Morphium 4.2 Documentation

What is *Morphium*

Morphium started as a feature rich access layer for MongoDB in java. It was built with speed and

flexibility in mind. So it supported cluster aware caching out of the box, lazy loading references and such.

But with time, one of the most popular features in *Morphium* is the messaging based on MongoDB. It is fast, reliable, customizable and stable.

Using *Morphium* as a messaging system

Morphium is simple to use, and easy to customise to your needs. The messaging implementation in *Morphium* relies on the `watch` functionality, that MongoDB offers since V3.6 (you can also use messaging with older versions of MongoDB, but it will result in polling for new messages). With that feature, the messages are *pushed* to all listeners. This makes it a very efficient messaging system based on MongoDB.

why *Morphium* messaging

There is a ton of messaging solutions out there. All of them have their advantages and offer lots of features. But only few of them offer the things that *Morphium* has:

- the message queue can easily be inspected and you can use mongo search queries to find the messages you are looking for
- the message queue can be altered
- possibility to broadcast messages, that are only processed by one client max (Exclusive Messages)
- multithreaded and thread safe
- pausing and unpausing of message processing without data loss
- *Morphium* Messaging picks up all pending messages on startup - no data loss.
- no need to install additional servers, provide infrastructure. Just use your MongoDB(fn)

Quick start Messaging

```
Morphium m=new Morphium();
Messaging messaging=new Messaging(m);

messaging.addMessageListener((messaging, msg) -> {
            log.info("Got message!");
            return null;  //not sending an answer
        });
```

This is a simple example of how to implement a message consumer. This consumer listens to *all* incoming messages, regardless of name.

Messages do have some fields, that you might want to use for your purpose. But you can create your own message type as well (see below). the Msg-Class defines those properties:

- `name` the name of the Message - you can define listeners only listening to messages of a specific

name using `addListenerForMessageNamed`. Similar to a *topic* in other messaging systems
- `msg`: String message
- `value`: well - a String value
- `mapValue`: for more complex use cases where you need to send more information
- `additional`: list value - used for more complex use cases
- all messages do store some values for the processing algorithm, like `processed_by`, `in_answer_to`, `timestamp` etc.

So if you want to send a Message, that is also simple:

messaging.queueMessage(new Msg("name","A message","the value");

queueMessage is running asynchrounously, which means, that the message is *not* directly stored. If you need more speed and shorter reaction time, you should use `storeMessage` instead (directly storing message to mongo).

Answering messages

Morphium is able to answer any message for you. Your listener implementation only needs to return an instance of the `Msg`-Class(fn). This will then be sent back to the sender as an answer.

When sending a message, you also may wait for the incoming answer. The Messaging class offers a method for that purpose:

```
//new messaging instance with polling frequency of 100ms, not multithreaded
//polling only used in case of non-Replicaset connections and in some
//cases like unpausing to find pending messages

    Messaging sender = new Messaging(_Morphium_, 100, false);
    sender.start();

    gotMessage1 = false;
    gotMessage2 = false;
    gotMessage3 = false;
    gotMessage4 = false;

    Messaging m1 = new Messaging(_Morphium_, 100, false);
    m1.addMessageListener((msg, m) -> {
        gotMessage1 = true;
        return new Msg(m.getName(), "got message", "value", 5000);
    });

    m1.start();
    Thread.sleep(2500);

    Msg answer = sender.sendAndAwaitFirstAnswer(new Msg("test", "Sender", "sent", 15000), 15000);
    assert (answer != null);
    assert (answer.getName().equals("test"));
    assert (answer.getInAnswerTo() != null);
    assert (answer.getRecipient() != null);
    assert (answer.getMsg().equals("got message"));
    m1.terminate();
    sender.terminate();
```

As the whole communication is asynchronous, you will have to specify a timeout after wich the wait

for answer will be aborted with an exception. And, there might be more than one answers to the same message, hence you will only get the first one.

in the above example, the timeout for the answer is set to 15s (and the TTL for messages also).

more advanced settings

Custom message classes

As mentioned above, you can define your own Message-Class to be send back and forth. This class just needs to extend the standard `Msg`-Class. When adding a listener to messaging, you have the option to also use generics to specify the Msg-Type you want to use.

Message priorities

Every message does have a priority field. That is used for giving queued messages precedence over others. The priority could be changed *after* a message is queued directly in MongoDB (or using *Morphium*).

But as the messaging is built on pushing of messages, when is the priority field used? Several cases:

- when starting up messaging. When starting Messaging, the system does look for pending messages in the queue, highes prio is used first
- when unpausing a messaging instance, it will look for any messages in the queue and will process them according to their priority.

Pausing / unpausing of messaging

In some cases it might be necessary to pause message processing for a time. That might be the case, if the message is triggering some long running task or so. If so, it would be good not to process any additional messages (at least of that type).

You can call `messaging.pauseProcessingOfMessagesNamed` to *not* process any more messages of a certain type.

Attention: if you have long running tasks triggered by messages, you should pause processing in the onMessage method and unpause it when finished.

Multithreading / Multimessage processing

When instantiating Messaging, you can specify two booleans:

- multithreading: if true, every incoming message will be processed in an own thread (Executor - see MorphiumConfig below). That means, several messages can be processed in parallel

- processMultiple: this setting is only important in case of startup or unpausing(fn). If true, messaging will lock all(fn) messages available for this listener and process them one by one (or in parallel if multithreading is enabled).

  These settings are influenced by other settings:
- `messagingWindowSize` in MorphiumConfig or as constructor parameter / setter in Messaging: this defines how many messages are marked for processing at once. Those might be processed in parallel (depending whether `processMultiple` is true, and the executor configuration, how many threads can be run in parallel)
- `useChangeStream` in Messaging. Usually messaging determines by the cluster status, whether or not to use the changestream or not. If in a cluster, use it, if not use polling. But if you explicitly want to use polling, you can set this value to `false`. The advantage here might be, that the messages are processed by priority with every poll. This might be useful depending on your usecase. If this is set to false (or you are connected to an single instance), the `pause` configuration option (aka polling frequency) in Messaging will determine how fast your messages can be consumed. **Attention** high polling frequency (a low `pause` value), will increase the load on MongoDB.
- `ThreadPoolMessagingCoreSize` in MorphiumConfig: If you define messaging to be multithreaded it will spawn a new thread with each incoming message. this is the core size of the corresponding thread pool. If your messaging instance is not configured for multithreading, this setting is not used.
- `ThreadPoolMessagingMaxSize`: max size of the thread pool. similar to above.
- `ThreadPoolMessagingKeepAliveTime`: time of threads to live in ms

  some examples to clarify that:
- your messaging instance is configured for multithreaded processing, multiple processing, having a `windowSize` of 100 and a `ThreadPoolMessagingMaxSize` of 10, then there will be 100 messages in queue marked for being processed by this specific messaging instance, but only 10 will be processed in parallel.
- multithreaded processing is false, then the `windowSize` determines how many messages are marked for being processed, but are only processed one by one
- multithreaded processing and multiple processing is false, then only one message is marked for being processed at a time. As soon as this processing is finished, the next message is being taken.
- having `multithreaded` set to true and `processMultiple` set to false would result in running each message processing in one separate thread, but only one at a time. This is very similar to having `multithreaded` and `process multiple` both set to false.

