100 Core MongoDB Interview Questions

You can also find all 100 answers here 👉 Devinterview.io - MongoDB

1. What is *MongoDB* and what are its main features?

MongoDB is a robust, document-oriented NoSQL database designed for high performance, scalability, and developer agility.

Key Features

Flexible Data Model

- Employs **JSON-like documents** (BSON format), facilitating complex data representation, deep nesting, and array structures.
- Provides dynamic schema support, allowing **on-the-fly data definition and data types**.
- Permits multi-document transactions within a replica set (group of nodes). **Sharding** extends this to support large distributed systems.

Indexed Queries

- Offers extensive indexing capabilities, such as single and multi-field support, **text**, **geospatial**, and **TTL** (Time to Live) Indexes for data expiration.
- Gives developers the tools needed to design and optimize query performance.

High Availability & Horizontal Scalability

- Uses replica sets for data redundancy, ensuring **auto-failover** in the event of a primary node failure.
- Adopts sharding to **distribute data across clusters**, facilitating horizontal scaling for large datasets or high-throughput requirements.

Advanced Querying

- Engages in **ad-hoc querying**, making it easy to explore and analyze data.
- Provides **aggregation pipeline**, empowering users to modify and combine data, akin to SQL GROUP BY.
- Specialized query tools like **Map-Reduce** and **Text Search** cater to distinctive data processing needs.

Embedded Data Management

- Encourages a rich, document-based data model where you can **embed related data** within a single structure.
- This denormalization can enhance read performance and data retrieval simplicity.

Rich Tool Suite

- Further augmented by several desktop and web-supported clients, MongoDB Atlas offers a seamless and unified experience for database management.
- Web-based MongoDB Compass handles query optimization and schema design.

Code Sample: Data Interaction with MongoDB

Here is the Python code:

```
from pymongo import MongoClient

client = MongoClient()  # Connects to default address and port
db = client.get_database('mydatabase')

# Insert a record
collection = db.get_collection('mycollection')
inserted_id = collection.insert_one({'key1': 'value1', 'key2': 'value2'}).inserted_id

# Query records
for record in collection.find({'key1': 'value1'}):
    print(record)

# Update record
update_result = collection.update_one({'_id': inserted_id}, {'$set': {'key2': 'new_value'}})
print(f"Modified {update_result.modified_count} records")

# Delete record
delete_result = collection.delete_one({'key1': 'value1'})
print(f"Deleted {delete_result.deleted_count} records")
```

2. How does *MongoDB* differ from relational databases?

While both **MongoDB** and relational databases handle data, they do so in fundamentally different ways. Let's explore the key distinctions.

Data Model

Relational Databases

- Use tables with predefined schemas that **enforce relationships and data types**.
- Often use normalization techniques to minimize data redundancy.

MongoDB

- Stores data as **flexible, schema-less** sets of key-value pairs inside documents.
- Relationships can be represented through embedded documents or referencing via keys, providing more granular control and allowing for a more natural representation of real-world data.

Data Integrity

Relational Databases

- Rely on **ACID transactions** to ensure data consistency.

MongoDB

- Offers **ACID guarantees** at the document level, though transactions across multiple documents happen within the same cluster to ensure consistency.
- Provides **multi-document transactions** for more complex operations.

Query Language

Relational Databases

- Use SQL, a **declarative** query language.

MongoDB

- Employs **JSON-like** queries, which are **imperative** and resemble the structure of the data it operates on.

Scalability

Relational Databases

- Traditionally use a **vertical scaling** approach, featuring limits on a single server's resources such as CPU, storage, and memory.

MongoDB

- Designed for **horizontal scaling**, making it easier to handle larger datasets and heavier loads by distributing data across multiple servers. This scalability also supports cloud-based setups.

Performance

Relational Databases

- Can handle complex queries efficiently but might require multiple joins, potentially degrading performance.

