Distributed Key-Value Database

Kyrillos Ntronov
kyrillos.ntronov@studio.unibo.it

March 2021

The aim of this project is to produce an eventually consistent distributed key-value database. The modern web-based application often require some kind of temporary or persistent memory to perform simple look-up operations, for example cookie storage, shopping carts, configurations etc... This project attempts to provide a solution by offering a rather simple CRUD REST interface for a cluster of heterogeneous nodes of a replicated key-value database that guarantee an eventual consistency and a degree of fault and partitioning tolerance. In regards of the CAP theorem[1], the database stands in the AP category, continuing to serve the client's requests even if the cluster is partially inconsistent. Furthermore, the database also offers a consistent writes replication via consensus protocols. Being both AP and consensus-based allows the proposed database to be flexible, solid and scalable.

1 Goal/Requirements

To produce the described system a p2p architecture is chosen, where each node can act without an explicit "master". In other words, each node is a self-contained consensus protocol server. Furthermore, each node should also act as a self-contained HTTP-server exposing a public API for the database interactions and a simple webpage-based GUI for the cluster monitoring. In case of the consensus' server fault, the server should still continue to act as an HTTP server serving only the read requests, until reset and its persistence synchronized.

Adding to the persistence requirements, the node should be able to store data both on filesystem and in-memory and the implementation of the persistence should remain transparent to the cluster.

In case of a node's fault or connectivity issues, a basic mechanism to synchronise missed writes should be employed.

After a preliminary analysis, the goals may be summarized as follows:

Primary Objectives:

- Studying the problem's domain and designing the system
- Studying and using modern and stable frameworks and libraries to implement the system
- Using CI pipelines and quality control methods (Test coverage, Static Code Anal-

ysis...)

- Creating a database server exposing HTTP CRUD interface
- Creating a consensus protocol server
- Creating a simple interface for GUI and node's information retrieval
- Using a build automation tool
- Although the project has only one developer, I will be using a git workflow (gitflow[2])
- Achieving a good test coverage across most of the classes

Secondary (Optional) Objectives:

- Creating a basic persistence synchronization mechanism to allow nodes to catch-up after a stoppage of service instead of manually assuring the data consistency.
- Produce a swagger documentation for the API.

Also, the following considerations has been accounted for:

- The GUI graphical UI and UX quality should not be given much consideration and it should remain more as a tool for demonstration, debugging and a proof-of-concept
- The database efficiency and features are not the primary focus, so more advanced concepts such as indexing, storage efficiency, complex queries (for example filters, projections, aggregations...) will not be given priority, although it is expected to leave the code easily extendable to accommodate such improvements.

After an initial analysis of the problem, the following backbone technologies and frameworks have been selected:

- Atomix [3] - *an event-driven framework for coordinating fault-tolerant distributed systems using a variety of proven distributed systems protocols.* In particular, only the RAFT consensus protocol feature will be used to create the cluster and establish connectivity between the nodes.
- SpringBoot [4] - A famous all-in-one framework for webservice development, it will be used for its IoC (Inversion of Control) and DI (Dependency Injection) capabilities, its HTTP-webservice and REST capabilities and other utilities such as REST communication template.

- Thymeleaf - A template parser easily integrated with Spring to simplify the GUI development
- Log4J2 - For logging
- Mockito [5] - A testing framework that allows mocking (stubbing) of methods and classes to improve unit tests quality and ease of development
- Maven [6] - A build-automation tool widely used in java ecosystem

The final product is a simple .jar to be launched configured via arguments (or configured via java's .properties).

1.1 Scenarios

A user should be able to interact with any node to access database via its HTTP REST API. A user may perform the following CRUD actions:

Read: Provided a list of keys, the user should expect in return the stored records if such a key exists in the database.

Create/Update: For the convenience of use, the creation and the update have been mapped to a single endpoint where the user provides a list of records (a key-value pair, where the value is a set of named attributes)

Delete: Provided a list of keys, the user should expect in return a positive response status, indicating that the deletion of any records matched by the provided keys has been performed.

A user should be able to interact with node's cluster layer via REST HTTP API.

A user may perform the following actions:

Stop a running node (i.e. stop consensus features and let node enter in read-only mode)

Start a stopped node

Retrieve information on the cluster status

Retrive information on the database state

A user may also want to use the monitoring dashboard, in order to do this the user may simply access the node's HTTP webpage at "/dashboard".

In the dashboard, the user can expect to perform the following operations:

- Provide a list of nodes' addresses to monitor (the dashboard does no assumptions in regars of node's relationship to the cluster)
- Allow stopping or restarting the node
- View node's internal database state (i.e. all of the record contained, useful to assert the correct functioning of the writes propagation)

1.2 Self-assessment policy

To asses the quality and the correctness of the code, a good coverage of meaningful unit tests is required. In addition to the unit tests, it is also a good practice to test the distributed system as a whole by providing an integration test simulating a cluster of nodes, and assert the correctness of their interactions. The coverage should also be considered in the CI pipeline to avoid allowing failing tests in production. In addition to coverage, it is also a good idea to use a static code analyzer (such as SonarLint[7]) to avoid code smells and trivial bugs.

To asses the effectiveness of the project, besides gathering users' feedback after a reasonable amount of trial usage, some local runtime tests also should be performed. The tests should include all of the basic use cases and as many edge cases as possible.

2 Requirements Analysis

The project aims to satisfy the following functional requirements:

- Be able to persist key-value records.

  The database module of a single node can be launched in either the filesystem or the in-memory persistence mode.

  The filesystem persistence should store records directly in the host's filesystem as .dkv files. A configurable base directory path should be provided as more than one node can run on a single host.

The in-memory persistence mode stores records directly in volatile memory.

The nodes with different persistence modes must be interchangeable and transparent in the handling of the intra-cluster communications.

A record consists of a unique string key that serves as the record's identifier and a set of named attributes with string values.

- Offer a simple HTTP REST interface for CRUD operations on the database records.

  The interface must provide the following operations that consume the same json object that simply consists of a list of key-attributes pairs:

  Read - produces a list of records matching the requested list of keys, may be empty if no matches are found.

  Save - acts as both save and update by creating a new record, overwriting the old one if present.

  Delete - deletes a list of records matching the requested list of keys if present in the database.

- Allow singular nodes to form a cluster for the consistent writes replication.

  To guarantee a write's (update or delete) propagation across the cluster, a consensus protocol must be used.

  If a node fails in any way (faults, network partitioning...), instead of shutting down it should continue to serve read requests and reject writes. The node is then expected to be manually reset after the problem has been identified and corrected.

- Implement a simple automatic database synchronization mechanism to catch-up persistence after a node comes back online after a failure or a partitioning.

  While a consensus protocol can offer a consistent writes propagation, considering the volatile nature of an in-memory database and the missed updates on a stopped or an unreachable node, a mechanism should be implemented to automatically synchronize a node's persistence state on startup/cluster join via indicating a fallback cluster member that, by the user's discretion, should act as the authoritative copy of the data.

- Produce a simple dashboard to allow basic nodes monitoring and manipulation.

To avoid producing separate artifacts with an independent lifetime, any node in a cluster can act as a simple HTML dashboard.

The dashboard isn't a "production ready" artefact and should be used only for debugging and as a proof-of-concept.

The dashboard may begin to "listen" to a number of node by registering their REST interface addresses.

The dashboard must show the node's cluster visibility, database contents and current status (Running, Stopped....)

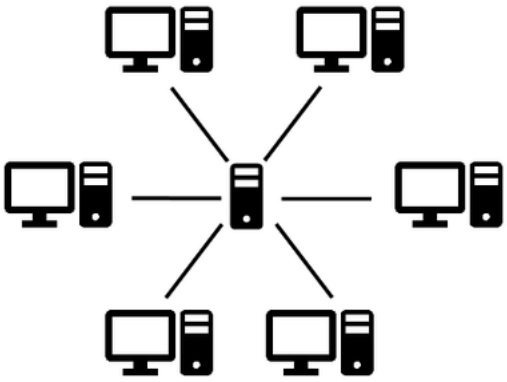
The project also aims to satisfy the following non-functional requirements:

- The project should be easily deployable.
- The project should be reasonably easy to scale.

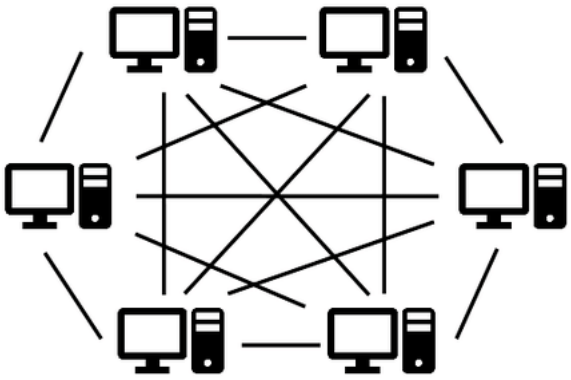
3 Design

3.1 Cluster Design

In the proposed architecture a cluster is a collection of masterless p2p nodes rather than a client-server model:



Client-Server network



P2P network

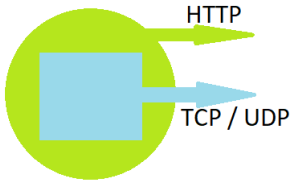
- In Client-Server Networks clients and server are differentiated and the management is delegated to the server.