Custom MessageQueue name

When creating a Messaging instance, you can set a collection name to use. This could be compared to having a separate message queue in the system. Messages sent to one queue are not being registered by another.

Examples

Simple producer consumer setup:

```
Morphium m=new Morphium(config);
// create messaging instance with default settings, meaning
// no multithreading, windowSize of 100, processMultiple false
Messaging producer=new Messaging(m);

producer.queueMessage(new Msg("name","a message","a value"));

the receiver needs to connect to the same mongo and the same database:

Morphium m=new Morphium(config);
Messaging consumer=new Messaging(m);
consumer.start(); //needed for receiving messages

consumer.addMessageListener((messaging, msg) -> {
         //Incoming message
         System.out.println("Got a message of name "+msg.getName());
         return null; //no answer to send back
        });
```

you can also register listeners only for specific messages:consumer.start(); //needed for receiving messages

```
consumer.addListenerForMessageNamed("name",(messaging, msg) -> {
       //Incoming message, is always named "name"
         System.out.println("Got value: "+msg.getValue());
         Msg answer=new Msg(msg.getName(),"answer","the answerValue");
         return answer; //no answer to send back
        });
```

Attention: the producer will only be able to process incoming messages, if `start()` was called!

The message sent there was a broadcast message. All registered listeners will receive that message and will process it!

Direct messages

In order to send a message directly to a specific messaging instance, you need to get the unique ID of it. This id is add as sender to any message.

```
Msg m=new Msg("Name","Message","value");
m.addRecipient(messaging1.getId());
//you could add more recipients if necessary
```

Background: This is used to send answers back to the sender. If you return a message instance in `onMessage`, this message will be sent directly back to the sender.

You can add as many recipients as needed, if no recipient is defined, the message by default is sent to all listeners.

Exclusive Broadcast messages

Broadcast messages are fine for informing all listeners about something. But for some more complex scenarios, you would need a way to queue a message, and have only one listener process it - no matter which one (load balancing?)

Morphium supports this kind of messages, it is called "exclusive broadcast". This way, you can easily scale up by just adding listener instances.

Sending a exclusive broadcast message is simple:

```
Msg m=new Message("exclusive","The message","and value");
m.setExclusive(true);
messaging.queueMessage(m);
```

The listener only need to implement the standard `onMessage`-Method to get this message. Due to some sophisticated locking of messages, *Morphium* makes this message exclusive - which means, it is only processed once!

Since Morphium V4.2 it is also possible to send an exclusive message to certain recipients[1].

The behaviour is the same: the message will only be processed by *one* of the specified recipients, whereas it will be processed by *all* recipients, if not exclusive.

InMemory Driver

one main purpose of the InMemoryDriver is to be able to do testing without having a MongoDB installed. The InMemoryDriver adds the opportunity to let all MongoDB-code run in Memeory, with a couple of exceptions

- unfortunately, the InMemoryDriver cannot do aggregations. It will throw an Exception, when trying Aggregations with this driver
- the inMemoryDriver is also not capable to return cluster information, runn mongodb commands
- it does not support spacial indexes or queries

if you want to mock those things in testing, you need to:

1. create a subclass of the inMemoryDriver
2. override the corresponding method, for example `aggregate()` for aggregation and return the properly mocked data
3. set the driver back to default in order to have it work

```
@Test
public void mockAggregation() throws Exception{
    MorphiumDriver original=morphium.getDriver();
    morphium.setDriver(new InMemoryDriver(){
        @Override
        public List<Map<String, Object>> aggregate(String db, String collection, List<Map<String, Object>>
pipeline, boolean explain, boolean allowDiskUse, Collation collation, ReadPreference readPreference) throws
```

```
MorphiumDriverException {
            return Arrays.asList(Utils.getMap("MockedData",123.0d));
        }
    });

Aggregator<UncachedObject, Map> agg = morphium.createAggregator(UncachedObject.class, Map.class);
//...
assert(agg.aggregate().get(0).get("MockedData").equals(123.0d)); //checking mocked data
morphium.getDriver().close();
morphium.setDriver(original);

}
```

how to use the inMemory Driver

you just need to set the Driver properly in your *Morphium* configuration.

```
MorphiumConfig cfg = new MorphiumConfig();
cfg.addHostToSeed("inMem");
cfg.setDatabase("test");
cfg.setDriverClass(InMemoryDriver.class.getName());
cfg.setReplicasetMonitoring(false);
morphium = new Morphium(cfg);
```

Of course, the *InMemDriver* does not need hosts to connect to, but for compatibility reasons, you need to add at least one host (although it will be ignored).

You can also set the Driver in the settings, e.g. in properties:

```
morphium.driverClass = "de.caluga.morphium.driver.inmem.InMemoryDriver"
```

After that initialization you can use this *Morphium* instance as always, except that it will "persist" data only in Memory.

Morphium POJO Mapping

Ideas and design criteria

In the early days of MongoDB there were not many POJO mapping libraries available. One was called *morphia*. Unfortunately we had a lot of problems adapting this to our needs.

Hence we built **Morphium** and we named it similar to *morphia* to show where the initial idea came from.

Morphium is built with flexibility, thread safety, performance and cluster awareness in mind.

- flexibility: it is possible to exchange most of the internal implementations of *Morphium*. You could have your own Driver class for connecting to MongoDB(fn) or have a custom implementation for the query processing.
- thread safety: all aspects of *Morphium* were tested multithreaded so that it can be used in

production

- performance: one of the main goals of *Morphium* was to improve performance. The Object Mapping in use is a custom implementation that was built especially for *Morphium*, is very fast and to improve speed even further, caching is part of the core features of *Morphium*
- cluster awareness: this is essential nowadays for high availability or just mere speed. _Morphium_s caches are all cluster aware which means you will not end up with dirty reads in a clustered environment when using *Morphium*(fn)

Concepts

Morphium is built to be very flexible and can be used in almost any environment. So the architecture needs to be flexible and sustainable at the same time. Hence it's possible to use your own implementation for the cache if you want to.

There are four major components of *Morphium*:

1. the *Morphium* Instance: This is you main entrypoint for interaction with Mongo. Here you create Queries and you write data to mongo. All writes will then be forwarded to the configured Writer implementation, all reads are handled by the Query-Object
2. Query-Object: you need a query object to do reads from mongo. This is usually created by using `_Morphium_.createQueryFor(Class<T> cls)`. With a Query, you can easily get data from database or have some things changed (update) and alike.
3. the Cache: For every request that should be sent to mongo, *Morphium* checks first, whether this collection is to be cached and if there is already a result being stored for the corresponding request.
4. The Writers: there are 3 different types of writers in *Morphium*: The Default Writer (`_Morphium_Writer`) - writes directly to database, waiting for the response, the BufferedWriter (`BufferedWriter`) - does not write directly. All writes are stored in a buffer which is then processed as a bulk. The last type of writer ist the asynchronous writer (`AsyncWriter`) which is similar to the buffered one, but starts writing immediately - only asynchronous. *Morphium* decides which writer to use depending on the configuration and the annotations of the given Entities. But you can *always* use asynchronous calls just by adding a`AsyncCallback` implementation to your request.

Simple rule when using *Morphium*: You want to read -> Use the Query-Object. You want to write: Use the *Morphium* Object.