MongoDB

- Optimized for quick CRUD operations and can efficiently handle large volumes of read and write requests.

Indexing

Relational Databases

- Tables can have a multitude of indexes, which can be a mix of clustered, non-clustered, unique, or composite.

MongoDB

- Collections can have several indexes, including single field, compound, and multi-key indexes.

Data Joins

Relational Databases

- Use joins to merge related data from different tables during a query, ensuring data integrity.

MongoDB

- Offers **embedded documents** and **manual reference** to achieve similar results, but multi-collection joins have performance and scalability considerations.

3. Can you describe the structure of data in *MongoDB*?

In **MongoDB**, data units are organized into **collections**, which group related documents. Each **document** corresponds to a single **record** and maps to fields or **key-value pairs**.

JSON-Like Format

Data in MongoDB is stored using a **BSON** (Binary JSON) format that can handle a maximum depth of 100 levels. This means a BSON object or element can be a document consisting of up to 100 sub-elements, such as fields or values.

Example: Nested Document

Here is a nested document:

```
{
  "_id": "123",
  "title": "My Blog Post",
  "author": {
    "name": "John Doe",
    "bio": "Tech enthusiast"
  },
  "comments": [
    {
      "user": "Alice",
      "text": "Great post"
    },
    {
      "user": "Bob",
      "text": "A bit lengthy!"
    }
  ]
}
```

In the example above, the "author" field is an embedded document (or sub-document), and the "comments" field is an array of documents.

Key Features

- **Ad-Hoc Schema**: Documents in a collection don't need to have the same fields, providing schema flexibility.
- **Atomicity at the Document Level**: The `ACID` properties (Atomicity, Consistency, Isolation, Durability) of a transaction, which guarantee that the modifications are successful or unsuccessful as a unit of work.
- **Index Support**: Increases query performance.
- **Support for Embedded Data**: You can nest documents and arrays.
- **Reference Resolution**: It allows for processing references across documents. If a referenced document is modified or deleted, any reference to it from another document also needs to be updated or deleted in a multi-step atomic operation.
- **Sharding and Replication**: For horizontal scaling and high availability.

Data Model Considerations

1. **One-to-One**: Typically achieved with embedded documents.
2. **One-to-Many (Parent-Child)**: This can be modelled using embedded documents in the parent.
3. **One-to-Many (Referenced)**: Achieved through referencing, where several documents contain a field referencing a single document. For better efficiency with frequent updates, consider referencing.
4. **Many-to-Many**: Modeled similarly to "One-to-Many" relationships.
5. **You should avoid** using “repeatable patterns”, such as storing data in separate arrays or collections, to ensure smooth data manipulation and effective query operations.

   For example, using separate collections for similar types of data based on a category like "users" and "admins" instead of a single "roles" array with multiple documents.

The above best practice example prevents **data redundancy** and ensures **consistency** between similar documents. Redundant storage or separating non-redundant data can lead to inconsistencies and increase the effort required for maintenance.

4. What is a *Document* in *MongoDB*?

In MongoDB, a **document** is the basic data storage unit. It's a JSON-like structure that stores data in key-value pairs known as fields.

Document Structure

Each **document**:

- Is a top-level entity, analogous to a row in a relational database.
- Is composed of **field-and-value** pairs, where the value can be a variety of data types, including arrays or sub-documents.
- Has a unique `_id` or primary key that is indexed for fast lookups.

Here is the document structure:

```
{
  "_id": 1,
  "name": "John Doe",
  "age": 30,
  "email": "john.doe@email.com",
  "address": {
    "city": "Example",
```

```
    "zip": "12345"
  },
  "hobbies": ["golf", "reading"]
}
```

Collections

Documents are grouped into **collections**. Each collection acts as a container with a unique namespace within a database. Collections don't enforce a predefined schema, which allows for flexibility in data modeling.