- In Peer-to-Peer Networks clients and servers are not differentiated and every node is itself client and server.

3.2 Node Design

The choice is motivated by the ease of scalability and fault tolerance. By eliminating the single-point-of-failure, such qualities may be achieved with more ease.

A node has a dual nature of both being a peer in a cluster and an HTTP server. Thus is was decided to develop a node as a self-contained REST webservice with cluster capability rather than the other way around, since the node can continue to operate in read-only mode even if consensus layer fails.



3.3 HTTP Server Design

To implement HTTP communications, two architectures may be considered:

- REST - Representational state transfer model. A RESTful service exposes its web resources in a textual representation and allow them to be read and modified with a stateless protocol such as HTTP and a predefined set of operations.

- SOAP - Simple Object Access Protocol. In SOAP the WSDL file provides the client

with the necessary information which can be used to understand what services the web service can offer.

It was decided to utilize REST rather than SOAP for a number of reasons:

- REST is efficient (particularly coupled with json)
- REST is fast (no extensive processing required)
- REST allows for easier and faster client implementations

3.4 Consensus Protocol Choice and Implementation

Distributed consensus can be summarised as reaching agreement among a collection of machines (cluster) cooperating to solve a problem. Consensus algorithms have become essential tools for fault-tolerance in the distributed systems and enhance the resilience by eliminating the single points of failure.

Two consensus protocols were considered: PAXOS [8] and RAFT [9], both are grounded in a similar set of core principles. Ultimately, RAFT was chosen as it is designed to be easy to understand and it's equivalent to Paxos in fault-tolerance and performance.

Raft offers a generic way to distribute a state machine across a cluster of computing systems (nodes). The protocol achieves consensus via an elected leader, so is not a Byzantine fault tolerant algorithm. The leader regularly sends a heartbeat message the followers. If a follower doesn't receive the heartbeat message in a fixed amount of time, it becomes a candidate and starts a new *term*. The term starts a new leader election and upon reaching the majority of votes one of the nodes becomes the new leader. In case of a split vote or an error, a new election is held.

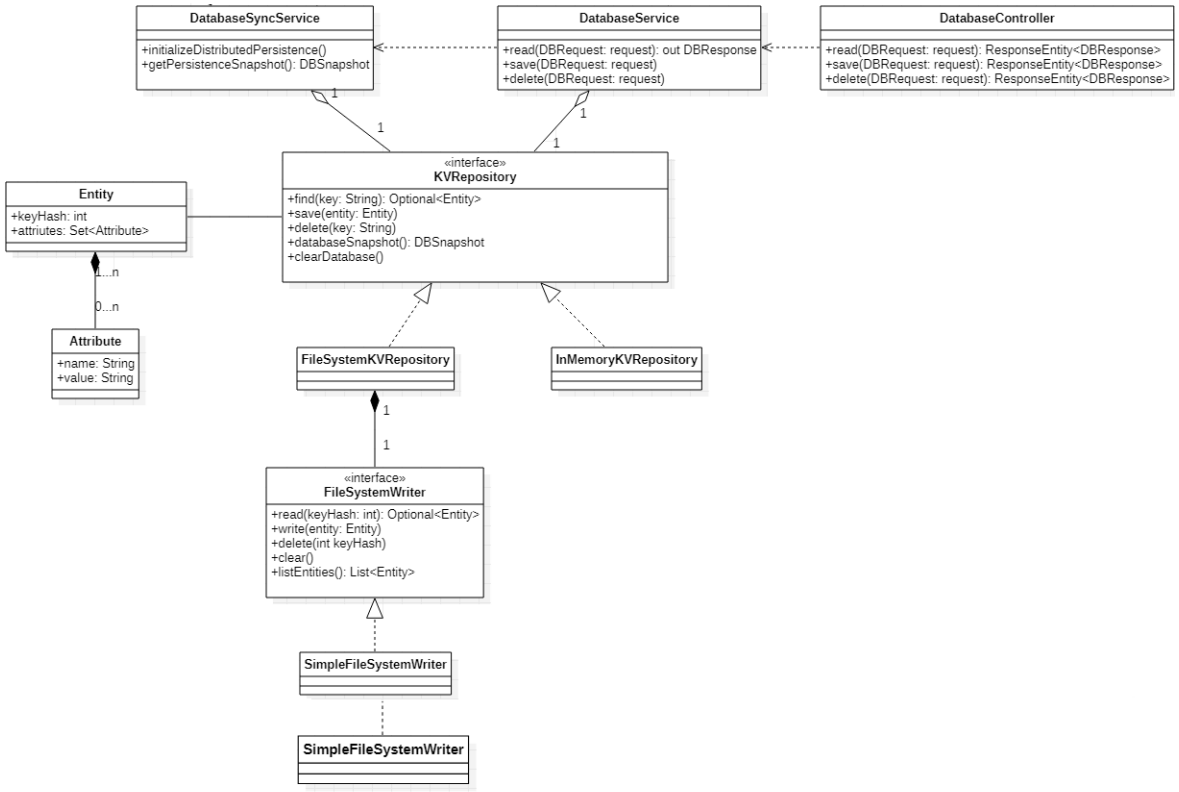
The leader is responsible for the **log replication**. The leader accepts a client request with a command to be be performed and append a new entry in its log. The leader then sends the same request to the other members of the cluster, upon receiving the confirmation of reception from the majority of members, the entry is considered to be *committed*. A committed entry is applied at the local node, and eventually in the followers once they learn that the entry has been committed.

In the case of leader crash, the logs can be left inconsistent. The new leader will then handle inconsistency by forcing the followers to duplicate its own log. The leader will compare its log with the logs from the followers, find the last synchronized entry, then

delete all the entries coming after and replace them with its own log entries.

3.5 Structure

The following UML class diagrams are rather complete, but the usage of some of the trivial "container" classes (e.g. DTOs) and some of the more specialized dependency/library classes is omitted to improve readability.



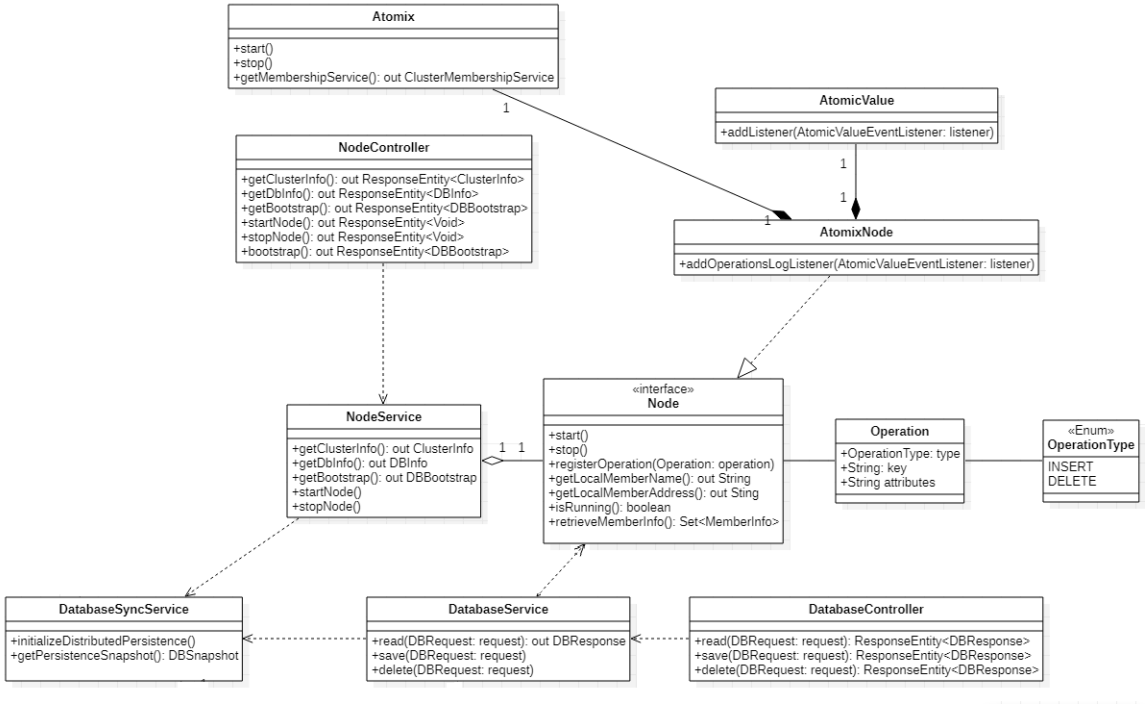
The persistence is organized around the **KVRepository** Interface. The interface allows to support further database persistence mode and neatly abstracts the data layer. Currently the project supports only the filesystem and the in-memory modes.

