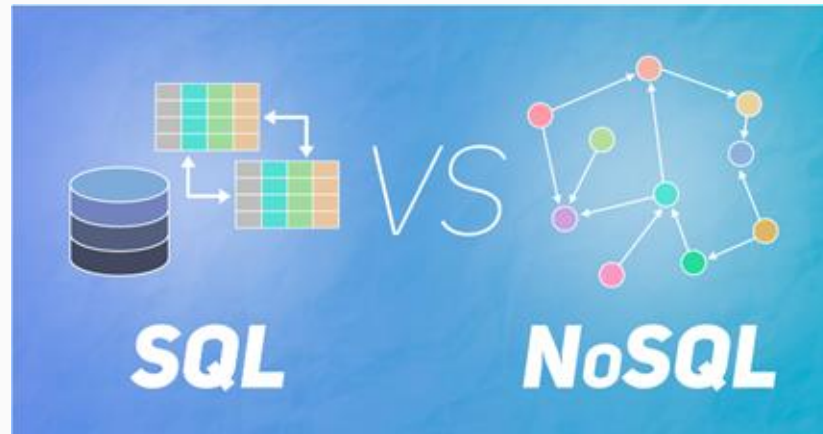


- Beyond Relational Databases
- Introducing MongoDB
- Installing MongoDB
- MongoDB Query Language
- MongoDB Data Modeling
- Application Design with MongoDB
- Data Migration and Indexing
- MongoDB Administration and Maintenance
- Monitoring
- Performance Tuning

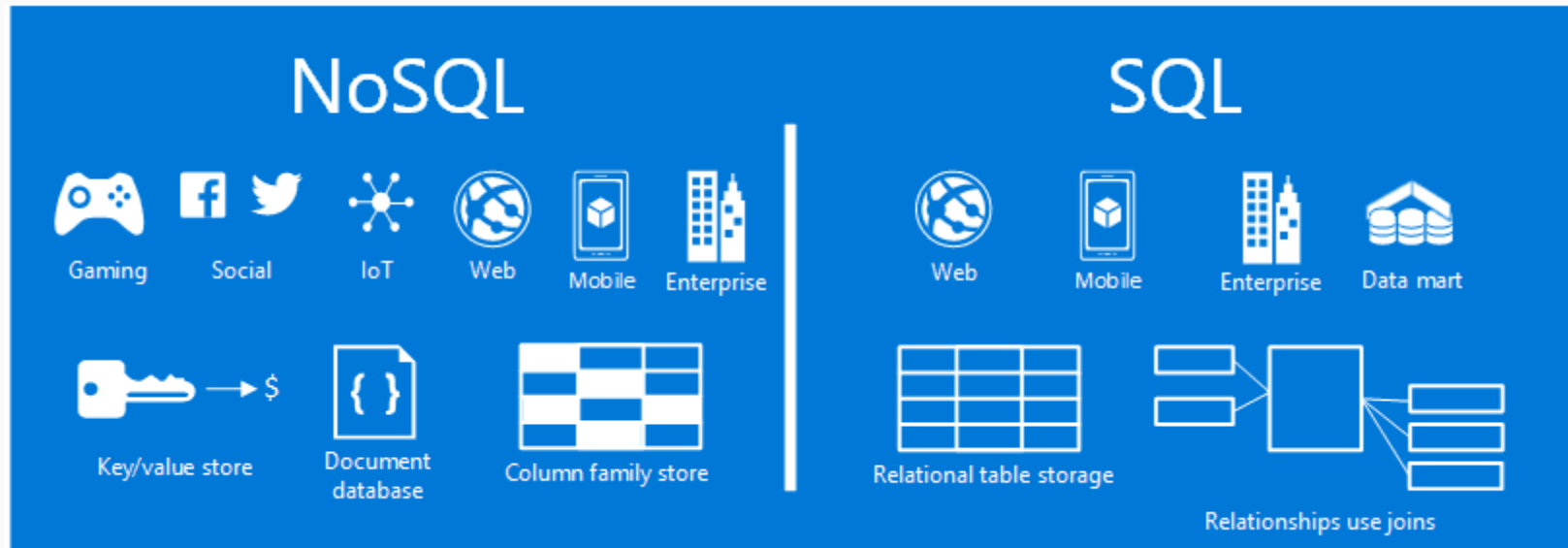


- Types of Databases, examples

- Relational Database
- NoSQL Database
- Spatial Database
- Unstructured Database
- Data Warehouse



- NoSQL vs SQL Databases



- NoSQL vs **SQL Databases**
 - SQL databases are relational databases that store data in tables with rows and columns.
 - They use a structured schema, meaning data must be organised into predefined tables with fixed columns.
 - SQL databases are typically used for applications where data consistency and integrity are paramount, such as financial systems or traditional transactional applications.
 - SQL databases excel at handling complex queries and transactions.
 - Examples of SQL databases include MySQL, PostgreSQL, Oracle, and Microsoft SQL Server.

- **NoSQL** vs SQL Databases
- No SQL stands for “Not Only SQL”
- NoSQL databases are non-relational databases that can store and handle unstructured, semi-structured, or structured data.
- They offer flexible schemas, allowing the storage of data in various formats without the need for a predefined schema.
- NoSQL databases are often chosen for applications that require scalability, high availability, and fast performance, such as web applications, IoT systems, and real-time analytics.

- **NoSQL vs SQL Databases**
- A No SQL database is used to store large quantities of complex and diverse data, such as product catalogs, logs, user interactions, analytics, and more Designed for scalability and high availability.
- NoSQL databases come in various types: document-oriented, key-value stores, column-oriented, and graph databases.
- NoSQL databases are well-suited for handling large volumes of data and distributing workloads across multiple nodes.
- Examples of NoSQL databases include MongoDB, Cassandra, Couchbase, Redis, and Amazon DynamoDB.

- **NoSQL** vs SQL Databases

Feature	RDBMS	NoSQL
Schema	Fixed schema, predefined structure	Flexible schema, dynamic structure
Data Model	Tabular (tables, rows, columns)	Various models (document, key-value, column-family, graph)
Normalization	Heavy use of normalization	Encourages denormalization
Scaling	Vertical scaling	Horizontal scaling

- Why NoSQL?
 - **Flexible Schema:** Agile development, changing requirements
 - **Scalability:** Horizontal scaling for big data, high-traffic apps
 - **High Performance:** Optimization for fast read/write operations
 - **Variety of Data Models:** Choose the best-fit data model for the use case
 - **Distributed Architecture:** Fault tolerance, high availability

NoSQL databases offer flexibility, scalability, and high-performance
Ideal for modern applications with dynamic data needs



We Work With Data

Thank you!

ailton.moreira@openup.pt



MongoDB – Administration

Introducing MongoDB

Ailton Moreira
ailton.moreira@openup.pt

VN Gaia 2024

- Understand how to handle data migration and indexing in MongoDB.

- Loading Bulk Data

Bulk data loading is the process of importing large volumes of data into a MongoDB database efficiently.

Tools:

mongoimport: Command-line tool for importing JSON, CSV, or TSV data.

mongorestore: Command-line tool for restoring data from binary BSON files.

```
mongoimport --db <mydatabase> --collection <mycollection> --file data.json --jsonArray
```

Migrating historical sales data from a CSV file into MongoDB

```
mongoimport --db salesdb --collection sales --type csv --headerline --file sales_data.csv
```

- JSON Format Import and Export

Importing and exporting data in JSON format is crucial for data exchange and backup purposes.

Tools:

mongoexport: Export data to JSON or CSV.

mongoimport: Import data from JSON or CSV.

Exporting to JSON

```
mongoexport --db <mydatabase> --collection <mycollection> --out data.json
```

Importing from JSON

```
mongoimport --db <mydatabase> --collection >mycollection> --file data.json --jsonArray
```

- JSON Format Import and Export

Importing and exporting data in JSON format is crucial for data exchange and backup purposes.

Backup and restore a MongoDB collection.

Export data before making major changes.

```
mongoexport --db <mydatabase> --collection <mycollection> --out  
/tmp/backups/backup.json
```

Restore data if necessary.

```
mongoimport --db <mydatabase> --collection <mycollection> --file backup.json --  
jsonArray
```

- Using Primary Indexes

A primary index is an index on the `_id` field, which is automatically created and unique for each document.

Ensures fast retrieval of documents by their unique identifier.

Primary Index on `_id`

MongoDB automatically indexes the `_id` field.

Retrieving user details by their unique user ID.

```
db.users.find({ _id: ObjectId("60b8d295f1c2b2d9a6e8dcd7") })
```

- Using Secondary Indexes

Secondary indexes are additional indexes on fields other than `_id` to support efficient queries.

Types:

Single Field Index

Compound Index

Multikey Index

Geospatial Index

Text Index

Hashed Index

- Using Secondary Indexes

Creating a Single Field Index

```
db.collection.createIndex({ fieldname: 1 })
```

Creating a Compound Index

```
db.collection.createIndex({ field1: 1, field2: -1 })
```

Optimize query performance for frequent searches.

Analyze query patterns.

Create indexes on fields used in queries.

```
db.orders.createIndex({ customerId: 1, orderDate: -1 })
```

- Exercises use the MongoDB tools provided early to import/export data from the database and create a compound index on one collection



We Work With Data

Thank you!

ailton.moreira@openup.pt



MongoDB – Administration

Introducing MongoDB

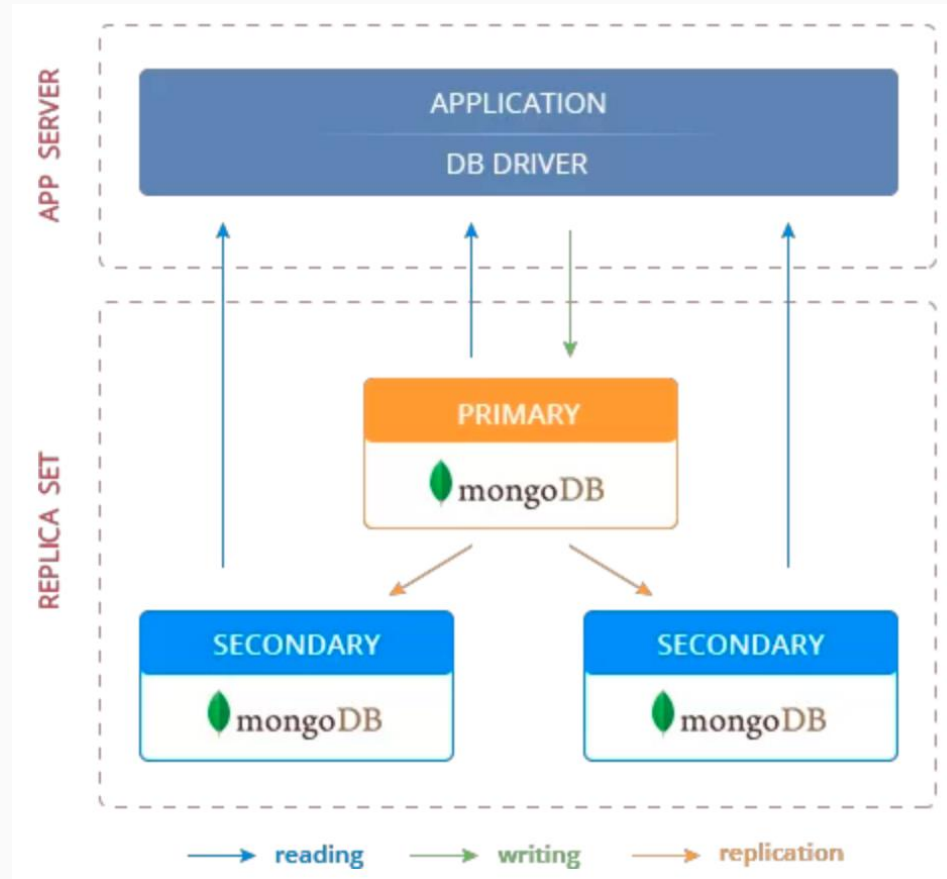
Ailton Moreira
ailton.moreira@openup.pt

VN Gaia 2024

What is replication?

- Replication is a way of keeping identical copies of your data on multiple servers
- With MongoDB, you set up replication by creating a replica set
- A replica set is a group of servers with one primary, the server taking writes, and multiple secondaries, servers that keep copies of the primary's data
- If the primary crashes, the secondary scan elects a new primary from amongst themselves.
- MongoDB Atlas provides replica sets for test, dev and production

Important points to consider



Important point to consider

- The members of the replica set should be started with ip other than local ip.

