AxonDB Guide

AxonIQ B.V.

Version 1.1-RC1, 2018-01-29

Table of Contents

Chapter 1. Introduction

1.1. About

AxonDB is a purpose-built database system optimized for the storage of event data of the type that is generated by applications that use the event sourcing architecture pattern. It has been primarily designed with the use case of Axon Framework-based Java applications in mind, although there is nothing in the architecture that restricts its use to these applications only. When used with Axon Framework, AxonDB is a drop-in replacement for the other event storage options supported by Axon Framework. Because AxonDB is designed specifically for this single purpose, its performance and scalability far exceed the performance and scalability of a general-purpose database system used in this role.

1.2. Core functionality

To describe the functionality of AxonDB in more detail, it’s useful to first introduce some of the key terminology of domain-driven design (DDD) and event sourcing that is also used throughout Axon Framework and AxonDB. The explanation here is the minimum to allow us to define AxonDB’s functionality. We refer to the DDD and event sourcing literature for more elaborate discussions of these topics.

Aggregate

An aggregate is an object or a group of objects that are treated as a single unit from a persistence and consistency point of view. Aggregates always have a type and a unique identifier, which can be used to load an instance of an aggregate.

Domain Event

An object that describes some form of change to an aggregate. (Where change also includes creation or deletion.) The core attributes of a domain event are the global sequence number (unique within all events), an aggregate identifier (linking the event to the aggregate it refers to), a sequence number (unique for the aggregate identifier), and of course the actual event payload.

Snapshot Event

An object that describes the full state of an aggregate at some point in time. Conceptually, these are not needed when using event sourcing. Technically, they are a performance optimization that can be very useful if aggregates contain large numbers of events. Snapshot events have an aggregate identifier, a sequence number (referring to one of the events of the aggregate), but conceptually don't have a global sequence number.

Event

Somewhat overloaded term that may either mean an event in general (which could be a domain or snapshot event), or may be shorthand for a domain event. The context should disambiguate between the two cases.

Event Stream

The full collection of events that have been generated.

Tracking Token

Represents a defined point in the event stream. When looking at AxonDB in isolation, a tracking token is essentially the same as the global sequence number. In more complex scenarios (such as using multiple event store systems in parallel), both concepts may diverge.

Expressed in these terms, AxonDB offers the following primary functions:

- Append one or more domain events to the event stream, with a given aggregate identifier, sequence number and optional metadata. The global sequence number is generated by AxonDB.
- Append a snapshot event to AxonDB, with a given aggregate identifier, sequence number and optional metadata.
- Read all domain events from the event stream, starting at a given tracking token, and continue to read new events real-time as they are appended. This functionality is geared towards event handlers, including those that are designed to create and maintain read models.
- Read the events belonging to a certain aggregate. Reading can take place from sequence number 0, or from a certain given sequence number. Reading can either take into account snapshots (returning the last snapshot and the domain events since that snapshot), or ignore snapshots altogether and simply return all domain events. This functionality is intended to support an aggregate repository / command handler.
- Retrieve the highest sequence number for a given aggregate. This is intended for applications that only want to register new domain events for an aggregate, without needing its current state. Akka Persistence is an example of a framework that needs this functionality in its back-end event storage system.

1.3. Non-functional characteristics

Organizations using Axon Framework with event sourcing that are using a traditional RDBMS to store those events, and produce substantial amounts of events, have regularly reported performance and scalability challenges. The main motivation for building AxonDB was to provide a better performing, easier to operate alternative for those use cases. As a result, the non-functional characteristics of AxonDB can be summarized as follows:

- AxonDB offers superior throughput (events per second) and scalability (number of events stored) compared to an RDBMS.
- It offers similar levels of reliability (failover, backup, ACID transactions etc.).
- To achieve this, it has much lower flexibility.

The architecture to implement these characteristics will be outlined in some detail in chapter 2. Here, we want to limit ourselves to describing the core ideas behind this.

General purpose database systems (including both RDBMSes as well as NoSQL systems) make very few assumptions about what application programmers want to do with the database. They either have flexible schemas, or are flexible because they are schema-less. They typically support full create/read/update/delete (CRUD) operations, and to maintain integrity in this process, have advanced locking mechanisms. They can be told to make some assumptions about the types of queries typically performed (by means of creating appropriate indices), but they generally don't make assumptions about which records are most likely to be accessed.

Looking at the tasks expected from an event store, we see some rather different characteristics. The schema can be completely fixed. Of course, event *payload* structure may very well evolve over time, but for an event store, the event payload is opaque. An event store doesn't need to support full CRUD operations: events can be appended (which is a restricted form of create), and be read (always in a given order), but are never updated and never deleted. And when using event sourcing, recent events are far more likely to be accessed for read than older events. (This s due to the nature of most business processes, and may technically be amplified by using snapshots.)

General purpose databases systems are unnecessarily slow when managing events, because they carry so much weight with them which is actually not helpful for the task. Also, since they can't exploit the specific properties of events, their performance degrades as the number of events stored grows. This is caused by an ever smaller portion of the events table fitting into memory buffers, and an increasing computational complexity of maintaining the indices.

AxonDB has been built from scratch to exploit the specific properties of events. Exploiting the fact that there is only append and read, it can use a much simpler mechanism to manage concurrent modification than general purpose locking algorithms. Given that the global sequence number of events is a primary key, the only index to maintain is the one on aggregate identifier and sequence number. Given that the most recent events are most likely to be read, it can keep those in memory and retrieve older ones from disk. Building on these ideas, AxonIQ has created its high-performance AxonDB. Some benchmark data about this can be found in Figure 1 below.

