LESSON

Transactions in MongoDB

Google slide deck available here

Overview

Learning Objectives

At the end of this lesson, learners will be able to:

- Define what a transaction is in regards to databases.
- Define what it means to be ACID compliant in regards to databases.
- Explain the background of transaction development in MongoDB.
- Identify the two approaches to transactions in MongoDB, with Documents and with MongoDB Transactions, and when to use each of them.

Suggested Uses

- Lecture spread out across a couple of class periods
- Handouts / asynchronous learning
- Supplemental reading material - read on your own / not part of formal teaching
- Complement to University course MongoDB for SQL Pros.

This lesson is a part of the course Introduction to Modern Databases with MongoDB.

At a Glance

Length:
60 minutes

Level:
Advanced

Prerequisites:
MongoDB Architecture

Share your feedback: We hope these curriculum materials will be a valuable resource for you and your learners. Let us know how the materials work for you, what we can improve on, and how MongoDB for Academia can support you via our brief feedback form.

MongoDB for Academia: MongoDB for Academia offers resources for educators and students to support teaching and learning MongoDB. Check out our educator resources and join the Educator Community. Students can receive $50 in Atlas credits and free certification through the GitHub Student Developer Pack.

Last Update: March 2025

What is a Transaction?

A single unit of logic composed of multiple different database operations, which exhibits the following properties:

Atomic: either completes in its entirety or has no effect whatsoever (rolls back and is not left only partially complete)

Consistent: each transaction observes the latest current database state in the correct write ordering

Isolated: the state of an inflight transaction is not visible to other concurrent inflight transactions (and vice versa)

Durable: changes are persisted and cannot be lost if there is a system failure

A transaction represents a single unit which consists of multiple different database operations, all of these operations exhibit the following set of properties of Atomic, Consistent, Isolated, and Durable or ACID.

The property atomic for a transaction means it is either completes in its entirety or has no effect whatsoever (rolls back and is not left only partially complete)
The property consistent for a transaction means each transaction observes the latest current database state in the correct write ordering
The property isolated for a transaction means the state of an inflight transaction is not visible to other concurrent inflight transactions (and vice versa)
The property durable for a transaction means changes are persisted and cannot be lost if there is a system failure

In databases, transactions are a solution to ensure writes are ACID compliant, which stands for Atomicity (transactions are all or nothing), Consistency (only valid data is saved), Isolation (transactions do not affect each other), and Durability (written data will not be lost). If all of these properties are satisfied then the transaction is guaranteed to be ACID compliant.