Use `--bind_ip` option

- Enable authentication and authorization
- `rs` is a global variable, containing methods to administrate replica sets
- Replica sets are limited to 50 members in total and only 7 voting members
- If you are creating a replica set that has more than seven members, every additional member must be given zero votes
- ```
> rs.add({"_id" : 7, "host" : "10.17.57.10:27025", "votes" : 0})
```

- Setting up replication
  - In this configuration, we have 3 servers. For simplicity we are starting all three servers from the same docker but on three different ports
  - Server lists
    - 10.17.57.10 27018
    - 10.17.57.10 27019
    - 10.17.57.10 27020
  - Create all the directories, required for the data directory and log the file
  - `$ mkdir /u01/data1 /u01/data2/ /u01/data3`

- Setting up replication
  - Start all the three mongod process
    - `mongod --bind_ip localhost,10.17.57.10 --replSet rs0 --dbpath/u01/data1 --port 27018 --oplogSize=200 --logpath/u01/data1/mongo.log --fork`
    - `mongod --bind_ip localhost,10.17.57.10 --replSet rs0 --dbpath/u01/data2 --port 27019 --oplogSize=200 --logpath/u01/data2/mongo.log --fork`
    - `mongod --bind_ip localhost,10.17.57.10 --replSet rs0 --dbpath/u01/data3 --port 27020 --oplogSize=200 --logpath/u01/data3/mongo.log --fork`

- Setting up replication

- Connect to one of the mongod processes, and create a variable with member information and replica set name. Make sure to use the replica set name used while starting up the mongod servers

```
rconfig={
 _id:"rs0",
 members:[
 {_id:0,host:"10.17.57.10:27018"},
 {_id:1, host:"10.17.57.10:27019"},
 {_id:2, host:"10.17.57.10:27020"}
]
}
```

- Initiate the replication configuration

- > rs.initiate(rconfig)

- Setting up replication
  - Replica set will elect a primary soon
  - The mongo shell prompt displays if it's primary or secondary
  - `> rs.status()` will show the status of the replication configuration
  - `rs0:PRIMARY>` indicates, you are connected to the primary server of "rs0", replication configuration
  - `> db.isMaster()` helps to determine if we have a connection to the primary. Mostly used in programming
  - Both commands show the list of all the members of the replica set

- Reading from the secondary
  - By default reading from secondary is not enabled.
    - "errmsg" : "not master and slaveOk=false",
  - To solve
    - Connect to secondary
    - > rs.secondaryOk()
  - Try to select documents which were inserted on primary
    - > use movies
    - > db.movies.find()

- Testing the failover secondary
  - Connect to primary and shut it down
    - use admin
    - `> db.shutdownServer()`
  - Connect to any other secondary
    - `> rs.status()`
    - Look for `"stateStr" : "PRIMARY"`, to understand which is the current primary

- Adding & Removing members

- Start a new mongod

- `mongod --bind_iplocalhost,10.17.57.10 --replSet rs0 --dbpath/u01/data4 --port 27021 --oplogSize=200 --logpath/u01/data4/mongo.log --fork`

- Connect to primary

- Add the new member

- `> rs.add("10.17.57.10:27021")`

- Check the status of the configuration

- `> rs.config()`

- Remove a member

- `> rs.remove("10.17.57.10:27021")`

- Reconfiguring Replica set
  - Let's assume, that one of the members is starting on a new port. To solve this, we have to change the port number of that member.
  - Stop the member, and restart using the new port number
    - `lastHeartbeatMessage` : "Error connecting to 10.17.57.10:27021 :: caused by :: Connection refused", // rs.status displays this message
  - Modify the member with a new port number
    - `> varconf=rs.config()`
    - `> conf.members[3].host="10.17.57.10:27022"`

- Reconfiguring Replica set
  - Reconfigure the replica set
    - `> rs.reconfig(conf)`
  - Check the configuration and status
    - `> rs.config()`
    - `> rs.status()`

- Considerations while designing a replica set
  - Design configuration in such a way that primary can reach more than half of the members
  - Majority is designed by the number of members in the configuration, not based on the available members.
  - Members will ping other members every 2 seconds.
  - If a heartbeat does not return from a member within 10 seconds, the other members mark the delinquent member as inaccessible.
  - If the member is unable to reach the primary in the specified time, it will propose itself as the primary
  - Other members after receiving the new primary request check the following.
    - Are they able to reach the old primary
    - Is the member who requests to primary have the latest transaction
    - Priority of the member

- States that indicate a problem in replication
  - DOWN
    - If a member was up but then becomes unreachable
  - UNKNOWN
    - If a member has never been able to reach another member, it will not know its state, so it will report it as UNKNOWN.
  - REMOVED
    - This is the state of a member that has been removed from the set.
  - ROLLBACK
    - This state is used when a member is rolling back data

- Different States in Replication

- STARTUP

- This is the state a member is in when it's first started, while MongoDB is attempting to load its replica set configuration. Once the configuration has been loaded, it transitions to STARTUP2.

- STARTUP2

- This state lasts throughout the initial sync process, typically taking just a few seconds. The member forks off several threads to handle replication and elections and then transitions into the next state: RECOVERING.

- RECOVERING

- This state indicates that the member is operating correctly but is not available for reads

- ARBITER

- Arbiters have a special state and should always be in this state during normal operation.

- Increase the size of the Ops log
  - Connect to the replica set member whose Ops log size is to be increased.
  - Verify the current size of the oplog:
    - `> use local`
    - `> db.oplog.rs.stats(1024*1024).maxSize`
  - Change the oplog size of the replica set member:
    - `> db.adminCommand({replSetResizeOplog: 1, size: 16000})`

- Advantages

**Provides High Availability** - Data replication ensures that your system continues to function even if one server or data center experiences a failure. Avoiding downtime and maintaining uninterrupted services keeps your data secure, ensures your brand reputation remains strong, and helps maintain customer satisfaction.

**Improves Performance** - Data replication not only enhances availability but also contributes to improved performance. By distributing data across multiple locations, businesses can reduce the load on the primary system, which leads to faster response times and reduced latency.

- Advantages

**Facilitates Load Balancing** - Data replication helps with load balancing by efficiently distributing data requests across multiple backend servers. Doing this ensures a single server isn't overloaded, maintaining fast response times during peak usage. Organizations can maintain their infrastructure and scale resources if needed through load balancing.

**Allows for Geographical Data Distribution** - Because many organizations have customers across the globe, it's important to ensure they can access information from their nearest server. Data replication allows organizations to store data in different locations worldwide, reducing data retrieval time, enhancing performance, and delivering a positive user experience no matter where your customer lives.

- Advantages

**Aids in Disaster Recovery** - Data replication is critical to disaster recovery planning. In the event of cyber attacks or natural disasters, replicated data can minimize downtime and ensure operations resume as quickly as possible. The ability to recover from disaster efficiently helps avoid customer loss and avoids staining your brand's reputation.

**Facilitates Testing Initiatives** - Data replication is critical when it comes to testing as developers can also use replication to have access to realistic, current data in their test environments. Having copies of data in non-production environments allows developers to work with data without impacting live systems. The opportunity to safely work on debugging initiatives and figure out how to speed up development cycles is important for continuously improving application quality and better catering to customers.

- Disadvantages

**Creates Challenges with Data Synchronization** - When keeping multiple copies of data, ensuring they stay synchronized and up-to-date is crucial. Data inconsistencies and application conflicts could arise if synchronization processes aren't well-managed, potentially leading to data integrity issues and inaccuracies when making critical business decisions.

**Increases Storage Costs** - Because data replication requires keeping multiple copies of data, it can lead to higher storage costs. While these costs might not significantly affect large organizations, maintaining additional storage can negatively affect budgets for smaller organizations.

- Disadvantages

**Increases Bandwidth Requirements** - Replicating data across multiple servers requires significant network bandwidth. Transferring data between servers can strain your network and potentially lead to increased operational costs.

**Increases Maintenance Complexity** - Data replication can be complicated to implement and even more complicated to maintain. It's important that organizations design a robust replication architecture from the beginning and continuously monitor the replication process to ensure operations run smoothly. To keep up with this process, you should have skilled employees ready to assist with ongoing maintenance and tackle any issues that may arise.

- Disadvantages

**Introduces Data Security Risks** - While data replication can enhance security by providing redundancy, it can also introduce security risks if poorly implemented. Replicating data to remote servers could expose it to unauthorized access and other potential cyber threats. It's important to note that your data replication efforts must also comply with data protection and privacy laws in the countries you're operating, which introduces additional complexity.

**Introduces Data Corruption and Loss Risks** - Errors in the data replication process can lead to data corruption or loss. If no one on your team knows about the data corruption, it could lead to additional corruption across servers. Delegating team members to monitor your systems regularly helps to detect and mitigate issues quickly.

Exercise use docker compose file provided to create a mongodb replica set database system



We Work With Data

Thank you!

[ailton.moreira@openup.pt](mailto:ailton.moreira@openup.pt)



# MongoDB – Administration

## Introducing MongoDB

Ailton Moreira  
ailton.moreira@openup.pt

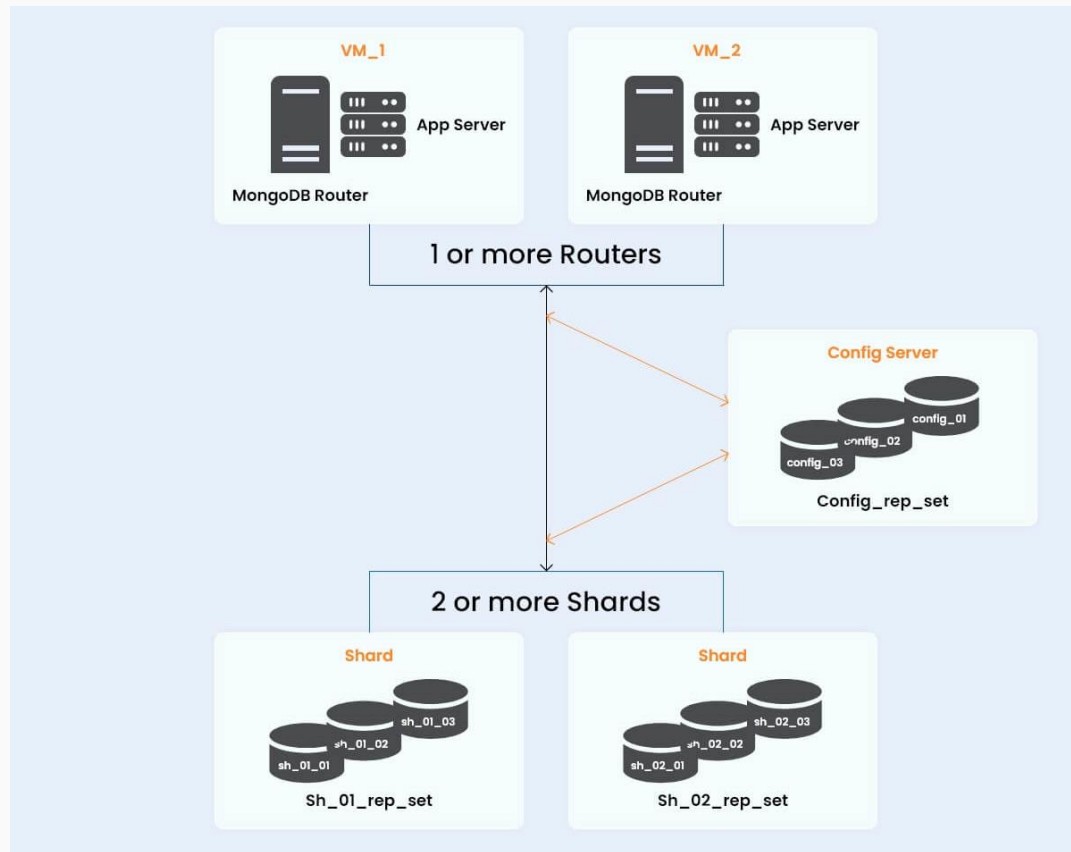
VN Gaia 2024

## What is sharding?

- Sharding refers to the process of splitting data up across machines;
- A subset of data is placed on different servers.
- MongoDB supports auto-sharding
- The application always to the sharding server as it connects to standalone server
- The data moves around the cluster automatically.
- The recommended means of configuring and deploying sharded clusters is through either MongoDB Ops Manager or MongoDB Atlas

What is sharding?

Each shard has its own replica set



- Mongos
  - In a sharded cluster, there can be 2, 3, 10, or even hundreds of members or shards, which look like a single machine to your application.
  - We run one or more routing processes called mongos in front of the shards to hide these details from the application.
  - A mongos keeps a “table of contents” telling which shard contains which data.
  - Applications can connect to mongos and issue commands normally.

- Config Server
  - They can be considered as the brain of the cluster
  - They hold all of the metadata about which servers hold what data.
  - The config servers must be started before any mongos processes, as mongos pulls its configuration from them.
  - `mongod --configsvr --replSet configRS --bind_ip localhost,10.17.57.10 --dbpath/u01/data1 --port 27018 --logpath/u01/data1/mongod.log --fork`
  - •The `--configsvr` option indicates to the `mongod` that you are planning to use it as a config server.

- Config Server

The `--configsvr` option indicates to the `mongod` that you are planning to use it as a config server.

On a server running with this option, clients (i.e., other cluster components) cannot write data to any database other than `config` admin.

```
PRIMARY> db.user.insert({name:"test"})
WriteResult({
 "nInserted" : 0,
 "writeError" : {
 "code" : 14037,
 "errmsg" : "can't create user databases on a --configsvr instance"
 }
})
```

- Config Server
- While starting mongos, specify the config server
- By default, mongos runs on port 27017. Note that it does not need a data directory
- mongos holds no data itself; it loads the cluster configuration from the config servers on startup

- Balancer

- The balancer is responsible for migrating data. It regularly checks for imbalances between shards and, if it finds an imbalance, will begin migrating chunks.
- The balancer is a background process on the primary of the config server replica set, which monitors the number of chunks on each shard
- It becomes active only when a shard's number of chunks reaches a specific migration threshold.

- Removing a shard
  - `db.adminCommand({"removeShard" : "shard02"})`
- Disabling Balancer
  - `sh.setBalancerState(false)`
- Enabling Balancer
  - `sh.setBalancerState( true )`

- Changing Chunk Size

- `> db.settings.findOne()`
  - `{ "_id" : "chunksize", "value" : 64 }`
- `db.settings.save({"_id" : "chunksize", "value" : 32})`

- Manually moving chunks

- `> sh.moveChunk("video.movies", {imdbId: 500000}, "shard02")`
  - Here “shard02” is the shard name .

- Advantages
- **Increased read/write throughput** — By distributing the dataset across multiple shards, both read and write operation capacity is increased as long as read and write operations are confined to a single shard.
- **Increased storage capacity** — Similarly, by increasing the number of shards, you can also increase overall total storage capacity, allowing near-infinite scalability.
- **High availability** — Finally, shards provide high availability in two ways. First, since each shard is a replica set, every piece of data is replicated. Second, even if an entire shard becomes unavailable since the data is distributed, the database as a whole remains partially functional, with part of the schema on different shards.

- Disadvantages

**Query overhead** — Each sharded database must have a separate machine or service which understands how to route a querying operation to the appropriate shard. This introduces additional latency on every operation. If the data required for the query is horizontally partitioned across multiple shards, the router must then query each shard and merge the results together. This can make an otherwise simple operation quite expensive and slow down response times.

- Disadvantages
- **The complexity of administration** — With a single unsharded database, only the database server itself requires upkeep and maintenance. With every sharded database, on top of managing the shards themselves, there are additional service nodes to maintain. Plus, in cases where replication is being used, any data updates must be mirrored across each replicated node. Overall, a sharded database is a more complex system which requires more administration.
- **Increased infrastructure costs** — Sharding by its nature requires additional machines and compute power over a single database server. While this allows your database to grow beyond the limits of a single machine, each additional shard comes with higher costs. The cost of a distributed database system, especially if it is missing the proper optimization, can be significant.