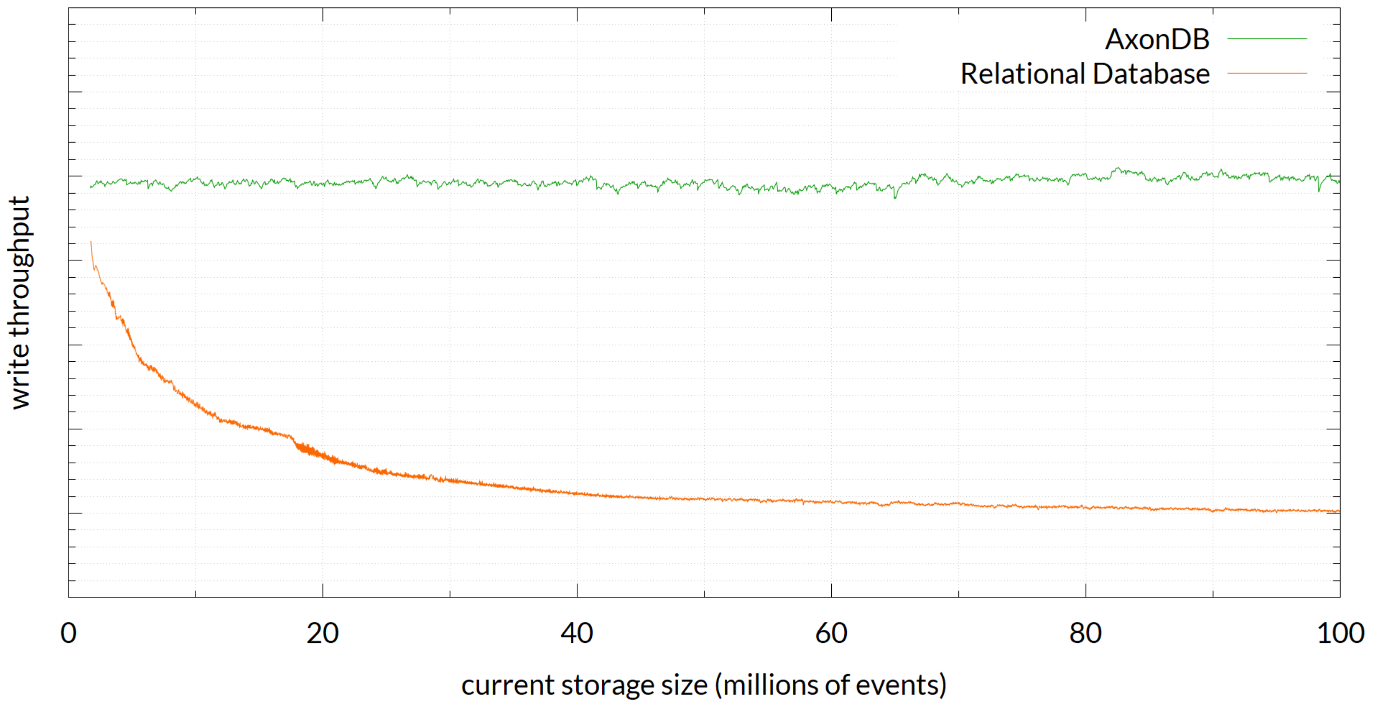

Figure 1. AxonDB scalability

1.4. Event bus versus event store

Aside of the non-functional benefits outlines above, the AxonDB has another great characteristic that makes it a great for event sourced applications: it is an event bus and event store at the same time. We will clarify this below.

When building an application without event sourcing, but which does use domain events for communication between components, one needs a form of an event bus that handles the distribution of events. In the simplest case, this may be a simple Java component that manages distribution of events within a JVM. In a more complex case, this may be an adapter to an external messaging system like RabbitMQ or Kafka.

When building an application with event sourcing, one needs a place to append and read events: the event store. Of course, distribution of the events via an event bus is also still required. In Axon Framework 2, the two functionalities were modeled as separate Java interfaces: EventBus and EventStore.

In reality, it turns out that these two concepts are not that clearly separated:

- Some event handlers may wish to exclusively receive new events as they happen, but other may be interested in reading all events that have occurred, including the ones that are already in the past. In Axon Framework parlance, the first kind is called a *subscribing* event processor and the second kind a *tracking* event processor. A tracking processor needs characteristics of both the event store and event bus.
- From a reliability perspective, it is undesirable if storing in the event store would succeed, but sending on the event bus would fail (or vice versa). Publishing an event to event store and event bus should be an atomic operation.

Because of these considerations, things were modeled differently in Axon Framework 3 compared to Axon Framework 2: in the newer version, the EventStore interface extends the EventBus interface. It's possible to have an EventBus that's not an EventStore, but every EventStore is also an EventBus.

Although this is conceptually clear, it's not straightforward to implement on a general purpose database. There's no easy way to actively push changes from an events table in a database to the various nodes in a distributed application. Plain SQL simply doesn't have the functionality for that - it assumes that the client queries the server, not that the server actively wants to report something to the client. Something could be set up using database triggers and messaging systems, but it would become pretty involved and highly database vendor specific. To get over this problem, Axon Framework tracking event processors actually poll databases containing events, at a configurable interval which defaults to 1 second. This is clearly not ideal.

AxonDB doesn't rely on SQL, but has its own dedicated protocols building on gRPC. This allows full bidirectional traffic. When one client appends an event, AxonDB will store this event *and* at the same time push this events to event processors that are currently listening. It is a clean implementation of the EventStore-is-an-EventBus concept.

1.5. License

The Axon Framework itself is open source and licensed under the Apache License v2.0.

AxonDB is a commercial software product by AxonIQ B.V. Please contact us via https://axoniq.io or sales@axoniq.io if you are interested in acquiring a license and support contract. A free, non-clustered Developer edition can be downloaded after registration on our web site. Free evaluation licenses of the other editions can be made available on request.

The Java client to AxonDB is open source under the Apache License 2.0. It is available through GitHub (https://github.com/AxonIQ) and can be downloaded from Maven Central.

Chapter 2. Architecture

2.1. Overview

In the introductory chapter, we have described the primary event and snapshot storage functionality of the event store. Of course, there is also a set of secondary functionality to support this. The systems needs to be managed and monitored. Also, AxonDB will usually operate in a cluster, which means that there has to be functionality for cluster discovery and data replication.

All of this functionality needs to have network connectivity. We have chosen to use open, documented protocols which can be directly used by other systems. In particular, the following protocols are used:

- **gRPC** (which is a protocol for remote procedure calls based on binary serialization with protobuf, using HTTP/2 as a transport mechanism). This is the protocol used for all high performance needs. AxonDB uses specific gRPC features such a data streaming (to allow pushes from the server to the client) and flow control.
- **JSON over HTTP/1.1**. This is the protocol used for things that do not require high performance, but that do require easy interoperability, for instance with tools like *curl*.

An outline of the AxonDB architecture, its components and network interfaces is provide in Figure 2 below.