Local and Global Transactions

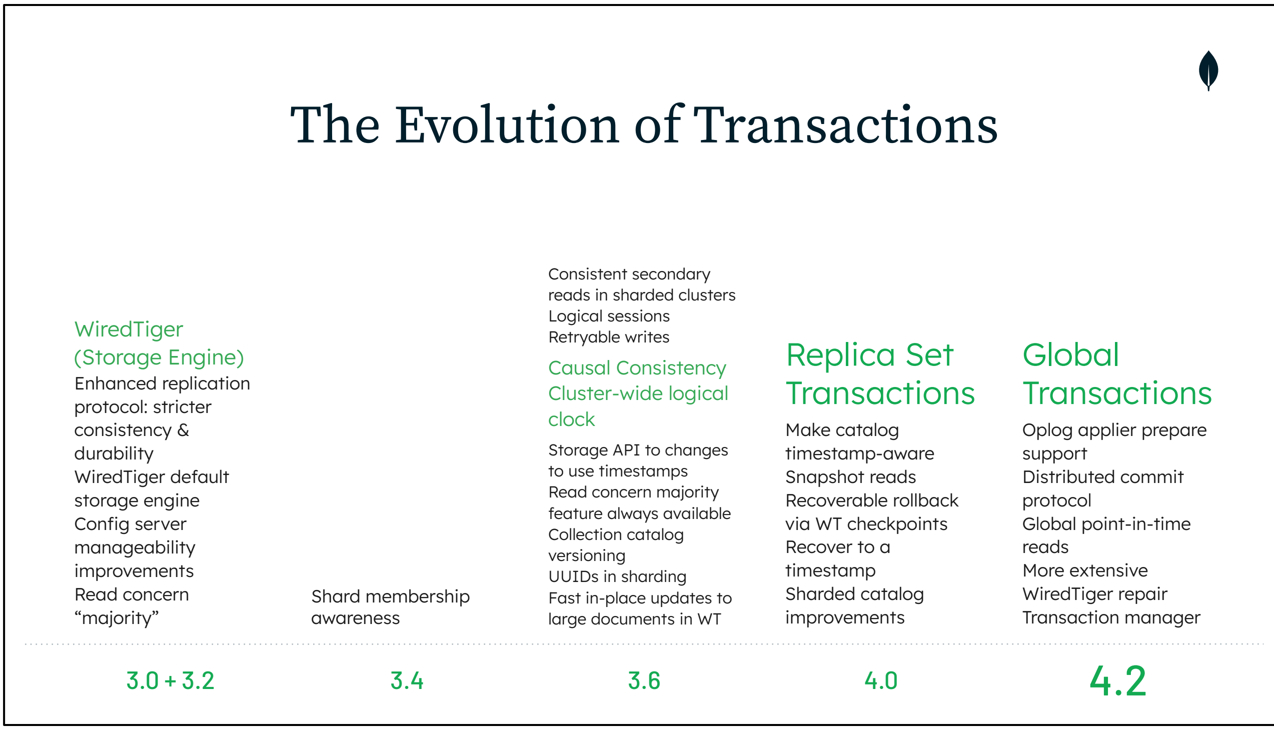
Local Transactions is when the single transaction and its operations are performed against the same database instance.

Global Transaction is when the single transaction and its operations are performed against two or more different instances.

Transactions can be defined as either local or global.

A local transaction is where the single transaction and the related operations are performed against the same database instance. This is one process on one machine, it isn’t distributed in the distributed systems sense.

A global transaction is where the single transaction and the related operations are performed against two or more different database instances. This is where the transaction occurs within a distributed system.



The path to transactions for MongoDB represents a multi-year engineering effort, which began with the integration of the WiredTiger storage engine. This required changes and enhancements to almost every part of the server – from the storage layer itself, to the replication consensus protocol, sharding architecture, consistency and durability guarantees, the introduction of a global logical clock, and refactored cluster metadata management and more.

This integration started with MongoDB 3.0.

WiredTiger is, and has been from its inception, an inherently transactional storage engine. One of the key reasons why WiredTiger was acquired back in 2014 was to provide the foundation for future transactions support within a distributed cluster. It was introduced in MongoDB version 3.0 and was subsequently made the default storage engine for MongoDB version 3.2 and beyond.WiredTiger has supported transactions for a long time. In versions 3.x, MongoDB used WiredTiger's 'transactions capability' to guarantee the modification atomicity of corresponding document, indexes, and oplog together.

MongoDB 3.6 introduced Causal Consistency. This allows developers to increase the strength of data consistency and provide guarantees around the ordering of their read and write operations. With causal consistency, MongoDB executes operations in an order that respects their causal relationship, and clients observe results that are consistent with this causal relationship.

MongoDB 3.6 also added a global ‘logical clock’. This enabled the capture of chronological and causal relationships of changing data across a sharded cluster. The sharded cluster has no concept of a physically synchronous global clock or absolute common time. Instead, the logical clock allows global ordering on events occurring on different shards of the same cluster. As a result, the cluster-wide ordering of asynchronous operations across distributed nodes can be established. This was an essential requirement to providing global transactions.

In MongoDB 4.0 Replica Set transactions were introduced. These were the first set towards true global transaction and only allowed global transactions within a single MongoDB replica set.

In MongoDB 4.2, Global Transactions were introduced. These built atop these to provide true global transaction abilities for sharded clusters.