Exercise use docker compose file provided to create a mongodb sharding database system



We Work With Data

Thank you!

[ailton.moreira@openup.pt](mailto:ailton.moreira@openup.pt)



# MongoDB – Administration

## Introducing MongoDB

Ailton Moreira  
ailton.moreira@openup.pt

VN Gaia 2024

MongoDB offers two primary methods for scaling a database: replication and sharding. Both techniques have different purposes and use cases.

- **Replication**

Replication primarily provides high availability and data redundancy. It ensures that data is replicated across multiple servers.

**Replica Set:** A group of MongoDB servers that maintain the same data set. A replica set typically has:

**Primary Node:** The main server that receives all write operations.

**Secondary Nodes:** Servers that replicate the data from the primary node.

These can take over in case the primary node fails.

- **Sharding**

Sharding is designed for horizontal scaling. It distributes data across multiple servers, or shards, to handle large datasets and high throughput applications.

**Config Servers:** Store metadata and configuration settings for the sharded cluster.

**Mongos:** Acts as a query router that directs operations to the appropriate shard(s).

**Shard:** Each shard holds a portion of the data.

- **Use Cases**

**Replication:** Suitable for applications that require high availability, disaster recovery, and read scalability. For example, applications that need to ensure uptime can benefit from read replicas.

**Sharding:** Ideal for applications with large-scale data and high throughput requirements where write operations need to be distributed. For example, big data applications, social networks, or any application with rapidly growing data.

- **Combining Replication and Sharding**

In large-scale applications, it's common to combine both replication and sharding to leverage the benefits of both methods. Each shard in a sharded cluster can be configured as a replica set, providing both horizontal scaling and high availability.

| Features          | Replication                                                                                                    | Sharding                                                                                                         |
|-------------------|----------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| Purpose           | High availability and data redundancy                                                                          | Horizontal scaling for large datasets and high throughput                                                        |
| Components        | <ul style="list-style-type: none"><li>- Primary Node</li><li>- Secondary Nodes</li><li>- Replica Set</li></ul> | <ul style="list-style-type: none"><li>- Shard</li><li>- Config Servers</li><li>- Mongos (Query Router)</li></ul> |
| High Availability | Yes, through automatic failover                                                                                | Limited, relies on replication within each shard for high availability                                           |
| Data Redundancy   | Yes, multiple copies of data across nodes                                                                      | Not inherent, but can be combined with replication for redundancy                                                |
| Read Scaling      | Yes, secondary nodes can handle read operations                                                                | Yes, reads can be distributed across multiple shards                                                             |

| Features             | Replication                                                 | Sharding                                                                   |
|----------------------|-------------------------------------------------------------|----------------------------------------------------------------------------|
| Write Scaling        | No, all writes go through the primary node                  | Yes, writes are distributed across multiple shards                         |
| Complexity           | Lower, easier to set up and manage                          | Higher, more complex to set up and manage                                  |
| Operational Overhead | Lower, simpler management                                   | Higher, requires careful shard key selection and data balancing            |
| Data Consistency     | Possible replication lag on secondary nodes                 | Potential for uneven data distribution if shard key is not chosen properly |
| Use Cases            | Applications needing high availability and read scalability | Applications with large-scale data and high write/read throughput          |

| Features    | Replication                                                    | Sharding                                                                |
|-------------|----------------------------------------------------------------|-------------------------------------------------------------------------|
| Examples    | Disaster recovery, read-heavy applications                     | Big data applications, social networks, rapidly growing data apps       |
| Scalability | Limited to read scaling and redundancy                         | Highly scalable for both read and write operations                      |
| Combining   | Can be combined with sharding for more comprehensive solutions | Often combined with replication for high availability within each shard |



We Work With Data

Thank you!

[ailton.moreira@openup.pt](mailto:ailton.moreira@openup.pt)



# MongoDB – Administration

## Introducing MongoDB

Ailton Moreira  
ailton.moreira@openup.pt

VN Gaia 2024

Understand how to monitor MongoDB effectively to ensure optimal performance and reliability.

- **Health Check**

- Regularly assessing the health of your MongoDB instances to ensure they are functioning correctly.
  - **rs.status()**: Checks the status of the replica set.
  - **db.serverStatus()**: Provides an overview of the server
  - **status.db.stats()**: Retrieves statistics about the database.

- **Health Check**

## Checking Replica Set Status

- `rs.status()`

Ensure the replica set is operating correctly and all members are in sync.

Run `rs.status()` to check the health of the replica set.

Analyze the output for any issues, such as lagging secondaries or members in the wrong state.

- **Logging and Log Rotation**

Recording events and activities for troubleshooting and auditing purposes, and rotating logs to manage disk space.

**mongod** options: --logpath, --logRotate

**logRotate** command: command is an administrative command that allows you to rotate the MongoDB server log and/or audit log to prevent a single logfile from consuming too much disk space.

## ● Logging and Log Rotation

```
db.adminCommand(
 {
 logRotate: <integer or string>,
 comment: <string>
 }
)
```

| Field     | Type              | Description                                                                                                                                                                                                                                      |
|-----------|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| logRotate | integer or string | The log or logs to rotate, according to the following: <ul style="list-style-type: none"><li>1 -- Rotates both the server and audit logs</li><li>"server" -- Rotates only the server log</li><li>"audit" -- Rotates only the audit log</li></ul> |
| comment   | string            | <i>Optional.</i> A message logged by the server to the log file and audit file at time of log rotation.                                                                                                                                          |

```
db.adminCommand({logRotate: 1})
```

Manage log files to prevent them from consuming too much disk space.

Configure mongod to log to a specific path.

Set up log rotation using the logRotate command or by configuring it in the MongoDB configuration file.

- **The CLI Monitoring Tool**

Command Line Interface (CLI) tools for real-time monitoring of MongoDB instances.

**mongostat**: Provides a quick overview of the status of a running mongod instance.

**mongotop**: Tracks the time a MongoDB instance spends reading and writing data.

```
mongostat --host localhost --port 27017 --rowcount=20 1
```

```
mongostat -u <username> -p <password> --authenticationDatabase admin --
rowcount=20 1
```

Monitor real-time performance metrics such as queries per second, lock percentage, and network traffic.

Run mongostat to get real-time statistics.

Use the output to identify performance bottlenecks.

- **System-Level Monitoring**

Monitoring the underlying system resources to ensure MongoDB has adequate CPU, memory, disk, and network resources.

**Prometheus:** Open-source system monitoring and alerting toolkit.

**Grafana:** Open-source analytics and monitoring platform.

**Telegraf:** Plugin-driven server agent for collecting metrics from systems.

Ensure that the system resources are sufficient and performing well.

- Regular health checks and monitoring are essential for maintaining MongoDB performance and reliability.
- Proper logging and log rotation help in managing disk space and troubleshooting issues.
- CLI and GUI tools provide valuable insights into MongoDB's performance.
- System-level monitoring ensures that the infrastructure supporting MongoDB is functioning optimally.



We Work With Data

Thank you!

[ailton.moreira@openup.pt](mailto:ailton.moreira@openup.pt)



# MongoDB – Administration

## Introducing MongoDB

Ailton Moreira  
ailton.moreira@openup.pt

VN Gaia 2024

Understand how to optimize MongoDB performance through various tuning techniques.

- **Storage and OS Tuning**

- Optimizing the underlying storage and operating system settings to improve MongoDB performance.
  - **File System:** Use appropriate file systems like XFS for better performance.
  - **Disk I/O:** Use SSDs for faster read/write operations.
  - **Memory Management:** Ensure enough RAM is allocated to MongoDB.
  - **Swappiness:** Adjust swappiness to reduce swapping.
  - **I/O Scheduler:** Use noop or deadline I/O schedulers for SSDs.

- **Storage and OS Tuning**

- Adjusting Swappiness

- `sudo sysctl vm.swappiness=1`

- Optimize MongoDB for a high-read workload.
  - Configure the file system to XFS.
  - Use SSDs for data storage.
  - Adjust swappiness to reduce unnecessary swapping.
  - Set the I/O scheduler to noop.

- **Caching Strategies**

Efficient use of caches to improve data retrieval speed.

- **Key Concepts:**

- **WiredTiger Cache:** MongoDB's storage engine uses an internal cache.
- **Memory-Mapped Storage:** For MMAPv1, the operating system's page cache is used.
- **Read Preference:** Optimize read operations to use the appropriate replica.

- **Caching Strategies**

## Configuring WiredTiger Cache

```
storage:
 wiredTiger:
 engineConfig:
 cacheSizeGB: 4
```

Optimize MongoDB for a high-write workload.

Increase WiredTiger cache size.

Monitor cache usage with `db.serverStatus().wiredTiger.cache`.

- **Compaction and Compression**

Reducing the size of data on disk and improving read/write efficiency.

## Key Concepts

**Compaction:** Reclaim space by compacting collections.

**Compression:** Use WiredTiger's compression options to reduce data size.

## Enabling Compression

storage:

wiredTiger:

collectionConfig:

blockCompressor: zlib

Optimize storage usage for a large dataset.

Enable compression in the MongoDB configuration. Compact collections periodically.

- **Benchmarking and Stress Testing**

Assessing MongoDB's performance under various load conditions.

- **sysbench**: General-purpose benchmarking tool.
- **YCSB (Yahoo! Cloud Serving Benchmark)**: A benchmarking tool for evaluating database performance.
- **MongoDB Bench (mongobench)**: MongoDB-specific benchmarking tool.

Running a YCSB Benchmark

```
ycsb load mongodb -s -P workloads/workloada
```

- **Benchmarking and Stress Testing**

Running a YCSB Benchmark

```
ycsb load mongodb -s -P workloads/workloada
```

Assess MongoDB's performance under high-concurrency scenarios.

Set up YCSB with MongoDB.

Run the benchmark to simulate high-concurrency load.

Analyze the results to identify performance bottlenecks.

- **Key Points:**

- Storage and OS tuning involve configuring the underlying infrastructure for optimal performance.
- Caching strategies help in utilizing memory efficiently to speed up data retrieval.
- Compaction and compression reduce storage requirements and improve read/write efficiency.
- Benchmarking and stress testing provide insights into MongoDB's performance under different loads.



We Work With Data

Thank you!

[ailton.moreira@openup.pt](mailto:ailton.moreira@openup.pt)



# MongoDB – Administration

## Introducing MongoDB

Ailton Moreira  
ailton.moreira@openup.pt

VN Gaia 2024

- Understand the key tasks and tools involved in MongoDB administration and maintenance.
  - Flush, Cleanup  
Operations to ensure efficient use of disk space and memory.

**db.collection.reIndex():** Rebuilds all indexes on a collection.

**db.runCommand({ compact: 'collection' }):** Compacts a collection to reclaim space.

The db.collection.reIndex() drops all indexes on a collection and recreates them.

This operation may be expensive for collections that have a large amount of data and/or a large number of indexes.

- Repair

Fixes issues in data files, ensuring data integrity.

`mongod --repair`: Repairs all databases.

Recover from data corruption due to hardware failure.

1. Stop the MongoDB server.
2. Run the repair command.
3. Restart the server.

`mongod --dbpath /data/db --repair`

- Rebuilding Index

Rebuilds indexes to optimize query performance and reclaim disk space.

**db.collection.reIndex()**: Rebuilds indexes for a collection.

Improve query performance after heavy updates and deletions.

- Adding Nodes

Expands the MongoDB cluster to increase capacity and availability.

Add node configuration to mongod.conf. (in a docker container the change should be done in docker compose file)

Start the new node.

Add the node to the replica set.

```
rs.add("new_node_hostname:27017")
```

Scale the MongoDB cluster to handle the increased load.

- Configure the new node.

- Add the node to the replica set.

- Verify the node addition.

- Handling Node Failures

Manage and recover from node failures to maintain availability.

Replica Set Configuration.

**rs.status()**: Checks the status of the replica set.

Automatically failover to secondary node in case of primary node failure.

Configure replica set with secondary nodes.

Monitor replica set status.

- Backup and Recovery

Create backups to restore data in case of failures.

**mongodump**: Creates a backup.

**mongorestore**: Restores a backup.

Examples

```
mongodump --out /backup/mydatabase
```

```
mongorestore --db mydatabase /backup/mydatabase
```

- Utilities

**mongo:** MongoDB shell.

**mongotop:** Tracks and reports the time a MongoDB instance spends reading and writing data.

**mongostat:** Provides a quick overview of the status of a currently running mongod or mongos instance.

Using mongostat

```
mongostat --host myhost --port 27017
```

- Utilities

**mongo:** MongoDB shell.

**mongotop:** Tracks and reports the time a MongoDB instance spends reading and writing data.

**mongostat:** Provides a quick overview of the status of a currently running mongod or mongos instance.

Using mongostat

```
mongostat --host myhost --port 27017
```

Using mongotop

```
mongotop 5
```



We Work With Data

Thank you!

[ailton.moreira@openup.pt](mailto:ailton.moreira@openup.pt)



# MongoDB – Administration

## Introducing MongoDB

Ailton Moreira  
ailton.moreira@openup.pt

VN Gaia 2024

Understand how to design applications using MongoDB, especially in the context of microservices.

- MongoDB and Microservice Architecture
- Applied Examples
- Identifying Services
- Designing a Persistence Solution with Microservices

- MongoDB and Microservice Architecture

Using MongoDB as the database in a microservices architecture to manage data in a distributed and scalable manner.

- **Key Concepts:**

- **Decentralized Data Management:** Each microservice manages its own database.
- **Schema Flexibility:** MongoDB's flexible schema supports diverse data models across different services.
- **Scalability:** MongoDB can scale horizontally, supporting the scalability needs of microservices.

- MongoDB and Microservice Architecture

## Using MongoDB in a Microservice

An e-commerce application with separate microservices for user management, product catalog, and orders.

Implementing an order service in an e-commerce application.

- Create a dedicated MongoDB instance for the order service.
- Define the schema for orders, leveraging MongoDB's flexibility.
- Implement the order service to interact with the MongoDB instance.