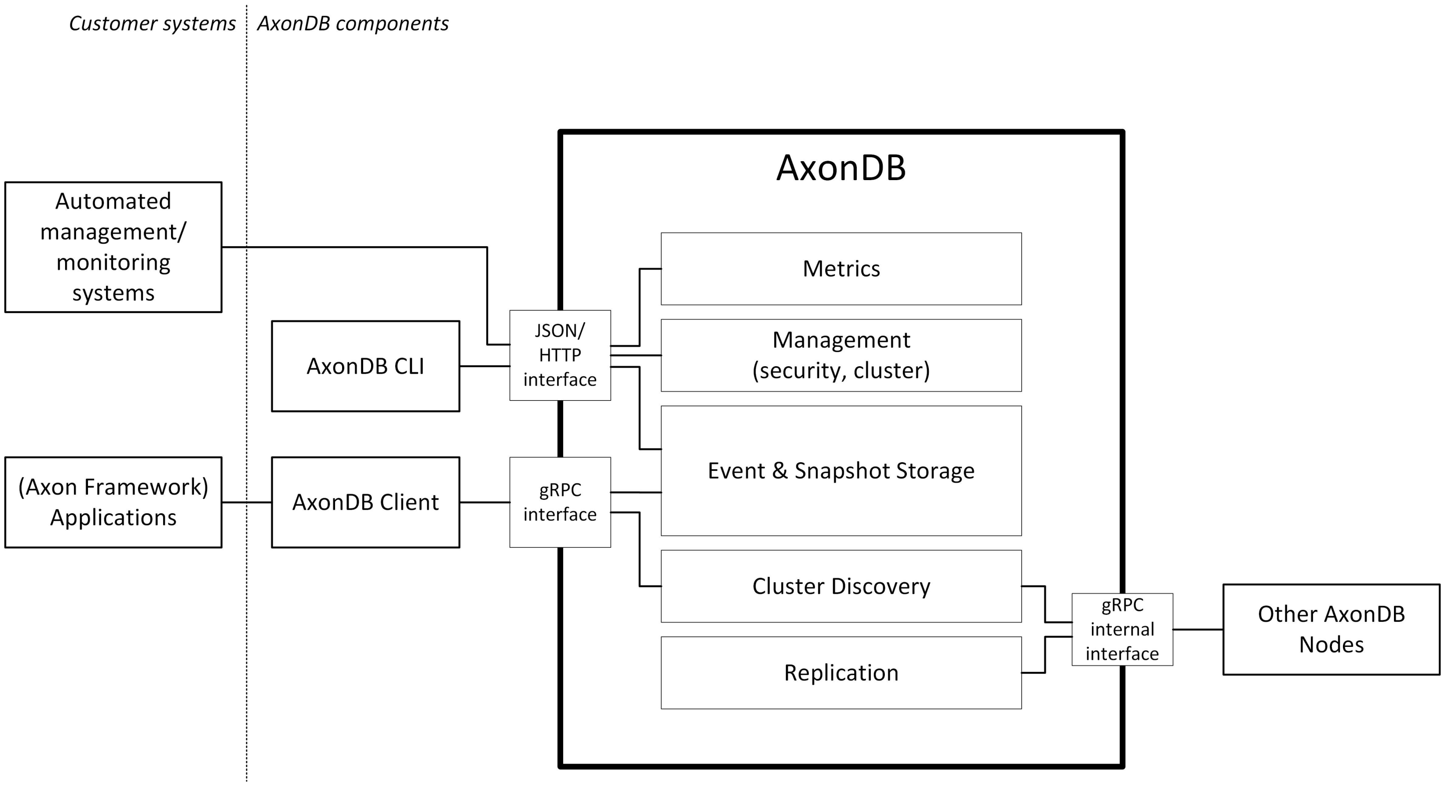

Figure 2. AxonDB architecture overview

AxonDB has 5 logical components that implement the primary storage functionality and the 4 supporting functionalities. The JSON/HTTP interface connects to the metrics, the management functionality, and to the event & snapshot storage. For the event & snapshot storage, this interface should be regarded as a testing facility. It is a lot slower than the gRPC interface and is not cluster-aware.

The main gRPC interface connects to both the event & snapshot storage component, and the cluster discovery component. A client can use the latter to retrieve information about the current master, and then connect to that master for actual append and read actions.

There is a secondary gRPC interface, the internal interface, which connects the cluster discovery and replication components. This is used for intra-cluster traffic. This interface always runs on a different network port than the primary gRPC interface. It cannot be used directly by applications and the gRPC protocol used on this connection is not publicized.

In addition to AxonDB itself, the distribution contains AxonDB CLI (command-line interface). This program uses the JSON/HTTP interface to perform management operations. There is also an AxonDB client that connects to the gRPC interface, which is open source.

The use of the JSON/HTTP interface and AxonDB CLI will be described in more detail in the Operations chapter. The use of the gRPC interface and AxonDB Client will be described in more detail in the Application Development chapter.

2.2. Implementation technology

AxonDB has been developed fully in Java, building on Spring Boot. It is distributed as a single jar file containing all libraries used through shading.

It does not use any underlying database management system for event & snapshot storage. The storage engine has been built from scratch on file system access. AxonDB does create and use small local H2 relational database. This is used to support the secondary functionality, but not to store events.

2.3. Storage mechanism

The storage mechanism separately stores domain events and snapshots. The functionality needed to store snapshots is a subset of what is needed to store domain events. For simplicity, AxonIQ has designed an single storage mechanism which has fulfills all the requirements for storing domain events. AxonDB employs two different instances of this mechanism: one for domain events and one for snapshot events. Because of this, snapshots have their own files rather than being included in the files containing domain events.

A core notion in the storage mechanism is the *segment*. A segment is a contiguous subset of the full event stream, identified by the global sequence number of its first event. When creating an initial/new segment, the segment gets a predefined (configurable) size in bytes. When the segment is full, it will be closed and a new segment will be initialized. Because of the append-only functionality of AxonDB, a segment that has been closed cannot change anymore. If the configured segment size changes over time, this new size will be applied to new segments but not to old ones. This is perfectly fine: AxonDB does not assume that all segments have the same size.

For each segment, AxonDB creates 3 files:

- A **.events** file. This is the file containing the actual event data.
- A **.index** file. This is an index into the corresponding .events file with aggregate identifier + sequence number as index key.
- A **.bloom** file. This is a Bloom filter on aggregate identifier for the segment. This is a fast, probabilistic filter that may provide certainty that the aggregate identifier is not in the segment.