Ordering Events in Database

How to order the events in a MongoDB cluster

Industry 2: Storage & Indexing SIGMOD '19, June 30–July 5, 2019, Amsterdam, Netherlands

Implementation of Cluster-wide Logical Clock and Causal Consistency in MongoDB

Misha Tyulenev
MongoDB, Inc.
misha@mongodb.com

Andy Schwerin
MongoDB, Inc.
schwerin@mongodb.com

Asya Kamsky
MongoDB, Inc.
asya@mongodb.com

Randolph Tan
MongoDB, Inc.
randolph@mongodb.com

Alyson Cabral
MongoDB, Inc.
alyson.cabral@mongodb.com

Jack Mulrow
MongoDB, Inc.
jack.mulrow@ mongodb.com

ABSTRACT

MongoDB is a distributed database that supports replication and horizontal partitioning (sharding). MongoDB replica

KEYWORDS: Causal Consistency, Eventual Consistency

ACM Reference Format

The ordering of events in a database is particularly important for transaction, we'll look at some of the work done in MongoDB to order the events to support transactions.

MongoDB's Logical Clock is based on the Lamport Clock mechanism (a monotonically increasing software counter). This is the core foundation to ordered events within MongoDB. A full write up of the work and issues faced by MongoDB in implementing this is discussed in the paper "Implementation of Cluster-wide Logical Clock and Causal Consistency in MongoDB". The link to the paper is https://dl.acm.org/doi/10.1145/3299869.3314049

Cluster participating nodes (mongod, mongos, config server, client/driver) always track and include the greatest known 'cluster time' when sending a message. Cluster time for Sharded clusters is required because Oplog time (optime) can only track the chronological order for a single replica set.

Every node keeps track of the maximum value of 'cluster time' it has ever seen and every node adds the latest time it is aware of to each message that it sends. Any node that receives that message jumps to that time. The only nodes that can increment time are the primary mongod nodes of each Shard.

The next aspect we'll look at that is important to transactions is the consistency of operations.

Consistency of operations

Balancing the performance and the safety of operations in terms of consistency

Tunable Consistency in MongoDB

William Schultz
MongoDB, Inc.
1633 Broadway 38th floor
New York, NY 10019
william.schultz@mongodb.com

Tess Avitabile
MongoDB, Inc.
1633 Broadway 38th floor
New York, NY 10019
tess.avitabile
@mongodb.com

Alyson Cabral
MongoDB, Inc.
1633 Broadway 38th floor
New York, NY 10019
alyson.cabral@mongodb.com

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352125

The consistency of operations is a key element to transactions. The implementation of transactions need to balance this along with performance to have usable transactions in terms of performance as well as consistency.

Causal consistency was a key feature required for transactions. To give an example of causal consistency, a write operation that deletes all documents based on a specified condition, followed by a read operation that verifies the delete operation means there is a causal relationship.

The work done by the engineering team at MongoDB on causal consistency was written up in a paper titled “Tunable Consistency in MongoDB”. Here’s the link to the paper https://dl.acm.org/doi/10.14778/3352063.3352125

The ability to leverage a cluster-wide logical clock was a linked requirement to develop this this feature. This two features combined then enabled every recorded CRUD operation to be tagged with its globally unique operation time to be able to infer the order.

In terms of the transactions, these were the two most visible and important features required after addressing the storage engine aspects with WiredTiger.

Quiz

Quiz

Which of the following were key features in MongoDB that enabled global transactions in MongoDB? More than one answer choice can be correct.

- A. WiredTiger storage engine
- B. Cluster-wide logical clock
- C. Change streams
- D. Causal consistency

Quiz

Which of the following were key features in MongoDB that enabled global transactions in MongoDB? More than one answer choice can be correct.

- ✅ A. WiredTiger storage engine
- ✅ B. Cluster-wide logical clock
- ❌ C. Change streams
- ✅ D. Causal consistency

CORRECT: WiredTiger storage engine - the ability to provide 'transactions capability' to guarantee the modification atomicity of corresponding document, indexes, and oplog together was a necessary feature to enabling global transactions.
CORRECT: Cluster-wide logical clock - the ability to the capture of chronological and causal relationships of changing data across a sharded cluster to provide the ordering was another necessary feature to enable global transactions.
INCORRECT: Change streams - these were not a necessary feature for global transaction but did come about due to the cluster-wide logical clock, that feature also enabled change streams.
CORRECT: Causal consistency - the ability to ensure operations in an order that respects their causal relationship, and which then allowed clients to observe results that are consistent with this causal relationship was a key feature to enabling global transactions in MongoDB.