- Applied Examples

Practical implementation of MongoDB in microservices.

## User Management Service

Service to handle user registration, login, and profile management.

```
{
 "_id": "ObjectId",
 "username": "string",
 "password": "string",
 "email": "string",
 "profile": {
 "firstName": "string",
 "lastName": "string",
 "address": {
 "street": "string",
 "city": "string",
 "state": "string",
 "zip": "string"
 }
 },
 "status": "string"
}
```

- Applied Examples

Practical implementation of MongoDB in microservices.

Product Catalog Service

Service to manage product information.

```
{
 "_id": "ObjectId",
 "name": "string",
 "description": "string",
 "price": "number",
 "category": "string"
}
```

Implementing the user/catalog management service.

Define the user/catalog schema.

Implement CRUD operations for user/catalog data.

Integrate with the authentication service.

- Identifying Services

Determining the different services required in an application.

- **Key Concepts:**

- **Domain-Driven Design (DDD):** Identify bounded contexts and define services around them.
- **Service Boundaries:** Each service should have a single responsibility and own its data.

- Identifying Services

## Identifying Services in an E-Commerce Application Services:

**User Management Service:** Handles user data and authentication.

**Product Catalog Service:** Manages product information.

**Order Service:** Manages orders and transactions.

**Inventory Service:** Tracks product inventory.

- Designing a Persistence Solution with Microservices

Creating a data storage strategy that aligns with microservices architecture.

- **Key Concepts:**

- **Database per Service:** Each microservice uses its own database to ensure loose coupling.
- **Event-Driven Architecture:** Services communicate asynchronously using events.
- **Data Consistency:** Ensuring data consistency across services using techniques like eventual consistency.

- Designing a Persistence Solution with Microservices

## Implementing a Persistence Solution

Designing the data layer for a microservice-based order management system.

- Define Database for Order Service:

- Use MongoDB for storing order details.

- Integrate with Event Bus:

- Use an event bus (e.g., Kafka) for inter-service communication.
- Publish events like OrderPlaced, OrderShipped, etc.

```
{
 "_id": "ObjectId",
 "userId": "ObjectId",
 "productId": "ObjectId",
 "quantity": "number",
 "status": "string",
 "orderDate": "date"
}
```

- **Key Points:**

- MongoDB's flexibility and scalability make it ideal for microservice architectures.
- Proper service identification and clear boundaries are crucial.
- Each microservice should manage its own database to ensure loose coupling.
- Use event-driven architecture for asynchronous communication between services.



We Work With Data

Thank you!

[ailton.moreira@openup.pt](mailto:ailton.moreira@openup.pt)



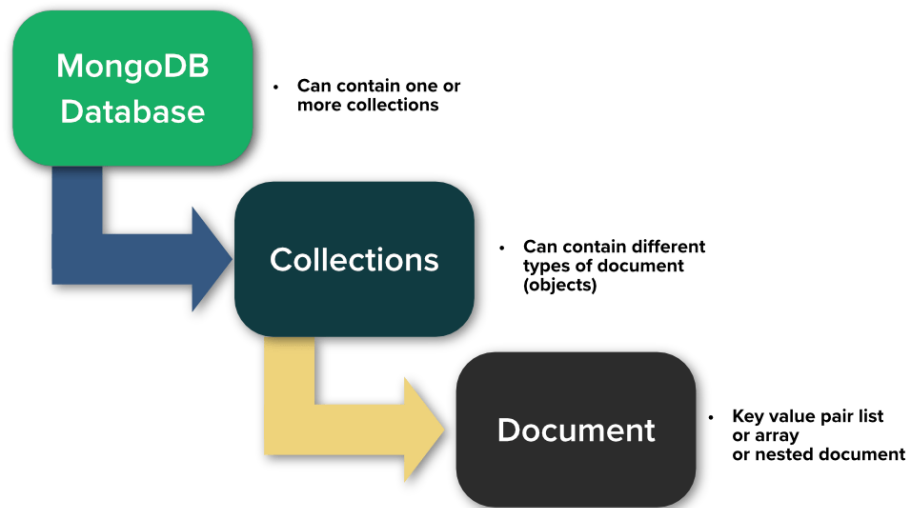
# MongoDB – Administration

## Introducing MongoDB

Ailton Moreira  
ailton.moreira@openup.pt

VN Gaia 2024

- MongoDB is a cross-platform, document-oriented NoSQL database, designed with flexibility and high-volume data storage.
- Stores data in flexible, JSON-like documents, making it easy to model complex data structures (semi-structured or unstructured).
- Follows a schema-less architecture



- Collections and Documents
  - MongoDB stores the data in the form of collections and documents.
  - Documents are the basis of data storage and it stores the data in the form of a "key-value" pair.
  - There is no pre-defined schema for documents, which enables the developer to add or remove fields as required.
  - **OBS:** two documents in a single collection can have different fields
  - Collections consists of set of documents and certain built in functions like "count" , which will help to get the count of documents in a collection.
  - MongoDB allows to create the collection "on the fly", which means the developer can create the collection along with the data insert statement.

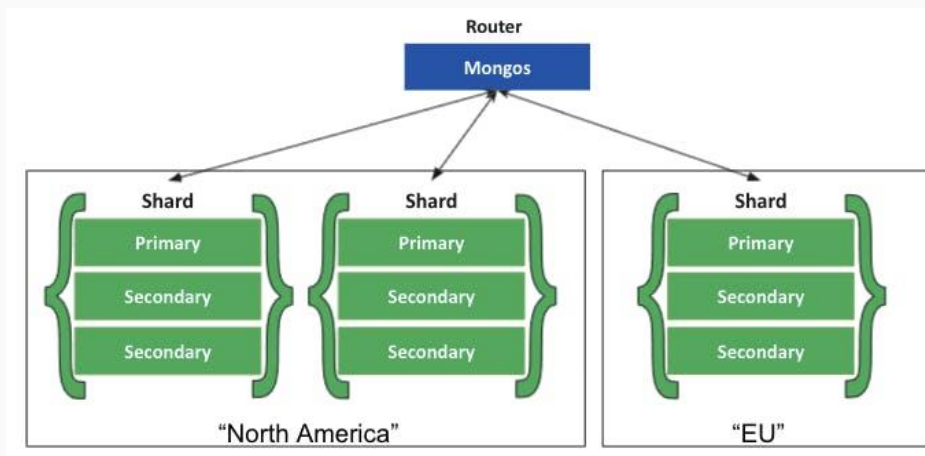
- Characteristics of MongoDB

- Document-Oriented
- High level of scalability
- High level of availability
- Optimized for heavy reads
- Atomicity at the document level
- Rich Query Language
- Flexible Data Modelling
- Indexing
- Built-in Aggregation Framework
- Community-Driven

- Distributed and Decentralized

## Distributed Architecture

MongoDB's architecture is inherently designed to be distributed, allowing it to handle large datasets and high traffic by distributing data and load across multiple servers.



- Distributed Architecture

- **Sharding:** Sharding is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with large datasets and high throughput operations.
- **Shard Keys:** A shard key is chosen to determine how data is distributed across shards. It can significantly impact the performance and scalability of the database.
- **Shard Components:**
  - **Config Servers:** Store metadata and configuration settings for the cluster. They manage the sharding process and keep track of where data resides.
  - **Mongos:** A routing service that directs queries from applications to the appropriate shard based on the shard key.
  - **Shards:** Each shard is an independent database that holds a subset of the data. Shards can be replica sets themselves to ensure high availability.

- Distributed Architecture
  - **Replica Sets:** A replica set is a group of MongoDB servers that maintain the same data set, providing redundancy and high availability.
  - **Replica sets Components:**
    - Primary Node
    - Secondary Nodes
    - Arbiter

- Decentralized Architecture

MongoDB's decentralized architecture refers to its ability to operate without a single point of failure and to distribute data and operations across a cluster of servers. This decentralization enhances fault tolerance and ensures continuous availability.

- No single Point of Failure
  - Redundancy
  - Automatic Failover
- Distributed Data Storage
  - Sharding
  - Replica Sets within Shards

- Distributed and Decentralized - Benefits
  - Scalability
  - High Availability
  - Performance
  - Fault Tolerance

- Elastic Scalability

## What is Elastic Scalability?

- Elastic scalability refers to the ability of a system to dynamically scale up (add resources) or scale down (remove resources) in response to changing workloads. This is essential for maintaining performance and cost-effectiveness as application demands vary.
- **MongoDB Elastic Scalability**
  - One of the key features of MongoDB that allows it to handle varying workloads and large datasets efficiently.
  - This capability is particularly important for modern applications that experience fluctuating traffic and require the database to scale up or down quickly without downtime.

- Elastic Scalability - How MongoDB Achieve

- **Sharding**

- Horizontal Scaling
    - Shard Key
    - Adding Shards

- **Replica Sets**

- High Availability
    - Read Scaling

- **Dynamic Cluster Management**

- Mongos Instances
    - Auto-Balancing

- **Cloud-Native Deployments**

- MongoDB Atlas
    - On-Demand Scaling

- Elastic Scalability - Benefits

- **Performance Optimization**

- Load Balancing
    - Adaptability

- **Cost Efficiency**

- Resource Utilization
    - Pay-as-You-Go

- **High Availability and Fault Tolerance**

- Replication
    - Failover

- **Flexibility**

- Dynamic Adjustments

- High Availability and Fault Tolerance

Critical features of MongoDB, ensuring that applications remain operational, and data remains accessible even in the face of hardware failures, network issues, or other disruptions.

- **High Availability**

- Ensures that a database system remains accessible and operational with minimal downtime.

MongoDB achieves this through several mechanisms

- Replica Sets
- Automatic Failover
- Read Preferences

- High Availability and Fault Tolerance

- **Fault Tolerance**

- Ensures that the system can continue to operate correctly even in the event of component failures.

MongoDB incorporates several features to provide fault tolerance:

- Data Redundancy
    - **Replication:** Data is replicated across multiple nodes in a replica set, ensuring that a copy of the data is available even if one or more nodes fail.
    - **Sync and Acknowledgement:** Write operations can be configured to require acknowledgments from multiple nodes before considering the write successful (write concern), enhancing data durability.

- Fault Tolerance

MongoDB incorporates several features to provide fault tolerance (cont.)

- Durability

- **Journaling:** MongoDB uses journaling to ensure data durability. The journal records write operations before they are applied to the database, enabling recovery in case of a crash.
- **Write Concern:** Configurable settings allow specifying the level of acknowledgment needed for write operations, ranging from acknowledgment from the primary node to acknowledgment from all replica set members.

- Fault Tolerance

MongoDB incorporates several features to provide fault tolerance (cont.)

- Geographic Distribution

- **Multi-Region Deployments:** Deploying replica set members across multiple geographic regions, MongoDB ensures that data remains accessible even in the event of a regional failure.
- **Read and Write Latency Optimization:** Geo-distributed deployments can be optimized to reduce read and write latencies by directing operations to the nearest available data center.

- Fault Tolerance

MongoDB incorporates several features to provide fault tolerance (cont.)

- Monitoring and Alerting

- **Built-in Monitoring:** Tools like MongoDB Atlas provide comprehensive monitoring and alerting capabilities, allowing administrators to detect and respond to issues promptly.
- **Automated Backups:** Regular automated backups and point-in-time recovery options ensure that data can be restored in case of corruption or accidental deletion.

- High Availability and Fault Tolerance - Benefits
  - Minimal Downtime
  - Data Integrity
  - Operational Continuity
  - Flexibility

- Tunable Consistency
  - The ability to configure the balance between consistency and performance according to the needs of your application.
  - Different applications have different requirements for data consistency, availability, and latency.

## Consistency Models in MongoDB

- MongoDB provides several mechanisms to adjust the consistency level for read and write operations:
  - Write Concern
  - Read Concern

- Write Concern
  - Setting that determines the level of acknowledgment requested from MongoDB when a write operation is performed.
  - Levels
    - **"w:1"**: Acknowledgment from the primary node only.
    - **"w:0"**: No acknowledgment. This is a "fire-and-forget" mode, where the client does not wait for any response from the server.
    - **"w:<number>"**: Acknowledgment from a specified number of replica set members, providing a balance between performance and reliability.
    - **"w"**: Acknowledgment from the majority of the replica set members.

- Read Concern
  - Setting that determines the consistency level of data read from the database.
  - Levels
    - **"local"**: Returns the data from the instance that received the read operation, without waiting for replication.
    - **"available"**: Similar to "local," but it does not guarantee the most recent data.
    - **"majority"**: Ensures that the read operation returns data that has been acknowledged by a majority of the replica set members.
    - **"linearizable"**: Ensures that the read operation reflects all successful writes that were acknowledged prior to the read operation.
    - **"snapshot"**: Ensures that the read operation returns data from a single, consistent snapshot of the data, providing a balance between consistency and performance.

- Tunable Consistency – Practical use
  - Write Operations
    - **High Availability with Basic Durability:** For applications where high availability is critical and occasional data loss is acceptable, use "**w:1**" or "**w:0**".
    - **Balanced Approach:** For applications requiring a balance between performance and data durability, use "**w:2**" or another specific number that reflects your needs.
    - **Strong Consistency:** For applications where data integrity is crucial, such as financial systems or critical data stores, use "**w:majority**" and set `j` to true.

- Tunable Consistency – Practical use
  - Read Operations
    - **Low Latency:** For applications requiring low latency and can tolerate reading slightly stale data, use "**local**" or "**available**".
    - **Strong Consistency:** For applications requiring the most up-to-date data, such as banking or inventory systems, use "**majority**" or "**linearizable**".
    - **Transactional Consistency:** For applications requiring consistent views of data, such as during multi-document transactions, use "**snapshot**".

- CAP Theorem and Eventual Consistency

The CAP theorem, also known as Brewer's theorem, is a fundamental principle that applies to distributed database systems. It states that a distributed database can only guarantee two out of the following three properties simultaneously:

1. **Consistency (C):** Every read receives the most recent write or an error.
2. **Availability (A):** Every request receives a (non-error) response, without the guarantee that it contains the most recent write.
3. **Partition Tolerance (P):** The system continues to operate, even if there is a network partition that separates nodes into isolated groups.