Key Advantages

1. **Flexibility**: Documents can be tailored to the specific data needs of the application without adherence to a rigid schema.
2. **Data Locality**: Related data, like a user's profile and their posts, can be stored in one document, enhancing performance by minimizing lookups.
3. **JSON Familiarity**: Documents, being JSON-like, enable easier transitions between application objects and database entities.
4. **Indexing**: Fields within documents can be indexed, streamlining search operations.
5. **Transaction Support**: Modern versions of MongoDB offer ACID-compliant, multi-document transactions that ensure data consistency.

Example Use Case

Consider an online library. Instead of having separate tables for users, books, and checkouts as in a relational database, you could store all the pertinent data about a user, including their checked-out books, in a **single document** within a `users` collection:

```
{
  "_id": 1,
  "name": "John Doe",
  "email": "john.doe@email.com",
  "address": { "city": "Example", "zip": "12345" },
  "checkedOutBooks": [
    { "bookId": 101, "dueDate": "2022-02-28" },
    { "bookId": 204, "dueDate": "2022-03-15" }
  ]
}
```

This approach enables swift retrieval of all pertinent user information in one go.

Considerations

- **Atomicity**: While single-document operations are atomic by default in MongoDB, transactions and atomicity guarantee apply to multi-document operations primarily.
- **Size Limitations**: Documents can't exceed 16MB in size. In most cases, this limit should not be a practical concern.

5. How is data stored in *collections* in *MongoDB*?

In **MongoDB**, data is stored in **types of collections**, ensuring flexibility and efficiency in data modeling.

Collection Basics

- Collections are the **primary data storage structures** in MongoDB, akin to tables in relational databases.
- They are schema-less, meaning that documents within a collection can have varying structures. This offers superior flexibility, while still allowing for structure validation through the use of JSON schema.

Documents

- **Documents** serve as the unit of data storage in MongoDB. These are akin to rows in relational databases or objects in languages such as JavaScript.
- Documents are represented in **BSON** (Binary JSON) format, a binary representation closely mirroring JSON's attribute-value data model.

Data Organization Hierarchy

- Data in MongoDB is organized in a **hierarchical structure**, with each database having one or more **collections**, each of which stores multiple **documents**, all of which can possess distinct structures.

Key Data Principles

- MongoDB collections are designed to **optimize** data access rather than just serving as containers.
- To maximize efficiency, it's crucial to design collections that cater to common query patterns.

Types of Database Collections

- By understanding the nuances of each collection type, you can better customize your MongoDB system to **cater to specific use-cases and performance requirements**.

AJAX Comments

- To effectively and iteratively store and manage comments, the AJAX Comments feature is engineered to provide a blend of flexibility and ease of access.
- It leverages **JSON-like documents** and the native power of MongoDB, such as **rich indexing** for efficient interactions.

Newsfeed Posts

- Tailored for sequential, feed-like content, such as posts from a social media platform or a messaging app.
- It benefits greatly from the ordered nature of **BSON documents**, making sure newer posts are easy to fetch.

User Profiles

- Focusing on user-defined, diverse, and possibly unstructured details, the User Profile collection is an ideal repository for self-descriptive user profiles.
- The **flexibility** of schema allows for comprehensive storage with minimal overhead.

Metadata

- For persistent and global configurations, the Metadata collection provides a secure space to cache system information.

Product Catalog

- Bolsters browsing and shopping activities by housing consistent, structured details related to products or services on offer.
- This attention to **consistency** helps in easy data retrieval and optimized user experiences.

Logging

- Ideally suited to record system interactions and debugging info, the Logging collection maintains an organized trail of system activity, nurturing a culture of informed decision-making.

6. Describe what a *MongoDB database* is.

A **MongoDB database** is a document-oriented, NoSQL database consisting of collections, each of which in turn comprise documents.

Core Concepts

1. Collection

- A collection is a grouping of MongoDB documents. A collection is the **equivalent of a table** in a relational database.