Quiz

Which of the following were key features in MongoDB that enabled global transactions in MongoDB? More than one answer choice can be correct.

- ✓ A. WiredTiger storage engine
- ✓ B. Cluster-wide logical clock
- ✗ C. Change streams
- ✓ D. Causal consistency

This is correct. The ability to provide 'transactions capability' to guarantee the modification atomicity of corresponding document, indexes, and oplog together was a necessary feature.

CORRECT: WiredTiger storage engine - the ability to provide 'transactions capability' to guarantee the modification atomicity of corresponding document, indexes, and oplog together was a necessary feature to enabling global transactions.

Quiz

Which of the following were key features in MongoDB that enabled global transactions in MongoDB? More than one answer choice can be correct.

- ✓ A. WiredTiger storage engine
- ✓ B. Cluster-wide logical clock
- ✗ C. Change streams
- ✓ D. Causal consistency

This is correct. The ability to the capture of chronological and causal relationships of changing data across a sharded cluster to provide the ordering was another necessary feature.

CORRECT: Cluster-wide logical clock - This is correct. The ability to the capture of chronological and causal relationships of changing data across a sharded cluster to provide the ordering was another necessary feature.

Quiz

Which of the following were key features in MongoDB that enabled global transactions in MongoDB? More than one answer choice can be correct.

- ✓ A. WiredTiger storage engine
- ✓ B. Cluster-wide logical clock
- ✗ C. Change streams
- ✓ D. Causal consistency

This is incorrect. These were not a necessary feature for global transaction but did come about due to the cluster-wide logical clock, as that feature also enabled change streams.

INCORRECT: Change streams - This is incorrect. These were not a necessary feature for global transaction but did come about due to the cluster-wide logical clock, as that feature also enabled change streams.

Quiz

Which of the following were key features in MongoDB that enabled global transactions in MongoDB? More than one answer choice can be correct.

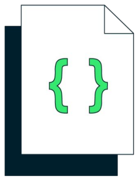
- ✓ A. WiredTiger storage engine
- ✓ B. Cluster-wide logical clock
- ✗ C. Change streams
- ✓ D. Causal consistency

This is correct. The ability to ensure operations in an order that respects their causal relationship, and which then allowed clients to observe results that are consistent with this causal relationship was a key feature.

CORRECT: Causal consistency - This is correct. The ability to ensure operations in an order that respects their causal relationship, and which then allowed clients to observe results that are consistent with this causal relationship was a key feature.

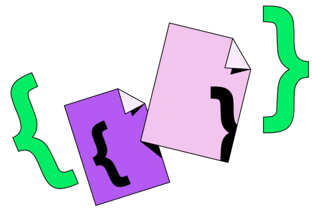
Approaches to Transactions in MongoDB

Two Approaches to Transactions



MongoDB Document

>= 1.0 MongoDB



MongoDB Transactions

>= 4.2 MongoDB

In terms of how you do transactions in MongoDB, there are two different approaches.

The first and recommended approach is to use a single MongoDB document.

The second approach is MongoDB Transactions which supports multiple documents in a single transaction.

Transactions with a Document

Patient records from a doctor's visit.

- date of visit
- doctor's notes
- drugs prescribed
- current weight

Update the document in one operation

```
patients collection
{
  "_id": 2395652,
  "name": "AJ",
  "current_weight": 210,
  "next_physical": "2021-06-13",
  "visits": [
    { "date": "2018-12-24",
      "notes": "Torn right calf" },
    { "date": "2020-02-01",
      "notes": "Strained left hamstring"
}
  ],
  "drugs": [
    { "date": "2018-12-24",
      "drug": "Ibuprofen" },
    { "date": "2020-02-01",
      "drug": "Paracetamol" }
  ]
}
```

Let's first look at using a document to perform a transaction in MongoDB. Let's look at an example of storing records related to a patient's visit to their doctor

This will include storing data such as the date of the visit, the doctor's notes, what the doctor prescribed in terms of drugs, and the current weight of the patient.