- CAP Theorem

MongoDB, like other distributed databases, must make trade-offs according to the CAP theorem.

How MongoDB aligns with the CAP theorem

- **Partition Tolerance (P):** Inherently partition-tolerant as it is designed to run across distributed clusters. This means that it can handle network partitions and continue operating, albeit with some trade-offs regarding consistency and availability.
- **Consistency and Availability Trade-off:**
  - **Consistency (C):** Provide strong consistency with appropriate configurations, such as setting write concern to "majority" and read concern to "majority."
  - **Availability (A):** Relaxing consistency requirements, to achieve higher availability.

- **Eventual Consistency**

Eventual consistency is a consistency model used in distributed systems. It guarantees that, given enough time all replicas in the system will converge to the same state.

How MongoDB implements it

- **Replica Sets:**

- Replication Lag
    - Read Preferences

- **Sharded Clusters:**

- Distributed Data
    - Consistency Levels

- Eventual Consistency – Practical use case
  - High Read and Write Throughput Applications
  - Geographically Distributed Systems
  - Scalable Systems

## ● MongoDB Architecture

### 2. Native Language Driver

```
db.users.insert({name: "Jane Doe", age: 20})
db.users.find({name: "John Doe"})
```

### 3. High Availability

- Replica Sets

### 1. Dynamic Document Schema

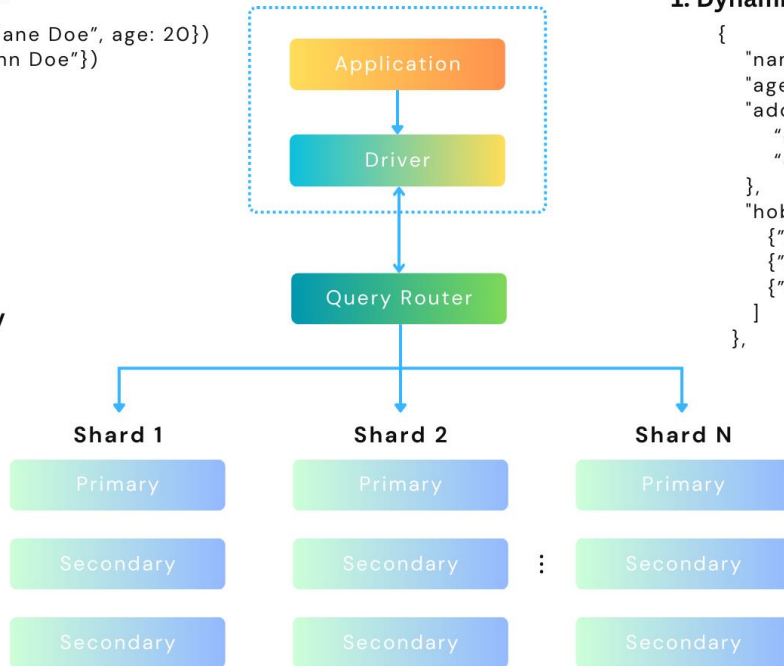
```
{
 "name": "John Doe",
 "age": 30,
 "address": {
 "city": "New York",
 "country": "USA"
 },
 "hobbies": [
 {"name": "sports"},
 {"name": "cooking"},
 {"name": "reading"}
],
}
```

### 4. High Performance

- Data Locality
- Rich Index
- RAM

### 5. Horizontal Scalability

- Sharding



- MongoDB Architecture (Standalone)

**1. Application and MongoDB Driver** - The application uses the MongoDB Node.js driver to make a connection with MongoDB and run the programs.

**2. Query Router** - The query router component is also called the mongos process that works as an interface and an entry point for applications. The application can connect with the query router despite underlying shards and replica sets.

- MongoDB Architecture

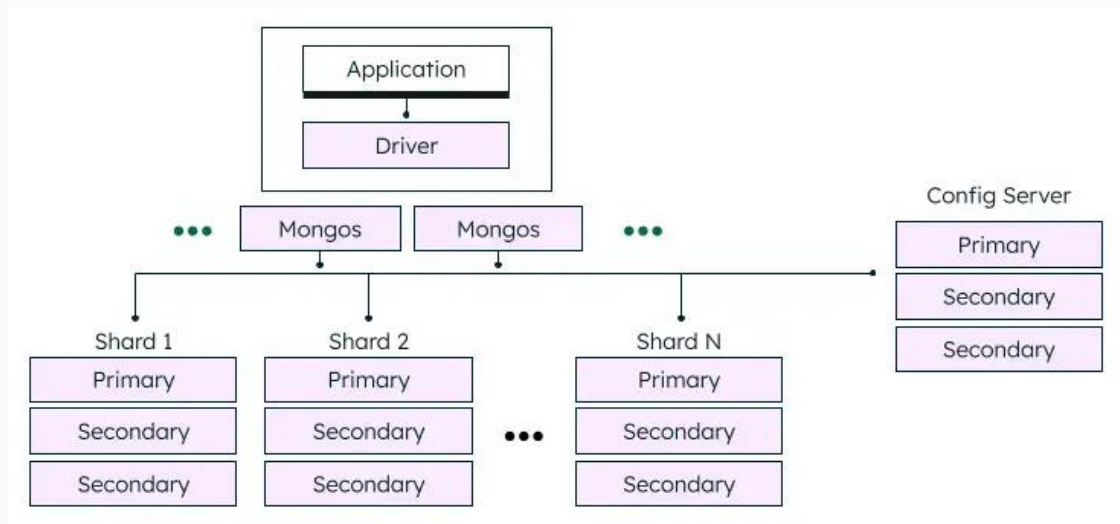
**3. Shard** - Technique to distribute the data across multinode. Using Sharding MongoDB supports large datasets and delivers high throughput operation. It uses horizontal scaling to distribute the data on multiple systems and depending upon the requirement add the nodes as well.

**4. Primary Replica Set Member** - The Primary Replica Set Member is used to receive all the write and read operations and process them. It maintains the **oplog** for all write operations being performed on the data set.

**5. Secondary Replica Set Member** - The Secondary Replica Set Member is used to maintain the primary data sets. It reads the primary's **oplog** and applies the changes in its dataset asynchronously.

- Cluster Architecture

MongoDB's cluster architecture is designed to provide high availability, scalability, and performance. The key components of MongoDB's cluster architecture include replica sets, sharding, and the associated configuration and routing components.



- Cluster Architecture

## **Shards:**

Each shard is a subset of the data. Shards are typically replica sets to ensure high availability and redundancy.

Stores a portion of the database's data.

## **Config Servers:**

Store metadata and configuration settings for the sharded cluster.

Manage the cluster configuration and track the distribution of data across shards.

## **Mongos Instances:**

Query routers that direct client requests to the appropriate shard(s) based on the shard key.

Ensure that queries are efficiently routed and balanced across the cluster.

- Cluster Architecture - Features
  - Horizontal Scaling
  - Data Partitioning
  - Load Balancing
  - Fault Tolerance

- Cluster Architecture - Benefits
  - High Availability
  - Scalability
  - Fault Tolerance
  - Load Balancing

- Editions of MongoDB
  - MongoDB Community Edition (Free for Production use)
  - MongoDB Enterprise Advanced Server (Paid License)
  - MongoDB Atlas
  - MongoDB Real for mobile applications
  - Can upgrade from community edition to enterprise edition

- MongoDB Community vs Enterprise

MongoDB Enterprise provides various features not available in the MongoDB Community edition, such as:

- In-Memory Storage Engine
- Auditing
- Kerberos Authentication
- LDAP Proxy Authentication and LDAP Authorization
- Encrypt at Rest

- MongoDB Atlas
  - Database as a service (DaaS) available on AWS, Azure, and GCP
  - Infrastructure provisioning, setup, and deployment is fully automated with MongoDB Atlas
  - Just select a cloud provider, region, instance size, memory, and additional configurations in the Cluster Builder or via the API
  - Free tier is available for testing

- MongoDB deployment Models
  - Single-Server (standalone) Deployment
  - Replica Set Deployment
  - Sharded Cluster Deployment
  - Multi-Cloud and Hybrid Deployment
  - MongoDB Atlas

- Deployment Model Use Cases
  - **Single-Server Deployment:** Development, testing, and small-scale applications.
  - **Replica Set Deployment:** Medium to large-scale applications requiring high availability and redundancy.
  - **Sharded Cluster Deployment:** Large-scale applications with significant data and high write/read throughput requirements.
  - **Multi-Cloud and Hybrid Deployment:** Applications requiring high flexibility, disaster recovery, and avoidance of vendor lock-in.
  - **MongoDB Atlas:** Organizations looking for a fully managed service with minimal operational overhead and easy scalability.



We Work With Data

Thank you!

[ailton.moreira@openup.pt](mailto:ailton.moreira@openup.pt)

- Single-Server Deployment

Standalone installation is a single-machine installation and is meant mainly for development or experimental purposes.

Suitable for development, testing, and low-traffic applications.

| Pros                                                                 | Cons                                                                                    |
|----------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| Easy to set up and manage.<br>Suitable for small-scale applications. | No high availability or redundancy.<br>Limited scalability.<br>Single point of failure. |

- Replica Set Deployment

Group of processes or servers that work together to maintain the same data set and to provide data redundancy and high availability. It requires at least three servers in a cluster. These servers are configured as the primary, secondaries, or arbiters.

| Pros                                                                                                                                                          | Cons                                                                             |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------|
| Provides high availability.<br>Automatic failover: if the primary node fails, a secondary can become the new primary.<br>Data redundancy and fault tolerance. | Requires more hardware/resources.<br>Complexity in management and configuration. |

- Sharded Cluster Deployment

Sharding distributes data across multiple machines, i.e., allows you to store the data in a distributed way. MongoDB uses sharding to support deployments with large data sets and high throughput operations.

A shard contains a subset of the data, and each shard must use a replica set to provide redundancy of the data that it holds. Multiple shards working together provide a distributed and replicated dataset.

| Pros                                                                                                                                | Cons                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| Horizontal scalability.<br>Can handle large data volumes and high throughput.<br>Can handle large data volumes and high throughput. | Complex to set up and manage.<br>Requires careful planning of shard keys.<br>Requires careful planning of shard keys. |

- Multi-Cloud and Hybrid Deployment

MongoDB can be deployed across multiple cloud providers or in a hybrid environment (combination of on-premises and cloud deployments).

| Pros                                                                                                                                                      | Cons                                                                                                                                                 |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| Flexibility in deployment.<br>Can leverage multiple cloud providers to avoid vendor lock-in.<br>Suitable for disaster recovery and geographic redundancy. | Suitable for disaster recovery and geographic redundancy.<br>Potential latency issues between different cloud providers or on-premises data centers. |

- MongoDB Atlas

MongoDB Atlas is a fully managed, cloud-based database service that automates deployment, scaling, and backup processes.

| Pros                                                                                                                                                 | Cons                                                                                           |
|------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|
| Fully managed service with automated backups and updates.<br>Easy to scale up or down based on demand.<br>Built-in monitoring and security features. | Higher cost compared to self-managed deployments.<br>Dependence on the cloud service provider. |



# MongoDB – Administration

## Introducing MongoDB

Ailton Moreira  
ailton.moreira@openup.pt

VN Gaia 2024

- Editions of MongoDB

- MongoDB Community Edition (Free for Production use)

Start MongoDB

```
service mongod start
```

Verify MongoDB status

```
service mongod status
```

Stop MongoDB

```
service mongod stop
```

MongoDB Shell (mongosh)

```
mongosh
```

```
mongosh "mongodb://<hostname>:<port>"
```

- Editions of MongoDB

Uninstall steps

Type the following commands one by one to uninstall MongoDB:

1. Stop MongoDB process:

```
sudo service mongod stop
```

1. Completely remove the installed MongoDB packages:

```
sudo apt-get purge mongodb-org*
```

1. Remove the data directories, MongoDB database(s), and log files:

```
sudo rm -r /var/log/mongodb /var/lib/mongodb
```

To check if MongoDB is successfully uninstalled, type:

```
service mongod status
```

```
vboxuser@MongoDB:~$ sudo service mongod start
[sudo] password for vboxuser:
sudo: service: command not found
vboxuser@MongoDB:~$ sudo service mongod start
vboxuser@MongoDB:~$ mongosh
Current Mongosh Log ID: 665b9392b2a3b5c1b7a26a12
Connecting to: mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.2.6
Using MongoDB: 7.0.11
Using Mongosh: 2.2.6
```

For mongosh info see: <https://docs.mongodb.com/mongodb-shell/>

To help improve our products, anonymous usage data is collected and sent to MongoDB periodically (<https://www.mongodb.com/legal/privacy-policy>).

You can opt-out by running the `disableTelemetry()` command.

