Release: 20230414

Topics we cover

Error handling

Idempotent vs Non-idempotent operations

Write concern

Authentication and connections

Object document mappers (ODM)

Data access layer models

Codecs, Drivers and APIs

Idempotent operations

For scale, operations need to be independent of each other.

An operation changing many things should not be in a transaction.

Use roll-forward, not roll-back

Ensure all operations are idempotent

Idempotent operations can be resubmitted and result in the same state.

- One key idea when coding with MongoDB is to try and make your writes idempotent - that way, you can easily redo them if they fail
- For example - if part way through, there is a failover event. This is handled automatically for single document updates, but for multi-document updates (updateMany), it's important to understand the idea of retrying on failure, not rolling back.

Idempotent operations

These write operations are idempotent:

- Inserts - with a supplied `_id` (2nd attempt will fail)
- Deletes - cannot delete a thing twice
- Updates - if using explicitly set values

Example: Set { paid: false } in 100,000 documents with updateMany() - If it fails part way through, we can retry the whole operation

For efficiency, we might only set it where it's not set already

- If you want to update every record in the database, then do it in a way that on failure, you can submit the request again - and ideally, it only changes those records it didn't change the first time.
- This is much better than anything that locks lots of data - some operations are naturally idempotent.

Non Idempotent operations

Example: **Give every employee a 10% pay rise**

```
db.hr.updateMany(
     {employee: true},
     { $mul : { salary: 1.1 }}
)
```

If stops part way through, not safe to just resubmit, but can fix by adding a flag:

```
db.hr.updateMany(
     {employee: true, junepayrise: null},
     { $mul : { salary: 1.1 }, $set:{ junepayrise: true }}
)
```

How can we remove the flag when we are done?

- Using $inc or $mul is a change based on the current value - so you cannot do it twice
- To get around this, we can add a new field, or have an array field and add a value in that, or even do something with a date or a single bit (MongoDB has bitwise query operators)

Error Handling

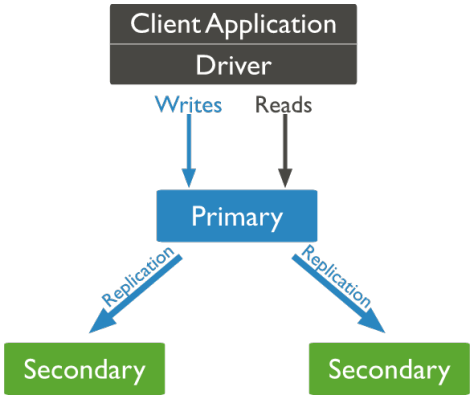
In a distributed system, transient errors are possible

Catch and handle Exceptions correctly:

- Some exceptions you can ignore
- Some you should retry
- Some you should retry with a change
- Some are fatal and need investigation

Can you think of examples?

Replication - The Essentials



- We cover replication and replica rets in detail later in the MongoDB developer training track. Here, we give a brief overview of replication as a primer for the one essential replication-related thing that a developer needs to know *when writing code* - namely, how to specify the right durability semantics for data written to the database.
- MongoDB achieves high availability by the use of a replica set.
- A replica set is a group of mongod instances that host the same data.
- In a replica set, one node is elected the primary node - the primary receives all write operations.
- All other nodes in the replica set replicate operations from the primary asynchronously.
- A client can choose to read from secondary nodes if desired; write operations can only be performed against the primary node.
- With this basic understanding about MongoDB replication, let's look briefly at write concerns ...

Write Concerns

Write concern is the level of acknowledgment requested from MongoDB for write ops

- Determines how durable we want the data to be before a 'success' response
- If a client specifies a weak write concern, data can be rolled back if failover
- Use w : "majority" when writing data that must survive an automatic failover

- As a developer, one of the most essential things to think about when writing your code is - when writing data to the database, does MongoDB wait for your writes to be durable, and durable to which failure events (server crash, replica set failover, etc.)?
- Starting in MongoDB 5.0, the implicit default write concern for most deployments is w: majority, meaning that MongoDB waits for a write operation to have propagated to the majority of the data-bearing voting members in the replica set before acknowledging the write.
- Write concern can be specified in several places with different scope - for example, in the connection string for all write operations using the connection, or on individual write operations.

Authentication and Connection

Authentication is complicated - don't just send a password

- Computationally expensive to prevent brute force attacks
- So it is important not to do it too much
- It is good practices to have a Singleton Client object
- This provides an efficient connection pool

Connection pools in the drivers manage actual connections

- Each connection has a queue - so the default is many, many threads
- Client should not really have many thousands of active threads
- If threads are blocked by the server - are there server operations that are too slow?

- Challenge response means computing expensive hashes at each end
- X509 requires some computation
- Kerberos and LDAP need network communications and computation
- Connection pools can be tuned but normally don't need to be - default is large and queues on each connection.
- Nothing should be in the connection queue more than a few ms.

Object Document Mappers (ODM)

ODMs are a popular for those new to MongoDB:

- Familiar as the same ODM/ORM has been used with an RDBMS
- Rigid schemas and class mapping feel reassuring
- Reduce complexity of learning a new database
- Common ODMs: Spring Data, Morphia and Mongoose

Downsides of using ODMs:

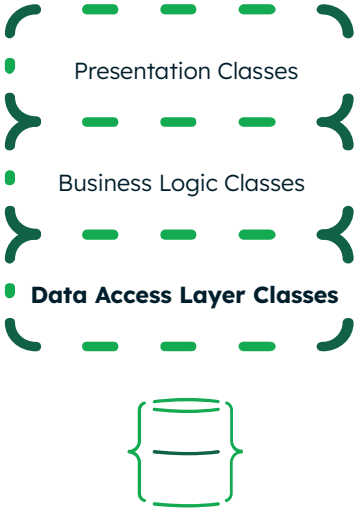
- Not officially supported by MongoDB
- Less efficient than the native MongoDB drivers
- Not allow developers to use best practises in schema design
- Use older drivers and prevent upgrading them
- Restrict what database functionality is available to an application

