Decentralized Cluster-Based NoSQL DB System

Table of Contents

Introduction to the Project

In this report, I will discuss my implementation of the decentralized cluster-based NoSQL database system. The system consists of two main parts, a bootstrapping node, and several worker nodes (or NoSQL nodes). The bootstrapping node supports the decentralization of the system by acting as an entry point for new users, and initiating the worker nodes with the needed information, such as other worker nodes addresses, so worker nodes can communicate with each other, also it provides each node with its assigned users, where every new user is assigned a node in a load-balanced manner. Each node must have a list of the users assigned to it so it can authenticate any coming commands. The system is decentralized so it can minimize the risks of a single point of failure, where a centralized system will fall apart if the master node (bootstrapping node) fails. In such systems, the number of worker nodes can vary at runtime, which is known as auto-scaling, but for simplicity we are going to assume that the number of worker nodes is always constant. In terms of the communication between worker nodes, I will use a docker network for the bootstrapping node and all other the worker nodes, and I will use Spring to create web APIs to make the communication easier between nodes themselves, and between the users and the nodes. Finally, we will have a demo application to test the correctness and load-balancing of the system.

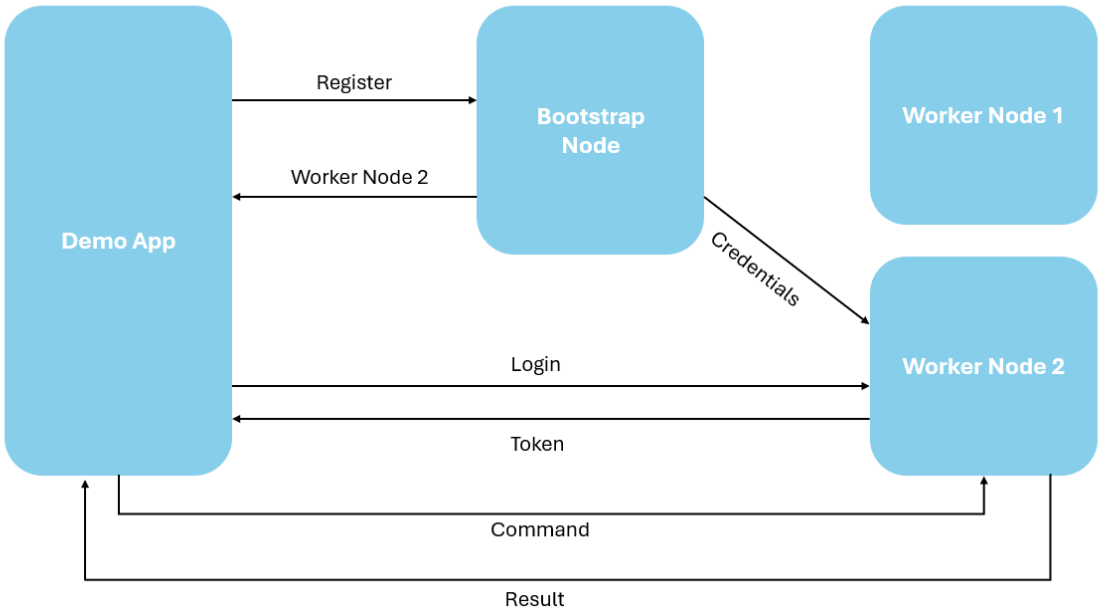
Introduction to NoSQL Databases

In NoSQL databases, each database contains several collections, a collection corresponds to a table in SQL systems, a collection has a schema that decides what structures of a document it accepts. Within each collection we have several documents which represent the entry (or the row) in SQL tables. NoSQL databases are used when we have big data, that may contain images, videos, and non-textual data. One of the key distinguishing features of NoSQL databases is their horizontal scalability. Traditional RDBMS often encounter scalability limitations when dealing with massive datasets or high transaction volumes. NoSQL databases, on the other hand, are inherently distributed, enabling them to scale horizontally across multiple nodes or servers seamlessly. This distributed architecture not only ensures high availability and fault tolerance but also facilitates linear scalability, where performance improves proportionally with the addition of new nodes. Furthermore, NoSQL databases offer specialized data models optimized for specific use cases, such as key-value stores, document stores, column-family stores, and graph databases. Each data model excels in different scenarios, empowering developers to choose the most suitable database technology based on their application's requirements. For instance, document stores like MongoDB excel in handling JSON-like documents, while

graph databases like Neo4j excel in managing highly interconnected data sets, such as social networks or recommendation engines. In summary, NoSQL databases represent a paradigm shift in database technology, offering developers the flexibility, scalability, and performance needed to tackle the challenges of today's data-driven applications. As organizations continue to embrace digital transformation and harness the power of big data, NoSQL databases are poised to play a crucial role in shaping the future of data management.