There are some additional features built upon this architecture:

- messaging: *Morphium* has its own production grade messaging system. Its has a lot of features, that are unique for a messaging system.
- cache synchronization: Synchronize caches in a clustered environment. Uses messaging.
- custom mappers - you can tell *Morphium* how to map a certain type from and to MongoDB. For example there is a "custom" mapper implementation for mapping `BigInteger` instances to MongoDB.

- every one of those implementations can be changed: it is possible to set the class name for the `BufferedWriter` to a custom built one (in `MorphiumConfig`). Also you could replace the object mapper with your own implementation by implementing the `ObjectMapper` interface and telling *Morphium* which class to use instead. In short, these things can be changed in *Morphium* / MorphiumConfig:
  - MorphiumCache
  - ObjectMapper
  - Query
  - Field
  - QueryFactory
  - Aggregator
  - AggregatorFactory
  - MorphiumDriver (> V3.0, for connecting to MongoDB or any other data source if you want to. For example, there is an In-Memory-Driver you might want to use for testing. As an example, there is also an InfluxDB-Driver available.)
- Object Mapping from and to Strings (using the object mapper) and JSON.
- full support for the Aggregation Framework
- Transaction support (for supporting MongoDB versions)
- Automatic encryption of fields (this is a re-implementation of the MongoDB enterprise feature in pure java - works declarative)

Advantages / Features

POJO Mapping

Morphium is capable of mapping standard Java objects (POJOs - plain old java objects) to MongoDB documents and back. This should make it possible to seemlessly integrate MongoDB into your application.

Declarative caching

When working with databases - not only NoSQL ones - you need to consider caching. *Morphium* integrates transparent declarative caching by entity to your application, if needed. Just define your caching needs in the `@Cache` annotation.(fn)

The cache uses any JavaCache compatible cache implementation (like EHCache), but provides an own implementation if nothing is specified otherwise.

cache synchronization

as mentioned above, caching is of utter importance in production grade applications. Usually, caching in a clustered Environment is kind of a pain. As you need consider dirty reads and such. But *Morphium* caching works also fine in a clustered environment. Just start (instantiate) a `CacheSynchronizer` -

and you're good to go!

Auto-Versioning

When it comes to dirty reads and such, you might want to use the auto-versioning feature of *Morphium*. This will give every entity a version number. If you want to write to MongoDB and the version number differs, you'd get an exception - meaning the database was modified before you tried to persist your data. This so called *optimistic locking* will help in most cases to avoid accidental overwriting of data.

To use auto-Versioning, just set the corresponding flag in the `@Entity`-annotation to `true` and define a `Long` in your class, that should hold the version number using the `@Version`-annotation.

Attention: do not change the version value manually, this will cause problems writing and will most probably cause loss of data!

Type IDs

usually *Morphium* knows which collection holds which kind of data. When de-serializing it is easy to know, what class to instanciate.

But when it comes to polymorphism and containers (like lists and maps), things get compicated. *Morphium* adds in this case the class name as property to the document. Up until version 4.0.0 this was causing some problems when refactoring your Entities. If you changed the classname or the package name of that class, de-serializing was impossible (the classname was obviously wrong).

now you can just set the `typeId` in `@Entity` to be able refactor more easily. If you already have data, and you want to refactor your entitiy names, just add the *original* class name as type id!

transparent encryption of values

Morphium implemented a client side version of auto encrypted fields. When defining a property, you can specifiy the value to be encrypted. Morphium provides an implementation of AESEncryption, but you could implement any other encryption.

In order for encryption to work, we need to provide a `ValueEncryptionProvider`. This is a very simple interface:

```
package de.caluga.morphium.encryption;

public interface ValueEncryptionProvider {
    void setEncryptionKey(byte[] key);

    void setEncryptionKeyBase64(String key);

    void setDecryptionKey(byte[] key);

    void sedDecryptionKeyBase64(String key);

    byte[] encrypt(byte[] input);
```

```
        byte[] decrypt(byte[] input);

    }
```

There are two implementations available: `AESEncryptionProvider` and `RSAEncryptionProvider`.

Another interface being used is the `EncryptionKeyProvider`, a simple system for managing encryption keys:

```
    package de.caluga.morphium.encryption;

    public interface EncryptionKeyProvider {
        void setEncryptionKey(String name, byte[] key);

        void setDecryptionKey(String name, byte[] key);

        byte[] getEncryptionKey(String name);

        byte[] getDecryptionKey(String name);

    }
```

The `DefaultEncrptionKeyProvider` acutally is a very simple key-value-store and needs to be filled manually. The implementation `PropertyEncryptionKeyProvider` reads those keys from *encrypted* property files.

Here is an example, on how to use the transparent encryption:

```
@Entity
public static class EncryptedEntity {
    @Id
    public MorphiumId id;

    @Encrypted(provider = AESEncryptionProvider.class, keyName = "key")
    public String enc;

    @Encrypted(provider = AESEncryptionProvider.class, keyName = "key")
    public Integer intValue;

    @Encrypted(provider = AESEncryptionProvider.class, keyName = "key")
    public Float floatValue;

    @Encrypted(provider = AESEncryptionProvider.class, keyName = "key")
    public List<String> listOfStrings;

    @Encrypted(provider = AESEncryptionProvider.class, keyName = "key")
    public Subdoc sub;


    public String text;
}



@Test
public void objectMapperTest() throws Exception {
    morphium.getEncryptionKeyProvider().setEncryptionKey("key", "1234567890abcdef".getBytes());
    morphium.getEncryptionKeyProvider().setDecryptionKey("key", "1234567890abcdef".getBytes());
    MorphiumObjectMapper om = morphium.getMapper();
    EncryptedEntity ent = new EncryptedEntity();
    ent.enc = "Text to be encrypted";
```

```
    ent.text = "plain text";
    ent.intValue = 42;
    ent.floatValue = 42.3f;
    ent.listOfStrings = new ArrayList<>();
    ent.listOfStrings.add("Test1");
    ent.listOfStrings.add("Test2");
    ent.listOfStrings.add("Test3");

    ent.sub = new Subdoc();
    ent.sub.intVal = 42;
    ent.sub.strVal = "42";
    ent.sub.name = "name of the document";

    //serializing the document needs to encrypt the data
    Map<String, Object> serialized = om.serialize(ent);
    assert (!ent.enc.equals(serialized.get("enc")));

    //checking deserialization used decryption
    EncryptedEntity deserialized = om.deserialize(EncryptedEntity.class, serialized);
    assert (deserialized.enc.equals(ent.enc));
    assert (ent.intValue.equals(deserialized.intValue));
    assert (ent.floatValue.equals(deserialized.floatValue));
    assert (ent.listOfStrings.equals(deserialized.listOfStrings));
}
```

Please note, that the key *name* used for encryption and decryption is to be defined in the property configuration of the corresponding entity.

configuring *Morphium*: `MorphiumConfig`

MorphiumConfig is the class to encapsulate all settings for *Morphium*. The most obvious settings are the host seed and port definitions. But there is a ton of additional settings available.

Different sources

Json

The standard `toString()`method of MorphiumConfig creates an Json String representation of the configuration. to set all configuration options from a json string, just call `createFromJson`.

Properties

the configuration can be stored and read from a property object.

`MorphiumConfig.fromProperties(Properties p);` Call this method to set all values according to the given properties. You also can pass the properties to the constructor to have it configured.

To get the properties for the current configuration, call `asProperties()` on a configured MorphiumConfig Object.