-----

The server generated these startup warnings when booting

2024-06-01T22:32:53.253+01:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. See <http://docs.mongodb.org/core/prodnotes-filesystem>

2024-06-01T22:32:54.348+01:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted

2024-06-01T22:32:54.349+01:00: vm.max\_map\_count is too low

-----

test>

## Basic MongoDB Shell Commands

Here are some basic commands to get you started with mongosh:

### Show Databases:

```
show dbs
```

### Use a Database:

```
use <myDatabase>
```

### Show Collections:

```
show collections
```

Create collections

```
db.createCollection("demo")
```

Insert into collection

```
db.demo.insertOne({name: "teste 1", age: 28})
```

```
db.demo.insertMany([
 {name: "teste 1", age: 28},
 {name: "teste 2", age: 29, address: "Porto"}])
```

## Advanced MongoDB Shell Commands

### Advanced MongoDB Shell Operations

#### Database Statistics:

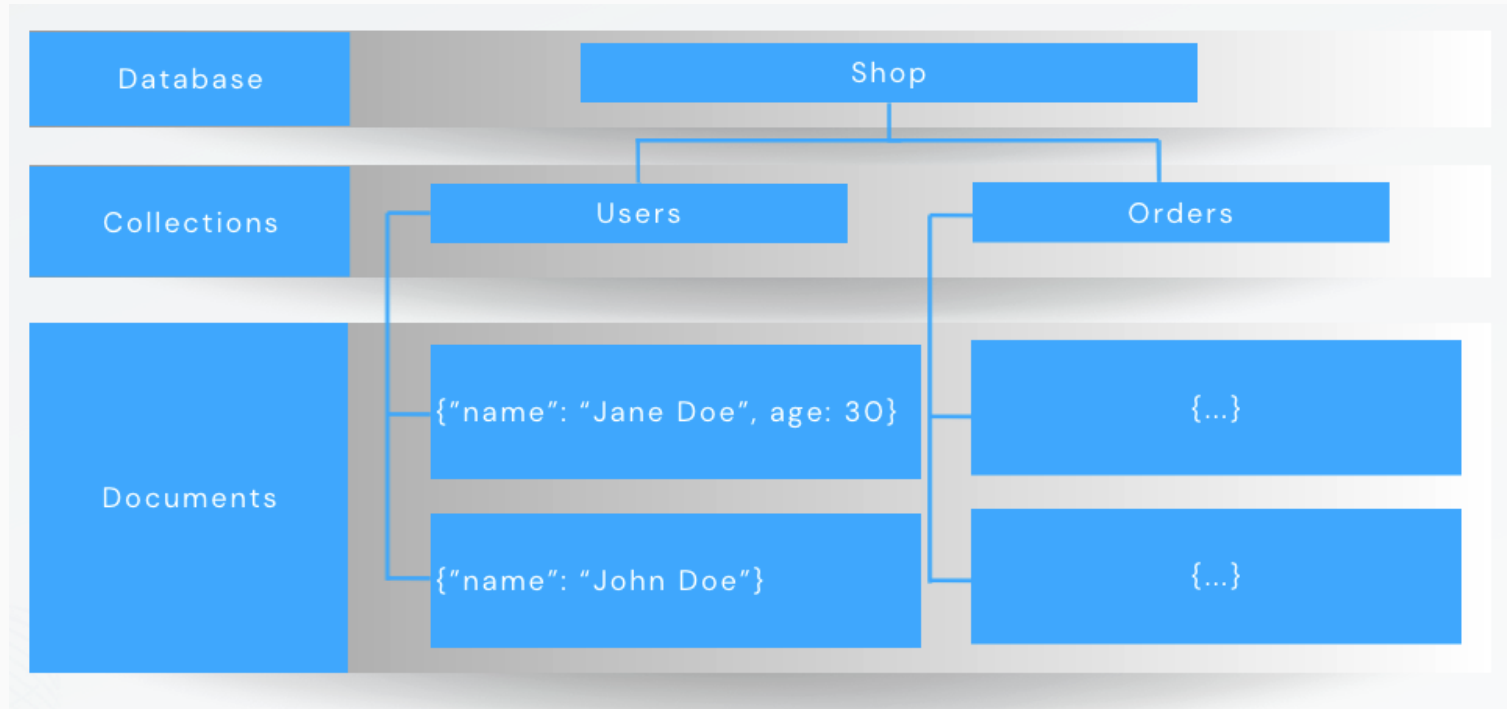
```
db.stats()
```

#### Collection Statistics:

```
db.<myCollection>.stats()
```

#### Explain Query Performance:

```
db.<myCollection>.find({ age: { $gt: 25 } }).explain("executionStats")
```





We Work With Data

Thank you!

[ailton.moreira@openup.pt](mailto:ailton.moreira@openup.pt)



# MongoDB – Administration

## Introducing MongoDB

Ailton Moreira  
ailton.moreira@openup.pt

VN Gaia 2024

The MongoDB Query Language (MQL) is a powerful, flexible, and easy-to-use language designed for working with MongoDB's document-oriented data model.

MQL is based on JavaScript and provides a variety of commands and operators to interact with MongoDB databases.

- MongoDB's Data Model

MongoDB's data model is designed to provide flexibility, scalability, and performance for a variety of applications. Unlike traditional relational databases that use tables, rows, and columns, MongoDB uses a document-oriented data model, which stores data in BSON (Binary JSON) format.

## Key Features of MongoDB's Data Model

### Databases:

Collections are organized into databases. A single MongoDB instance can host multiple databases, each with its own collections and documents.

### Collections:

Documents are grouped into collections, which are analogous to tables in relational databases.

A collection holds multiple documents, and each document within a collection can have a different schema (set of fields).

### Documents:

The fundamental unit of data in MongoDB is a document, which is a BSON object.

Documents are similar to JSON objects and can contain embedded documents and arrays.

Each document is a collection of key-value pairs, where the keys are strings and the values can be a variety of data types, including strings, numbers, dates, arrays, and embedded documents.

## Flexible Schema

MongoDB's schema is flexible and dynamic, meaning documents within a collection do not need to have the same set of fields or data types.

This allows for easy and rapid iteration and evolution of the data model without requiring migrations or schema changes.

## Embedded Documents

Documents can contain other documents as values. This is known as embedding. For example, the "address" field in the above document is an embedded document.

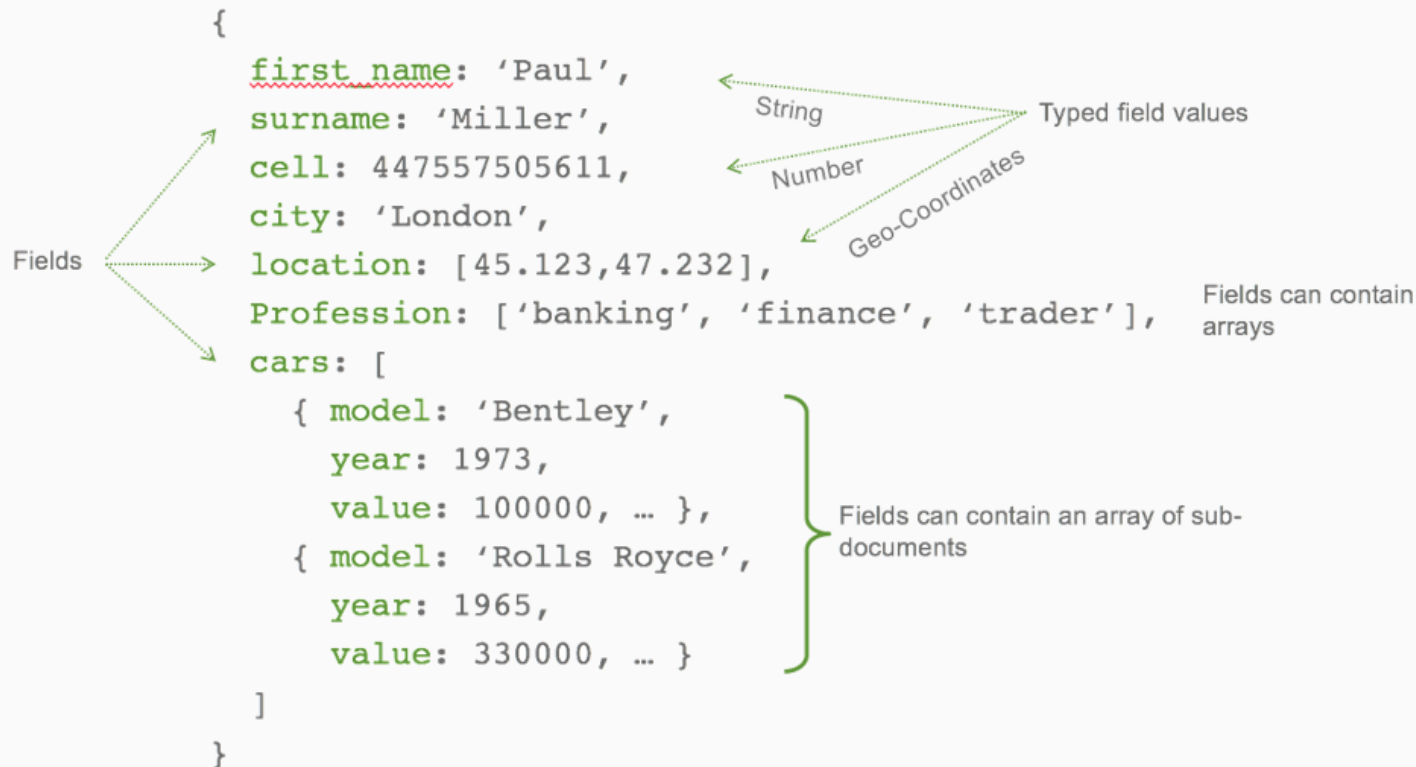
## Arrays

Documents can also contain arrays as values. For example, the "phoneNumbers" field in the above document is an array of embedded documents.

## Document Structure

```
{
 "_id": ObjectId("5f50c31f1c4ae5d1f0e8b7b7"),
 "name": "Alice",
 "age": 30,
 "address": {
 "street": "123 Main St",
 "city": "Anytown",
 "state": "CA",
 "zip": "12345"
 },
 "phoneNumbers": [
 { "type": "home", "number": "555-555-5555" },
 { "type": "work", "number": "555-555-5556" }
],
 "email": "alice@example.com",
 "createdAt": ISODate("2023-01-01T12:00:00Z")
}
```

- MongoDB Documents - JSON (BSON) Data Format – { "key" : "value" }



## Advantages of MongoDB's Data Model

- Schema Flexibility
- Rich Data Types
- Ease of Use
- Efficient Data Retrieval
- Scalability

## Clusters

**Cluster Architecture** - Replica sets and sharding

**Replica Sets** - Ensures high availability and data redundancy

**Sharding** - Enables horizontal scaling

## Keyspaces

**Keyspaces in MongoDB** - Equivalent to databases

**Namespace Management** - Organize collections and documents

**Database Operations** - CRUD operations within keyspaces

## Tables

**Tables in MongoDB** - Referred to as collections

**Document Storage** - Collections store JSON-like documents

**Schema-less Nature** - Collections do not enforce a schema

## Columns

**Columns in MongoDB** - Represented as fields in documents

**Flexible Fields** - Fields can vary across documents

**Nested Fields** - Support for embedded documents and arrays

## Timestamps

**Timestamp Type:** BSON data type

**Automatic Timestamps:** createdAt and updatedAt fields

**Usage:** Track document creation and modification times

## Time to Live (TTL)

**TTL Indexes** - Automatic deletion of documents

**Expiration** - Based on a specific field

**Use Cases** - Cache data, temporary data storage

## Types

**BSON Types** - String, Number, Boolean, Date, Array, Object, etc.

**Rich Data Types** - Support for complex data structures

**Type Checking** - Ensures data integrity

## Collections

**Definition** - Groups of documents

**No Fixed Schema** - Dynamic schema design

**Indexing** - Support for efficient querying

## Tuples

**No Direct Equivalent** - Not natively supported

**Similar Structures** - Use arrays or embedded documents

**Example Usage** - Storing ordered pairs or related items

## User-Defined Types

**Custom Data Structures** - Define and use complex data types

**Schema Validation** - Enforce structure using validation rules

**Application** - Enhance data integrity and structure



We Work With Data

Thank you!

[ailton.moreira@openup.pt](mailto:ailton.moreira@openup.pt)



# MongoDB – Administration

## Introducing MongoDB

Ailton Moreira  
ailton.moreira@openup.pt

VN Gaia 2024

- Conceptual Data Modeling

Conceptual Data Modeling is the process of creating a high-level model of the data without considering how it will be implemented.

It focuses on what data is stored and its relationships.

## Key Points:

Define entities, attributes, and relationships.

Use ER diagrams to visualize data structures.

Focus on business requirements

- Design Differences Between RDBMS and MongoDB

**RDBMS vs MongoDB Design** focuses on understanding the differences in data modelling between traditional relational databases and MongoDB.

Key Points:

RDBMS uses tables, rows, and columns; MongoDB uses collections and documents.

MongoDB supports embedded documents and arrays.

Schema flexibility in MongoDB vs. fixed schema in RDBMS.

Handling relationships: joins in RDBMS vs. **embedding and referencing in MongoDB**.

- Query-first Approach

Query-first Approach in MongoDB involves designing the schema based on the queries that will be executed.

## Key Points:

Identify common queries and design schema to optimize these queries.

Use indexing and data structures to enhance query performance.

Balance between read and write optimization.

- Logical Data Modelling

Logical Data Modelling involves structuring the data in a way that it can be mapped to MongoDB collections and documents.

## Key Points:

Define collections and documents.

Determine how to model relationships: embedding vs. referencing.

Plan for data growth and scalability.

- Logical Data Modelling
- Embedded Document

```
{
 "post_id": "101",
 "content": "Hello, world!",
 "created_at": "2023-05-01",
 "comments": [
 {
 "comment_id": "201",
 "user_id": "2",
 "content": "Hi Alice!",
 "created_at": "2023-05-02"
 },
 {
 "comment_id": "202",
 "user_id": "2",
 "content": "That a nice spot!",
 "created_at": "2023-05-02"
 }
]
},
```

- Logical Data Modelling

```
{
 "post_id": "101",
 "content": "Hello, world!",
 "created_at": "2023-05-01",
 "comment_ids": ["201", "202"]
},
```

- Referenced Document

```
{
 "comment_id": "201",
 "user_id": "2",
 "content": "Hi Alice!",
 "created_at": "2023-05-02"
},
{
 "comment_id": "202",
 "user_id": "2",
 "content": "That a nice spot!",
 "created_at": "2023-05-02"
}
```

- Logical Data Modelling

## Questions and points for embedding:

**Simplicity:** Would keeping the pieces of information together lead to a simpler data model and code?

**Go together:** Do the pieces of information have a “has-a”, “contains”, or similar relationships?

**Query atomicity:** Does the application query the pieces of information together?

**Update complexity:** Are the pieces of information updated together?

**Archival:** Should the pieces of information be archived at the same time?

If answer is yes, you should then opt for embedding document.

- Logical Data Modelling

## Questions and points for referencing:

**Cardinality:** Is there a high cardinality (current or growing) in a “many” side of the relationship?

**Data duplication:** Would data duplication be too complicated to manage and undesired?

**Document size:** Would the combined sizes of the pieces of information take too much memory or transfer bandwidth for the application?

**Document growth:** Would the embedded piece grow without bound?

**Workload:** Are the pieces of information written at different times in a write-heavy workload?

**Individuality:** For the children's side of the relationship, can the pieces exist by themselves without a parent?

If answer is yes, you should then opt for referencing.