User Authentication

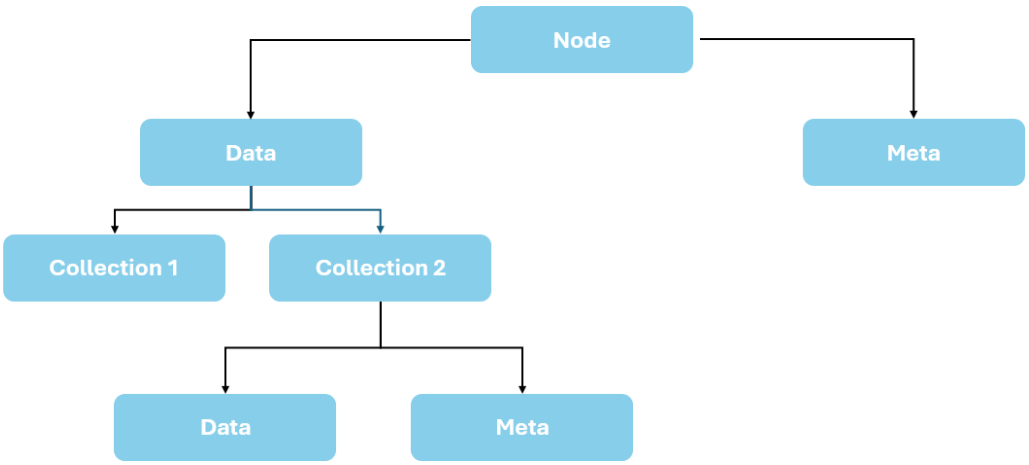
Each database system has a single user at least. User roles may vary between admins and normal users. A normal user can only select data from the database, where admins can execute any command. To control the permissions of each user, I have utilized the strategy design pattern. When constructing each command, we pass the user role that is trying to construct it. If the strategy implementation validates the role successfully then it is okay to execute the command by this user, otherwise an access denied exception is thrown. A database user's credentials are sent from the application that uses the database system (the demo application). In this implementation, any new user must first communicate with the bootstrapping node to register a new account, the bootstrapping node will respond back to the user with his assigned worker node address and sends the user credentials to the assigned worker node. After that, the user will communicate with the worker node with the credentials to login, the worker node validates the credentials of the user, if valid, the worker node responds with a JWT token with no expiration date (for simplicity). The user can use the token to send commands to the worker node, without the token, the user is forbidden to execute any commands. To persist database users, they are stored in a directory inside the bootstrapping node, then on each cluster boot, the users are distributed to their assigned nodes so each node can authenticate its users again. Regarding user registration, the database has a pre-defined admin in the system with the username "Admin" and the password "Atypon", then you can register new normal users by sending a request to the bootstrapping node with the username and the password of the new user. Below, we can find a figure that explains the process.



Database Implementation

Database Architecture

Each node is structured with two main directories, a data directory, and a meta directory. The data directory contains the actual data of the database such as the documents and collection-related meta-data. While the meta directory contains meta-data on the system level. More on that below.



The system consists of a single common database that contains several user-defined collections, this is the upper data directory specified in the figure above. Each collection contains a group of documents that belong to the schema of that collection. This way, the demo application does not send a database name along with the user credentials to the database since there is only one database. The upper meta directory in the figure will be discussed below in the document affinity section. While the lower meta directory contains some important information about the collection, it contains a file that persists the last assigned id in this collection, so the next time we add a new document to the collection it is given a built-in id from the system, this id is an existing attribute inside the document and is named "_id", the document itself is also named with the chosen id. This attribute name is preserved and if a user tries to assign it, the assigned value will be ignored. Also, the meta directory contains the schema JSON file that is used to validate any inserted documents into the collection. And finally, the directory contains another directory called "indexes" containing the files created for each index used in the collection.

Document Affinity

Let's first dive into the meta directory since it contains less information. The directory only contains the data needed to load-balance the document's affinity between the nodes. The document-node affinity is a term used to determine the node and the only node that is responsible for modifying or deleting this document. If a user sends a command to his assigned node to update or delete a document, the node only updates the documents that belong to it, then it forwards the command to all other worker nodes so they also can do the same thing. Document-node affinity is load-balanced between the nodes, the meta directory contains the number of documents assigned to each node as a HashMap with the node address as a key and the number of documents assigned to that node as a value. The currently implemented load balancer suffers from one disadvantage, the load between the nodes becomes unbalanced when a delete process is executed on the documents. In addition to what is mentioned, each document in the database has a built-in attribute that is called "_affinity", this attribute is assigned a value by the load balancer whenever a document is inserted to a collection. The value cannot be overridden by the user by trying to update the document or inserting a new one with a user-chosen value.

Database Commands

In this section, I will mention all the available commands a database user can execute. Every command starts with a unique keyword that allows the system to determine the type of the command.

Unique Keyword	Must-Have Keywords	Optional Keywords	Description
CREATE_COLLECTION(name)	ATRIBUTES(attr1 : type ...)	None	Creates a collection.
DELETE_COLLECTION(name)	None	None	Deletes a collection.
INSERT_INTO(collectionName)	ATTRIBUTES(attr1= value ...)	None	Inserts a document into a collection.
UPDATE(collectionName)	ATTRIBUTES(attr1= value ...)	WHERE(attr=value)	Updates documents.
DELETE_FROM(collectionName)	None	WHERE(attr=value)	Deletes documents.
SELECT_FROM(collectionName)	ATTRIBUTES(attr1: value ...)	WHERE(attr=value)	Reads documents.
CREATE_INDEX(attributeName)	INTO(collectionName)	None	Creates an index.
DESCRIBE()	None	None	Returns all collection names.