We can make all of these updates in a single document and single operation making this a transaction. Here's an example of the fields in that document that would be updated.

This example is taken from the online course "M100: MongoDB for SQL Pros" on MongoDB University and you might enjoy that course and the deeper coverage of this example there. See the web site for more details: M100 MongoDB for SQL Pros

Transactions with a Document

Atomic: All writes to one document are done at once

Consistency: No dependency on other documents

Isolation: Document being modified not seen by other reads

Durability: Guaranteed by doing a write with a "majority" concern

```
patients collection
{
  "_id": 2395652,
  "name": "AJ",
  "current_weight": 210,
  "next_physical": "2021-06-13",
  "visits": [
    { "date": "2018-12-24",
      "notes": "Torn right calf" },
    { "date": "2020-02-01",
      "notes": "Strained left
hamstring" }
  ],
  "drugs": [
    { "date": "2018-12-24",
      "drug": "Ibuprofen" },
    { "date": "2020-02-01",
      "drug": "Paracetamol" }
  ]
}
```

So we said this is a transaction and we did it with a single document, let's break down how this is indeed an ACID transaction.

Firstly, it is atomic because all writes to the document are done at once.
Secondly, it is consist as there are no other dependency on other documents.
Thirdly, it is isolated as the document being modified is not seen by other reads.
Fourthly and finally, it is durability because it is guaranteed by doing the write with a "majority" write concern. We cover read and write concerns in more depth in our lesson on replication, please refer to that lesson for more details on these kinds of concerns.

Similarly, the same fields in the document will be updated regardless of the approach of how they are updated.

ACID with a MongoDB Document

- **Since MongoDB version 1.0**
- **Preferred way to achieve ACID**
- **Design your model to have documents**
  embedding the different relational tables updated together

Let’s recap on ACID transactions in MongoDB using a single MongoDB document.

It’s been possible since MongoDB version 1.0

It’s the preferred way to achieve ACID in MongoDB. It’s preferred because it this approach favors good schema design and because of the nature of being a single document update/write operation, it is more performant than multi-document transactions.

It does require you to design you model to have documents and it does require embedding data to the same document to ensure all changes can be made within the same/single document.

Transactions with a Document

Patient records and payments from a doctor's visit.

date of visit

doctor's notes

drugs prescribed

current weight

payment information

```
patients collection
{
  "_id": 2395652,
  "name": "AJ",
  "current_weight": 210,
  "next_physical": "2021-06-13",
  "visits": [
       { "date": "2018-12-24",
       "notes": "Torn right calf" },
       { "date": "2020-02-01",
       "notes": "Strained left hamstring"
} ],
  "drugs": [
       { "date": "2018-12-24",
       "drug": "Ibuprofen" },
       { "date": "2020-02-01",
       "drug": "Paracetamol" } ]
}
payments collection
{
  "date": "2020-02-01",
  "patient_id": 2395652,
  "amount": 250.00,
}
```

Transactions with a MongoDB Transaction

Atomic: All writes to all documents are committed at once

Consistency: All checks are done within the transaction

Isolation: Guaranteed through a "snapshot" isolation level

Durability: Guaranteed by default, the write has a "majority" concern

```
patients collection
{
  "_id": 2395652,
  "name": "AJ",
  "current_weight": 210,
  "next_physical": "2021-06-13",
  "visits": [
       { "date": "2018-12-24",
       "notes": "Torn right calf" },
       { "date": "2020-02-01",
       "notes": "Strained left hamstring"
} ],
  "drugs": [
       { "date": "2018-12-24",
       "drug": "Ibuprofen" },
       { "date": "2020-02-01",
       "drug": "Paracetamol" } ]
}
payments collection
{
  "date": "2020-02-01",
  "patient_id": 2395652,
  "amount": 250.00,
}
```

So we said this is a transaction and we did it with multiple documents, let’s break down how this is indeed an ACID transaction.
In MongoDB Transactions, it is atomic because all the writes to all the documents are committed at once or if any fail the entire transaction is rolled back.
It is consistent because all of the checks are done within the transaction.
It is isolated as the “snapshot” isolation level is used to guarantee this.
It is durable as it uses the write concern of “majority” to commit the data.