The filesystem repository uses a **FileSystemWriter** interface that mirrors the repository functions with a singular implementation. The reasoning behind abstracting the writing process is to support further optimizations and alternatives, for example indexing support.

The data passed between the webservice and the persistence layer is asbtracted with the **Entity** and the **Attribute** classes. A singular key holds reference to multiple named attributes.

The webservice layer is structured according to the controller-service-model layered architecture, with controllers handling the HTTP mappings and responses, while services act as the middle layer between persistence or application logic and the controllers.

Two services make use of the **KVRepository**, **DatabaseService** and **DatabaseSyncService**. The **DatabaseService** is responsible for the CRUD operations handling and warpping raw persistence data with DTOs. The **DatabaseService** has **DatabaseSyncService** as a dependency to initialize database synchronization logic via attaching the listeners and performing the database synchronization on startup. **DatabaseSyncService** is also used to form the database dump object to be passed for the synchronization process.



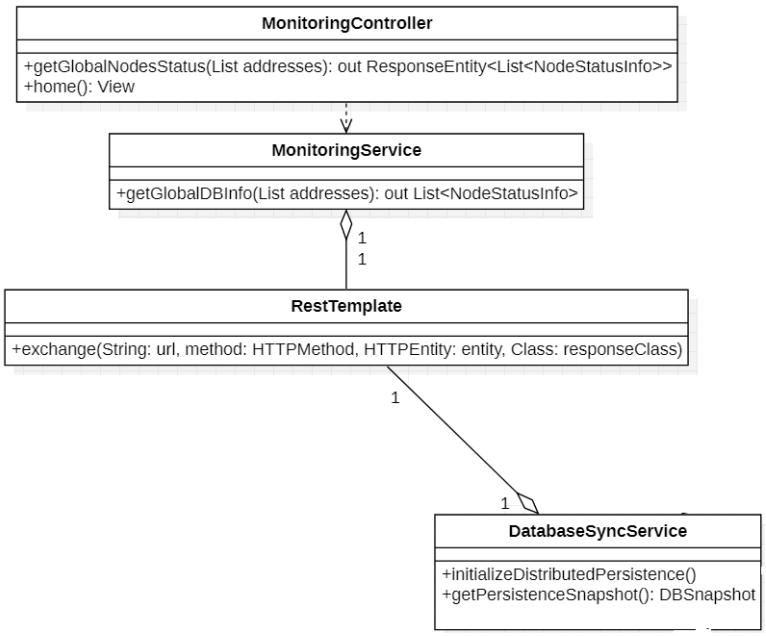
The distributed layer of the application is build separately from the persistence layer.

The center of this class group is the **Node** interface. The usage of the interaface is intended to facilitate the change of the cluster logic if needed, or allow support for multiple types of consensus and cluster membership management.

The Operations in the database are replicated via the **Operation** class, representing a singular write to be propagated among the nodes. A Node that performs a write registers an operation to propagate to the rest of the cluster and the rest of the nodes receive the operation and replicate it.

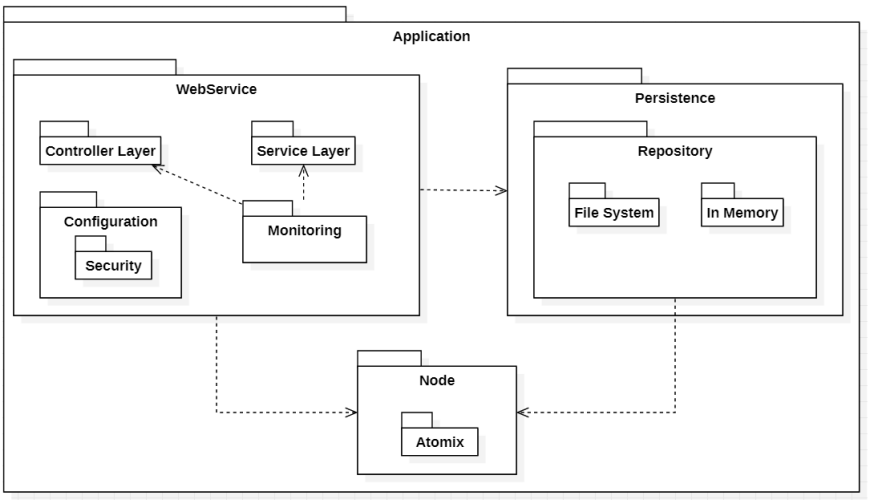
The **AtomixNode** implementation uses the **AtomicValue** structure (provided by the Atmoxi framework) that guarantees replication on the other nodes (and the consequent event caught by the listeners) via the RAFT consensus. The framework incapsulates almost all of its functionality in the **Atomix** class, that provides an api to interact with the distributed layer.

The node controller-service group is used to both provide the runtime information on node's status in the cluster and interact with the distributed functionality, i.e. stopping and starting the node.



The dashboard is rather simple, a controller renders a view and uses the underlying service to perform the HTTP calls to the request addresses to receive the cluster status as seen by the called node.

To perform HTTP calls, **RestTemplate** a SpringBoot class is used, in particular the *exchange* method that marshals the received json into a desired class. It also allows to attach headers to the request, for example for the BASIC authentication.

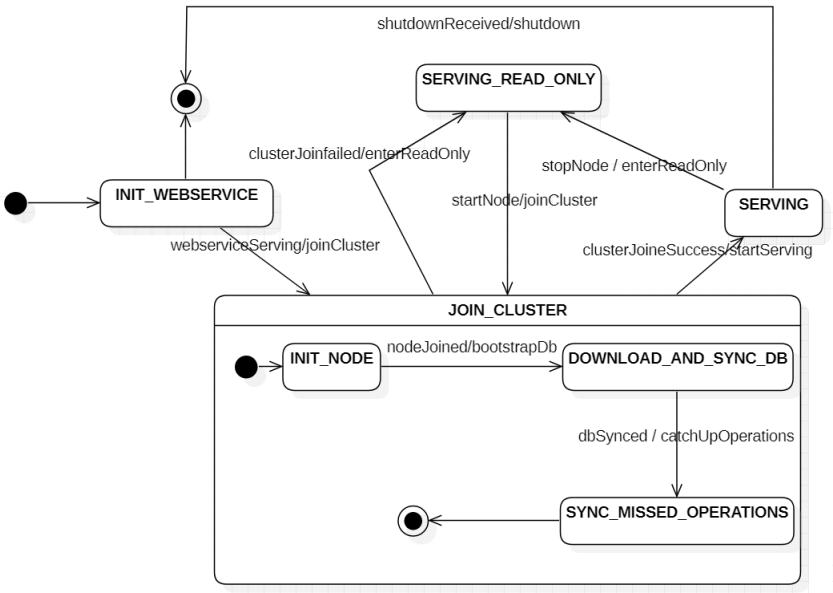


The Application is structured in three "blocks":

- The Webservice package containing controllers, services, application configurations (in particular the security configuration) and the monitoring (dashboard) package.
- The Persistence package is related to the handling and the performing of the CRUD database operations, the repository package is is further subdivided in in-memory and filesystem related packages
- The Node package simply contains all the logic related to the distributed nature of the node.

3.6 Behaviour

The node's startup follows predetermined steps and can be seen as a state-machine:



1. The first essential step is to initialize the webservice and the Spring's IoC container.

2. If the first step fails the node cannot function in any way and terminates.

3. The second step is to join the cluster and configure the writes replcation.

4. It is a compound process beginning with node's initialization via the atomix framework. then the database synchronization takes place, if successful, the nodes cathes up with the missed writes (if any exist).

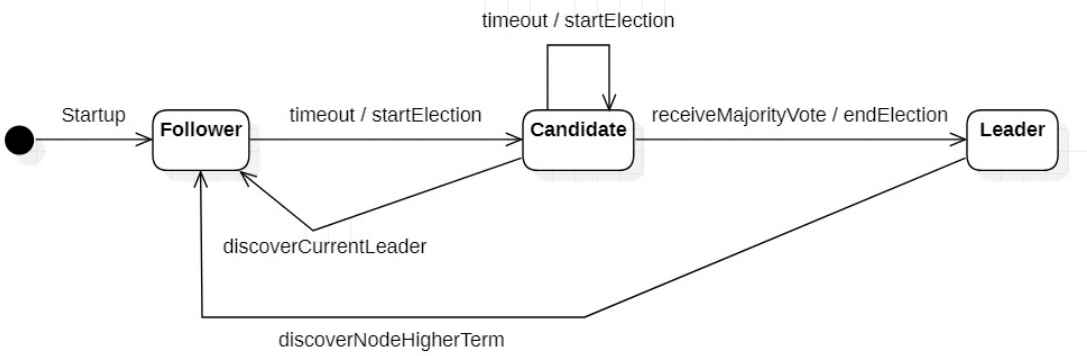
5. If at any of these steps a failure is encountered, the node enters the read-only mode where only read operations are allowed.

6. If all of these steps are successful, the node is operation and begins serving. At this point it could either be shutdown (via external means) or be stopped and enter read-only mode.