AxonDB uses a tiered storage mechanism for the segments. A currently open file will be fully available in RAM in addition to its storage on disk. The second tier of recently used files will be accessed using memory-mapped files. The third tier of all other files will be access using regular file input streams.

In the current release, all segments files are stored in a single directory. Future versions of AxonDB may support a more advanced storage tiers mechanism which supports having segment files in separate directories, for instance one being a mount point for a fast SSD and another for a slower HDD.

Future versions of AxonDB may support the notion of having several different contexts instead of just one global event stream. In anticipation of this, the event files are stored in a subdirectory called "default" under the configured storage directory. Instead of "default", future versions may support arbitrary context names.

2.4. Transactional guarantees

AxonDB offers a fully ACID compliant transactional model, as needed for event sourcing.

Atomicity

Multiple events may need to be appended in a single transaction, and atomicity means that they are either all committed, or none of them. This is fully supported by AxonDB. A single client request is identified with a single transaction. It is the responsibility of the AxonDB Client to buffer individual append event operations that belong to the same transaction, and submit these as a single request upon commit. AxonDB will then manage the set of events as a single operation on the event stream, which will either succeed or fail.

Consistency

Consistency requirements on the application level are not a responsibility of an event store - the content of an event doesn't matter to the event store. The key consistency requirement is in sequence number. For every aggregate identifier, the first sequence number should be 0, and sequence numbers should be unique, contiguous and increasing. AxonDB enforces this and will deny a transaction if the rule is violated. This is a normally occurring scenario in an event sourced application when aggregates are concurrently modified. The normal response from the application would be to automatically retry the command, which will lead to new events with new sequence numbers.

Isolation

Isolation is achieved along the same lines as atomicity.

Durability

To allow high throughput, AxonDB does not do an fsync before committing every transaction, but only periodically with a 1 second interval. This does leave a window of 1 second in which transactions could exist that are confirmed to the client, but not actually durable in case of a sudden shutdown. To compensate for this, we recommend to run AxonDB in a clustered set-up. A transaction will then only be confirmed after a quorum of nodes has received the transaction. This ensures full durability even in the face of individual failing nodes.

2.5. Clustering

AxonDB can run either stand-alone or in a clustered mode. For production, we recommend to run in clustered mode, both for reasons of general system availability and to guarantee durability of all transactions.

The cluster set-up has a single master. All other nodes are replicas. All read and append actions have to take place at the master. Changes at the master are pushed to all the replicas. Currently, all nodes have all data. There is no notion of sharding. This may change in future releases.

Every node can potentially be a master. To determine which one will be the master, the nodes execute a leader election algorithm. One of the key parts of this algorithm is that a new master has to be one of the nodes that is aware of the highest global sequence number. Additional nodes can be added to the cluster at will. They will start to replicate with the master and will can potentially become a master themselves after they have caught up with the current master.

The clustering algorithm uses the quorum concept in two ways. When the client wants to write to the master, the master won’t acknowledge the transaction until it has been received by a quorum of nodes. This quorum is half + 1 of the nodes, rounded down, including the master itself. So in a 2-node cluster, the master needs acknowledgment by the one replica. In a 3-node cluster, by 1 of the 2 replicas. In a 4-node cluster, by 2 of the 3 replicas. In a 5-node cluster, by 2 of the 4 replicas. Etc.

A similar logic applies to the leader election process. In order to become a master, a node needs approval by the same quorum of nodes. The main reason for this is to prevent a *split brain* situation in which the cluster gets split in smaller clusters that each have a node claiming to be master. As long as all nodes agree on the number of nodes that should be in the cluster, the quorum logic prevents this situation.

Although AxonDB supports clusters of any number of nodes, it follows from the above logic that it makes sense to use clusters with an odd number of nodes. The simplest cluster of 2 requires both nodes to be alive to function, so doesn’t add any availability guarantees compared to a single node. Adding a 3rd one does allow one node to fail. Adding a 4th one doesn’t make that better, but adding a 5th one does. Etc.

2.6. Security

AxonDB offers the following security features:

- All 3 interfaces support SSL. This is currently limited to server-side SSL, in other words, it protects the confidentiality and integrity of data-in-motion but doesn’t perform client authentication.
- There is an access control mechanism based on the following concept: access to AxonDB is usually performed by applications (rather than people). These applications authenticate themselves by including an access token in their header. Within AxonDB, applications have a particular role. These roles map to specific endpoints on AxonDB.
- Encryption of data-at-rest is currently supported by the client. When this feature is used, the client will encrypt all event payloads before sending them to AxonDB. AxonDB itself only has access to the metadata in the clear. This functionality is further explained in the Application Development chapter.

There is a specific security-related issue regarding event sourcing that keeps popping up in the context of the GDPR (the new EU privacy framework becoming fully enforceable on 25 May 2018). Under this law, data subjects have a conditional right "to be forgotten". When they exercise this right, data processors are mandated to delete personally identifiable information they have on that subject. This is somewhat hard to reconcile with the idea of event sourcing, to keep all events indefinitely without mutations. AxonIQ offers a specific module that solves this issue using cryptographic algorithms. This module can be used with AxonDB, but also with RDBMS or NoSQL event stores. Please contact sales@axoniq.io if you want to learn more about this.

Chapter 3. Installation

3.1. Prerequisites

A Java 8+ JRE should be installed on the system.

3.2. Installation procedure

Copy axondb-server-1.1-RC1-exec.jar to a directory of your choice.

Create a file axoniq.properties with configuration parameters. The minimal content of this configuration file should be:

```
axoniq.axondb.file.storage=[storage-location]
```

where storage-location is the directory where AxonDB stores its datafiles.

This will set up a default server, which is no cluster enabled, no SSL enabled and requires no authentication from clients. This is to get started, but of course for a production environment all these options should be enabled.

For testing, you can simply execute the jar with `java -jar`, but for production purposes you would wrap this in some service startup/shutdown script. Such a script is currently not a part of the distribution.

If you start AxonDB without a license parameter, it will look for a file axoniq.license in the working directory. If this is not found it will run in development mode which means it is not possible to run in a cluster and the database size is restricted. To specify a license and unlock higher throughput, place the axoniq.license file in the working directory or specify the 'license' system property and make it point to a license file, e.g. `-Dlicense=axoniq.license` if your license file is axoniq.license.