These commands are interpreted and checked using regular expressions, which allows us to manipulate any type of command, either with optional keywords or not. A command factory class , supporting the factory design pattern, takes the received command as a parameter and generates the command object depending on the unique keyword that the command starts with. All the commands classes implement a single interface called Command; it is a functional interface with a single execute method.

```
public interface Command {
    9 implementations
    Result execute() throws NoSuchMethodException;
}
```

Each command implementing class takes several parameters for its constructor, the received command string itself, so it can check the correctness of the command after specifying the type of it. Then an instance of the Data Access Layer so the command gets the ability to modify the file system of the cluster. And finally, a list of the executing user roles to check the permissions, more about that later.

```
public CreateCollectionCommand(String command, DataAccessLayer dal, List<UserRole> userRole) throws AccessDeniedException {
    this.command = command;
    this.dal = dal;
    this.permissionStrategy = new WritePermissionStrategy();
    if (!permissionStrategy.checkPermission(userRole))
        throw new AccessDeniedException("Access denied: You role is not allowed to perform the selected operation!");
}
```

After executing any command, an object of type Result is returned to the demo application. It contains two main attributes, a Boolean indicating whether the execution of the command was successful or not. And a list of JSON strings representing the retrieved documents or the affected ones.

Data Access Layer

Any command will use the storage to read or write to it. A Data Access Layer interface is there to help. As we can see below, it contains all the methods that correspond to the list of available commands in our system.

```
@Component
public interface DataAccessLayer {
    2 usages  1 implementation
    Result select(String collectionName, List<String> columns, List<Condition> conditions);
    1 usage  1 implementation
    Result insert(String collectionName, Map<String, String> attributes);
    2 usages  1 implementation
    Result update(String collectionName, Map<String, String> modifications, List<Condition> conditions, boolean byPass);
    1 implementation
    Result delete(String collectionName, List<Condition> conditions, boolean byPass);
    1 usage  1 implementation
    Result createCollection(String collectionName, String schema) throws IOException;
    1 usage  1 implementation
    Result deleteCollection(String collectionName);
    1 usage  1 implementation
    Result createIndex(String collectionName, String columnName);
    1 usage  1 implementation
    Result describe();
}
```

Data Types and Conditions

The supported data types to be used in my system are:

- String
- Boolean
- Integer
- Float

The only supported condition to be used inside the where keyword is:

- Equals (the only one currently implemented).

Indexing

Indexing is the process of mapping entries to specific values inside a specific data structure. In the realm of database management, indexing stands as a technique aimed at optimizing data retrieval efficiency. Indexes serve as structured data arrangements within a database, organizing and sorting specific columns to improve data retrieval processes. By strategically implementing indexes on frequently accessed columns like primary keys or commonly filtered attributes, database systems can swiftly locate records, particularly within extensive datasets. This acceleration significantly reduces the time and computational resources required to execute queries, thereby improving system performance. However, the implementation of indexes must be balanced with the associated overhead, as excessive indexing can lead to increased storage requirements and slower data modification operations, such as inserts, updates, and deletes. In my system, I am using the B-Tree data structure for each created index. The B-Tree is a self-balancing data structure where each node can contain several keys and children. It works in a way that gives Log(N) complexity for all operations since it is always balanced, and all leaf nodes are the same height.

B-tree Implementation

1. For each node x, the keys are stored in increasing order.
2. In each node, there is a Boolean value x.leaf which is true if x is a leaf.
3. If n is the order of the tree, each internal node can contain at most n – 1 keys along with a pointer to each child.
4. Each node except root can have at most n children and at least n/2 children.
5. All leaves have the same depth (height-h of the tree).
6. The root has at least 2 children and contains a minimum of 1 key.

In my case, a regular B-Tree would not work, I had to customize the default implementation of it, so it suits my case. In regular cases, each node contains several keys, each key is a value, let's say a number. Then each node has several children where they are sorted according to the keys of the parent node. But in my implementation, each node is key-value pair, where the key is the value of the

attribute we are indexing on, and the value is a list of documents paths where these documents satisfy the key to that pair for the indexed attribute.

Data Consistency Issues

Since we are dealing with a cluster of nodes, then each node will have its own file system, so if a user writes to a document or modifies the file system, this modification must be broadcast to all other nodes. I have utilized the singleton design pattern by creating a class called Data Synchronizer. This class contains methods that help the node to communicate with other nodes and send them either commands to maintain the correctness of the affinity rule, or what is defined as a Data Syncing Action. Each command in the database system corresponds to a data syncing action implementation. This action is sent using the data Synchronizer to all other nodes which will receive the action on a specifi" API endpoint. Also, some of the actions is called Write File Sync Action that takes a file and a byte array and syncs them across all nodes. This might be useful if we want to synchronize some files like the affinity file. Another implementation is the Delete File Syncing Action, which does what it says. The Data Syncing Action interface has a bunch of annotations that help it to get converted to and from JSON successfully. This is needed since all nodes are receiving an abstract type on the previously mentioned API endpoint and it needs to know to what implementing type it must be cast to.