7. If a node is found in read-only mode, the user may attempt to restart it by performing the cluster join process.

Please note that the synchronization step is recommended, but optional, a node can join a cluster without synchronizing its database as per configuration.

The following diagram represents the basic RAFT Node's statechart:

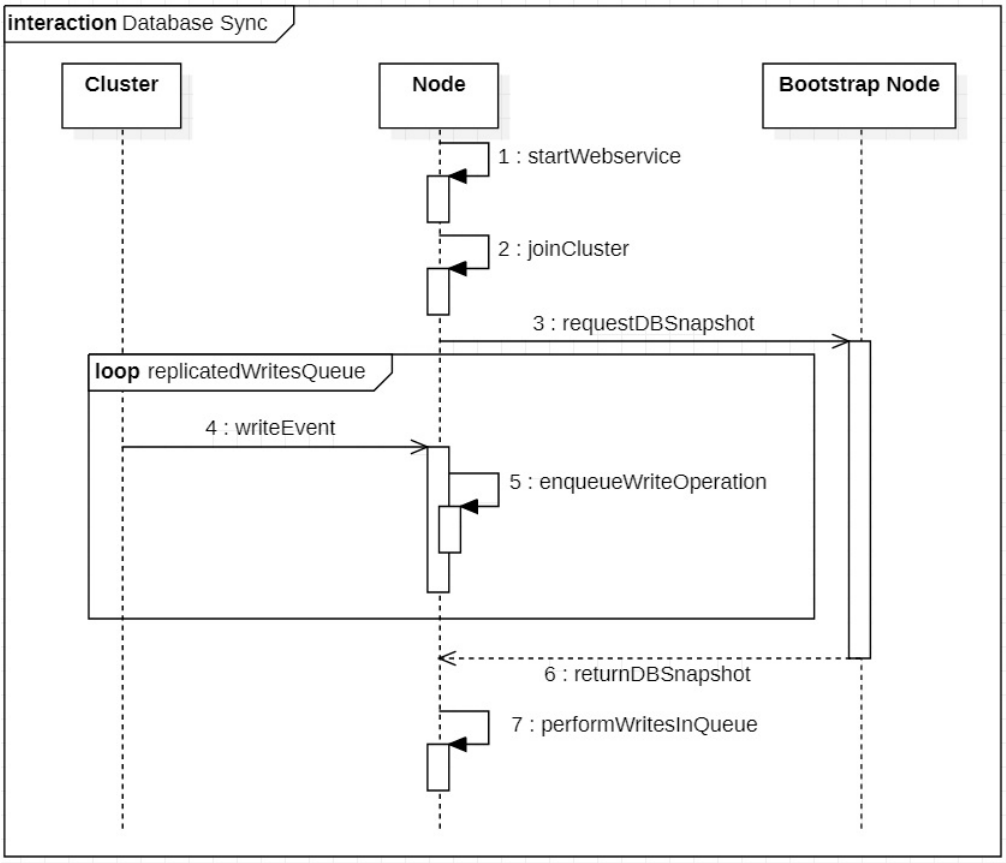


Raft protocol achieves consensus via leader election. A node starts as a follower, whenever the current leader becomes unavailable, the node begins an election. During the election the node becomes a candidate. The node that receives the majority vote becomes a leader responsible for log replication to the followers.

3.7 Interaction

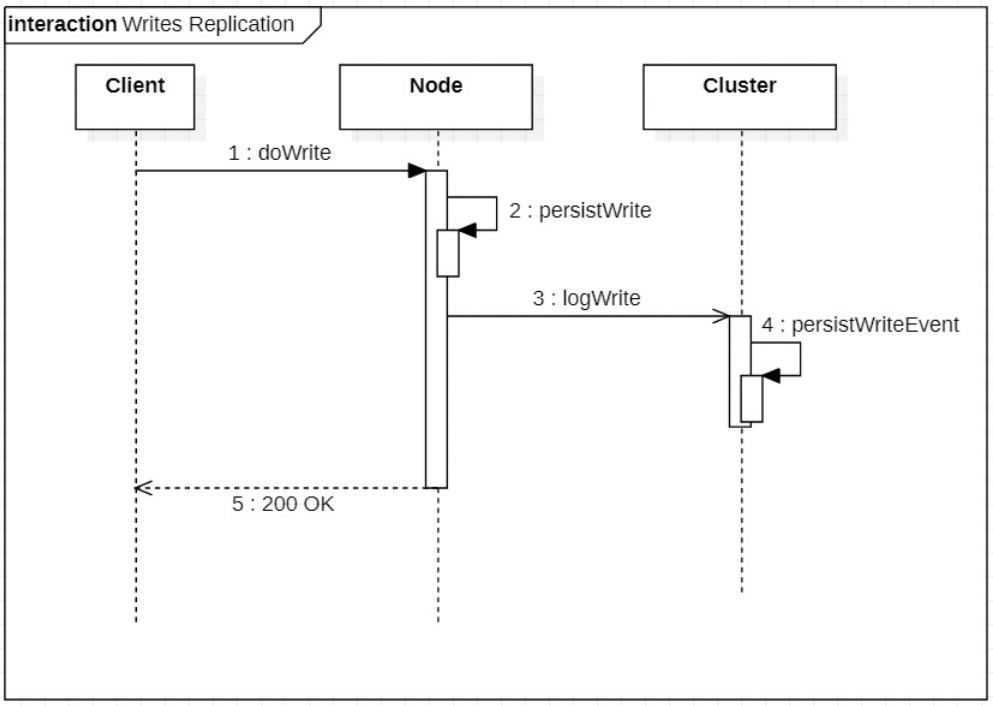
To make the following sequence diagrams more readable, a distinction between a particular node and the cluster as a whole was made, however it should be noted that any node considered is still a cluster member.

The following sequence describes the database synchronization process:



It should be noted that while the database is being synchronized (a potentially time-consuming task), the writes are not being performed, but stored in a queue. Once the synchronization is completed, the queue is consumed and the operations are performed.

The next sequence describes the write replication process:



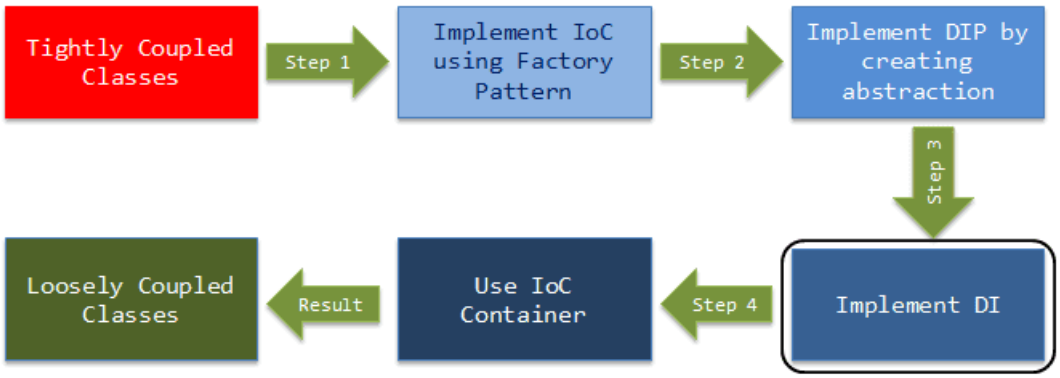
It should be noted that any IO fault results in the node entering the read-only mode.

4 Implementation Details

Frameworks

The project uses two frameworks, SpringBoot for HTTP webservice capabilities and dependency injection and Atomix for RAFT consensus.

Dependency Injection is a design pattern used to implement Invesrion of Control (IoC). It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways.



Spring handles the creation of an IoC container that allows annotation-driven injection into the desired classes.

Atomix is an event-driven framework for coordinating fault-tolerant distributed systems. Atomix provides a series of building blocks for distributes systems, such as cluster management, messaging, partitioning... Being an event-driven framework, the operations in the nodes are performed via event callbacks on the distributed atomic object (which is maintained consistent via consensus protocol) that holds the information about the latest write operation. Atomix guarantees that the the log events for a distributed primitive will be processed in the same order as the changes were received. However this framework presents a limitation in the fact that the events are *not* replicated for an unavailable node (shutdown or partitioned) due to the distributed primitives and their events being abstractions based on replicated state machines and due to technical limitations (in particular the log compaction). Upon re-joining the cluster it will not receive the missed events. Hence the need to synchronise node's data on startup / rejoin, especially for the in-memory databases.

Atomix RAFT cluster requires a node to be configured with cluster members' names and addresses to correctly form the cluster. However it also supports member's address discovery via multicast discovery, so the only thing to declare is the nodes' names.

Testing with JUnit was done using **Mockito** mocking/stubbing framework. This framework allows mocking an object and its behavior. Doing this allows to unit test code that uses "problematic" components, for example by mocking **RestTemplate** you can simulate HTTP responses with values or throw desired exceptions. Furthermore, it allows verifying interactions with mocks, making testing interactions and messages passed between the objects easy.