Data Access Layer Models

Using a Data Access Layer (DAL) model is a good way to develop for MongoDB

With a DAL a specific class or classes are responsible for all database interaction for a given business entity

DAL allows you to decouple the application object from the underlying schema and update mechanisms



- With a DAL a specific class or classes is responsible for all database interaction for a given business entity.
  - This class will model the business entity.
  - This class will provide access to members and functions on it
  - This class will explicitly manage database interactions relating to it.
- Using a DAL allows you to decouple the application object from the underlying schema and update mechanisms
  - This allows you to optimise queries and update operations.
  - It allows you to support older data seamlessly without performing data conversion
  - It allows you to avoid a 1:1 mapping between a business object and a document.

CODECs

Codecs are classes, registered with drivers, to convert native language to/from BSON
Many MongoDB Drivers support default and Custom Codecs

Default POJO/POCO (Plain old Java/CSharp) allow developers to persist class instances in the database and retrieve them seamlessly

Custom CODECs let developers define mapping between class and document

CODECs are not as flexible as DALs in abstracting away the database access but allow developers to save language objects without mapping them to documents explicitly

- Many MongoDB Drivers now support both default and Custom Codecs
- Codecs are classes, registered with the driver to convert a native language class to and from BSON
- The Default POJO/POCO (Plain old Java/CSharp) allow developers to persist class instances in the database and retrieve them seamlessly
- Custom CODECs let developers define the mapping between a class and the document that represents it
- CODECs are not as flexible as DALs in abstracting away the database access but do allow developers to easily save language objects without mapping them to documents explicitly

Driver Helpers

Everything is generally objects

```
query = new Document("firstname","bob").add("lastname","smith")
```

Queries, Update, Aggregations, UpdateOptions, etc. - all BSON documents
There are helpers for example Filters in Java, LINQ in C# - make queries more readable

```
query = and(eq("firstname","bob"),eq("lastname","smith"))
```

Avoid using JavaScript if that's not the language you are developing in.

```
✗ query = Document.parse("{firstname:" + name + "}")
```

- All communication with the server is via BSON objects - updates, data, queries, config options are all constructed as objects in the client.
- Some languages and drivers have helpers to make it easier to create objects to describe a specific thing - like a query.
- These are a perfectly good way of doing this - on the other hand, creating JSON then parsing it is a huge antipattern.
- If you are using MongoDB Compass or the Aggregaton Builders in Atlas or Ops Manager - these can generate code samples too.

Stable APIs

Stable API allows the DB to be upgraded without changing app behaviour

- Developers specify the API version when connecting to MongoDB to use it
- MongoDB guarantee to keep maintain compatibility with an existing API version
- Stable API v1 was introduced with MongoDB 5.0
- If behaviour changes in API version 2, apps that specify v2 see new behaviour
- Those connecting and specifying v1 the old behaviour remains

APIs and behaviour will be maintained "For many years to come"

Quiz Time!

#1. Long running update operations should be designed for:

A	Parallelism, run as fast as possible by using all the resources
B	Resumability, be able to efficiently restart on failure
C	Transactions, ensure to always succeed or fail atomically and instantly
D	Roll Back, undo changes made on failure
E	Idempotency, run multiple times with the same eventual outcome

Answer in the next slide.

#1. Long running update operations should be designed for:

A	Parallelism, run as fast as possible by using all the resources
B	Resumability, be able to efficiently restart on failure
C	Transactions, ensure to always succeed or fail atomically and instantly
D	Roll Back, undo changes made on failure
E	Idempotency, run multiple times with the same eventual outcome

Answer: B & E

#2. Which of the following write operations are always Idempotent?

A	InsertOne with a supplied _id
B	UpdateMany using $addToSet
C	UpdateOne using $inc
D	ReplaceOne
E	DeleteMany

Answer in the next slide.

#2. Which of the following write operations are always Idempotent?

A. InsertOne with a supplied _id

B. UpdateMany using $addToSet

C. UpdateOne using $inc

D. ReplaceOne

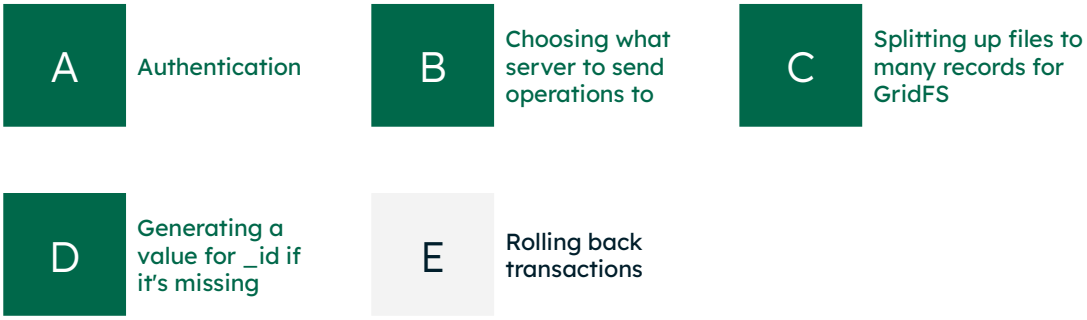
E. DeleteMany

#3. Which of the following should be natively coded in a driver?

A	Authentication
B	Choosing what server to send operations to
C	Splitting up files to many records for GridFS
D	Generating a value for _id if it's missing
E	Rolling back transactions

Answer in the next slide.

#3. Which of the following should be natively coded in a driver?



Answer: A, B , C & D