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, include = JsonTypeInfo.As.PROPERTY, property = "type")
@JsonSubTypes({
        @JsonSubTypes.Type(value = CreateCollectionAction.class, name = CreateCollectionAction.TYPE),
        @JsonSubTypes.Type(value = DeleteCollectionAction.class, name = DeleteCollectionAction.TYPE),
        @JsonSubTypes.Type(value = WriteFileSyncAction.class, name = WriteFileSyncAction.TYPE),
        @JsonSubTypes.Type(value = DeleteFileSyncAction.class, name = DeleteFileSyncAction.TYPE)
})
public interface DataSyncingAction {
    1 usage   4 implementations
    void sync(FileSystem fileSystem);
}
```

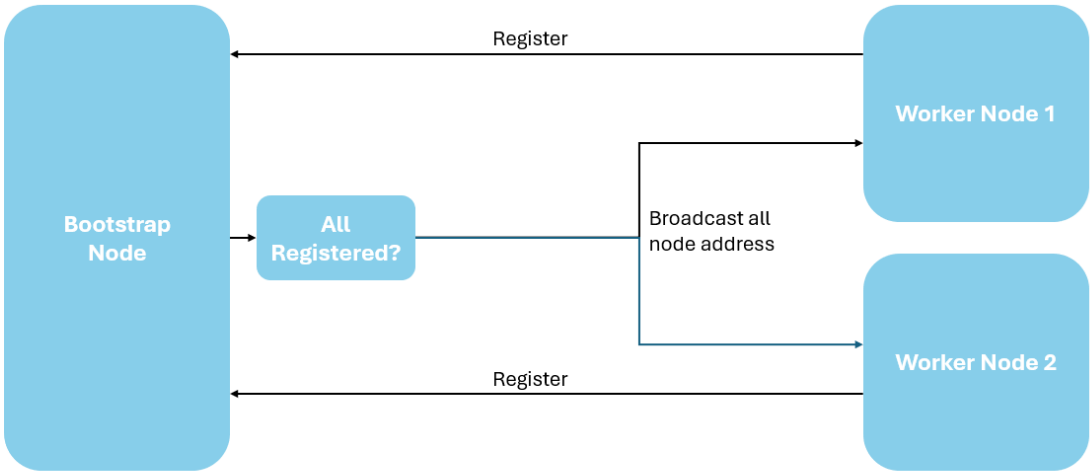
Bootstrapping Node

Endpoints

I have utilized what we have learnt in the training about Spring framework. The bootstrapping node contains a rest controller that has a couple of API endpoints:

- /register-user: takes the username in the body of the request and returns the created user if the process was successful. After creating the user, the user must be sent to the node assigned to him, which is a worker node (or NoSQL node), and it must be saved to the storage of the bootstrapping node so every time the cluster is started the already existing users are sent to the NoSQL nodes.
- /nosql/register-node: This endpoint is used by the NoSQL nodes to register their existence and address. When all nodes are registered, the addresses of the nodes are broadcast to all nodes so they can find a way to communicate with each other.

Below, we can see how worker nodes know about each other.



Multi-threading

Since I am using Spring framework, requests handling in terms of multi-threading is already built-in. All I need to do is to make sure all multi-threaded methods are synchronized properly. In the bootstrap node, we only have a few cases.

1- The endpoint that registers new users: the whole method is synchronized on the declared list of users.
2- The endpoint that registers new nodes: the whole method is synchronized on the declared list of nodes.

Security

We must protect the APIs mentioned above, not all of them. Registering a node is a very sensitive process, imagine if anyone can send a request and register his own server to our cluster. For that reason, a private API key is used to communicate between all nodes, to protect the "/nosql/register-node" endpoint. An interceptor is added to the configuration of Spring framework that intercepts the requests before they are handled by the server. It checks whether the authorization header contains the API key or not.

NoSQL Node

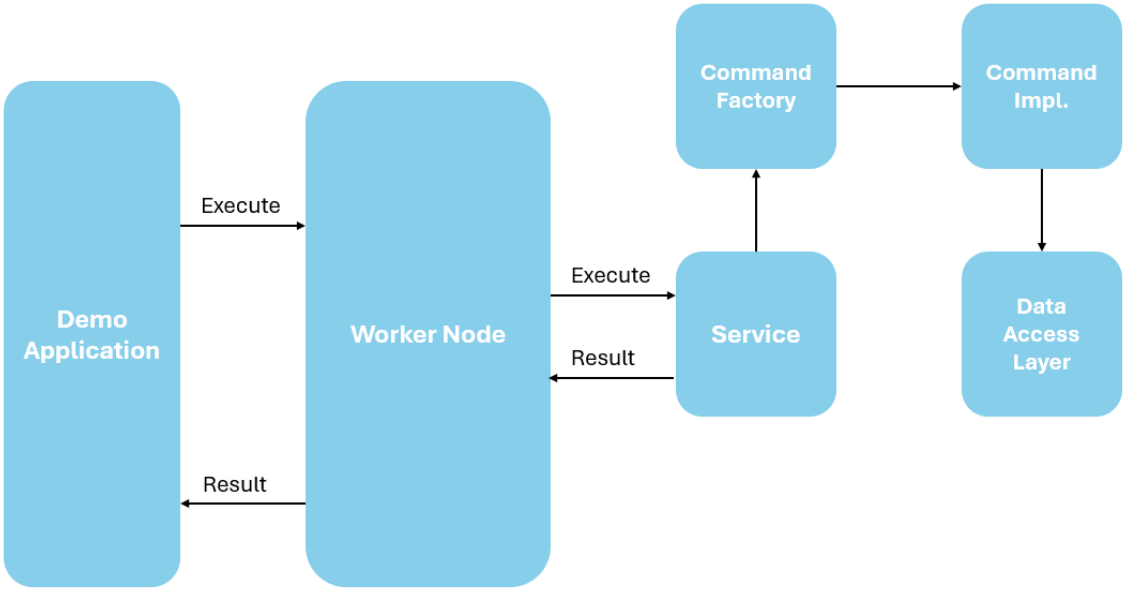
Endpoints

It is also a Spring application with the following API endpoints:

- /bootstrap/nodes-addresses: Takes a collection of nodes so the nodes addresses are saved in each node to be able to communicate with each other.
- /bootstrap/node-users: Takes a collection of the users assigned to this node so the node can authenticate the user sending the command to it.
- /bootstrap/add-user: This endpoint is used when a new user wants to join the cluster.
- /bootstrap/stats: This used for testing, more about it later.

- /execute: Takes the command to execute as a string, and an authorization JWT token to authenticate the user.
- /nosql/data-broadcast: This endpoint is used by the other NoSQL nodes to broadcast any changes happening to the file system.
- /login: This one is used to authenticate the credentials of the user and return a token.

Regarding the NoSQL node, whenever it receives a command on the "/execute" endpoint, it goes through the following process.



Multi-threading

Again, requests handling in terms of multi-threading is already built-in. All I need to do is to make sure all multi-threaded methods are synchronized properly. Below are the cases where we need to synchronize:

1- Receiving the addresses of all nodes coming from the bootstrap node is synchronized based on the declared list of addresses.
2- Adding a newly registered user is synchronized on the users list too.
3- The token or credentials authentication process is synchronized on the list of users too.
4- Insert document command is synchronized inside the data access layer based on a defined lock.

5- Update document command is synchronized inside the data access layer based on a defined lock.
6- Delete document command is synchronized inside the data access layer based on a defined lock.
7- Create collection command is synchronized inside the data access layer based on a defined lock.
8- Delete collection command is synchronized inside the data access layer based on a defined lock.
9- Create index command is synchronized inside the data access layer based on a defined lock.

Here we can find the list of defined objects to be used as locks:

```
private final Object insertLock = new Object();
1 usage
private final Object updateLock = new Object();
1 usage
private final Object deleteDocumentLock = new Object();
1 usage
private final Object createCollectionLock = new Object();
1 usage
private final Object deleteCollectionLock = new Object();
1 usage
private final Object createIndexLock = new Object();
```

Security

Same goes here, some specific APIs must be secured. Security is implemented here in two ways, the first one is using interceptors, I have a single interceptor that checks for any URLs starting with “/nosql” or “/bootstrap”, then it reads the authorization header searching for the private API key. And the other way of securing the APIs is checking the “/execute” endpoint by searching the authorization header, but for a token this time, since only authenticated users are allowed to execute commands on the NoSQL node.

Demo Application (Email Demo Application)

I have chosen to create a simple email application that allows new users to register into the system, login to their accounts, read the received emails, and send new emails.

Endpoints

1- /register: used to register new users.
2- /login: used to authenticate users using their credentials, then generating a token.
3- /compose: used to send new emails.

Multi-threading

Since I am using Spring then threads are automatically created, below we can find the cases where I needed to synchronize application:

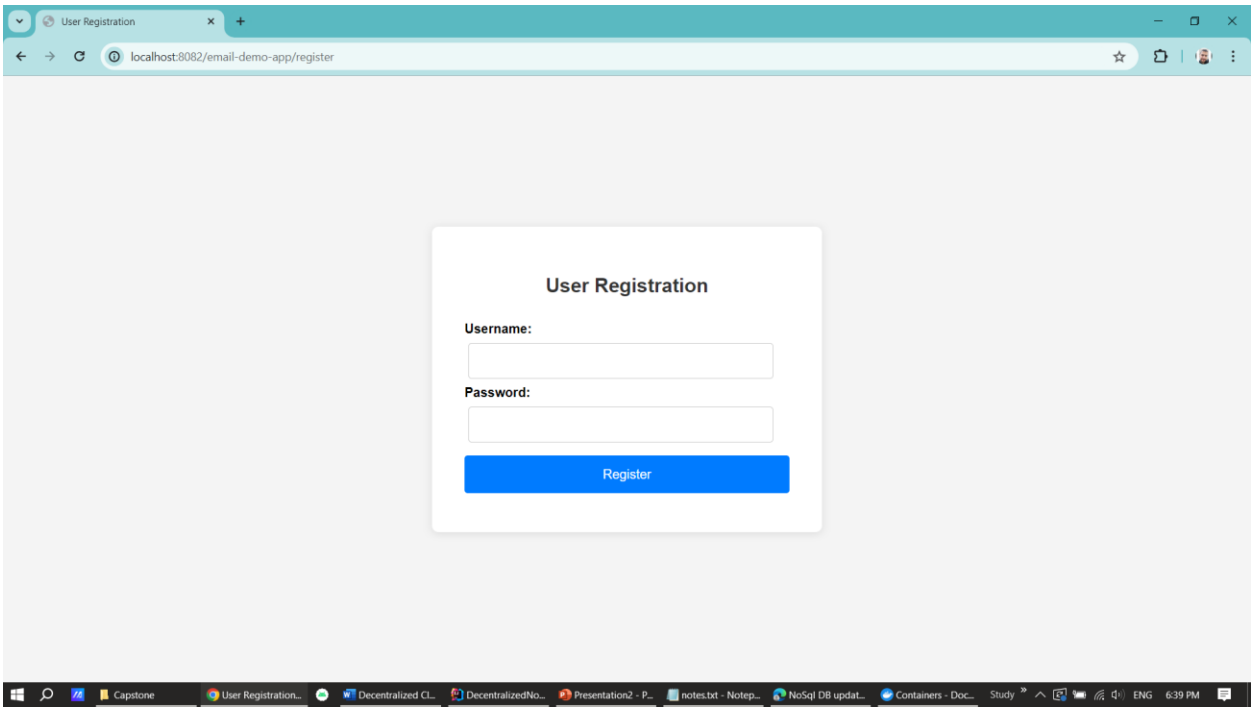
1- Registering a new user: since two people might try to register at the same time.
2- Logging in: Since two people might try to login at the same time.

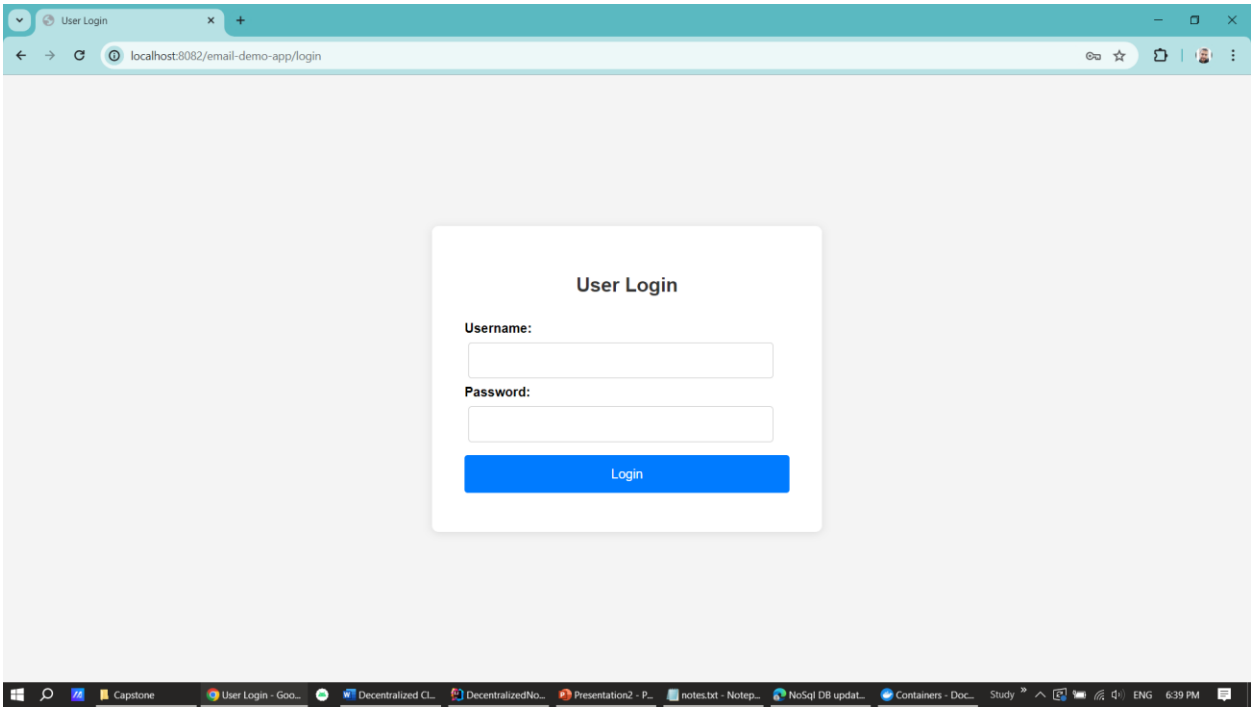
Security

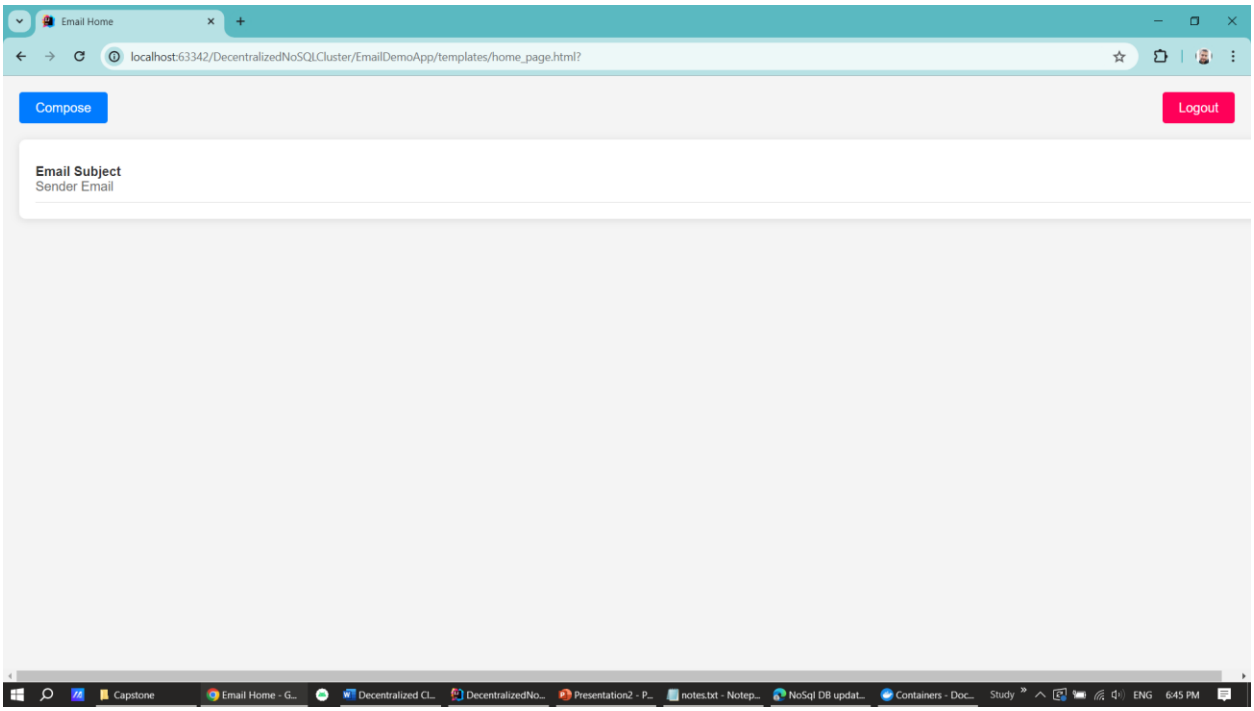
Security in the demo application is simple, whenever a user tries to login, the user's credentials are checked based on the database collection that contains the registered users. If the user is authenticated successfully, a token is stored as a cookie to avoid entering the credentials every time the user tries to access a page in the web application. But the server needs to store the tokens somewhere so it can authenticate users. For simplicity, I have chosen to store the tokens inside the database itself as an attribute in the Email App Users collection. The password is also stored in the database, usually it