Swagger Doc

For a complete list of the endpoints and the API's features, please refer to the swagger doc by calling the following endpoint:

```
/v2/api-docs
```

5 Self-assessment / Validation

5.1 Objectives Assessment

An assessment of the stated objectives has been done:

Primary Objectives:

- Studying the problem's domain and designing the system

  A number of similar projects and projects have been analysed and a simplified solution was designed

- Studying and using modern and stable frameworks and libraries to implement the system

  SpringBoot is a mature and very popular HTTP webservice framework and Atomix is a solid and highly configurable framework for distributed systems. Both have been thoroughly studied by reading the documentation and by examples.

- Using CI pipelines and quality control methods (Test coverage, Static Code Analysis...)

  A basic maven testing pipeline has been created on the protected branches.

- Creating a database server exposing HTTP CRUD interface

  A webservice was created that is capable of the CRUD operations

- Creating a consensus protocol server

A consensus protocol server has been embedded into the webservice via Atomix.

- Creating a simple interface for GUI and node's information retrieval

  A simple dashboard has been created to interact with the nodes and check on their status.

- Using a build automation tool

  Maven build automation tool was used.

- Although the project has only one developer, I will be using a git workflow (gitflow) Gitflow workflow has been used, however a couple "hotfix" commits on the develop have been allowed as a part of the development progress.

- Comprehensive and Meaningful Tests

  The project is well-covered with meaningful tests.

- Produce a swagger documentation for the API. The automatic swagger documentation generation was integrated with the project at the */v2/api-docs* endpoint

Secondary (Optional) Objectives:

- Creating a basic persistence synchronization mechanism to allow nodes to catch-up after a stoppage of service instead of manually assuring data correctness

  A basic synchronization mechanism has been implemented.

- Achieving a good test coverage across most of the classes

  The project is well-covered by the tests (both unit and system), in particular on on all of the critical parts.

5.2 Test Coverage

The development was done according to the test-driven principles. It wasn't possible to cover the code to completion due to the time constraints, however a number of Unit and System tests on the most critical components were written and performed:

Controllers

The controllers were tested via SpringBoot's convenient *@WebMvcTest*. It allows mocking the MVC layer and testing calls to controller via http request builders and matchers. The beans injected into the controller can also be mocked to simulate different system states and responses.

Element	Class, %	Method, %	Line, %
DatabaseController	100% (1/1)	100% (8/8)	96% (26/27)
MonitoringController	100% (1/1)	100% (3/3)	100% (8/8)
NodeController	100% (1/1)	100% (7/7)	100% (25/25)

All of the possibile calls and responses were covered by the test and almost every possible path.

Services

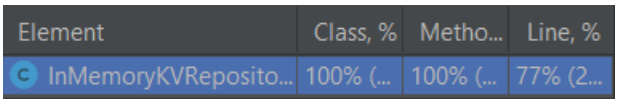
The services were tested independently by mocking the persistence layer and other dependencies (for example consensus layer)

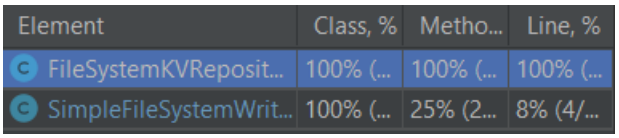
Element	Class, %	Method, %	Line, %
DatabaseService	100% (1/1)	100% (5/5)	76% (40/52)
DatabaseSyncService	100% (2/2)	100% (14/14)	81% (80/98)
MonitoringService	100% (2/2)	100% (6/6)	100% (34/34)
NodeService	100% (1/1)	71% (5/7)	57% (24/42)

The Node service doesn't have a good coverage because some of the trivial methods (simple dto creation) were not covered with tests and the node interaction methods are already tested in the system test.

Persistence

The persistence layer was covered with a single integration *@ParameterizedTest* that runs a suite of operations covering all of the basic functionality with a mock in-memory database (in case of the filesystem test, the file writer is mocked to write to the in-memory database).

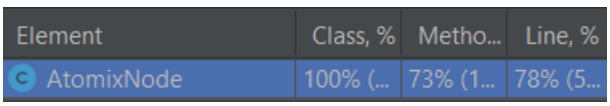




Some of the most trivial exception cases and the file writer class (due to requiring writes in file system) weren't covered with tests.

Node Test

The node functionality with a mocked distributed system was thoroughly tested.



Distributed System Tests

Finally the system was tested with a System test. The test launches three instances of the actual application running on localhost. Two tests are performed:

- Database Test

  The whole database CRUD functionality is tested by performing write operations on one node via HTTP calls and checking the correctness of the replicated operations on the other nodes. The test is parametrized to run three times, each time using random data and selecting a different combination of the nodes doing writes and reads.

  All of the CRUD operation are tested.

- Database Sync Test

  This tests checks the correctness of the database synchronization process by disabling a node, performing writes on the other nodes, then restarting the stopped node to assert that upon rejoining the cluster, the database state is consistent with the rest of the cluster. In addition, during the bootstrap process, some writes are performed to assert that the node will still receive them. The test is parametrized to be repeated 3 times on different data.

Total test coverage statistics:

```
91% class coverage, 79% method coverage, 74% line coverage
```

However it should be noted that most critical code is covered, while some of the trivial code (such as getters/setters of dtos or exception constructors) isn't tested due to time constraints.

In addition to tests, another tool for QA was used. The **SonarLint** plugin was used to assure the code quality, the plugin intercepts possible bugs, code smells, suggests refactor ideas and in general provides other quality-of-life features.

6 Deployment Instructions

To prepare the environment and compile the project, the following steps must be taken:

1. Install and configure Java 14 on your machine. Make sure java –version shows the correct version.

2. Install and configure maven 3.x.x. Make sure mvn –version shows the correct version. If you wish to use maven bundled with an IDE, you may skip this step.

3. Clone the gitlab repo.

4. The project uses Lombok to streamline the boilerplate code generation. An IDE may require a plugin to function correctly. Please refer to `https://projectlombok.org/setup/overview`.

5. Point the terminal at the poject's root (where the pom.xml is located) and compile the code with

   ```
   mvn clean install
   ```

   . If running tests, please make sure to have the ports 8080-8082, 8090-8092 available and the folder **/dbkv/** writable for the system tests. To skip the tests launch with the

   ```
   -Dmaven.test.skip=true
   ```

   property.

6. Copy the compiled .jar in the /target folder

7. Launch the compiled jar with java -jar... and all the desired configuration parameters (see below)

The application takes multiple java parameters (–foo=bar or via application.properties on the classpath) to configure database and consensus node launch:

- 'server.port' - HTTP REST server port for API interactions.

- 'db.mode' - Persistence mode, "fs" for file system persistence, "mem" for in-memory persistence.

- ‘db.base‘ - Base folder for filesystem persistence storage and consensus protocol data.

- ‘node.address‘ - Node’s cluster communication address (host:port) IMPORTANT the cluster port must not be same as the HTTP port.

- ‘node.name‘ - Node’s name in cluster, must be unique

- ‘node.members‘ - A list of the nodes in the cluster to be discovered via multicast. Should be a list of names separated by a comma (member1,member2,member3...). Must also include the node’s own name.

- ‘node.bootstrap.address‘ [Optional] - Optional HTTP api address of a node used to populate/sync database on the startup.

- ‘node.partitions‘ [Optional] - Optional number of partitions for distributed data. Defaults to 1.

- ‘node.startup.timeout‘ [Optional] - Optional timeout (in seconds) for node’s cluster startup. If startup fails, node will enter the read-only mode. Defaults to 60 seconds.

- ‘node.bootstrap.attempts‘ [Optional] - Optional number of maximum attempts a node can make to bootstrap via provided node address. Defaults to 5.

- ‘security.enabled‘[Optional] - Optional flag to enable basic auth. **IMPORTANT doing this will break CORS support** (some of the **dashboard functionality** will not work without CORS). Defaults to false.

- ‘security.username‘ [Optional] - Optional BASIC Authentication username. Defaults to ’admin’

- ‘security.password‘ [Optional] - Optional BASIC Authentication password. Defaults to ’admin’

The following sample script launches 5 nodes, 3 in in-memory mode, 2 in filesystem mode, with one of the filesystem nodes being used for sync on startup. Authentication defaults to *admin* username and password.

```
java -jar ./distributed-kv-1.3.jar --server.port=8080 --db.mode=mem
↪  --db.base=./node1 --node.bootstrap.address=localhost:8084
↪  --node.address=localhost:8090 --node.name=member1
↪  --node.members=member1,member2,member3,member4,member5

java -jar ./distributed-kv-1.3.jar --server.port=8081 --db.mode=mem
↪  --db.base=./node2 --node.bootstrap.address=localhost:8084
↪  --node.address=localhost:8091 --node.name=member2
↪  --node.members=member1,member2,member3,member4,member5

java -jar ./distributed-kv-1.3.jar --server.port=8082 --db.mode=mem
↪  --db.base=./node3 --node.bootstrap.address=localhost:8084
↪  --node.address=localhost:8092 --node.name=member3
↪  --node.members=member1,member2,member3,member4,member5


java -jar ./distributed-kv-1.3.jar --server.port=8083 --db.mode=fs
↪  --db.base=./node4 --node.bootstrap.address=localhost:8084
↪  --node.address=localhost:8093 --node.name=member4
↪  --node.members=member1,member2,member3,member4,member5

java -jar ./distributed-kv-1.3.jar --server.port=8084 --db.mode=fs
↪  --db.base=./node5 --node.address=localhost:8094 --node.name=member5
↪  --node.members=member1,member2,member3,member4,member5
```

The node will begin to output logs and is to be considered launched after the *Completed initialization* log. If the node fails to launch the cluster layer, it will still procede to serve in read-only mode, refer to logs for the startup information. The node's HTTP capabilities run on an embedded Tomcat.