All control functionality of AxonDB can be done through JSON/HTTP calls. The API can be explored via /swagger-ui.html. Actions can be automated using curl or similar tooling. In addition, we offer a command line interface that makes this simpler. This can be installed by copying axoniq-cli-1.1-RC1-exec.jar and simply executing with `java -jar`.

3.3. Cluster configuration

AxonDB runs in single master mode. To set up the cluster configuration the different AxonDB nodes need to know each other. This is done using the following properties in the axoniq.properties configuration file:

- **axoniq.axondb.hostname** - sets the hostname as this node advertises it to clients
- **axoniq.axondb.domain** - sets the domain as used in returning the server address to clients
- **axoniq.axondb.cluster.enabled** [true] - sets the server in cluster mode
- **axoniq.axondb.cluster.name** - logical name of the node in the cluster. This must be unique.
- **axoniq.axondb.cluster.internal-hostname** - hostname to be used by other nodes in the server cluster
- **axoniq.axondb.cluster.internal-domain** - domain to be used by other nodes in the server cluster
- **axoniq.axondb.cluster.internal-port** - internal gRPC port number for communication to other nodes

When there is no hostname specified it defaults to the hostname as returned by the **hostname** command. The node name defaults to the hostname.

Sample configuration:

```
axoniq.axondb.hostname=eventstore1.axoniq.io
axoniq.axondb.cluster.enabled=true
axoniq.axondb.cluster.name=eventstore1
```

Creating a cluster is done through the following steps:

- setup each AxonDB server individually.
- use the command line interface to connect the nodes, using the following command:

```
axoniq-cli-1.1-RC1-exec.jar register-node -S http://node-to-add:port -h node-in-
cluster -p internal-port-of-node-in-cluster
```

or when not running in a bash shell environment:

```
java -jar axoniq-cli-1.1-RC1-exec.jar register-node -S http://node-to-add:port -h
node-in-cluster -p internal-port-of-node-in-cluster
```

This sends the command to one of the nodes to join the cluster containing the node **node-in-cluster**. For example, say there are 3 nodes that need to be clustered (node1, node2 and node3), each runs on the default ports (http port: 8023, internal grpc port 8223), run the following commands:

```
axoniq-cli-1.1-RC1-exec.jar register-node -S http://node2:8023 -h node1 -p 8223
axoniq-cli-1.1-RC1-exec.jar register-node -S http://node3:8023 -h node1 -p 8223
```

To configure the cluster in the Axon client, specify the following property:

- axoniq.axondb.servers: comma separated list of servername:port tuples of the nodes in cluster

Sample:

```
axoniq.axondb.servers=eventstore1.axoniq.io:8123, eventstore2.axoniq.io:8123,
eventstore3.axoniq.io:8123
```

3.4. SSL configuration

Requires private key and public certificate. The same certificate is used to connect to all event store servers.

For gRPC communication add file locations to the axoniq.properties file:

- **axoniq.axondb.ssl.certChainFile** - location of the public certificate file
- **axoniq.axondb.ssl.privateKeyFile** - location of the private key file

Sample:

```
axoniq.axondb.ssl.enabled=true
axoniq.axondb.ssl.certChainFile=./resources/axoniq-public.crt
axoniq.axondb.ssl.privateKeyFile=./resources/axoniq-private.pem
```

For HTTPs configuration the certificate and key need to be installed in a p12 keystore. To install the keys use the following command:

```
openssl pkcs12 -export -in [public-cert-file] -inkey [private-key-file] -out
[keystore-file] -name [alias]
```

Configure AxonDB server to use HTTPs by adding the following properties to the axoniq.properties file:

```
server.port=8443
server.ssl.key-store=[keystore-file]
server.ssl.key-store-password=[password]
server.ssl.keyStoreType=PKCS12
server.ssl.keyAlias=[alias]
```

3.5. Access control

By default, all services have restricted access using an Access Token. To disable access control add the following property to axoniq.properties:

- **axoniq.axondb.accesscontrol.enabled**=false

To register an application and get an access token use the following command:

```
axoniq-cli-1.1-RC1-exec.jar register-application -S http://eventstore:8023 -a
applicationname -d description -r READ,WRITE
```

The address of the server specified in this command is the address of the current master node. The master will distribute the applications to all the replicas.

This command returns the generated token to use. *Note that this token is only generated once, if you loose it you must delete the application and register it again to get a new token*. Specify the access token in the client by setting the property:

- **axoniq.axondb.token**=[Token]

3.6. Flow control

Flow control is the process of managing the rate of data transmission between two nodes to prevent a fast sender from overwhelming a slow receiver.

In AxonDB server flow control is possible both between the axon-client and the AxonDB server, and between the nodes in an AxonDB service cluster.

3.6.1. Client - AxonDB Server

The client needs to set the following properties to configure flow control:

- **axoniq.axondb.flowControl.initialNrOfPermits** [100000] - number of messages that the server can initially send to client.

- **axoniq.axondb.flowControl.nrOfNewPermits** [90000] - additional number of messages that the server can send to client.
- **axoniq.axondb.flowControl.newPermitsThreshold** [10000] - when client reaches this threshold in remaining messages, it sends a request with additional number of messages to receive.

3.6.2. Replica - Master

Set the following properties to set flow control on the synchronization between master and replicas:

- **axoniq.axondb.synchronize.stream.initialNrOfPermits** [100000] - number of messages that the master can initially send to replica.
- **axoniq.axondb.synchronize.stream.nrOfNewPermits** [90000] - additional number of messages that the master can send to replica.
- **axoniq.axondb.synchronize.stream.newPermitsThreshold** [10000] - when replica reaches this threshold in remaining messages, it sends a request with additional number of messages to receive.

To disable flow control set **axoniq.axondb.synchronize.stream.initialNrOfPermits** to 0 (Not recommended).

3.7. Migration

The AxonDB package contains a migration tool to migrate from an already existing RDBMS event store to a new AxonDB instance. The tool reads events and snapshots from the existing store and pushes them to the AxonDB server.

The migration tool maintains the state of its migration, so it can be run multiple times.