must be hashed using BCrypt, but I skipped this step to save time, knowing that I have implemented this feature in previous assignments. Spring Security was an option to secure all endpoints, but it would be overkill for such a simple application.

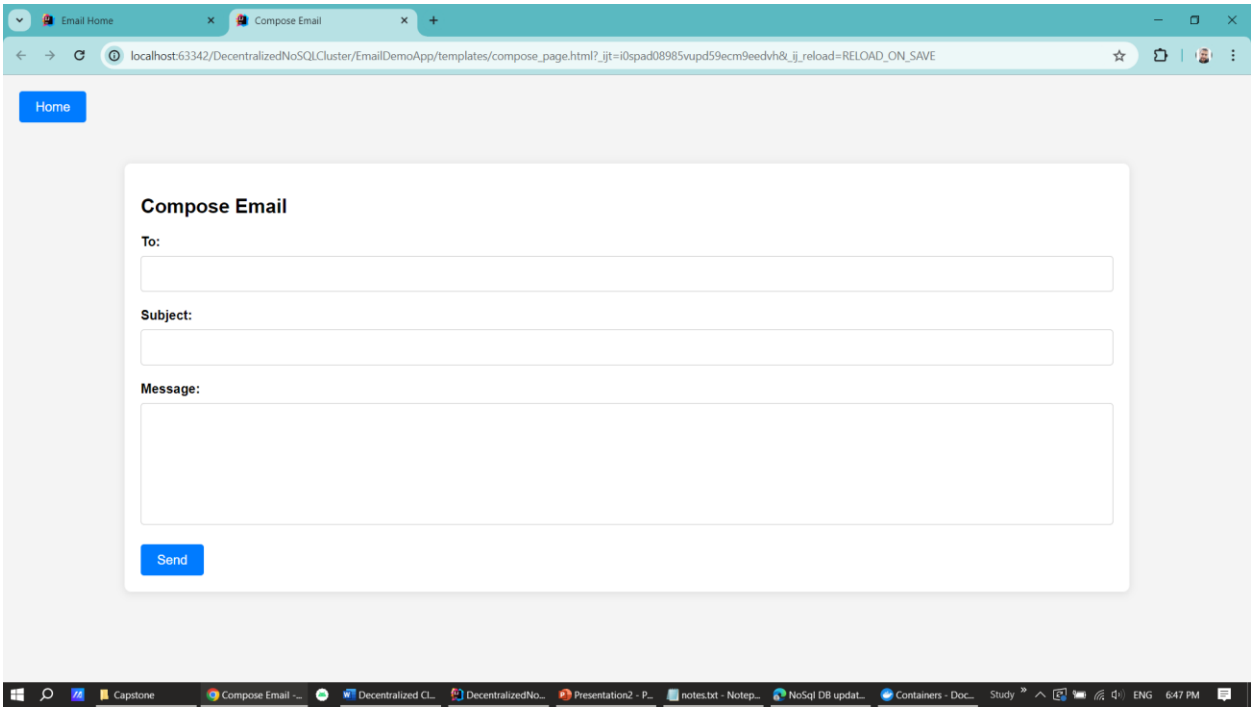
UI

The application has 4 pages in total, a register page, login page, home page, and a compose page.


User Registration
localhost:8082/email-demo-app/register
User Registration
Username:
Password:
Register


User Login
localhost:8082/email-demo-app/login
User Login
Username:
Password:
Login


Email Home
localhost:63342/DecentralizedNoSQLCluster/EmailDemoApp/templates/home_page.html?
Compose
Logout
Email Subject
Sender Email


Email Home
Compose Email
localhost:63342/DecentralizedNoSQLCluster/EmailDemoApp/templates/compose_page.html?_ijt=i0spad08985vupd59ecm9eedvh8&_ij_reload=RELOAD_ON_SAVE
Home
Compose Email
To:
Subject:
Message:
Send

Connecting to the Database

I have created an interface that corresponds to the JDBC driver, it allows us to communicate with the database in an abstract matter, by using the sendCommand method with a string parameter representing the command.