Advantages of Using Collections:

- **Flexibility**: Each document in a collection can have its own set of fields. Structural changes are easier to manage than in traditional, rigid SQL tables.
- **Scalability**: Collections can be distributed across multiple servers or clusters to handle large data volumes.

2. Document

- Synonymous with a record, a **document** is the main data storage unit in MongoDB. It is a set of key-value pairs.
    - Key: The field name
    - Value: The data

Document-Key Pairs:

- Each document maintains a unique ID, known as the **object ID** which is autogenerated. This ensures every document is distinct.
- Unlike SQL databases where each row of a table follows the same schema, a document can be more fluid, accommodating fields as required.

Considerations When Choosing the Level of Normalization:

- **Optimized Reads**: Normalization into separate collections may be beneficial if there are large amounts of data that might not always have to be fetched.
- **Batch Inserts and Updates**: Denormalization often leads to simpler write operations. If there will be a lot of changes or inserts, denormalization can be more efficient.
- **Atomicity**: When data that belongs together is split into different collections, ensuring atomicity can become difficult.

3. Field

- A **field** is a single piece of data within a document. It's synonymous with a database column.
    - **Field Type**: MongoDB supports multiple field types, including arrays.

- **Limit on Nested Fields**: Documents can be nested, which is like being able to have sub-documents within a main document. However, there is a depth limitation: you can't embed documents endlessly.

Schema

MongoDB is often regarded as **schema-less**, but a more accurate description is that it's **flexible**. While documents within a single collection can have different fields, a robust schema design process is still essential.

Adapting to Evolving Schemas:

- **Versioning**: Managed schema changes and versioning in the application layer.
- **Schema Validation**: Introduced in MongoDB 3.2, this feature allows for the application of structural rules to documents.
- **Education and Training**: Properly educating developers on the use of a database can minimize potential misuse of its flexibility.
- **Use of Techniques to Ensure Data Integrity**: Techniques such as double-entry bookkeeping can assure data accuracy, especially when dealing with multiple, occasionally outdated records.

Modeling vs. Tuning Approaches

- **Normalization**: Seeks to reduce redundancy and improve data consistency.
- **Denormalization**: Emphasizes performance gains. Redundancies are knowingly introduced for optimized and rapid reads.
- **Use Cases Dictate**: Neither is definitively superior; their suitability depends on the specific use case.

7. What is the default *port* on which *MongoDB* listens?

The default **port number** for MongoDB is 27017. While it is possible to run multiple instances of MongoDB on the same machine, each instance must have its unique port number to ensure they don't conflict.

8. How does *MongoDB* provide high availability and disaster recovery?

MongoDB ensures high availability and disaster recovery through a robust data architecture and a distributed system model. It integrates various mechanisms to maintain data integrity, uptime assurances, and data redundancy.

Key Components

1. **Replica Sets**: These are clusters of MongoDB nodes that use automatic failover to maintain data consistency.
2. **WiredTiger Storage Engine**: It powers numerous features including data durability, in-memory storage, and compression.
3. **Oplog**: Short for "operations log", it records all write operations in an append-only manner.
4. **Write Concerns**: These are rules that determine the level of acknowledgment required for write operations.
5. **Read Preferences**: They define which nodes in a cluster can satisfy read operations.
6. **Data Centers**: Hardware resilience can be achieved by distributing nodes across multiple data centers.

7. **Backups and Restores**: MongoDB offers built-in mechanisms to backup and restore data, further aiding in disaster recovery.
8. **Monitoring Tools**: For performance tracking and potential issue detection.
9. **Technology Agnostic**: Can deploy on multi-cloud, hybrid and on-premises architectures.

Data Recovery Modes

1. **Restore**: Achieved through the backup of data when the config server is the only component that is active and accurate. This method doesn't consider data changes made after the backup was captured.
2. **Oplog Replays**: This involves using oplogs that track changes, ensuring that even after a cluster restart, any missed transactions are reinstated.
3. **Snapshotting**: It is a consistent snapshot of data across the nodes in the replica set.