Set the following properties to define the existing event store and the target AxonDB server:

- **axoniq.axondb.servers** - comma separated list of hostnames and ports for the AxonDB cluster.
- **axoniq.datasource.eventstore.url** - URL of the JDBC data store containing the existing event store
- **axoniq.datasource.eventstore.username** - Username to connect to the JDBC data store containing the existing event store
- **axoniq.datasource.eventstore.password** - Password to connect to the JDBC data store containing the existing event store

The default settings expect the data in the current event store to be serialized using the XstreamSerializer. When the data is serialized using the JacksonSerializer add the following property:

- **spring.profiles.active**=migrate-from-jpa,jackson

To run the migration tool create a file axoniq.properties, containing the properties mentioned above, e.g.

```
axoniq.axondb.servers=eventstore1.axoniq.io:8123,eventstore2.axoniq.io:8123,events
tore3.axoniq.io:8123
axoniq.datasource.eventstore.url=jdbc:mysql://localhost:3306/applicationdb?useSSL=
false
axoniq.datasource.eventstore.username=myusername
axoniq.datasource.eventstore.password=mypassword
```

Run the command:

```
axondb-migration-1.1-RC1-exec.jar
```

Chapter 4. Operations

4.1. Starting an AxonDB Server

Start AxonDB with the following command:

```
axondb-server-1.1-RC1-exec.jar
```

or when not running in a bash shell:

```
java -jar axondb-server-1.1-RC1-exec.jar
```

4.2. Stopping an AxonDB Server

Either press Ctrl-C in the console window, or call the stop API.

4.3. SSL troubleshooting

Symptom: Client receives exception *INTERNAL: Connection closed with unknown cause*, no logging on the server.

Cause: Server is running in SSL mode, client is not.

Solution: Add certificate to client configuration.

Symptom: Client receives exception *UNAVAILABLE*. Server gives error: *HTTP/2 client preface string missing or corrupt*.

Cause: Client is using SSL, server is not.

Solution: Add SSL configuration to the server.

Symptom: Client receives exception *UNAVAILABLE*. Server gives no error.

Cause: Possibly the hostname used by the client does not match the hostname pattern in the certificate.

Solution: Check certificate values from client by starting the client with java option -Djavax.net.debug=all. This will output something like:

```
adding as trusted cert:
  Subject: CN=*.axoniq.io, O=AxonIQ
  Issuer:  CN=*.axoniq.io, O=AxonIQ
  Algorithm: RSA; Serial number: 0x92047296751fe115
  Valid from Mon Jul 31 15:18:29 CEST 2017 until Sat Jul 30 15:18:29 CEST 2022
```

Verify that the name for the server matches the common name (CN) for the subject.

Symptom: Client receives error *No name matching [hostname] found*

Cause: Server is not returning a fully qualified name. During the initial connection from client to server the server returns the address of the master node.

Solution: add property **axoniq.axondb.domain** to the server configuration.

4.4. Backup

The core strategy employed by AxonDB to keep data available, is to replicate it over various cluster nodes. These nodes should be in data centers that are isolated from each other in relevant disaster scenarios. In such a set-up, it is feasible to operate AxonDB without ever making explicit backups to off-line media. Nevertheless, there are also environments where such backups are a strict requirement and for that reason, AxonDB does support it.

There are two types of items to be backed up: the control database, and the event stream segments. It is possible to recover the data in an event store without the control database, but that might make recovery slower and therefore we recommend to back it up.

Within the endpoints documented in /swagger-ui.html, there are two endpoints of the Backup Info Rest Controller. The support in creating a consistent backup.

The control database is a relational H2 database. Although it's stored in a single file, this file cannot be simply copied for backup as it may not be in a safe state. A call to the POST endpoint /v1/backup/createControlDbBackup forces the creation of a proper backup file. It returns the full path to that file, which can then be used to move that file to another storage medium.

The segment are either closed and immutable, or still open for new events. An open segment cannot be safely backed-up. For the closed segments, it is feasible to only backup the ones that haven't been backed-up yet, since the once that have been are guaranteed not to change. This logic is supported by the GET endpoint /v1/backup/filenames. It takes an event type (either Event or Snapshot) and optionally the last segment that has already been backedup. It will return a list of filenames belonging to segments that haven't been backed up yet, but which are now safe to backup by simply copying them.

Because the control database contains a pointer to the last event that is known to be stored safely on the cluster (the *safepoint*), the proper order of doing this is to first create the control database backup and then backing up the segments. This will ensure that the segments may have events beyond the safepoint (which is ok) but are not missing events before the safepoint (which would be bad).

4.5. Monitoring

For monitoring, AxonDB includes Spring Actuator endpoints. The documentation can be accessed via HTTP on the /docs endpoint of AxonDB. In particular, you will find that the /health endpoint can be used to verify that the host is up and reports disk space left.

Other, AxonDB-specific endpoints can be browsed through /swagger-ui.html. Especially important for monitoring is the /v1/cluster/count endpoint. This reports the current number of events stored in a node. Under normal operations, this call should give the same result for all nodes in a cluster, within a very small margin.

Chapter 5. Application Development

5.1. Accessing AxonDB

Applications can interact with AxonDB on various levels:

- AxonDB Client offers an implementation of the Axon Framework AxonDB interface. This is a drop-in replacement AxonDB for Axon Framework 3 applications.
- One level below, AxonDB Client project has an AxonDBClient class which offers access to AxonDB functionality without Axon Framework.
- At the lowest level, AxonDB's gRPC protocol files are published. These can be used to create clients in any language.

AxonDB Client is open source. Its github repo is https://github.com/AxonIQ/axondb-client. In there, you will find the axondb-grpc-proto module (which contains only the gRPC protocol files), the axondb-client, which is the client itself, and a subdirectory 'examples' which contains examples on how to use this in various scenarios. We encourage you to study these examples.

The artifacts (excluding the examples) are also available through Maven Central:

```xml
<dependency>
    <groupId>io.axoniq</groupId>
    <artifactId>axondb-client</artifactId>
    <version>1.0.1</version>
</dependency>

<dependency>
    <groupId>io.axoniq</groupId>
    <artifactId>axondb-grpc-proto</artifactId>
    <version>1.0.1</version>
</dependency>
```

For the axondb-client artifact, several versions exist which can be selected through the 'classifier' element.