Here is an example property-file:

```
    maxWaitTime=1000
```

```
maximumRetriesBufferedWriter=1
maxConnections=100
retryWaitTimeAsyncWriter=100
maxAutoReconnectTime=5000
blockingThreadsMultiplier=100
housekeepingTimeout=5000
hosts=localhost\:27017, localhost\:27018, localhost\:27019
retryWaitTimeWriter=1000
globalCacheValidTime=50000
loggingConfigFile=file\:/Users/stephan/_Morphium_/target/classes/_Morphium_-log4j-test.xml
writeCacheTimeout=100
connectionTimeout=1000
database=_Morphium__test
maximumRetriesAsyncWriter=1
maximumRetriesWriter=1
retryWaitTimeBufferedWriter=1000
```

The minimal property file would define only `hosts` and `database`. All other values would be defaulted.

If you want to specify classes in the config (like the Query Implementation), you need to specify the full qualified class name, e.g. `de.caluga.morphium.customquery.QueryImpl`

Java-Code

The most straight forward way of configuring *Morphium* is, using the object directly. This means you call the getters and setters according to the given variable names above (like `setMaxAutoReconnectTime()`).

The minimum configuration is explained above: you only need to specify the database name and the host(s) to connect to. All other settings have sensible defaults, which should work for most cases.

Configuration Options

There are a lot of settings and customizations you can do within *Morphium*. Here we discuss *all* of them:

- *loggingConfigFile*: can be set, if you want *Morphium* to configure your log4j for you. *Morphium* itself has a dependency to log4j (see Dependencies).
- *camelCaseConversion*: if set to false, the names of your entities (classes) and fields won't be converted from camelcase to underscore separated strings. Default is `true` (convert to camelcase)
- *maxConnections*: Maximum Number of connections to be built to mongo, default is 10
- *houseKeepingTimeout*: the timeout in ms between cache housekeeping runs. Defaults to 5sec
- *globalCacheValidTime*: how long are Cache entries valid by default in ms. Defaults to 5sek
- *writeCacheTimeout*: how long to pause between buffered writes in ms. Defaults to 5sek
- *database*: Name of the Database to connect to.
- *connectionTimeout*: Set a value here (in ms) to specify how long to wait for a connection to mongo to be established. Defaults to 0 (⇒ infinite)

- *socketTimeout*: how long to wait for sockets to be established, defaults to 0 as well
- *checkForNew*: This is something interesting related to the creation of ids. Usually Ids in mongo are of type `ObjectId`. Anytime you write an object with an `_id` of that type, the document is either updated or inserted, depending on whether or not the ID is available or not. If it is inserted, the newly created ObjectId is being returned and add to the corresponding object. But if the id is not of type ObjectId, this mechanism will fail, no objectId is being created. This is no problem when it comes to new creation of objects, but with updates you might not be sure, that the object actually is new or not. If this obtion is set to `true` *Morphium* will check upon storing, whether or not the object to be stored is already available in database and would update.
- *writeTimeout*: this timeout determines how long to wait until a write to mongo has to be finshed. Default is `0`⇒ no timeout
- *maximumRetriesBufferedWriter*: When writing buffered, how often should retry to write the data until an exception is thrown. Default is 10
- *retryWaitTimeBufferedWriter*: Time to wait between retries
- *maximumRetriesWriter*, *maximumRetriesAsyncWriter*: same as *maximumRetriesBufferedWriter*, but for direct storage or asynchronous store operation.
- *retryWaitTimeWriter*, *retryWaitTimeAsyncWriter*: similar to *retryWaitTimeBufferedWriter*, but for the according writing type
- *globalW*: W sets the number of nodes to have finished the write operation (according to your safe and j / fsync settings)
- *maxWaitTime*: Sets the maximum time that a thread will block waiting for a connection.
- *serverSelectionTimeout*: Defines how long the driver will wait for server selection to succeed before throwing an exception
- *writeBufferTime:* Timeout for buffered writes. Default is 0
- *autoReconnect*: if set to `true` connections are re-established, when lost. Default is `true`
- *maxAutoReconnectTime*: how long to try to reconnect (in ms). Default is `0`⇒ try as long as it takes
- *mongoLogin*,*mongoPassword*: User Credentials to connect to MongoDB. Can be null.
- *mongoAdminUser*, *mongoAdminPwd*: Credentials to do admin tasks, like get the replicaset status. If not set, use mongoLogin instead.
- *autoValuesEnabled*: *Morphium* supports automatic values being set to your POJO. These are configured by annotations (`@LasChange`, `@CreationTime`, `@LastAccess`, ...). If you want to switch this off *globally*, you can set it in the config. Very useful for test environments, which should not temper with productional data. By default the auto values are *enabled*.
- *readCacheEnabled*: Globally enable or disable readcache. This only affects entities with a `@Cache` annotation. By default it's enabled.
- *asyncWritesEnabled*: Globally enable or disalbe async writes. This only affects entities with a `@AsyncWrites`annotation
- *bufferedWritesEnabled*: Globally enable or disable buffered writes. This only affects entities with a `@WriteBuffer` annotation
- `defaultReadPreference`: whether to read from primary, secondary or nearest by default. Can be defined with the `@ReadPreference` annotation for each entity.
- `replicaSetMonitoringTimeout`: time interval to update replicaset status.

- *retriesOnNetworkError*: if you happen to have an unreliable network, maybe you want to retry writes / reads upon network error. This settings sets the number of retries for that case.
- *sleepBetweenNetworkErrorRetries*: set the time to wait between network error retries.
- *autoIndexAndCappedCreationOnWrite*: This setting is by default *true* which means, that *Morphium* keeps a list of existing collections. When a collection would be created automatically by writing to it, morphium can then and only then have all indexes and capped settings configured for that specific collection. Causes a little overhead on write access to see, if a collection exists. Probably a good idea to switch off in production environment, but for development it makes things easier.

In addition to those settings describing the behaviour of *Morphium*, you can also define custom classes to be used internally:

- *omClass*: here you specify the class, that should be used for mapping POJOs (your entities) to `Documnet`. By Default it uses the `ObjectMapperImpl`. Your custom implementation must implement the interface `ObjectMapper`.
- *iteratorClass*: set the Iterator implementation to use. By default `MorphiumIteratorImpl`is being used. Your custom implementation must implement the interface `MorphiumIterator`
- *aggregatorClass*: this is *Morphium*'s representation of the aggregator framework. This can be replaced by a custom implementation if needed. Implements `Aggregator` interface
- *aggregatorFactoryClass*: this is *Morphium*'s representation of the aggregator framework. This can be replaced by a custom implementation if needed. Implements `AggregatorFactory` interface
- *queryClass* and *fieldImplClass*: this is used for Queries. If you want to take control over how queries ar built in *Morphium* and on how fields within queries are represented, you can replace those two with your custom implementation.
- *queryFactoryClass*: query factory implementation, usually just creates a Query-Object. Custom implementations need to implement the `QueryFactory` interface.
- *cache*: Set your own implementation of the cache. It needs to implement the `MorphiumCache` interface. Default is `MorphiumCacheImpl`. You need to specify a fully configured cache object here, not only a class object.
- *driverClass*: Set the driver implementation, you want to use. This is a string, set the class name here. E.g. `MorphiumConfig.setDriverClass(MetaDriver.class.getName()`. Custom implementations need to implement the `MorphiumDriver` interface.