Code Example: Write Concerns and Oplog

Here is the Python code:

```
# Import the MongoClient class from pymongo.
from pymongo import MongoClient

# Establish connection to the MongoDB server using MongoClient.
client = MongoClient('mongodb://localhost:27017/')

# Assign the test database to a variable
db = client.test

# Assign the collection within the test database to a variable
collection = db.test_collection

# Insert a document into the collection and set the write concern to 'majority'
result = collection.insert_one({'test_key': 'test_value'}, write_concern={'w': 'majority'})

# Fetch the oplog entry associated with the insert operation.
oplog_cursor = db.local.oplog.rs.find({'ns': 'test.test_collection', 'op': 'i'})

# Access the result and compare the count to ensure the operation was recorded in the oplog.
operation_count = oplog_cursor.count()
```

Recommendations

- Employ consistent and comprehensive **backup** strategies in conjunction with multi-faceted recovery plans.

9. What are *indexes* in *MongoDB*, and why are they used?

Indexes are employed in **MongoDB** to optimize database queries by providing faster access to data. Without indexes, MongoDB performs full collection scans.

Common Types of Indexes in MongoDB

- **Single Field Index**: The most basic form of index.
- **Compound Index**: Generated across multiple fields; used for queries involving these fields.
- **Multikey Index**: Specially designed for arrays or embedded documents.

Batch Insert Operations on an Indexed Collection Describe any performance bottlenecks you anticipate.

- **Text Index**: Suited for text searches, often leveraging stemming and stop words.

Unique Explain in which situations it's beneficial to manage a unique index. Discard icon GEO Index Describe the purpose of this index type and the type of queries it can optimize.

- **TTL (Time-to-Live) Index**: Deletes documents after a specified duration, suitable for logs and cached data.

Common Performance Bottlenecks with Indexes

- **Index Overuse**: Too many indexes can degrade write performance.
- **Index Size**: Larger indexes consume more RAM and might slow down read and write operations.
- **Index Inefficiency**: Inaccurate or non-selective index usage can render them ineffective.
- **Write Penalties**: Indexes incur an overhead during writes, impacting their efficiency in write-heavy systems.
- **Index Maintenance**: Regular maintenance, like rebuilding or reorganizing indexes, is often necessary.
- **Workload Misalignment**: An index might not be beneficial if it's not aligned with the actual query workload.

Make sure to keep the indexes required and remove any unnecessary ones.

10. What is the role of the *id field* in *MongoDB documents*?

The `_id` Field in MongoDB serves as a **primary key** and provides several key functionalities:

- **Uniqueness Guarantee**: Each document must have a unique `_id`, which ensures data integrity.
- **Automatic Indexing**: Automated indexing based on `_id` enhances query efficiency.
- **Inherent Timestamp**: The `_id` can have an embedded timestamp, useful for time-based operations.

  For instance, with an **ObjectId**, the first 8 characters represent a 4 byte timestamp:

  $$\text{timestamp} = \text{substr}(\text{ObjectId}, 0, 8)$$

- **Concurrency Control**: If multiple write operations with the same `_id` occur simultaneously, MongoDB uses a technique called **last-write wins** to manage the conflict:

  The document with the most recent `_id` value, or timestamp if using an ObjectId, supersedes the others.

- ***Modify and Return***: When executing an operation to insert a new document or find & modify an existing one, you can request to return the modified document and its `_id`.

ObjectId vs. Custom `_id`

README

- **Custom Representations**: Unleash flexibility by using custom strings, numbers, or other valid BSON types for the `_id` field.
- **Controlled Uniformity**: Design your own `_id` strategy to align with data, such as employing natural keys for documents originating from specific, external sources.
- **Migrate with Care**: Once an application is live, altering the structure can be intricate. Transition plans are vital for a seamless shift.