7 Usage Examples

Dashboard

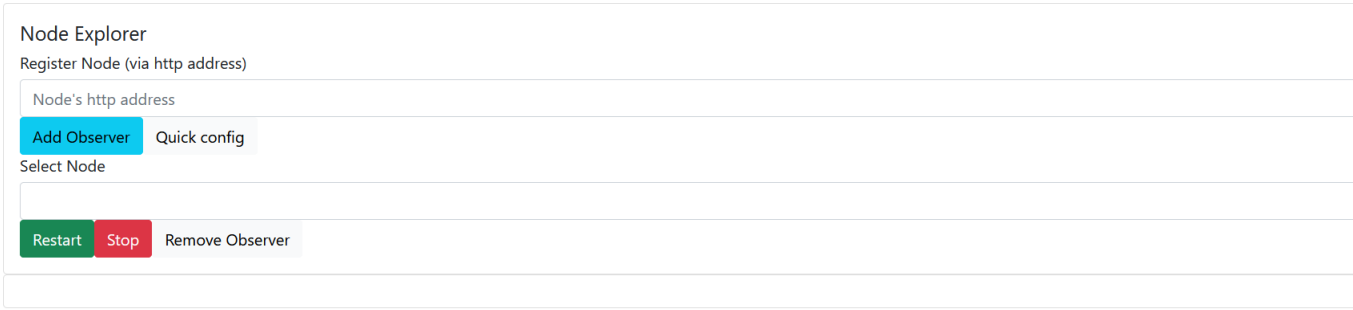
Once a node is in serving mode, its dashboard can be access via a browser. In order to do so, simply connect on the node's **/dashboard/** path. Please note that it is best to avoid using the authorization feature (security.enabled=false) with the dashboard when

running multiple nodes on localhost as some incompatibilities between CORS support and Spring Security were encountered.

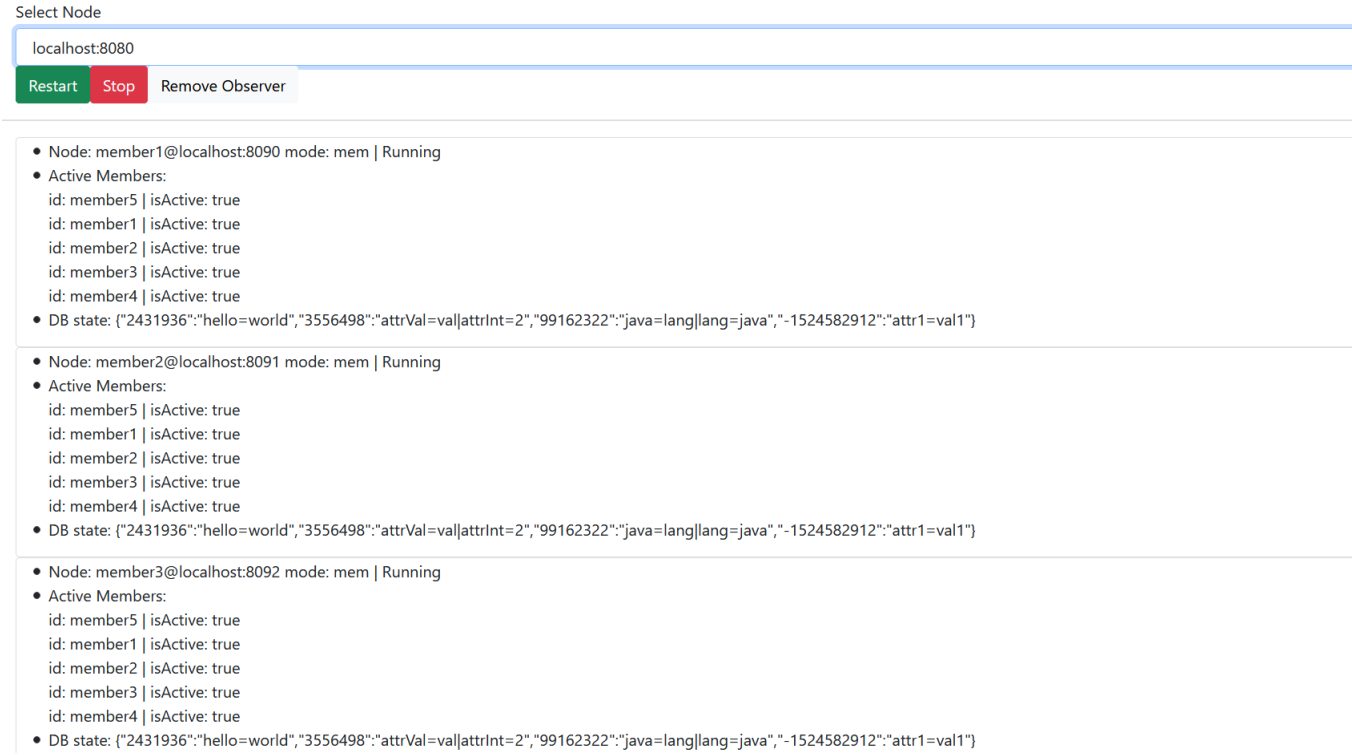
For example:

```
http://localhost:8080/dashboard/
```

Will open the dashboard:



The upper form section allows to begin listening to a node by providing a HTTP address. The quick config button simply adds five nodes, from localhost:8080 to localhost:8084 as per the sample script.



For example here the nodes have synchronized their database with one of the filesystem mode nodes that already had some entries.

The dashboard will listen to the provided nodes with a 2.5s interval.

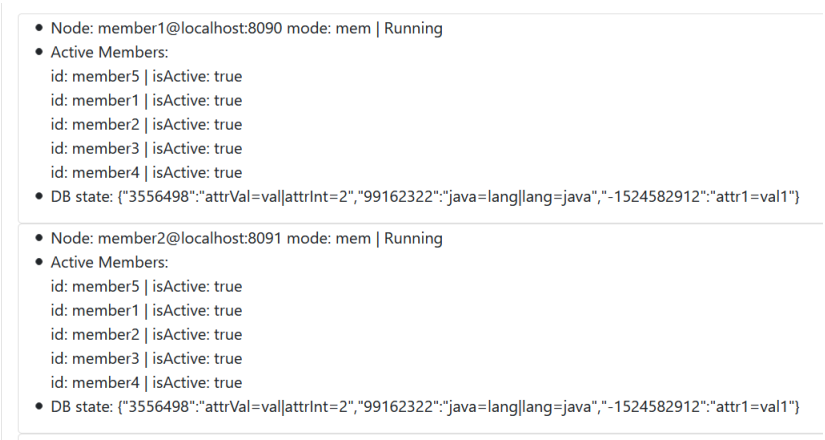
For each node, it's ID and cluster address will be provided, as well as a list of all of the seen nodes. Finally, the entire database snapshot will be displayed to easily verify the database consistency.

Selecting a node will allow to press the buttons to stop or start it's cluster capabilities via the dashboard.

For example, you may stop a node. The stopped node will not receive the operations propagated on the cluster.

Note how the node's cluster module is shutdown and it is both removed from the cluster's active nodes and has no cluster visibility.

If you restart the node, it will synchronize its database.