Entity Definition

Entitys in *Morphium* ar just "Plain old Java Objects" (POJOs). So you just create your data objects, as usual. You only need to add the annotation `@Entity` to the class, to tell *Morphium* "Yes, this can be stored". The only additional thing you need to take care of is the definition of an ID-Field. This can be any field in the POJO identifiying the instance. Its best, to use `ObjectID` as type of this field, as these can be created automatically and you don't need to care about those as well.

If you specify your ID to be of a different kind (like String), you need to make sure, that the String is

set, when the object will be written. Otherwise you might not find the object again. So the shortest Entity would look like this:

```
@Entity
 public class MyEntity {
    @Id private ObjectId id;
    //.. add getter and setter here
 }
```

indexes

Indexes are *critical* in mongo, so you should definitely define your indexes as soon as possible during your development. Indexes can be defined on the Entity itself, there are several ways to do so: - @Id always creates an index - you can add an `@Index` to any field to have that indexed:

@Index

private String name;

you can define combined indexes using the `@Index` annotation at the class itself:

```
 @Index({"counter, name","value,thing,-counter"}
 public class MyEntity {
```

This would create two combined indexes: one with `counter` and `name` (both ascending) and one with `value`, `thing` and descending `counter`. You could also define single field indexes using this annotations, but it's easier to read adding the annotation directly to the field.

Indexes will be created automatically if you *create* the collection. If you want the indexes to be created, even if there is already data stores, you need to call`morphium.ensureIndicesFor(MyEntity.class)`- You also may create your own indexes, which are not defined in annotations by calling`morphium.ensureIndex()`. As parameter you pass on a Map containing field name and order (-1 or 1) or just a prefixed list of strings (like`"-counter","name"`).

Every Index might have a set of options which define the kind of this index. Like `buildInBackground` or `unique`. You need to add those as second parameter to the Index-Annotation:

```
@Entity
 @Index(value = {"-name, timer", "-name, -timer", "lst:2d", "name:text"},
               options = {"unique:1", "", "", ""})
   public static class IndexedObject {
```

here 4 indexes are created. The first two ar more or less standard, wheres the `lst` index is a geospacial one and the index on `name` is a text index (only since mongo 2.6). If you need to define options for one of your indexes, you need to define it for all of them (here, only the first index is unique).

We're working on porting *Morphium* to java8, and there it will be possible to have more than one `@Index` annotation, making the syntax a bit more legible

capped collections

Similar as with indexes, you can define you collection to be capped using the `@Capped` annotation. This annotation takes two arguments: the maximum number of entries and the maximum size. If the collection does not exist, it will be created as capped collection using those two values. You can always ensureCapped your collection, unfortunately then only the `size` parameter will be honoured.

Querying

Querying is done via the Query-Object, which is created by *Morphium* itself (using the Query Factory). The definition of the query is done using the fluent interface:

```
    Query<MyEntity> query=_Morphium_.createQueryFor(MyEntity.class);
    query=query.f("id").eq(new ObjectId());
    query=query.f("valueField").eq("the value");
    query=query.f("counter").lt(22);
    query=query.f("personName").matches("[a-zA-Z]+");
    query=query.limit(100).sort("counter");
```

In this example, I refer to several fields of different types. The Query itself is always of the same basic syntax:

```
    queryObject=queryObject.f(FIELDNAME).OPERATION(Value);
queryObject=queryObject.skip(NUMBER); //skip a number of entreis
queryObject=queryObject.limig(NUMBER); // limit result
queryObject.sort(FIELD_TO_SORTBY);`
```

As field name you may either use the name of the field as it is in mongo or the name of the field in java. If you specify an unknown field to *Morphium*, a `RuntimeException` will be raised.

For definition of the query, it's also a good practice to define enums for all of your fields. This makes it hard to have mistypes in a query:

```
    public class MyEntity {
          //.... field definitions
          public enum Fields { id, value, personName,counter, }
    }
```

There is a plugin for intelliJ creating those enums automatically. Then, when defining the query, you don't have to type in the name of the field, just use the field enum:

`query=query.f(MyEntity.Fields.counter).eq(123);`

After you defined your query, you probably want to access the data in mongo. Via *Morphium*,there are several possibilities to do that: - `queryObject.get()`: returns the first object matching the query, only one. Or null if nothing matched - `queryObject.asList()`: return a list of all matching objects. Reads all data in RAM. Useful for small amounts of data - `Iterator<MyEntity> it=queryObject.asIterator()`: creates a `MorphiumIterator` to iterate through the data, whch does not read all data at once, but only a couple of elements in a row (default 10).

the Iterator

Morphium has support for a special Iterator, which steps through the data, a couple of elements at a time. By Default this is the standard behaviour. But the *_Morphium_*Iterator ist quite capable:

- `queryObject.asIterable()` will stepp through the result list, 10 at a time
- `queryObject.asIterable(100)` will step through the result list, 100 at a time
- `queryObject.asIterable(100,5)` will step through the result list, 100 at a time and keep 4 chunks of 100 elements each as prefetch buffers. Those will be filled in background.
- `MorphiumIterator it=queryObject.asIterable(100,5); it.setmultithreadedAccess(true);` use the same iterator as before, but make it thread safe.

Storing

Storing is more or less a very simple thing, just call `_Morphium_.store(pojo)` and you're done. Although there is a bit more to it: - if the object does not have an id (id field is `null`), there will be a new entry into the corresponding collection. - if the object does have an id set (!= `null`), an update to db is being issued. - you can call `_Morphium_.storeList(lst)` where lst is a list of entities. These would be stored in bulkd, if possible. Or it does a bulk update of things in mongo. Even mixed lists (update and inserts) are possible. *Morphium* will take care of sorting it out - there are additional methods for writing to mongo, like update operations `set`, `unset`, `push`, `pull` and so on (update a value on one entity or for all elements matching a query), `delete` objects or objects matching a query, and a like - The writer that acutally writes the data, is chosen depending on the configuration of this entity (see Annotations below)

Names of entities and fields

Morphium by defaults converts all java CamelCase identifiers in underscore separated strings. So, `MyEntity` will be stored in an collection called `my_entity` and the field `aStringValue` would be stored in as `a_string_value`.

When specifying a field, you can always use either the transformed name or the name of the corresponding java field. Collection names are always determined by the classname itself.

CamelCase conversion

But in *Morphium* you can of course change that behaviour. Easiest way is to switch off the transformation of CamelCase globally by setting `camelCaseConversionEnabled` to false (see above: Configuration). If you switch it off, its off completely - no way to do switch it on for just one collection or so.

If you need to have only several types converted, but not all, you have to have the conversion globally enabled, and only switch it off for certain types. This is done in either the `@Entity` or `@Embedded` annotation.

```
@Entity(convertCamelCase=false)
public class MyEntity {
     private String myField;
```

This example will create a collection called `MyEntity` (no conversion) and the field will be called `myField` in mongo as well (no conversion).

Attention: Please keep in mind that, if you switch off camelCase conversion globally, nothing will be converted!

using the full qualified classname

you can tell *Morphium* to use the full qualified classname as basis for the collection name, not the simple class name. This would result in createing a collection `de_caluga_morphium_my_entity` for a class called `de.caluga.morphium.MyEntity`. Just set the flag `useFQN` in the entity annotation to `true`.

```
@Entity(useFQN=true)
public class MyEntity {
```

Recommendation is, not to use the full qualified classname unless it's really needed.