- **Custom Indexing**: Managing an index on a uniquely generated custom `_id` turns the data into a compact, high-throughput structure.

Schema Design and the `_id` Field

The choice between automatic ObjectId and custom `_id` values links back to the **intended data model, data access patterns**, and specific **domain requirements**.

While using the automatic ObjectId brings about benefits like **ease of use** and **embedded timestamp**, custom `_id` generation provides finer control and helps in scenarios where a specific data structure is favored or where external data sources need to be integrated.

11. How do you create a new *MongoDB collection*?

The process for creating a new collection in MongoDB is simple and instantaneous.

Benefits of Instantaneous Creation

- MongoDB collections are schemaless, leading to immediate collection creation.
- Document structure and content drive schema design.
- No predefined schema requirements allow for dynamic, evolving data models.

Steps to Create a Collection

1. **Select the Database:** Ensure you are connected to the intended database for the collection's creation. Switch to the desired database using `use` in the `mongo` shell or select the database programmatically in your driver's API.
2. **Perform a Write Operation:** The new collection is created the moment you execute a write operation such as `insert`, `update`, or `save`.
3. **Check Collection Existence (Optional):** While not necessary for the creation process, you can verify the collection is created using the listCollections method.

12. What is the syntax to insert a *document* into a *MongoDB collection*?

To **insert a document** into a **MongoDB collection**, you can use the `insertOne()` method, which accepts the document as an argument:

```
db.collectionName.insertOne({
  key1: "value1",
  key2: 2,
  key3: [1, 2, 3],
  key4: { nestedKey: "nestedValue" }
});
```

Alternatively, you can use the `insertOne()` method, supply an array of documents with `insertMany()`:

```
db.collectionName.insertMany([
  { key: "value1" },
  { key: "value2" }
]);
```

13. Describe how to read data from a *MongoDB collection*.

To **read** data from a **MongoDB collection**, you use the `find` method with various options for querying and data manipulation.

Key Methods

- **find**(filter, projection): Retrieves documents based on filter conditions. You can specify which fields to include or exclude in the result (**projection**).
- **findOne**(filter, projection): Similar to `find` but retrieves only the first matching document.
- **distinct**(field, filter): Returns a list of distinct values for a specific field, optionally filtered.

Query Operators

- **Comparison**: `$eq`, `$gt`, `$lt`, `$in`, `$nin`, etc.
- **Logical**: `$and`, `$or`, `$not`, `$nor`, etc.
- **Element**: `$exists`, `$type`
- **Evaluation**: `$regex`, `$mod`, `$text`
- **Geospatial**: `$geoNear`, `$geoWithin`, etc.

Aggregation

MongoDB also provides the **aggregation framework** for complex operations, using a pipeline of various stages like `match`, `group`, `sort`, `limit`, etc.

Example: Basic Find Query

Here is a Python code:

```
import pymongo

client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["mydatabase"]
collection = db["mycollection"]

# Retrieve all documents
all_documents = collection.find()

# Alternatively, you can iterate through the cursor:
for doc in all_documents:
    print(doc)
```

Example: Querying with Filters

Here is a Python code:

```
# Let's say we have the following documents in the collection:
# [{
#    "name": "John",
#    "age": 30,
#    "country": "USA"
#  },
#  {
#    "name": "Jane",
#    "age": 25,
#    "country": "Canada"
# }]
```

```
# Retrieve documents where the name is "John"
john_doc = collection.find_one({"name": "John"})
print(john_doc)  # Output: {"name": "John", "age": 30, "country": "USA"}

# Retrieve documents where age is greater than or equal to 25 and from country "USA"
filter_criteria = {"age": {"$gte": 25}, "country": "USA"}
docs_matching_criteria = collection.find(filter_criteria)
for doc in docs_matching_criteria:
    print(doc)
    # Output: {"name": "John", "age": 30, "country": "USA"}
```

Projection

Projection helps control the fields returned. It uses a dictionary where fields to include are marked with 1, and those to exclude with 0.