- The default build is a jar that only contains the client's own class files and expects dependencies to be included transitively as specified in the pom.xml.
- The 'complete' build contains all dependencies in a single jar. To avoid classpath collisions in case you happen to use different versions of the same dependencies, shading and relocation is applied. The gRPC implementation library depends on Netty, which in turn supports different options to get an SSL implementation (which is needed because the standard in Java 8 is insufficient for HTTP/2). In

the 'complete' build, the full BoringSSL implementation is included for various platforms.

- The 'nossl' build is like the 'complete' build, but excludes SSL dependencies, providing you with the option to configure that as needed.

We recommend to use the default build unless there is a specific reason not to.

5.2. Encryption of data at rest

AxonDB client optionally supports encryption of data-at-rest. All cryptographic operations take place at the client side. This works by encrypting the payload bytes of each event message, while leaving the other fields like aggregateIdentifier, sequenceNumber, as well as the metadata in the clear. This approach protects the confidentiality of the event data while still allowing AxonDB to operate efficiently. The encryption applies to both regular events and snapshots.

The encryption algorithm is standard AES, with all AES key sizes supported. Each event is encrypted/decrypted separately, using CBC mode and PKCS#5 padding. The IV is all zeroes. To ensure semantic security, a 6-byte random nonce is prepended to the cleartext of each event before encrypting and stripped off after decrypting. Also, a 4-byte fixed magic number is prepended before encrypting and verified after decrypting.

Encryption of data at rest is disabled by default. There is a simple and a flexible way to enable it. The simple method assumes that there is just a single AES secret key for all events. In this case, an `io.axoniq.axondb.client.util.EventCipher` object can be created with the secret key bytes as the single constructor parameter. This object can be provided to the `io.axoniq.axondb.client.AxonDBConfiguration` object. If you're using Spring, this can be performed automatically with a client-side axoniq.properties like this:

```
axoniq.axondb.eventSecretKey: Ah2b3218sd2K*3#a
```

In this case, keys are limited to contain only bytes from the US-ASCII charset.

The more flexible way of setting up encryption is to use the 2-argument constructor of `EventCipher`. The first argument is a key selection function. It takes an Event as input and should return an index of an encryption key to be used (or -1 in case the event shouldn't be encrypted). The second argument is the list of encryption keys. With this mechanism, multiple encryption keys can be used which are applied on various criteria, for instance Event metadata or sequence number.

Appendix A: AxonDB configuration scenarios

A.1. Cluster in same domain as application

A.1.1. Without SSL

All the servers, both servers running the application and the servers running the event store are in the same domain. For instance we have two servers running Axon Framework applications called appsvr1 and appsvr2, and there are 3 event store servers eventstore1, eventstore2 and eventstore3.

On the clients the configuration property axoniq.axondb.servers should be set to eventstore1:8123, eventstore2:8123, eventstore3:8123 (assuming the gRPC port for the event store servers is kept to the default port). The configuration property axoniq.axondb.ssl.enabled must be set to false.

In the server the property axoniq.axondb.cluster.enabled must be set to true. When none of the other configuration properties are set the node name will default to the hostname. Also the servers will communicate their hostname to the clients (the Axon application), so the application servers must be able to resolve these names.

To ensure that the AxonDB server nodes are in the same cluster execute the following AxonDB command line commands:

```
axoniq-cli-1.1-RC1-exec.jar register-node -S http://eventstore2:8023 -n
eventstore1 -p 8223
axoniq-cli-1.1-RC1-exec.jar register-node -S http://eventstore3:8023 -n
eventstore1 -p 8223
```

These two commands will make nodes 2 and 3 register themselves to be in the same cluster as eventstore1. The -p argument is the portnumber used by the event store to communicate between event store servers. This is 8223 by default.

A.1.2. With SSL

When using SSL the basic setup is similar, but we need to specify the key and certificate files. To use SSL 2 files are needed:

- key file (PEM)
- certificate (CRT) file, use wildcard certificate to access all event stores

These files must be configured in the event store servers and also put in an keystore for web service access.

As the wildcard certificate checks on a qualified name, all hostnames must be qualified. In our example all servers are in the domain example.com. The certificate is for *.example.com. On the client specify the servers (axoniq.axondb.servers) as eventstore1.example.com:8123, eventstore2.example.com:8123, eventstore3.example.com:8123. Set the property axoniq.axondb.ssl.enabled to true and specify the location of the certificate in axoniq.axondb.ssl.certChainFile.

On the server add the property axoniq.axondb.domain and set it to example.com. This will add the domain to all the host names. Add the following properties for SSL configuration:

- axoniq.axondb.ssl.enabled=true
- axoniq.axondb.ssl.certChainFile - location of the certificate file (CRT)
- axoniq.axondb.ssl.privateKeyFile - location of the key file (PEM)

Use the following commands to register the nodes in the cluster:

```
axoniq-cli-1.1-RC1-exec.jar register-node -S https://eventstore2.example.com:8443
-n eventstore1.example.com -p 8223
axoniq-cli-1.1-RC1-exec.jar register-node -S https://eventstore3.example.com:8443
-n eventstore1.example.com -p 8223
```

A.2. Cluster in other domain than application

When the event store servers are for instance hosted in the cloud, there is a difference in the fully qualified name that the clients use to communicate with the AxonDB servers and the names used for internal communication. So when the client sends a request to the AxonDB server it uses eventstore1.example.com, whereas if another AxonDB server sends a request it uses eventstore1.c.axoniq-platform.internal.

A.2.1. Without SSL

For the client the only change for this is that it needs to use fully qualified names for the AxonDB servers.

On the servers, the servers should communicate their addresses using the domain they are in, and based on their context. The axoniq.axondb.domain is still used to create a fully qualified name to be used be the clients. Next to that, the property axoniq.axondb.cluster.internal-domain is used to specify the domain that should be added to the hostname when other nodes in the cluster want to access the server.

Use the following commands to register the nodes in the cluster:

```
axoniq-cli-1.1-RC1-exec.jar register-node -S http://eventstore2.example.com:8023
-n eventstore1.c.axoniq-platform.internal -p 8223
axoniq-cli-1.1-RC1-exec.jar register-node -S http://eventstore3.example.com:8023
-n eventstore1.c.axoniq-platform.internal -p 8223
```

Note that the node names used in the -n option are the internal addresses.

A.2.2. With SSL

When using SSL in this situation the configuration needs two certificates, one for the external domain and one for the internal domain. The one for the external domain is configured as in the SSL example above. For the internal domain add the property axoniq.axondb.ssl.internalCertChainFile with the location of the second certificate file. Both certificates should have the same private key.

