# New MongoDB Transaction Support

## Introduction

As of Mongo 4.0 (released June 1018), Mongo has multi-document transaction support. Transaction support was further improved and modified in version 4.2 (June 2019). Transactions can be used across multiple operations, collections, databases, documents and shards, and are atomic.

In SEMANTIC ANTICS MongoDB is our primary datastore. We also maintain a complete history of the state of rules so rather than update a single rule document, we archive old versions by updating a timestamp and create a duplicate document as the working copy. To ensure data consistency it is important that this operation happens atomically.

## Documentation

Mongo documentation is good and comprehensive (Apache could learn a thing or two!): <https://docs.mongodb.com/manual/core/transactions/>

## Infrastructure Setup

Mongo only supports transactions on a replica set and so we need a minimum of three Mongo instances. If we are to use Mongo transactions in SEMANTIC ANTICS, we will need to scale out the Mongo deployment.

This has been tested out on three Centos 7.7 VMs hosted on Azure.

**Configure Mongo yum repo**

Using a Centos 7.7 image:

# sudo nano /etc/yum.repos.d/mongodb.repo

[MongoDB]

name=MongoDB Repository

baseurl=http://repo.mongodb.org/yum/redhat/$releasever/mongodb-org/4.2/$basearch/

gpgcheck=1

enabled=1

gpgkey=https://www.mongodb.org/static/pgp/server-4.2.asc

**Install Mongo as a Replica Set**

On each VM, install, and enable Mongo:

# sudo yum install mongodb-org

# sudo systemctl enable mongod.service

Change binding in /etc/mongod.conf to 0.0.0.0 (quick fix to get this working on Azure - 0.0.0.0 will bind to all IPv4 addresses on the box).

In the same file, uncomment #replication and change to look like this:

replication:

replSetName: demo

**Ensure the firewall is open for port 27017 for all mongo boxes in the replica set** (this process will differ depending on how the VMs are hosted).

Start Mongo:

# sudo systemctl start mongod.service

Connect to mongo:

# mongo

**On just one** of the boxes:

> rs.initiate({

"\_id": "demo",

members: [

{"\_id":0, "host": "52.178.67.90:27017"},

{"\_id":1, "host": "52.236.139.190:27017"},

{"\_id":2, "host": "52.157.93.161:27017"}

]

})

This will create the replica set. Note that one of the nodes will be elected the primary and this is the node that further operations should take place on. Whether a node is primary or secondary will be obvious from the command prompt once logged into Mongo. When creating a connection to Mongo, all hosts should be included in the connection string.

**Create the Mongo database & collection:**

PRIMARY> use SEMAntics

PRIMARY> db.createCollection(“ruleTest”)

## Code Samples

**NOTE: This is NOT intended to be production quality code**

### Pure Java Code Sample

A very basic Maven Java project has been created that uses unit tests to exercise some simple example transactional code.

The code will perform a rule insertion atomically by checking that a duplicate rule does not already exist before creating the new record.

The code will “update” a rule atomically. i.e. it will set the timestamp on the existing rule and create a duplicate of the rule in a single transaction.

A brief description of each file:

|  |  |
| --- | --- |
| MongoPlayground.java | Uses the Mongo driver directly to setup transactions and execute database operations within them. |
| TestMongoPlayground.java | A couple of tests to exercise MongoPlayground.java |

### Java Spring Code Sample

Similar to the above but using the Spring Framework to see what additional features and/or code saving measures it might offer us.

A brief description of each file:

|  |  |
| --- | --- |
| DemoApplication.java | Entry point.  insertRule() checks if rule already exists and if not creates the new rule in a single atomic transaction  updateRule() updates the timestamp on the existing rule to be current copied the rule (for a new version) setting the timestamp to Long.MAX\_VALUE.  Note that use of:   * @Transactional to indicate that the operations in the method should be wrapped in a single transaction |
| Rule.java | Data class defining the mapping between the Mongo document and the code.  Note the use of:   * @Document to specify the Mongo collection name * @Field to change the default field name mapping * @Indexed to indicate an indexed field (index is automatically created) |
| RuleRepository.java | An interface that defines interactions with MongoDB. Many standard operations are inherited from MongoRepository.  The interface is never implemented by a class, Spring maps the method argument names to the data class and uses naming conventions of the method names to infer the operation required. Bit dodge, but it seems to work!  <https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html> |

## Comparison to Existing Transactions Implementation

Caveat: This being written without active sight of the code

The existing implementation is custom written code due to historical insistence that Elasticsearch should be used as a datastore but proceeding on the assumption that this would change in future (as it did, to MongoDB), hence wanting a database agnostic solution.

Pros/cons of using Mongo’s built in transaction support

Pros:

* There is an opportunity to slim down the database layer of the SA codebase by directly using Mongo’s transactions API. Mongo is generally well tested and bug free so little concern around the use of the built-in mongo transaction facilities.
* Removes one of the learning curve hurdles for onboarding developers as API will be well known and publicly documented. There is a lot of custom code in SA (where more use of familiar 3rd party libraries may be more usual) which makes even small code changes very time consuming for new developers until each area has been understood sufficiently.

Cons:

* Codebase *may* become less database agnostic by being tied to Mongo’s transaction model. In practice assuming a single wrapInTransaction() like method in the database abstraction layer I suspect any change to support a different database would be quite small. Some code changes for a change in database technology are almost inevitable.
* Work is required to alter some low level areas of the code that have been stable for some time. “If it ain’t broke don’t fix it?”
* Mongo’s solution only works with replica sets – but should we not be using one anyway? Maybe this is also a ‘Pro’.