For instance, `{"name": 1, "age": 1, "_id": 0}` only includes `name` and `age` while excluding `_id`:

Here is a Python code:

```
# Retrieve the name and age fields, ignoring the _id field
docs_with_limited_fields = collection.find({}, {"name": 1, "age": 1, "_id": 0})
for doc in docs_with_limited_fields:
    print(doc)
    # Output: {"name": "John", "age": 30}
    #         {"name": "Jane", "age": 25}
```

Sort, Skip, and Limit

`sort`, `skip`, and `limit` help in reordering, pagination, and limiting the result size.

Here is a Python code:

```
# Sort all documents by age in descending order
documents_sorted_by_age = collection.find().sort("age", -1)

# Skip the first two documents and retrieve the rest
documents_after_skipping = collection.find().skip(2)

# Limit the number of documents returned to 3
limited_documents = collection.find().limit(3)
```

Distinct Values

Here is a Python code:

```
# Suppose, the collection has a "country" field for each document

# Get a list of distinct countries
distinct_countries = collection.distinct("country")
print(distinct_countries)  # Output: ["USA", "Canada"]
```

Indexes

Indexes improve read performance. Ensure to use appropriate indexes for frequent and complex queries to speed up data retrieval. If the queries differ from the indexing pattern or if the collection is small, the gain from indexing might be insignificant, or it could even affect the write performance of the database. Choose an indexing strategy based on your data and usage patterns.

For example, if you frequently query documents based on their "country" field, consider creating an index on that field:

Here is a Python, PyMongo code:

```
collection.create_index("country")
```

This would make lookups based on the "country" field more efficient.

14. Explain how to update *documents* in *MongoDB*.

MongoDB offers several ways to update documents (equivalent to SQL's "rows"). Let's look at the most common methods.

Update Methods

- **Replace**: Entire document is updated. This is the closest equivalence to SQL's `UPDATE` statement.
- **Update**: For selective field updates, you use `$set`, `$inc`, `$push`, `$unset`, and more. This resembles SQL's `UPDATE` with selective column updates.

Replace & Update in *MongoDB*

Top-Down Approach Using Replace

- **Method**: `db.collectionName.updateOne()`
- **Code**:

```
db.collectionName.updateOne(
    {"name": "John Doe"},
    {$set: {"age": 30}}
);
```

- **Use-Case**: When replacing an entire document isn't needed. For example, when changing a user's email address.

Bottom-Up Approach Using Update + $set

- **Method**: `db.collectionName.replaceOne()`
- **Code**:

```
db.collectionName.replaceOne(
    {"name": "John Doe"},
    {"name": "John Doe", "age": 30}
);
```

- **Use-Case**: When an entire document needs updating or replacing, such as a product detail or a user's information.

15. What are the *MongoDB commands* for deleting *documents*?

MongoDB offers several methods for deleting documents.

Deletion Methods in MongoDB

1. **deleteOne()**: Deletes the first matched document.
2. **deleteMany()**: Removes all matching documents.
3. **remove()**: Legacy function; use `deleteOne()` or `deleteMany()` instead.

General Syntax

- For `deleteOne()`, the syntax is:
  - **db.collection.deleteOne({filter}, {options})**
- For `deleteMany()`, the syntax is:
  - **db.collection.deleteMany({filter}, {options})**

Code Example: Deleting One or Many

Here is the MongoDB shell script:

```
// Connect to the database
use myDB;

// Delete a single document from 'myCollection'
db.myCollection.deleteOne({ name: "Document1" });

// Delete all documents from 'myCollection' with the condition 'age' greater than 25
db.myCollection.deleteMany({ age: { $gt: 25 } });
```

Explore all 100 answers here 👉 Devinterview.io - MongoDB

Releases

No releases published

Packages

No packages published