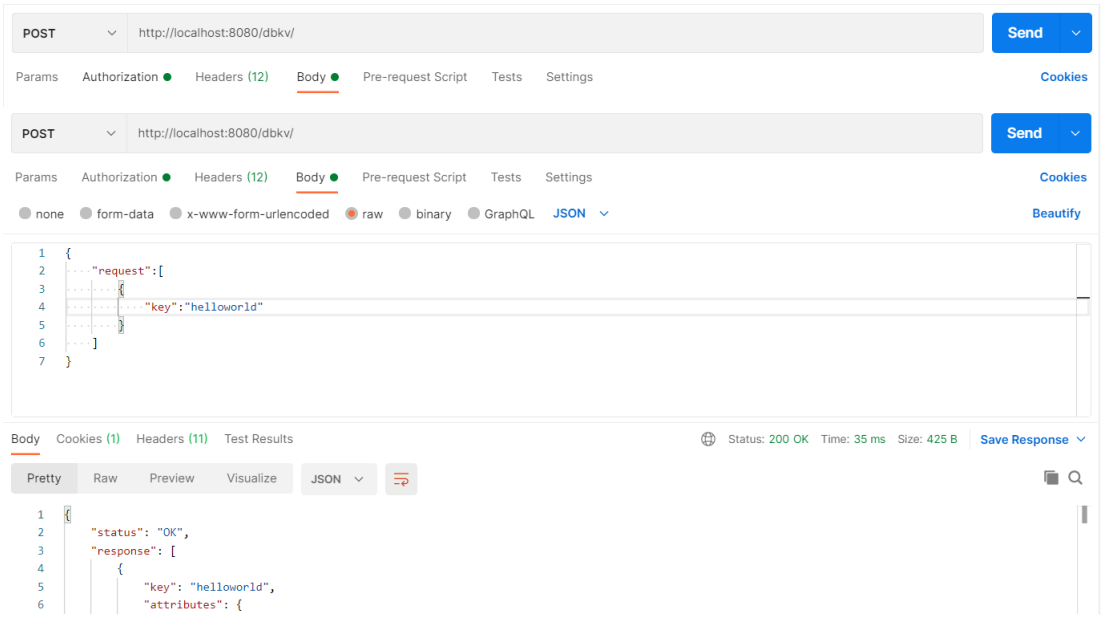


Once the node is running again, it will receive the replicated writes.

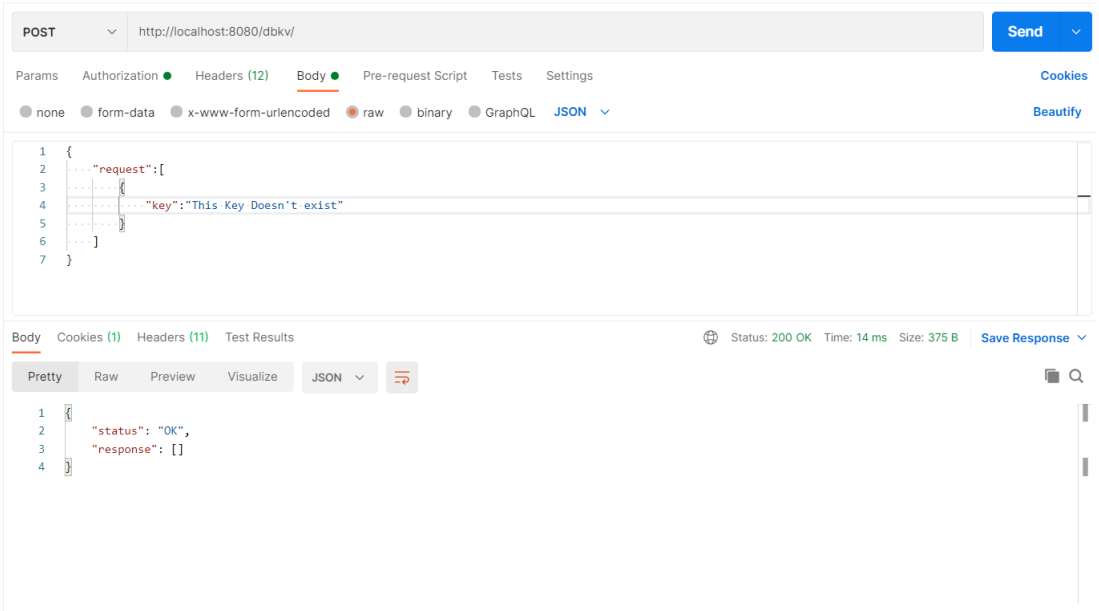
Database API

The database api's functionality will be demonstrated by using Postman.

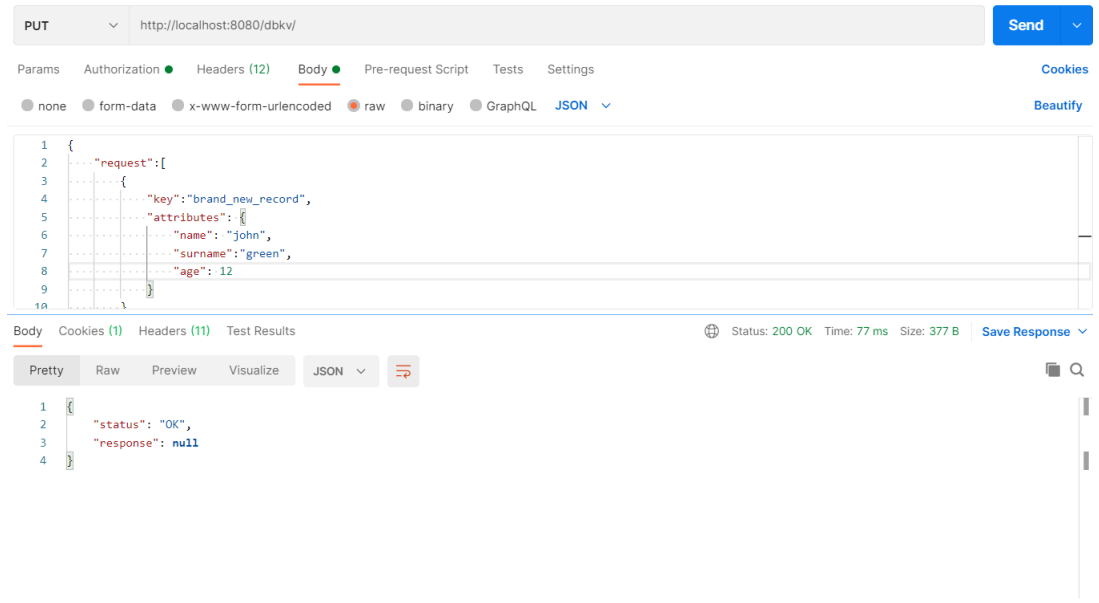
To retrieve a record, it is enough to call any of the nodes with a **POST** request with the list of keys in the body as such:



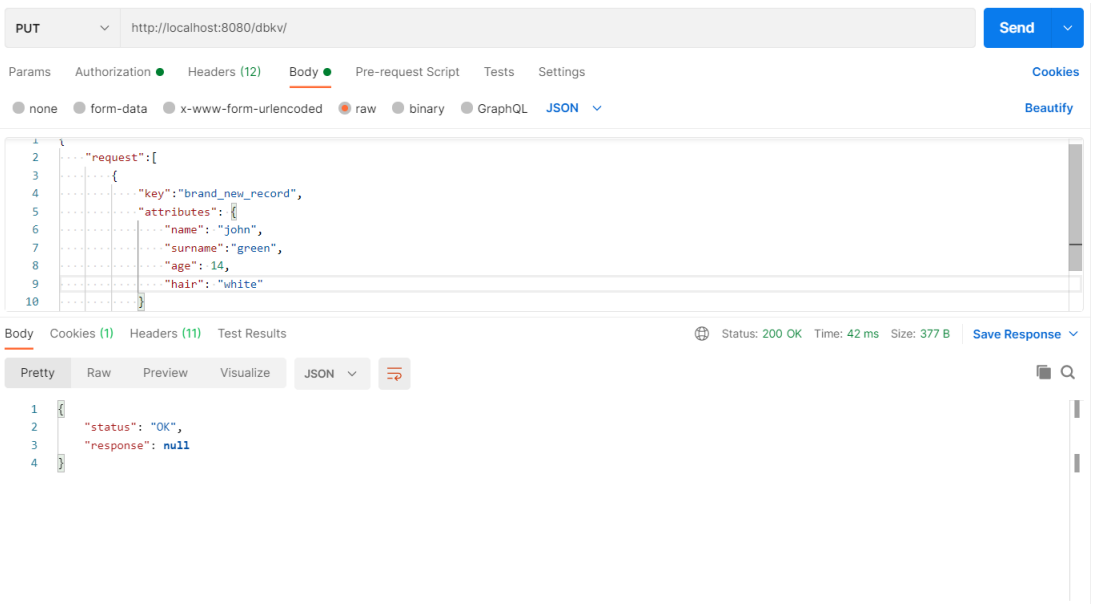
If no record with the provided key is found, the response is simply an empty list:



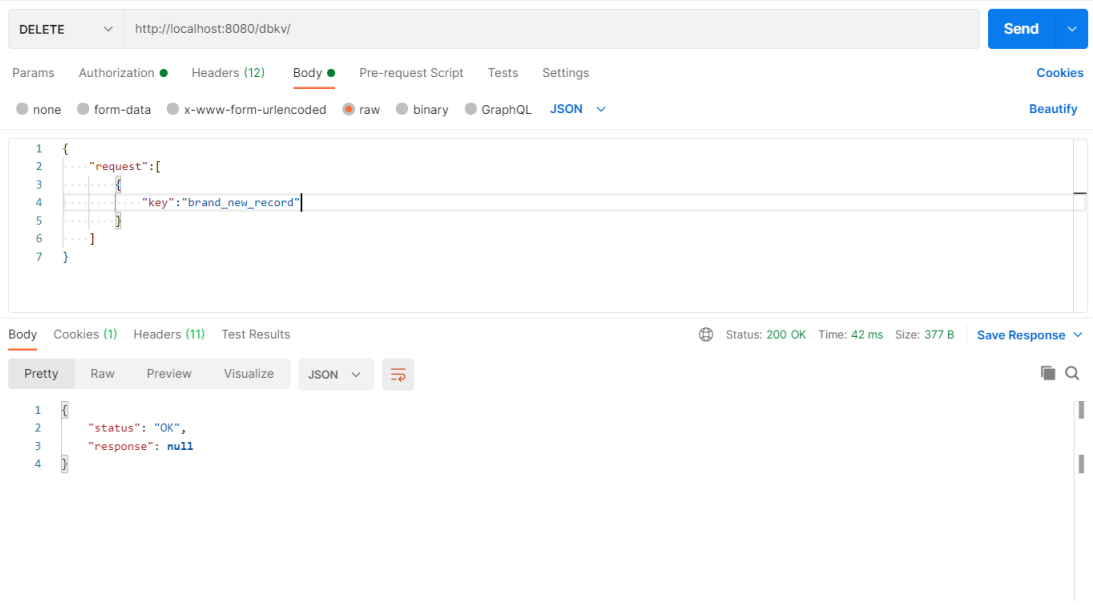
To add one or more records, it is enough to use the same template with a **PUT** request, however this time one or more attributes can be added to the entity



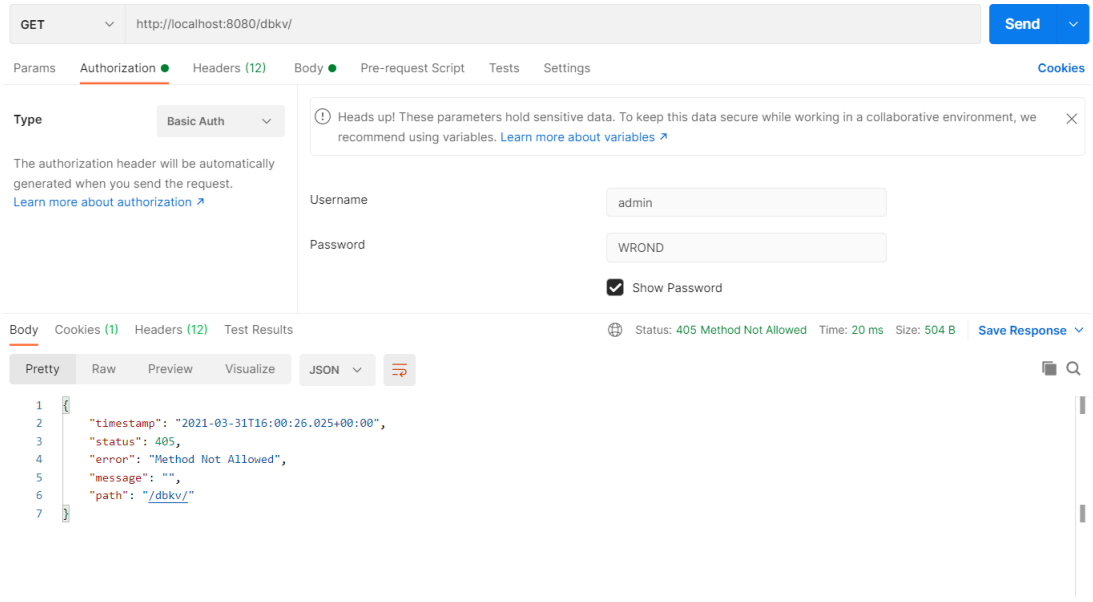
The same call may be used to update an already existing record (the attributes data is simply overwritten with the new data)



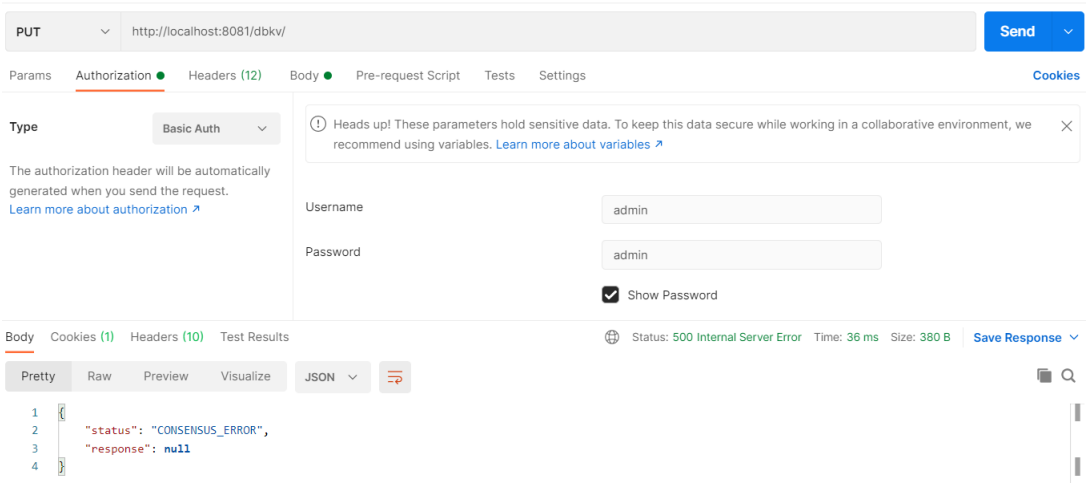
To delete some records, a **DELETE** call should be performed with a list of keys in the body:



If the security module is enabled (security.enabled=true) then a BASIC authorization header as per specification is required. Performing a request with wrong credentials will yield an "Unauthorized" error.



The node may still serve get requests in read-only mode, however if a write is requested, an error will be thrown:



Node API

You can start or stop a node via the

`/dbkv-node/`

endpoint. A **PATCH** call will start the node and a **DELETE** call will stop its consensus module.

The API also exposes the endpoints used in the dashboard, for example for the cluster and database status information. Please refer to the swagger documentations for a complete list of all the endpoints and features.

8 Conclusions

The project satisfies all of the declared objectives. The project is stable and functions correctly, although not production-grade it serves as a good proof-of-concept. It is quite easy to scale and the node is highly configurable to satisfy most requirements in most environments. The persistence functions correctly and the writes are replicated to the available cluster, in addition to that a crashed or a partitioned node can catch up without having to resort to manual maintenance. The dashboard is serviceable as a tool to visualize the consistency and perform some basic interactions with the nodes. The project is well-covered by the unit tests and system tests and the development was done according to the professional agile conventions.

8.1 Future Works

- The project's database module can be further developed to support complex queries, indexing and improve the filesystem writing mechanism. It would also be possible to implement the support for the structured data writes.

- The database mechanism can be further improved by creating a full-fledged operations log and recovery algorithms over the distributed primitives the framework provides. In alternative, it could be possible to use RAFT with a custom implementation optimized for the distributed database.

- The consensus module could offer more configurations and different protocol implementations.

- The authentication can be improved by holding users as special records in the database and a process to create a new user with a set of permissions can be created.

- The Dashboard could be further evolved and detached from the nodes to become a complete suite of tools and a development environment akin to, for example, phpmyadmin.

8.2 What did we learned

This project has provided me with a great opportunity to learn more about the distributes systems, in particular the challenges of creating p2p networks and maintaining consistency across clusters.

Another important topic of this project is the practical applications and limitations of the consensus protocols, in particular the RAFT consensus protocol, whose functionality was well researched.

Furthermore, this project has allowed me to learn and experiment with modern frameworks for http and consensus servers, a modern agile way of developing software and delivering a stable and documented product.

References

[1] Brewer's CAP Theorem
https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwiJ-97vwN3vAhXOmFwKHTOWDKEQFjADegQIFxAD&url=https%3A%2F%2Fedisciplinas.usp.br%2Fpluginfile.php%2F2541318%2Fmod_resource%2Fcontent%2F1%2FTeoremaDeBrewer.pdf&usg=AOvVaw1jVpz8Iur9NE-InCLu0JL6

[2] Gitflow
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[3] Atomix
https://atomix.io/docs/latest/user-manual/introduction/what-is-atomix/

[4] Spring / SpringBoot
https://spring.io/projects/spring-boot

[5] Mockito
https://site.mockito.org/

[6] Maven
https://maven.apache.org/

[7] Sonarlint
https://www.sonarlint.org/

[8] Paxos - Leslie Lamport
https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwj6xar2yd3vAhUxtXEKHQGjC8wQFjAAegQIAxAD&url=https%3A%2F%2Flamport.azurewebsites.net%2Fpubs%2Flamport-paxos.pdf&usg=AOvVaw1AHQDriKZ-fF2C7BTNr6qm

[9] Raft - Diego Ongaro and John Ousterhout

https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwj2zqD-yd3vAhVUuXEKHStiBmAQFjAAegQIAxAD&url=https%3A%2F%2Fraft.github.io%2Fraft.pdf&usg=AOvVaw3DlZnK2c75fpFT0lAXbwQn