Specifying a collection / fieldname

In addition to that, you can define custom names of fields and collections using the corresponding annotation (`@Entity`, `@Property`).

For entities you may set a custom name by using the `collectionName` value for the annotation:

```
@Entity(collectionName="totallyDifferent")
public class MyEntity {
    private String myValue;
}
```

the collection name will be `totallyDifferent` in mongo. Keep in mind that camel case conversion for fields will still take place. So in that case, the field name would probably be `my_value`. (if camel case conversion is enabled in config)

You can also specify the name of a field using the property annotation:

```
@Property(fieldName="my_wonderful_field")
private String something;
```

Again, this only affects this field (in this case, it will be called `my_wondwerful_field` in mongo) and this field won't be converted camelcase. This might cause a mix up of cases in your MongoDB, so please use this with care.

Accessing fields

When accessing fields in *Morphium* (especially for the query) you may use either the name of the

Field in Java (like myEntity) or the converted name depending on the config (camelCased or not, or custom).

Using NameProviders

In some cases it might be necessary to have the collection name calculated dynamically. This can be acchieved using the `NameProvider` Interface.

You can define a NameProvider for your entity in the `@Entity` annotation. You need to specify the type there. By default, the NameProvider for all Entities is `DefaultNameProvider`. Which acutally looks like this:

```
    public final class DefaultNameProvider implements NameProvider {

@Override
public String getCollectionName(Class<?> type, ObjectMapper om, boolean translateCamelCase, boolean useFQN, String
specifiedName, _Morphium_ _Morphium_) {

    String name = type.getSimpleName();

    if (useFQN) {
        name = type.getName().replaceAll("\\.", "_");
    }
    if (specifiedName != null) {
        name = specifiedName;
    } else {
        if (translateCamelCase) {
            name = _Morphium_.getARHelper().convertCamelCase(name);
        }
    }
    return name;
}


    }
```

You can use your own provider to calculate collection names depending on time and date or for example depending on the querying host name (like: create a log collection for each server separately or create a collection storing logs for only one month each).

Attention: Name Provider instances will be cached, so please implement them threadsafe.

Annotations

a lot of things can be configured in *Morphium* using annotations. Those annotations might be added to either classes, fields or both.

Entity

Perhaps *the* most important Annotation, as it has to be put on every class the instances of which you want to have stored to database. (Your data objects).

By default, the name of the collection for data of this entity is derived by the name of the class itself and then the camel case is converted to underscore strings (unless config is set otherwise).

These are the settings available for entities:

- `translateCamelCase`: default true. If set, translate the name of the collection and all fields (only those, which do not have a custom name set)
- `collectionName`: set the collection name. May be any value, camel case won't be converted.
- `useFQN`: if set to true, the collection name will be built based on the full qualified class name. The Classname itself, if set to false. Default is false
- `polymorph`: if set to true, all entities of this type stored to mongo will contain the full qualified name of the class. This is necessary, if you have several different entities stored in the same collection. Usually only used for polymorph lists. But you could store any polymorph marked object into that collection Default is false
- `nameProvider`: specify the class of the name provider, you want to use for this entity. The name provider is being used to determine the name of the collection for this type. By Default it uses the `DefaultNameProvider` (which just uses the classname to build the collection name). see above

Embedded

Marks POJOs for object mapping, but don't need to have an ID set. These objects will be marshalled and un-marshalled, but only as part of another object (Subdocument). This has to be set at class level.

You can switch off camel case conversion for this type and determine, whether data might be used polymorph.

Capped

Valid at: Class level

Tells *Morphium* to create a capped collection for this object (see capped collections above).

Parameters:

- *maxSize*: maximum size in byte. Is used when converting to a capped collection
- *maxNumber*: number of entries for this capped collection

AdditionalData

Special feature for *Morphium*: this annotation has to be added for at lease *one* field of type Map<String,Object>. It does make sure, that all data in Mongo, that cannot be mapped to a field of this entity, will be added to the annotated Map properties.

by default this map is read only. But if you want to change those values or add new ones to it, you can

set `readOnly=false`

Aliases

It's possible to define aliases for field names with this annotation (hence it has to be added to a field).

@Alias({"stringList","string_list"})

kList strLst;

in this case, when reading an object from MongoDB, the name of the field `strLst` might also be `stringList` or `string_list` in mongo. When storing it, it will always be stored as `strLst` or `str_lst` according to configs camelcase settings.

This feature comes in handy when migrating data.

CreationTime

has to be added to both the class and the field(s) to store the creation time in. This value is set in the moment, the object is being stored to mongo. The data type for creation time might be:

- `long` / `Long`: store as timestamp
- `Eate`: store as date object
- `String`: store as a string, you may need to specify the format for that

LastAccess

same as creation time, but storing the last access to this type. **Attention**: will cause all objects read to be updated and written again with a changed timestamp.

Usage: find out, which entries on a translation table are not used for quite some time. Either the translation is not necessary anymore or the corresponding page is not being used.

LastChange

Same as the two above, except the timestamp of the last change (to mongo) is being stored. The value will be set, just before the object is written to mongo.

DefaultReadPreference

Define the read preference level for an entity. This annotation has to be used at class level. Valid types are:

- PRIMARY: only read from primary node
- PRIMARY_PREFERED: if possible, use primary.

- SECONDARY: only read from secondary node
- SECONDARY_PREFERED: if possible, use secondary
- NEAREST: I don't care, take the fastest

Id

Very important annotation to a field of every entity. It marks that field to be the id and identify any object. It will be stored as `_id` in mongo (and will get an index).

The Id may be of any type, though usage of ObjectId is strongly recommended.

Index

Define indexes. Indexes can be defined for a single field. Combined indexes need to be defined on class level. See above.

Property

Can be added to any field. This not only has documenting character, it also gives the opportunity to change the name of this field by setting the `fieldName` value. By Default the fieldName is ".", which means "fieldName based".

ReadOnly

Mark an entity to be read only. You'll get an exception when trying to store.

Version

Mark a field to keep the current Version number. Field needs to be of type Long!

Reference

If you have a member variable, that is a POJO and not a simple value, you can store it as reference to a different collection, if the POJO is an Entity (and only if!).

This also works for lists and Maps. Attention: when reading Objects from disk, references will be de-referenced, which will result into one call to mongo each.

Unless you set `lazyLoading` to true, in that case, the child documents will only be loaded when accessed.

transient

Do not store the field.

UseIfnull

Usually, *Morphium* does not store null values at all. That means, the corresponding document just would not contain the given field(s) at all.

Sometimes that might cause problems, so if you add `@UseIfNull` to any field, it will be stored into mongo even if it is null.

WriteSafety

Sepcify the safety for this entity when it comes to writing to mongo. This can range from "NONE" to "WAIT FOR ALL SLAVES". Here are the available settings:

- timeout: set a timeout in ms for the operation - if set to 0, unlimited (default). If set to negative value, wait relative to replication lag
- level: set the safety level:
  - `IGNORE_ERRORS` None, no checking is done
  - `NORMAL` None, network socket errors raised
  - `BASIC` Checks server for errors as well as network socket errors raised
  - `WAIT_FOR_SLAVE` Checks servers (at lease 2) for errors as well as network socket errors raised
  - `MAJORITY` Wait for at least 50% of the slaves to have written the data
  - `WAIT_FOR_ALL_SLAVES`: waits for all slaves to have committed the data. This is depending on how many slaves are available in replica set. Wise timeout settings are important here. See [WriteConcern in MongoDB Java-Driver][3] for additional information

The Aggregation Framework

The aggregation framework is a very powerful feature of MongoDB and morphium supports it from the start. But with *Morphium* V4.2.x we made use of it a lot easier.