- Physical Data Modelling

Physical Data Modelling involves the actual implementation of the logical model in MongoDB, considering storage and performance.

## Key Points:

Define indexes to support query patterns.

Choose data types and storage formats.

Plan for sharding and replication for high availability and scalability.

- Evaluate Sizing

Evaluate Sizing involves estimating the storage and performance requirements for the MongoDB database.

## Key Points:

Estimate data growth and storage needs.

Plan for capacity based on read/write patterns.

Consider factors like document size, index size, and hardware resources.

- Implementing Database Schema

Implementing Database Schema is the process of creating collections, documents, and indexes in MongoDB.

## Key Points:

Use MongoDB commands to create and manage collections and indexes.

Apply best practices for schema design and implementation.

Test and validate schema against use cases and queries.

- Evaluate MongoDB Data Modelling Tools

Evaluate MongoDB Data Modelling Tools involves assessing tools that aid in designing and managing MongoDB schemas.

## Key Points:

Overview of tools like **MongoDB Compass**, Studio 3T, and NoSQLBooster.

Features and benefits of each tool.

How these tools integrate with development workflows.

| Feature              | MongoDB Compass                                   | Studio 3T                                                                      | NoSQLBooster                                                               |
|----------------------|---------------------------------------------------|--------------------------------------------------------------------------------|----------------------------------------------------------------------------|
| Schema Visualization | Provides basic schema visualization capabilities. | Offers comprehensive schema exploration with a hierarchical view.              | Provides visual schema exploration with tree-like representation.          |
| Index Management     | Allows users to create, view, and manage indexes. | Provides advanced index management with suggestions and optimization options.  | Offers index creation, editing, and management capabilities.               |
| Query Optimization   | Offers an intuitive query builder and profiler.   | Excels in query optimization with Visual Query Builder and IntelliShell.       | Provides a query builder, query execution, and performance analysis tools. |
| Data Modelling Tools | Provides basic data modelling tools.              | Offers advanced data modelling features like Schema Explorer and Data Compare. | Provides schema visualization, comparison, and modification tools.         |

| Feature              | MongoDB Compass                                        | Studio 3T                                                           | NoSQLBooster                                                              |
|----------------------|--------------------------------------------------------|---------------------------------------------------------------------|---------------------------------------------------------------------------|
| Data Import/Export   | Allows importing and exporting data from/to MongoDB.   | Offers data migration and import/export features.                   | Provides data import/export capabilities with customization options.      |
| Aggregation Pipeline | Supports building and analyzing aggregation pipelines. | Offers a visual aggregation editor and pipeline builder.            | Provides tools for constructing and optimizing aggregation pipelines.     |
| Collaboration        | Lacks collaboration features.                          | Offers collaboration features like sharing connections and queries. | Offers collaboration features including sharing snippets and collections. |
| Price                | Included with MongoDB subscription.                    | Available in paid versions with free trial.                         | Available in both free and paid versions.                                 |



We Work With Data

Thank you!

[ailton.moreira@openup.pt](mailto:ailton.moreira@openup.pt)



# MongoDB – Administration

## Introducing MongoDB

Ailton Moreira  
ailton.moreira@openup.pt

VN Gaia 2024

- Loading Bulk Data

Loading Bulk Data involves efficiently importing large volumes of data into MongoDB.

## Key Points:

Techniques for loading bulk data: **mongoimport**, **mongorestore**, and scripting.

Considerations for performance and scalability.

Monitoring and optimizing data loading processes.

- JSON Format Import and Export

JSON Format Import and Export enables seamless transfer of data between MongoDB and external systems using JSON format.

## Key Points:

MongoDB's support for JSON data format.

Exporting MongoDB collections to JSON files using **mongoexport**.

Importing JSON files into MongoDB collections using **mongoimport**.

- Using Primary Indexes

Using Primary Indexes involves leveraging primary indexes for efficient data retrieval in MongoDB.

## Key Points:

Understanding primary indexes in MongoDB.

Automatic creation of the `_id` field as the primary index.

Impact of primary indexes on query performance.

- Using Secondary Indexes

Using Secondary Indexes enhances query performance by creating secondary indexes on fields other than `_id` in MongoDB.

## Key Points:

Purpose and benefits of secondary indexes.

Creating secondary indexes using **`createIndex()`** method.

Considerations for index selection and query optimization.

- MongoDB Database Tools

- Mongodump
- mongorestore
- bsondump
- mongoimport
- mongoexport
- mongostat
- mongofiles



We Work With Data

Thank you!

[ailton.moreira@openup.pt](mailto:ailton.moreira@openup.pt)



# MongoDB – Administration

## Introducing MongoDB

Ailton Moreira  
ailton.moreira@openup.pt

VN Gaia 2024

- MongoDB Authentication and Authorization

While authentication and authorization are closely connected, it is important to note that authentication is distinct from authorization.

The purpose of authentication is to verify the identity of a user, while authorization determines the verified user's access to resources and operations.

- Creating Administrator User in MongoDB

Start mongoDB, without authentication enabled.

```
mongosh --port 27017
```

```
docker exec -it <container-id> mongosh
```

➔ use admin

➔ `db.createUser({user: "myUserAdmin",pwd: passwordPrompt(), // or cleartextpasswordroles: [ { role: "userAdminAnyDatabase", db: "admin" }, "readWriteAnyDatabase" ]})`

Start MongoDB with authentication Enabled

➔ `docker exec -it <container-id> mongosh -u <adminUser> -p <adminPWD> -- authenticationDatabase admin`

- Creating Administrator User in MongoDB

Start mongoDB, without authentication enabled.

```
mongosh --port 27017
```

```
docker exec -it <container-id> mongosh
```

➔ use admin

➔ `db.createUser( { user: "dba", pwd: "oracle", roles: ["root"] } )`

Start MongoDB with authentication Enabled

- Enabling Authentication

Enabling Authentication.

To enable authentication, change the `authorization=enabled` in `/etc/mongodb.conf` security:

```
authorization: enabled
```

Also , in the startup command of `mongosh` , specify `--auth` option, as in this example

```
$ mongosh --auth
```

- Connecting to MongoDB with authentication Enabled

## Authentication Methods

To authenticate as a user, you must provide a username, password, and the authentication database associated with that user.

To authenticate using the mongo shell, either

Use the mongo command-line authentication options (`--username`, `--password`, and `--authenticationDatabase`) when connecting to the mongod or mongos instance, or

- Connecting to MongoDB with authentication Enabled

Connect first to the mongosh or mongos instance, and then run the authenticate command or the db.auth() method against the authentication database.

- `$ mongosh -u dba -p oracle --authenticationDatabase admin`
- `$ mongosh`

-> use admin

-> `db.auth("dba","oracle")`

- Finding as what user connected to the database
  - `db.runCommand({ connectionStatus: 1 })`

```
[test> db.runCommand({ connectionStatus: 1 })
{
 authInfo: {
 authenticatedUsers: [{ user: 'dba', db: 'admin' }],
 authenticatedUserRoles: [{ role: 'root', db: 'admin' }]
 },
 ok: 1
}
```

- Creating a Regular User in MongoDB
  - use admin // This will be the authentication database for the user

```
db.createUser({
 user: "reportsUser",
 pwd: passwordPrompt(),
 roles: [
 {
 role: "read",
 db: "reporting"
 },
 {
 role: "read",
 db: "products"
 },
 {
 role: "read",
 db: "sales"
 },
 {
 role: "readWrite",
 db: "accounts"
 }
]
})
```

- Changing the password of the user
  - use admin // This will be the authentication database for the user
  - `db.changeUserPassword("reportsUser", passwordPrompt())`

```
[admin> db.changeUserPassword("reportsUser_2", passwordPrompt())
Enter password
[MongoServerError[UserNotFound]: User reportsUser_2@admin not found
```

- Listing the details of the users
  - Listing the details of the users
    - -> use admin
    - -> db.getUsers()
  - Listing the details of a specific user
    - -> use admin
    - -> db.getUsers({filter:{user:"reportsUser"}});
  - Or
    - -> use admin
    - -> db.getUser("dba")

- Assigning/Managing Roles to a User

To grant roles to user

```
-> db.grantRolesToUser("reportsUser",
[{"role":"readWrite",db:"local"}, {"role":"readWrite",db:"test"}])
```

To update the roles of an existing user

```
-> db.updateUser("reportsUser",{roles:[{"role":"readWrite", db: "admin"}]})
```

```
-> db.updateUser("dba", {roles:[{"role":"userAdminAnyDatabase",db:"admin"},
{"role":"readWriteAnyDatabase",db:"admin"}, {"role":"readWrite",db:"local"}]})
```

- Removing Roles from a user

→ `db.revokeRolesFromUser("reportsUser",[{role:"readWrite",db:"test"}])`

- Dropping a user

→ `db.dropUser("reportsUser")`

Or

→ `db.removeUser("reportsUser")`

- Creating User Defined Roles

```
db.createRole({role:"NewReadWrite", privileges:[],
roles:[{role:"readWrite",db:"test"},{role:"read",db:"admin"}]})
```

- Details of the roles

To get the details about a role

```
-> db.getRole("NewReadWrite")
```

To list all user Defined Roles

```
-> db.getRoles()
```

this command will list all user defined role.

- Privilege Management

## Granting Privileges to a Role

```
db.grantPrivilegesToRole("NewReadWrite", [{resource:{db:"test",collection:""},
action:["insert","find"]}]) //read not works
```

To revoke privileges from a role

```
db.revokePrivilegesFromRole("NewReadWrite", [{resource:{db:"test",collections:""},
action:["insert"]}])
```

- Privilege Management

To list the privileges given for a role

```
db.getRole("NewReadWrite",{showPrivileges:true})
```

To list all the user defined roles with privileges

```
db.getRoles({rolesInfo: 1, showPrivileges:true,showBuiltinRoles: false})
```

We can specify some parameters to getRoles()

- Listing built in Roles

use admin

```
db.getRoles({rolesInfo: 1, showPrivileges:true, showBuiltinRoles: true})
```

List all the database built-in roles

- Granting Roles to Roles

- > use admin

- > `db.grantRolesToRole("NewReadWrite", [{role:"read", db:"local"}])`

Grant roles to roles to another database

- Revoking Roles from Roles

- > use admin

- > `db.revokeRolesFromRole("NewReadWrite",[{role:"read",db:"local"}])`

Revoking roles to roles to another database

- How to change own password

To change own password , user should have `changeOwnPassword` and `changeOwnCustomData` privileges

## Steps

Create a user defined role with the privileges `changeOwnPassword` and `changeOwnCustomData` privileges

Create the user with the role or grant this role to the existing user.

```
-> db.createRole({ role: "passwdRole", privileges: [{ resource: { db: "admin",
collection: "" }, actions: ["changeOwnPassword", "changeOwnCustomData"] }], roles: [] })
```

```
-> db.grantRolesToUser("dba", [{role:"passwdRole",db:"admin"}])
```

- Built-in Roles in MongoDB

## Database User Roles

Every database includes the following client roles:

- read
  - Provides the ability to read data on all non-system collections and the system.js collection.
- readWrite
  - Provides all the privileges of the read role plus ability to modify data on all non-system collections and the system.js collection.

- Built-in Roles in MongoDB

## Database Administrator Roles

Every database includes the following database administration roles:

- dbAdmin
  - Provides the ability to perform administrative tasks such as schema-related tasks, indexing, and gathering statistics. This role does not grant privileges for user and role management.
- userAdmin
  - Create and modify roles and users on the current database
- dbOwner
  - The database owner can perform any administrative action on the database. This role combines the privileges granted by the readWrite, dbAdmin and userAdmin roles.

- Built-in Roles in MongoDB - userAdmin

Provides the ability to create and modify roles and users on the current database. Since the userAdmin role allows users to grant any privilege to any user, including themselves, the role also indirectly provides superuser access to either the database or, if scoped to the admin database, the cluster.

The userAdmin role explicitly provides the following actions:

|                  |                              |
|------------------|------------------------------|
| changeCustomData | dropUser                     |
| changePassword   | grantRole                    |
| createRole       | revokeRole                   |
| createUser       | setAuthenticationRestriction |
| dropRole         | viewRole                     |
|                  | viewUser                     |

- Built-in Roles in MongoDB - Cluster Administration Roles

The **admin** database includes the following roles for administering the whole system rather than just a single database. These roles include but are not limited to replica set and sharded cluster administrative functions.

clusterManager

Provides management and monitoring actions on the cluster. A user with this role can access the config and local databases, which are used in sharding and replication, respectively.

enableharding

Provides the ability to enable sharding for a collection and modify existing shard keys.

- Built-in Roles in MongoDB - Cluster Administration Roles

## clusterMonitor

Provides read-only access to monitoring tools, such as the MongoDBCloud Manager and Ops Manager monitoring agent.

## hostManager

Provides the ability to monitor and manage servers.

## clusterAdmin

Provides the greatest cluster-management access. This role combines the privileges granted by the clusterManager, clusterMonitor, and hostManager roles. Additionally, the role provides the dropDatabase action.

- Backup and Restore Roles

The **admin** database includes the following roles for backing up and restoring data:

## Backup

Provides minimal privileges needed for backing up data. This role provides sufficient privileges to use the MongoDB Cloud Manager backup agent, Ops Manager backup agent, or to use mongodump to back up an entire mongod instance.

## Restore

Provides privileges needed to restore data from backups.

- All Database Roles

The following roles are available on the admin database and provide privileges which apply to all databases except local and config

## readAnyDatabase

Provides the same read-only privileges as read on all databases except local and config. The role also provides the list Databases action on the cluster as a whole

## readWriteAnyDatabase

Provides the same privileges as readWrite on all databases except local and config. The role also provides the list Databases action on the cluster as a whole.

- All Database Roles

## userAdminAnyDatabase

Provides the same access to user administration operations as userAdmin on all databases except local and config.

## dbAdminAnyDatabase

Provides the same privileges as dbAdmin on all databases except local and config. The role also provides the listDatabases action on the cluster as a whole.