ACID with a MongoDB Transaction

Since MongoDB version 4.2

Similar to traditional relational databases

Let’s recap on ACID transactions in MongoDB using the MongoDB Transactions feature.

It’s been available since MongodB 4.2.
It’s very similar mechanism to those traditional relational databases that provide transactions. In the next slides, we’ll look at how similar it is to MySQL, a popular relational database which offers transactions, in terms of syntax.

Quiz

Quiz

Which of the following were key design choices/considerations in MongoDB for the Transactions syntax? More than one answer choice can be correct.

- A. Simple
- B. Exactly matched existing relational database syntax for transactions
- C. Supports only Python, Java, and Javascript
- D. Idiomatic

Quiz

Which of the following were key design choices/considerations in MongoDB for the Transactions syntax? More than one answer choice can be correct.

- ✓ A. Simple
- ✗ B. Exactly matched existing relational database syntax for transactions
- ✗ C. Supports only Python, Java, and Javascript
- ✓ D. Idiomatic

CORRECT: Simple - Yes, this is correct. The choice of syntax was deliberately designed to be simple to ensure no confusion and to make it easy to use MongoDB Transactions.
INCORRECT: Exactly matched existing relational database syntax for transactions. - This is not correct, the syntax was designed to be similar to existing relational databases but it wasn't designed to match it exactly.
INCORRECT: Supports only Python, Java, and Javascript - This is incorrect, all of the MongoDB company supported drivers support transactions. Python, Java, and Javascript are very popular drivers and programming languages but they are not the only ones supported.
CORRECT: Idiomatic - This is correct, each implementation of transactions for each programming language was designed to support the style and idioms of that specific programming language. This makes it much easier and natural for developers fluent in the specific language to use and understand MongoDB Transactions.

Quiz

Which of the following were key design choices/considerations in MongoDB for the Transactions syntax? More than one answer choice can be correct.

- A. Simple
- B. Exactly matched existing relational database syntax for transactions
- C. Supports only Python, Java, and Javascript
- D. Idiomatic

This is correct. The choice of syntax was deliberately designed to be simple to ensure no confusion and to make it easy to use MongoDB Transactions.

CORRECT: Simple - Yes, this is correct. The choice of syntax was deliberately designed to be simple to ensure no confusion and to make it easy to use MongoDB Transactions.

Quiz

Which of the following were key design choices/considerations in MongoDB for the Transactions syntax? More than one answer choice can be correct.

- ✓ A. Simple
- ✗ B. Exactly matched existing relational database syntax for transactions
- ✗ C. Supports only Python, Java, and Javascript
- ✓ D. Idiomatic

This is incorrect. The syntax was designed to be similar to existing relational databases but it wasn't designed to match it exactly.

INCORRECT: Exactly matched existing relational database syntax for transactions. - This is incorrect. The syntax was designed to be similar to existing relational databases but it wasn't designed to match it exactly.

Quiz

Which of the following were key design choices/considerations in MongoDB for the Transactions syntax? More than one answer choice can be correct.

- ✓ A. Simple
- ✗ B. Exactly matched existing relational database syntax for transactions
- ✗ C. Supports only Python, Java, and Javascript
- ✓ D. Idiomatic

This is incorrect. All of the MongoDB company supported drivers support transactions. Python, Java, and Javascript are very popular drivers and programming languages but they are not the only ones supported.

INCORRECT: Supports only Python, Java, and Javascript - This is incorrect. All of the MongoDB company supported drivers support transactions. Python, Java, and Javascript are very popular drivers and programming languages but they are not the only ones supported.

Quiz

Which of the following were key design choices/considerations in MongoDB for the Transactions syntax? More than one answer choice can be correct.

