```
{
    "title":
```

"Event Streaming & Message Queueing with MongoDB",

```
    "event":     "MongoDB Berlin 2013",
    "name":      "Adam Warski",
    "company":   "SoftwareMill",
    "e-mail":    "adam@warski.org",
    "twitter":   "@adamwarski"
}
```

About me

- CTO at SoftwareMill, http://softwaremill.com
- Programmer
- Engaged in some open-source projects
  - Hibernate Envers
  - ElasticMQ
  - Veripacks
  - SoftwareMill Bootstrap
- Blog @ http://www.warski.org/blog
- Current tech stack:
  - Scala
  - Akka
  - Spray
  - MongoDB

Plan

- Project background
  - High-performance messaging system
  - Accept requests from users
  - Process asynchronously
  - Provide reporting
- Using Mongo as a queue
- Using Mongo for Event Streaming in Java

Mongo Message Queue

The requirements

- **Persistent** messaging
  - Short-lived messages
  - Some may linger for a longer time
  - Messages shouldn't be lost
- **Fast**
  - But not insanely fast
  - Currently we need 1000s msgs / second
- If needed, possible to scale-up & **scale-out**

Queue interface

- **Send** a message (a String)
- **Receive** a message, blocking it for x seconds
- **Delete** a message

Sounds familiar?

- Amazon SQS semantics
- At-least-once delivery guarantee
- Also check out ElasticMQ, http://elasticmq.org

How to implement?

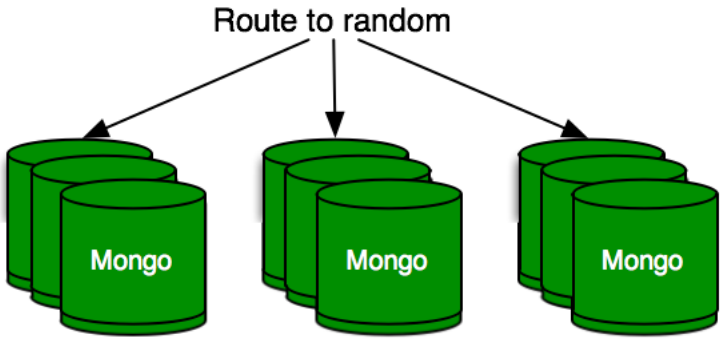
- Mongo **document structure**:
  - _id
  - Message content
  - Next delivery (timestamp)
- Message **send**:
  - Insert into collection
  - Next delivery := now
  - Return _id (message id)
- Message **delete**:
  - Delete document from collection

How to implement? (2)

- Message **receive**:
  - Find-and-modify
  - Find: next delivery must be <= now
  - Modify: next delivery := now + 10 seconds
- Why does this work?
  - Find-and-modify is **crucial**
  - Atomic operation

Meeting the requirements

- Replication OOTB
  - Replica Sets
- Scaling out
  - Starling/Kestrel model
  - Setup 2 identical replica sets (e.g. 2x3 servers)
  - Send/receive from a random server



mongoDB

softwaremill

Good/bad sides

- Good sides:
  - Easy to implement
  - Simple interface
  - Replication
- Bad sides:
  - Active polling
  - No batching

Write concerns

- Can we tolerate lost messages?
- Different write concerns during send
  - SAFE
  - REPLICA_SAFE

Mongo Event Streaming

General idea

- System generates a **series of events**
- Other components follow the stream
- Similar to Event Sourcing/CQRS
- Reading and writing of the events is **decoupled**
- Any following component may die & **catch up**
- Bursts of event activity don't cause an overall slowdown

The requirements

- Fast event writing
  - again, 1000s per second
- Main **source of truth** in the system
- Stream the events
  - as they are written
  - in batches
  - write reports to SQL DB
- Replicate data
- Store data up to Y GB
  - prevent lack of disk space

The collection

- Capped collection
  - By definition, size-constrained
  - We get a **circular buffer** for events
- Replicated
  - Hence an index on _id is mandatory
  - Until 2.2, capped collections didn't have an _id index by default

Writing events

- Insert
- Write concerns – how tolerant we are of event loss
- Events should be immutable
  - Nice (Java) code
  - Event sinks wouldn't know when events get updated
  - Changing document size – moving blocks on disk
  - Not possible in a capped collection

Reading events

- There may be multiple readers
- We want to get new events as they come in
  - But without active polling, if possible
- **Tailable cursors** are the answer
  - Need to provide a starting point – last read event
  - Will optionally block if no data is available
  - Can't be a TTL collection
- The reader must store the **last read event id**
  - Transactions can be useful here

Reading events (Java)

```
DBObject query = lastReceivedEventId.isPresent()
    ? BasicDBObjectBuilder.start("_id", BasicDBObjectBuilder
          .start("$gte", lookFrom(lastReceivedEventId.get()))
              .get())
          .get()
    : null;

DBObject sortBy = BasicDBObjectBuilder.start(
    "$natural", 1).get();

DBCollection collection = ... // must be a capped collection
DBCursor cursor = collection
    .find(query)
    .sort(sortBy)
    .addOption(Bytes.QUERYOPTION_TAILABLE)
    .addOption(Bytes.QUERYOPTION_AWAITDATA);
```