Use the following commands to register the nodes in the cluster:

```
axoniq-cli-1.1-RC1-exec.jar register-node -S https://eventstore2.example.com:8443
-n eventstore1.c.axoniq-platform.internal -p 8223
axoniq-cli-1.1-RC1-exec.jar register-node -S https://eventstore3.example.com:8443
-n eventstore1.c.axoniq-platform.internal -p 8223
```

A.3. Multiple AxonDB servers on one host

When running multiple AxonDB servers on the same host make sure that each one has a unique location for storing the files and unique gRPC ports (internal and external) and http ports. Check the following properties:

- server.port: unique http port
- axoniq.axondb.file.storage: location of this servers data files
- axoniq.axondb.port: unique gRPC port for clients
- axoniq.axondb.cluster.internalPort: unique gRPC port for intra-server communication

Appendix B: Release notes

New features in version 1.1-RC1:

- Ad-hoc query on AxonDB
- Authentication on AxonDB web pages

Changes in version 1.1-RC1:

- Properties file renamed from application.properties to axoniq.properties.
- Default http port changed to 8023
- Look and feel update for web pages

The current version 1.1-RC1 has the following known issues:

- Swagger doesn’t work in conjunction with the streams being returned. The swagger HTML page for GET /v1/events without an aggregateId does not return as this returns an event stream. This is a limitation in Swagger-UI which only effects manual access for testing purposes.
- Sample applications in axondb-client contain references to non-existing hosts. The sample client application still need to be cleaned up.

Appendix C: Configuration Properties

The following table gives a complete list of all the configuration properties available for AxonDB Server.

Property	Default value	Unit	Description
axoniq.axondb.accesscontrol.cache-ttl	30000	Millisecs	Cache time for a checked access token
axoniq.axondb.accesscontrol.enabled	false	Boolean	Access control enabled for AxonDB Server
axoniq.axondb.cluster.enabled	false	Boolean	Event Store running in cluster
axoniq.axondb.cluster.internal-domain		String	Domain to add to the internal hostname
axoniq.axondb.cluster.internal-hostname		String	Hostname used in internal communication, defaults to the standard hostname
axoniq.axondb.cluster.internal-port	8223	Integer	Port number used in internal communication
axoniq.axondb.cluster.internal-token		String	Token added to each request in internal communication
axoniq.axondb.cluster.leader-election-timeout	1000	Millisecs	Time that each node will wait for responses from other nodes in cluster during leader election
axoniq.axondb.cluster.leader-election-wait	1000	Millisecs	Time that the node will wait before starting leader election
axoniq.axondb.cluster.name		String	Unique name of the node in the cluster, defaults to fully qualified hostname
axoniq.axondb.cluster.notification-timeout	1000	Millisecs	Timeout for master notification messages
axoniq.axondb.cluster.start-sync-delay	500	Millisecs	Delay between the moment that replica recieves new master and it starts to synchronize
axoniq.axondb.domain		String	Domain name added to the hostname when returning the fully qualified hostname
axoniq.axondb.hostname		String	Hostname of the server, defaults to the hostname of the server. Used to contruct the fully qualified hostname
axoniq.axondb.port	8123	Integer	gRPC Port number of the AxonDB server to be used by clients
axoniq.axondb.ssl.cert-chain-file		Path	Path to the certificate file
axoniq.axondb.ssl.enabled	false	Boolean	Switch to enforce that all gRPC communication is SSL
axoniq.axondb.ssl.internal-cert-chain-file		Path	Path to the certificate file used for internal communication. Defaults to the value of axoniq.eventstore.ssl.cert-chain-file

Property	Default value	Unit	Description
axoniq.axondb.ssl.private-key-file		Path	Path to the private key file, same private key should be used for internal and external SSL
axoniq.axondb.synchronize.stream.initial-nr-of-permits	50000	Integer	Flow control initial number of permits for synchronization between master and replica
axoniq.axondb.synchronize.stream.max-number-waiting-for-sync	500	Integer	Maximum number of transactions waiting for synchronization, before the replica gives up and reconnects
axoniq.axondb.synchronize.stream.new-permits-threshold	10000	Integer	Flow control, threshold at which the replica sends a request with additional permits to the master
axoniq.axondb.synchronize.stream.no-event-timeout	15	Seconds	Timeout for replica, while synchronizing old transactions. If the replica does not receive any messages from master during this time, it gives up and reconnects
axoniq.axondb.synchronize.stream.no-safepoint-timeout	10	Seconds	Timeout for replica. If the replica does not receive any safepoints from master during this time, it gives up and reconnects
axoniq.axondb.synchronize.stream.nr-of-new-permits	40000	Integer	Flow control, number of additional permits granted when threshold is reached
axoniq.axondb.file.bloom-index-fpp	0.03	Percentage	Bloom filter, allowable number of false positives. Higher value reduces bloom filter size, but decreases performance
axoniq.axondb.file.bloom-index-suffix	.bloom	String	Bloom filter, suffix for bloom filter files for events
axoniq.axondb.file.datafile-window-size	10	Integer	Number of datafiles to maintain in memory
axoniq.axondb.file.events-suffix	.events	String	Suffix for the events files
axoniq.axondb.file.index-cleanup-hack-enabled	true	Boolean	Option to forcefully remove index files from memory
axoniq.axondb.file.index-suffix	.index	String	Suffix for index files
axoniq.axondb.file.index-window-size	10	Integer	Number of indexes to keep in memory
axoniq.axondb.file.max-request-size	10MB	Integer	Maximum size of a single transaction, must be less than axoniq.file.max-segment-size
axoniq.axondb.file.max-segment-size	250MB	Integer	Size of each segment
axoniq.axondb.file.max-segments-for-id	10	Integer	While adding a new aggregate, check in at most this number of segments
axoniq.axondb.file.snapshot-bloom-index-suffix	.sbloom	String	Bloom filter, suffix for bloom filter files for snapshots

Property	Default value	Unit	Description
axoniq.axondb.file.snapshot-index-suffix	.sindex	String	Suffix for snapshot index files
axoniq.axondb.file.snapshot-suffix	.snapshots	String	Suffix for snapshot files
axoniq.axondb.file.storage		Path	Directory where all the AxonDB files are created. Directory will be created if not exists