- Super User Roles

root

Provides access to the operations and all the resources of the following roles combined:

readWriteAnyDatabase  
dbAdminAnyDatabase  
userAdminAnyDatabase  
clusterAdmin  
restore  
backup

- Super User Roles

root

Provides access to the operations and all the resources of the following roles combined:

readWriteAnyDatabase  
dbAdminAnyDatabase  
userAdminAnyDatabase  
clusterAdmin  
restore  
backup

- Super User Roles

The following roles provide the ability to assign any user any privilege on any database, which means that users with one of these roles can assign themselves any privilege on any database:

- dbOwner role, when scoped to the admin database

- userAdmin role, when scoped to the admin database

- userAdminAnyDatabase role

- How to assign privileges to a user

As of version current MongoDB doesn't support assigning privileges directly to the user

## Steps

1. Create a user defined role
2. Assign the required privileges to the role
3. Assign the role to the user



We Work With Data

Thank you!

[ailton.moreira@openup.pt](mailto:ailton.moreira@openup.pt)



# MongoDB – Administration

## Introducing MongoDB

Ailton Moreira  
ailton.moreira@openup.pt

VN Gaia 2024

- MongoDB Authentication and Authorization

While authentication and authorization are closely connected, it is important to note that authentication is distinct from authorization.

The purpose of authentication is to verify the identity of a user, while authorization determines the verified user's access to resources and operations.

- Creating Administrator User in MongoDB

Start mongoDB, without authentication enabled.

```
mongosh --port 27017
```

```
docker exec -it <container-id> mongosh
```

➔ use admin

➔ `db.createUser({user: "myUserAdmin",pwd: passwordPrompt(), // or cleartextpasswordroles: [ { role: "userAdminAnyDatabase", db: "admin" }, "readWriteAnyDatabase" ]})`

Start MongoDB with authentication Enabled

➔ `docker exec -it <container-id> mongosh -u <adminUser> -p <adminPWD> -- authenticationDatabase admin`

- Creating Administrator User in MongoDB

Start mongoDB, without authentication enabled.

```
mongosh --port 27017
```

```
docker exec -it <container-id> mongosh
```

➔ use admin

➔ `db.createUser( { user: "dba", pwd: "oracle", roles: ["root"] } )`

Start MongoDB with authentication Enabled

- Enabling Authentication

Enabling Authentication.

To enable authentication, change the `authorization=enabled` in `/etc/mongodb.conf` security:

```
authorization: enabled
```

Also , in the startup command of `mongosh` , specify `--auth` option, as in this example

```
$ mongosh --auth
```

- Connecting to MongoDB with authentication Enabled

## Authentication Methods

To authenticate as a user, you must provide a username, password, and the authentication database associated with that user.

To authenticate using the mongo shell, either

Use the mongo command-line authentication options (`--username`, `--password`, and `--authenticationDatabase`) when connecting to the mongod or mongos instance, or

- Connecting to MongoDB with authentication Enabled

Connect first to the mongosh or mongos instance, and then run the authenticate command or the db.auth() method against the authentication database.

- `$ mongosh -u dba -p oracle --authenticationDatabase <db_name>`
- `$ mongosh`

-> use admin

-> `db.auth("dba","oracle")`

- Finding as what user connected to the database
  - `db.runCommand({ connectionStatus: 1 })`

```
[test> db.runCommand({ connectionStatus: 1 })
{
 authInfo: {
 authenticatedUsers: [{ user: 'dba', db: 'admin' }],
 authenticatedUserRoles: [{ role: 'root', db: 'admin' }]
 },
 ok: 1
}
```

- Creating a Regular User in MongoDB
  - use admin // This will be the authentication database for the user

```
db.createUser({
 user: "reportsUser",
 pwd: passwordPrompt(),
 roles: [
 {
 role: "read",
 db: "reporting"
 },
 {
 role: "read",
 db: "products"
 },
 {
 role: "read",
 db: "sales"
 },
 {
 role: "readWrite",
 db: "accounts"
 }
]
})
```

- Changing the password of the user
  - use admin // This will be the authentication database for the user
  - `db.changeUserPassword("reportsUser", passwordPrompt())`

```
[admin> db.changeUserPassword("reportsUser_2", passwordPrompt())
Enter password
[MongoServerError[UserNotFound]: User reportsUser_2@admin not found
```

- Listing the details of the users
  - Listing the details of the users
    - -> use admin
    - -> db.getUsers()
  - Listing the details of a specific user
    - -> use admin
    - -> db.getUsers({filter:{user:"reportsUser"}});
  - Or
    - -> use admin
    - -> db.getUser("dba")

- Assigning/Managing Roles to a User

To grant roles to user

```
-> db.grantRolesToUser("reportsUser",
[{"role":"readWrite",db:"local"}, {"role":"readWrite",db:"test"}])
```

To update the roles of an existing user

```
-> db.updateUser("reportsUser",{roles:[{"role":"readWrite", db: "admin"}]})
```

```
-> db.updateUser("dba", {roles:[{"role":"userAdminAnyDatabase",db:"admin"},
{role:"readWriteAnyDatabase",db:"admin"}, {"role":"readWrite",db:"local"}]})
```

- Removing Roles from a user

→ `db.revokeRolesFromUser("reportsUser",[{role:"readWrite",db:"test"}])`

- Dropping a user

→ `db.dropUser("reportsUser")`

Or

→ `db.removeUser("reportsUser")`

- Creating User Defined Roles

```
db.createRole({role:"NewReadWrite", privileges:[],
roles:[{role:"readWrite",db:"test"},{role:"read",db:"admin"}]})
```

- Details of the roles

To get the details about a role

```
-> db.getRole("NewReadWrite")
```

To list all user Defined Roles

```
-> db.getRoles()
```

this command will list all user defined role.

- Privilege Management

## Granting Privileges to a Role

```
db.grantPrivilegesToRole("NewReadWrite", [{resource:{db:"test",collection:""},
action:["insert","find"]}]) //read not works
```

To revoke privileges from a role

```
db.revokePrivilegesFromRole("NewReadWrite", [{resource:{db:"test",collections:""},
action:["insert"]}])
```

- Privilege Management

To list the privileges given for a role

```
db.getRole("NewReadWrite",{showPrivileges:true})
```

To list all the user defined roles with privileges

```
db.getRoles({rolesInfo: 1, showPrivileges:true,showBuiltinRoles: false})
```

We can specify some parameters to getRoles()

- Listing built in Roles

```
use admin
```

```
db.getRoles({rolesInfo: 1, showPrivileges:true, showBuiltinRoles: true})
```

List all the database built-in roles

- Granting Roles to Roles

- > use admin

- > `db.grantRolesToRole("NewReadWrite",[{role:"read",db:"local"}])`

Grant roles to roles to another database

- Revoking Roles from Roles

- > use admin

- > `db.revokeRolesFromRole("NewReadWrite",[{"role":"read",db:"local"}])`

Revoking roles to roles to another database

- How to change own password

To change own password , user should have `changeOwnPassword` and `changeOwnCustomData` privileges

## Steps

Create a user defined role with the privileges `changeOwnPassword` and `changeOwnCustomData` privileges

Create the user with the role or grant this role to the existing user.

```
-> db.createRole({ role: "passwdRole", privileges: [{ resource: { db: "admin",
collection: "" }, actions: ["changeOwnPassword", "changeOwnCustomData"] }], roles: [] })
```

```
-> db.grantRolesToUser("dba", [{role:"passwdRole",db:"admin"}])
```

- Built-in Roles in MongoDB

## Database User Roles

Every database includes the following client roles:

- read
  - Provides the ability to read data on all non-system collections and the system.js collection.
- readWrite
  - Provides all the privileges of the read role plus ability to modify data on all non-system collections and the system.js collection.

- Built-in Roles in MongoDB

## Database Administrator Roles

Every database includes the following database administration roles:

- dbAdmin
  - Provides the ability to perform administrative tasks such as schema-related tasks, indexing, and gathering statistics. This role does not grant privileges for user and role management.
- userAdmin
  - Create and modify roles and users on the current database
- dbOwner
  - The database owner can perform any administrative action on the database. This role combines the privileges granted by the readWrite, dbAdmin and userAdmin roles.

- Built-in Roles in MongoDB - userAdmin

Provides the ability to create and modify roles and users on the current database. Since the userAdmin role allows users to grant any privilege to any user, including themselves, the role also indirectly provides superuser access to either the database or, if scoped to the admin database, the cluster.

The userAdmin role explicitly provides the following actions:

|                  |                              |
|------------------|------------------------------|
| changeCustomData | dropUser                     |
| changePassword   | grantRole                    |
| createRole       | revokeRole                   |
| createUser       | setAuthenticationRestriction |
| dropRole         | viewRole                     |
|                  | viewUser                     |

- Built-in Roles in MongoDB - Cluster Administration Roles

The **admin** database includes the following roles for administering the whole system rather than just a single database. These roles include but are not limited to replica set and sharded cluster administrative functions.

- clusterManager

- Provides management and monitoring actions on the cluster. A user with this role can access the config and local databases, which are used in sharding and replication, respectively.

- enableharding

- Provides the ability to enable sharding for a collection and modify existing shard keys.

- Built-in Roles in MongoDB - Cluster Administration Roles

## clusterMonitor

Provides read-only access to monitoring tools, such as the MongoDBCloud Manager and Ops Manager monitoring agent.

## hostManager

Provides the ability to monitor and manage servers.

## clusterAdmin

Provides the greatest cluster-management access. This role combines the privileges granted by the clusterManager, clusterMonitor, and hostManager roles. Additionally, the role provides the dropDatabase action.

- Backup and Restore Roles

The **admin** database includes the following roles for backing up and restoring data:

## Backup

Provides minimal privileges needed for backing up data. This role provides sufficient privileges to use the MongoDB Cloud Manager backup agent, Ops Manager backup agent, or to use mongodump to back up an entire mongod instance.

## Restore

Provides privileges needed to restore data from backups.

- All Database Roles

The following roles are available on the admin database and provide privileges which apply to all databases except local and config

## `readAnyDatabase`

Provides the same read-only privileges as read on all databases except local and config. The role also provides the `listDatabases` action on the cluster as a whole

## `readWriteAnyDatabase`

Provides the same privileges as `readWrite` on all databases except local and config. The role also provides the `listDatabases` action on the cluster as a whole.

- All Database Roles

## userAdminAnyDatabase

Provides the same access to user administration operations as userAdmin on all databases except local and config.

## dbAdminAnyDatabase

Provides the same privileges as dbAdmin on all databases except local and config. The role also provides the listDatabases action on the cluster as a whole.

- Super User Roles

root

Provides access to the operations and all the resources of the following roles combined:

- readWriteAnyDatabase
- dbAdminAnyDatabase
- userAdminAnyDatabase
- clusterAdmin
- restore
- backup

- Super User Roles

root

Provides access to the operations and all the resources of the following roles combined:

- readWriteAnyDatabase
- dbAdminAnyDatabase
- userAdminAnyDatabase
- clusterAdmin
- restore
- backup

- Super User Roles

The following roles provide the ability to assign any user any privilege on any database, which means that users with one of these roles can assign themselves any privilege on any database:

- dbOwner role, when scoped to the admin database

- userAdmin role, when scoped to the admin database

- userAdminAnyDatabase role

- How to assign privileges to a user

As of version current MongoDB doesn't support assigning privileges directly to the user

## Steps

1. Create a user defined role
2. Assign the required privileges to the role
3. Assign the role to the user



We Work With Data

Thank you!

[ailton.moreira@openup.pt](mailto:ailton.moreira@openup.pt)



# MongoDB – Administration

## Introducing MongoDB

Ailton Moreira  
ailton.moreira@openup.pt

VN Gaia 2024

- Enable and Configure Audit Output

Use the `--auditDestination` option to enable auditing and specify where to output the audit events .

This option is set while starting the mongod using command line

The same can be set in the `mongod.conf`, using the following parameter  
auditLog:

destination:<value>

- Enable and Configure Audit Output

What all being audited

MongoDB will be auditing the following actions

- schema (DDL),
- replica set and sharded cluster,
- authentication and authorization, and
- CRUD operations (requires `auditAuthorizationSuccess` set to true).

- Enable and Configure Audit Output

What are the audit destination

MongoDB can write the audit information to any of the following output

Syslog ( Not available on Windows)

Console

JSON file

BSON File

- Enable and Configure Audit Output

Writing the auditing information to console

```
$ mongosh --dbpath/data/db --auditDestination console
```

Or in the configuration file

storage:

dbPath: /data/db

auditLog:

destination: console

- Enable and Configure Audit Output

Writing the auditing information to SYSLOG

```
$ mongoh --dbpath/u01/data --auditDestination syslog
```

Or in the configuration file

storage:

dbPath: /u01/data

auditLog:

destination: syslog

- Enable and Configure Audit Output

Writing the audit information to json

```
$ mongosh --dbpath/data/db--auditDestinationfile --auditFormat JSON --auditPath
data/db/auditLog.json
```

Or

storage:

dbPath: /data/db

auditLog:

destination: file

format: JSON

path: /data/db/auditLog.json

- Enable and Configure Audit Output

Writing the audit information to bson

```
$ mongosh --dbpath/data/db--auditDestinationfile --auditFormat BSON --auditPath
data/db/auditLog.bson
```

Or

storage:

dbPath: /data/db

auditLog:

destination: file

format: BSON

path: /data/db/auditLog.bson

- Enable and Configure Audit Output

## Auditing CURD Operations

```
$mongosh --port 27017 --dbpath/u01/data --auth--
setParameterauditAuthorizationSuccess=true --auditDestinationfile --auditFilter'{ atype:
"authCheck", "param.command": { $in: ["find", "insert", "delete", "update", "findandmodify"] }
' --auditFormatJSON --auditPath/u01/data/auditLog.json
```

- Enable and Configure Audit Output

Or update in the configuration file

storage:

dbPath: data/db

security:

authorization: enabled

auditLog:

destination: file

format: BSON

path: data/db/auditLog.bson

filter: '{ atype: "authCheck", "param.ns": "test.orders", "param.command": {

\$in: [ "find", "insert", "delete", "update", "findandmodify" ] } }'



We Work With Data

Thank you!

[ailton.moreira@openup.pt](mailto:ailton.moreira@openup.pt)