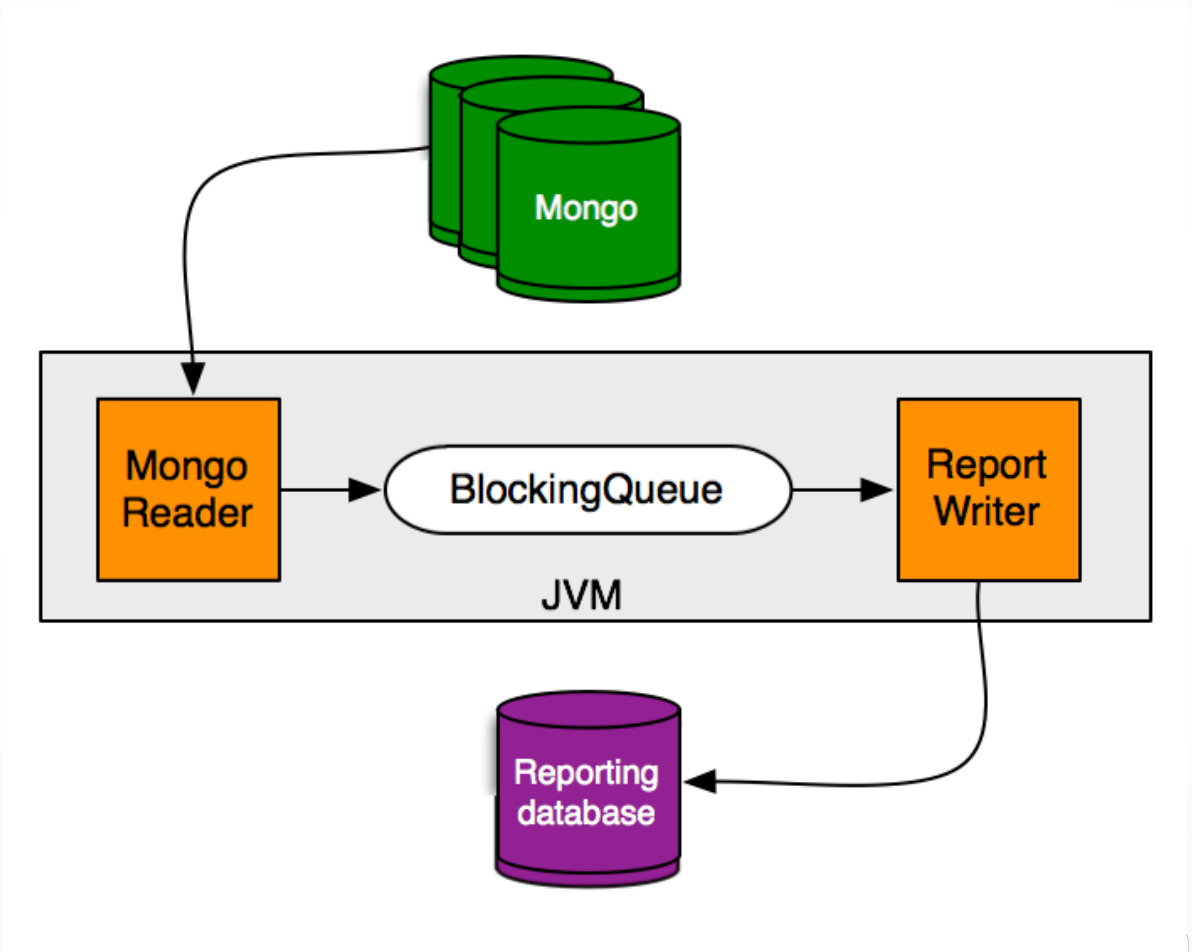
Reading events (Java)

- Note the **`$gte`**
  - Skip events until the last received event is found
  - Looking from … e.g. 10 minutes before the last received event
  - Cannot query for "documents created after a given document"
- We get a Java **`Iterator`**
- Data from Mongo is received in batches
  - Implemented by the Java driver
  - Only some calls to **`hasNext()/next()`** will cause network I/O
- The potentially blocking call is **`hasNext()`**

Intermediate queue

- To get events in batches without delays, we need an intermediate queue
- Two threads
  - One reading from Mongo
  - Second reading from the queue and processing
- E.g. a **`LinkedBlockingQueue`**
  - Has a size limit
  - When reading, first we do a blocking **`poll()`**
  - Then drain the queue

Intermediate queue



mongoDB

softwaremill

Use dedicated components?

- Maybe, in the future
  - If performance requirements rise
- The components are very easy to replace
  - If you write nice code, that is ;)
- Simplified setup & deployment
  - Both local, and on production
- Fewer external components
- Focus on the business problem, not on the infrastructure
  - As always, a question of balance

Thank you!

Blog: http://www.warski.org/blog

E-mail: adam@warski.org

Twitter: **@adamwarski**