- A. Simple
- B. Exactly matched existing relational database syntax for transactions
- C. Supports only Python, Java, and Javascript
- D. Idiomatic

This is correct. Each implementation of transactions for each programming language was designed to support the style and idioms of that specific programming language.

CORRECT: Idiomatic - This is correct. Each implementation of transactions for each programming language was designed to support the style and idioms of that specific programming language.

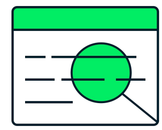
Transactions API

Let’s look at a little more depth about how you can use MongoDB Transactions in your applications, specifically let’s look now at the Application Programming Interfaces available.

Two Approaches

Core Transaction API

>= 4.0 drivers



Callback API

>= 4.2 drivers

There are two Application Programming Interfaces available to implement transactions in your application.

Firstly, the Core Transaction API, it was designed as the first API for Transactions. You can use this API with MongoDB Drivers that support version 4.0 or greater of the database. This is not the recommend API for development in terms of implementing transactions in applications.

The second approach is the Callback API. The learnings from the first API, the Core Transaction API, as well as significant feedback from the developer community led to the creation of a second API, the Callback API. This API is the recommended approach for developing your application and adding transactions to them.

It simplified how you program transactions and automatically handles a number of common errors/exceptions that can occur. This significantly improves the robustness of your application when it is uses MongoDB transactions.

We’re only going to look at the Callback API, the Core Transaction API should not be used for your development in the vast majority of cases.

Callback API

Callback API

```
def callback(session):
    employees_coll = session.client.hr.employees
    events_coll = session.client.reporting.events
    employees_coll.update_one( {"employee_id": 3},
                         {"$set”: {"status": "Inactive"}},
                         session=session )
    events_coll.insert_one( {"employee_id": 3, "status": {
                         "new": "Inactive", "old": "Active"},
                         session=session } } )
```

Here's an example in Python of the Callback API, this is the recommended API to use with your applications.

In this example, we define in Python our callback function. You can see the most important piece is that of the session. The callback function in this example, uses two collections, employees which it updates a document in and events where it inserts a new document.

The Callback API is more concise than the Transactions API, it automatically retries for the errors **TransientTransactionError** or **UnknownTransactionCommitResult**. It will also retry the write once if it fails per the default MongoDB behaviour.

For more details refer to the documentation page online at https://docs.mongodb.com/manual/core/transactions-in-applications/#txn-callback-api

Callback API

```
def callback(session):
     employees_coll = session.client.hr.employees
     events_coll = session.client.reporting.events
     employees_coll.update_one( {"employee_id": 3},
                                {"$set”: {"status": "Inactive"}},
                                session=session )
     events_coll.insert_one( {"employee_id": 3, "status": {
                             "new": "Inactive", "old": "Active"},
                               session=session } } )

def run_transaction(session):
     session.with_transaction( callback, read_concern=ReadConcern("local"),
                               write_concern=wc_majority,
                               read_preference=ReadPreference.PRIMARY )
```

Continuing our example, let’s introduce the run_transaction function. This performs the wrapping and additionally also configuration of the read and the write concerns as well as the read preference for the transaction.

Here’s the full callback and transactions code:

```
def callback(session):
     employees_coll = session.client.hr.employees
     events_coll = session.client.reporting.events
     employees_coll.update_one( {"employee_id": 3},
{"$set”: {"status": "Inactive"}}, session=session )
     events_coll.insert_one( {"employee_id": 3, "status":
{ "new": "Inactive", "old": "Active"}, session=session } }
)

def run_transaction(session):
     session.with_transaction( callback,
read_concern=ReadConcern("local"),
write_concern=wc_majority,
read_preference=ReadPreference.PRIMARY )
```

This concludes the material for this lesson. However, there are many more ways to learn about MongoDB and non-relational databases, and they are all free! Check out MongoDB's University page to find free courses that go into more depth about everything MongoDB and non-relational. For students and educators alike, MongoDB for Academia is here to offer support in many forms. Check out our educator resources and join the Educator Community. Students can receive $50 in Atlas credits and free certification through the GitHub Student Developer Pack.