Core of the aggregation Framework in *Morphium* is the `Aggregator`. This will be created (using the configured `AggregatorFactory`) by a `Morphium` instance.

```
Aggregator<Source,Result> aggregator=morphium.createAggregator(Source.class,Result.class);
```

This creates an aggregator, that reads from the entity `Source` and returns the results in `Result`. Usually you will have to define a `Result` entity in order to use aggregation, but with *Morphium* V4.2 it is possible to have a `Map` as a result class.

After preparing the aggregator, you need to define the stages. All currently available stages are also available in *Morphium*. For a list of available stages, just consult the mongodb documentation. Example:

```
Aggregator<UncachedObject, Aggregate> a = morphium.createAggregator(UncachedObject.class, Aggregate.class);
```

```
assert (a.getResultType() != null);
//reduce input
a = a.project("counter");
//Filter
a = a.match(morphium.createQueryFor(UncachedObject.class)
      .f("counter").gt(100));
//Sort, used with $first/$last
a = a.sort("counter");
//limit data
a = a.limit(15);
//group by - here we only have one static group, but could be any field or value
a = a.group("all").avg("schnitt", "$counter").sum("summe", "$counter").sum("anz", 1).last("letzter",
"$counter").first("erster", "$counter").end();

//result projection
HashMap<String, Object> projection = new HashMap<>();
projection.put("summe", 1);
projection.put("anzahl", "$anz");
projection.put("schnitt", 1);
projection.put("last", "$letzter");
projection.put("first", "$erster");
a = a.project(projection);

List<Aggregate> lst = a.aggregate();
assert (lst.size() == 1) : "Size wrong: " + lst.size();
log.info("Sum  : " + lst.get(0).getSumme());
log.info("Avg  : " + lst.get(0).getSchnitt());
log.info("Last :    " + lst.get(0).getLast());
log.info("First:   " + lst.get(0).getFirst());
log.info("count:  " + lst.get(0).getAnzahl());


assert (lst.get(0).getAnzahl() == 15) : "did not find 15, instead found: " + lst.get(0).getAnzahl();
```

Aggregation Expressions

MongoDB has support for an own expression language, that is mainly used in aggregation. _Morphium_s representation thereof is `Expr`.

Expr does have a lot of factory methods to create special `Expr` instances, for example `Expr.string()` returns a string expression (string constant), `Expr.gt()` creates the "greater than" expression and so on.

Examples of expressions:

```
Expr e = Expr.add(Expr.field("the_field"), Expr.abs(Expr.field("test")), Expr.doubleExpr(128.0));
Object o = e.toQueryObject();
String val = Utils.toJsonString(o);
log.info(val);
assert(val.equals("{ \"$add\" :  [ \"$the_field\", { \"$abs\" :  [ \"$test\"] } , 128.0] } "));

e = Expr.in(Expr.doubleExpr(1.2), Expr.arrayExpr(Expr.intExpr(12), Expr.doubleExpr(1.2), Expr.field("testfield")));
val=Utils.toJsonString(e.toQueryObject());
log.info(val);
assert(val.equals("{ \"$in\" :  [ 1.2,  [ 12, 1.2, \"$testfield\"]] } "));

e = Expr.zip(Arrays.asList(Expr.arrayExpr(Expr.intExpr(1), Expr.intExpr(14)), Expr.arrayExpr(Expr.intExpr(1),
Expr.intExpr(14))), Expr.bool(true), Expr.field("test"));
val=Utils.toJsonString(e.toQueryObject());
log.info(val);
assert(val.equals("{ \"$zip\" : { \"inputs\" :  [  [ 1, 14],  [ 1, 14]], \"useLongestLength\" : true, \"defaults\" :
\"$test\" }  } "));
```

```
e = Expr.filter(Expr.arrayExpr(Expr.intExpr(1), Expr.intExpr(14), Expr.string("asV")), "str", Expr.string("NEN"));
val=Utils.toJsonString(e.toQueryObject());
log.info(val);
assert(val.equals("{ \"$filter\" : { \"input\" :  [ 1, 14, \"asV\"], \"as\" : \"str\", \"cond\" : \"NEN\" }  } "));
```

the output of this little program would be:

```
{ "$add" :  [ "$the_field", { "$abs" :  [ "$test"] } , 128.0] }
{ "$in" :  [ 1.2,  [ 12, 1.2, "$testfield"]] }
{ "$zip" : { "inputs" :  [  [ 1, 14],  [ 1, 14]], "useLongestLength" : true, "defaults" : "$test" }  }
{ "$filter" : { "input" :  [ 1, 14, "asV"], "as" : "str", "cond" : "NEN" }  }
```

This way you can create complex aggregation pipelines:

```
Aggregator<UncachedObject, Aggregate> a = morphium.createAggregator(UncachedObject.class, Aggregate.class);
       assert (a.getResultType() != null);
       a = a.project(Utils.getMap("counter", (Object) Expr.intExpr(1)).add("cnt2", Expr.field("counter")));
       a = a.match(Expr.gt(Expr.field("counter"), Expr.intExpr(100)));
       a = a.sort("counter");
       a = a.limit(15);
       a = a.group(Expr.string(null)).expr("schnitt", Expr.avg(Expr.field("counter"))).expr("summe",
Expr.sum(Expr.field("counter"))).expr("anz", Expr.sum(Expr.intExpr(1))).expr("letzter",
Expr.last(Expr.field("counter"))).expr("erster", Expr.first(Expr.field("counter"))).end();
```

This expression language can also be used in queries:

```
       Query<UncachedObject> q = morphium.createQueryFor(UncachedObject.class);
   q.expr(Expr.gt(Expr.field(UncachedObject.Fields.counter), Expr.intExpr(50)));
   log.info(Utils.toJsonString(q.toQueryObject()));
   List<UncachedObject> lst = q.asList();
   assert (lst.size() == 50) : "Size wrong: " + lst.size();


   for (UncachedObject u : q.q().asList()) {
       u.setDval(Math.random() * 100);
       morphium.store(u);
   }

   q = q.q().expr(Expr.gt(Expr.field(UncachedObject.Fields.counter), Expr.field(UncachedObject.Fields.dval)));
   lst = q.asList();
```

Code Examples

Cache Synchronization

```
   @Test
   public void cacheSyncTest() throws Exception {
       morphium.dropCollection(Msg.class);
       createCachedObjects(1000);

       Morphium m1 = morphium;
       MorphiumConfig cfg2 = new MorphiumConfig();
       cfg2.setAdr(m1.getConfig().getAdr());
       cfg2.setDatabase(m1.getConfig().getDatabase());

       Morphium m2 = new Morphium(cfg2);
       Messaging msg1 = new Messaging(m1, 200, true);
       Messaging msg2 = new Messaging(m2, 200, true);
```

```
msg1.start();
msg2.start();

CacheSynchronizer cs1 = new CacheSynchronizer(msg1, m1);
CacheSynchronizer cs2 = new CacheSynchronizer(msg2, m2);
waitForWrites();

//fill caches
for (int i = 0; i < 1000; i++) {
    m1.createQueryFor(CachedObject.class).f("counter").lte(i + 10).asList(); //fill cache
    m2.createQueryFor(CachedObject.class).f("counter").lte(i + 10).asList(); //fill cache
}
//1 always sends to 2....


CachedObject o = m1.createQueryFor(CachedObject.class).f("counter").eq(155).get();
cs2.addSyncListener(CachedObject.class, new CacheSyncListener() {
    @Override
    public void preClear(Class cls, Msg m) throws CacheSyncVetoException {
        log.info("Should clear cache");
        preClear = true;
    }

    @Override
    public void postClear(Class cls, Msg m) {
        log.info("did clear cache");
        postclear = true;
    }

    @Override
    public void preSendClearMsg(Class cls, Msg m) throws CacheSyncVetoException {
        log.info("will send clear message");
        preSendClear = true;
    }

    @Override
    public void postSendClearMsg(Class cls, Msg m) {
        log.info("just sent clear message");
        postSendClear = true;
    }
});
msg2.addMessageListener(new MessageListener() {
    @Override
    public Msg onMessage(Messaging msg, Msg m) {
        log.info("Got message " + m.getName());
        return null;
    }
});
preSendClear = false;
preClear = false;
postclear = false;
postSendClear = false;
o.setValue("changed it");
m1.store(o);

Thread.sleep(1000);
assert (!preSendClear);
assert (!postSendClear);
assert (postclear);
assert (preClear);
Thread.sleep(60000);

long l = m1.createQueryFor(Msg.class).countAll();
assert (l <= 1) : "too many messages? " + l;
cs1.detach();
cs2.detach();
msg1.setRunning(false);
```

```
        msg2.setRunning(false);
        m2.close();
    }
```

Geo Spacial Search

```
    @Test
    public void nearTest() throws Exception {
        morphium.dropCollection(Place.class);
        ArrayList<Place> toStore = new ArrayList<Place>();
    //        morphium.ensureIndicesFor(Place.class);
        for (int i = 0; i < 1000; i++) {
            Place p = new Place();
            List<Double> pos = new ArrayList<Double>();
            pos.add((Math.random() * 180) - 90);
            pos.add((Math.random() * 180) - 90);
            p.setName("P" + i);
            p.setPosition(pos);
            toStore.add(p);
        }
        morphium.storeList(toStore);

        Query<Place> q = morphium.createQueryFor(Place.class).f("position").near(0, 0, 10);
        long cnt = q.countAll();
        log.info("Found " + cnt + " places around 0,0 (10)");
        List<Place> lst = q.asList();
        for (Place p : lst) {
            log.info("Position: " + p.getPosition().get(0) + " / " + p.getPosition().get(1));
        }
    }

    @Index("position:2d")
    @NoCache
    @WriteBuffer(false)
    @WriteSafety(level = SafetyLevel.MAJORITY)
    @DefaultReadPreference(ReadPreferenceLevel.PRIMARY)
    @Entity
    public static class Place {
        @Id
        private ObjectId id;

        public List<Double> position;
        public String name;

        public ObjectId getId() {
            return id;
        }

        public void setId(ObjectId id) {
            this.id = id;
        }

        public List<Double> getPosition() {
            return position;
        }

        public void setPosition(List<Double> position) {
            this.position = position;
        }

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
```

```
        }
    }
```

Iterator

```
    @Test
    public void basicIteratorTest() throws Exception {
        createUncachedObjects(1000);

        Query<UncachedObject> qu = getUncachedObjectQuery();
        long start = System.currentTimeMillis();
        MorphiumIterator<UncachedObject> it = qu.asIterable(2);
        assert (it.hasNext());
        UncachedObject u = it.next();
        assert (u.getCounter() == 1);
        log.info("Got one: " + u.getCounter() + "  / " + u.getValue());
        log.info("Current Buffersize: " + it.getCurrentBufferSize());
        assert (it.getCurrentBufferSize() == 2);

        u = it.next();
        assert (u.getCounter() == 2);
        u = it.next();
        assert (u.getCounter() == 3);
        assert (it.getCount() == 1000);
        assert (it.getCursor() == 3);

        u = it.next();
        assert (u.getCounter() == 4);
        u = it.next();
        assert (u.getCounter() == 5);

        while (it.hasNext()) {
            u = it.next();
            log.info("Object: " + u.getCounter());
        }

        assert (u.getCounter() == 1000);
        log.info("Took " + (System.currentTimeMillis() - start) + " ms");
    }
```

Asynchronous Read

```
      @Test
    public void asyncReadTest() throws Exception {
        asyncCall = false;
        createUncachedObjects(100);
        Query<UncachedObject> q = morphium.createQueryFor(UncachedObject.class);
        q = q.f("counter").lt(1000);
        q.asList(new AsyncOperationCallback<UncachedObject>() {
            @Override
            public void onOperationSucceeded(AsyncOperationType type, Query<UncachedObject> q, long duration,
List<UncachedObject> result, UncachedObject entity, Object... param) {
                log.info("got read answer");
                assert (result != null) : "Error";
                assert (result.size() == 100) : "Error";
                asyncCall = true;
            }

            @Override
            public void onOperationError(AsyncOperationType type, Query<UncachedObject> q, long duration, String error,
Throwable t, UncachedObject entity, Object... param) {
                assert false;
            }
        });
```

```
        waitForAsyncOperationToStart(1000000);
        int count = 0;
        while (q.getNumberOfPendingRequests() > 0) {
            count++;
            assert (count < 10);
            System.out.println("Still waiting...");
            Thread.sleep(1000);
        }
        assert (asyncCall);
    }
```

Asynchronous Write

```
    @Test
    public void asyncStoreTest() throws Exception {
      asyncCall = false;
      super.createCachedObjects(1000);
      waitForWrites();
      log.info("Uncached object preparation");
      super.createUncachedObjects(1000);
      waitForWrites();
      Query<UncachedObject> uc = morphium.createQueryFor(UncachedObject.class);
      uc = uc.f("counter").lt(100);
      morphium.delete(uc, new AsyncOperationCallback<Query<UncachedObject>>() {
          @Override
          public void onOperationSucceeded(AsyncOperationType type, Query<Query<UncachedObject>> q, long duration,
List<Query<UncachedObject>> result, Query<UncachedObject> entity, Object... param) {
                    log.info("Objects deleted");
          }

      @Override
      public void onOperationError(AsyncOperationType type, Query<Query<UncachedObject>> q, long duration, String
error, Throwable t, Query<UncachedObject> entity, Object... param) {
                assert false;
            }
     });

    uc = uc.q();
    uc.f("counter").mod(3, 2);
        morphium.set(uc, "counter", 0, false, true, new AsyncOperationCallback<UncachedObject>() {
            @Override
            public void onOperationSucceeded(AsyncOperationType type, Query<UncachedObject> q, long duration,
List<UncachedObject> result, UncachedObject entity, Object... param) {
                log.info("Objects updated");
                asyncCall = true;

            }

            @Override
            public void onOperationError(AsyncOperationType type, Query<UncachedObject> q, long duration, String error,
Throwable t, UncachedObject entity, Object... param) {
                log.info("Objects update error");
            }
    });

    waitForWrites();

    assert morphiuum.createQueryFor(UncachedObject.class).f("counter").eq(0).countAll() > 0;
    assert (asyncCall);
}
```

---

1. does only make sense, when there is more than one recipient usually ↩