```
@Component
public interface DatabaseConnector {
    8 usages  1 implementation
    Result sendCommand(String command);
}
```

How the connector works:

1- First, it checks for a locally stored database user in the file system of the server running the demo application.
2- If a database user exists, it tries to login to the assigned node to that user, then a token is retrieved.
3- If there aren't any database users saved locally, the connector tries to register a new user by communicating with the bootstrap node.
4- After the last two steps, the user is saved locally in the file system, so the next time we login we get the latest used user.
5- Then the connector is ready to trigger the sendCommand method by other classes from the demo application, it includes the token with the sent request to the NoSQL node.

Afterwards, the demo application has two data access objects, the first one is for the application users, it creates a collection in the cluster database named Email App Users where each document must contain:

1- Username of type string.
2- Password of type string.
3- Token of type string.

Then we have the Emails collection that stores all the sent emails in the application. It contains the following attributes:

1- Sender username of type string.
2- Receiver username of type string.

3- Email subject of type string.
4- Email message (body) of type string.

Containerization

Docker Compose

Docker was utilized to containerize the system, having each service separated in its own container. This way you decouple the components of your system. I have utilized Docker Compose to build multiple containers at once. I have tried to make it as scalable as possible by giving the developer the ability to choose the number of nodes and scale the cluster using a number that is specified in the compose file. This feature allows us to avoid creating a service for each worker node (NoSQL node).

Docker Files

I have three docker files in my implementation for bootstrap node, NoSQL node, and the email demo application. All of them look the same since I am running Java on all of them. I am using the openjdk-19 Alpine image to build the containers. Then each container takes the JAR file and runs it inside the container on a specific port.

Volumes, Networks, and dependencies

Two volumes are created in the compose file. One is created to persist the users of the cluster; it is attached to the bootstrap node. And the other volume is attached to the demo app to store the database user retrieved from the bootstrap node. Then a question might be, why doesn't the worker nodes have a volume attached to them? The answer simply exists at the title of this project; we need to preserve decentralization. The simplest way is to attach a volume to the service responsible for creating multiple worker nodes, but this defeats the purpose of the project as I mentioned. The harder way is to create a volume for each worker node, which is great for decentralization. The problem is that there isn't a simple way to assign different volumes to the same service in the compose file, because I have mentioned before that I use the scaling feature of docker to write down a single service that creates multiple containers as defined in the figure below.

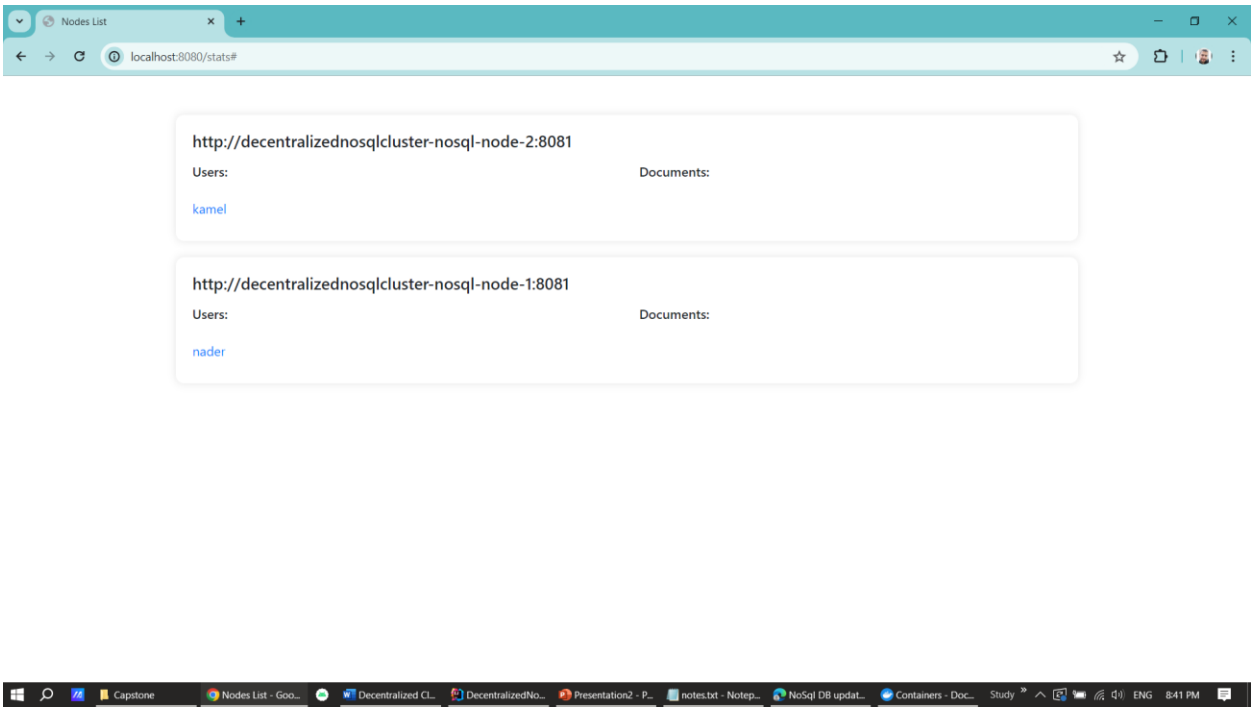
Otherwise, the solution would be to create a service for each worker node, which is not a scalable solution at all.

```
nosql-node:
  build: NoSQLNode
  ports:
    - "8081"
  environment:
    BOOTSTRAP_NODE_ADDRESS: "http://bootstrap-node:8080"
  restart: on-failure
  depends_on:
    - bootstrap-node
  networks:
    - cluster-network
```

Now regarding the networking, a single bridge network called cluster-network is created to connect the three services with each other using a software bridge network on the host machine. Finally, we can notice the depends-on attribute on the service, this allows the worker node to wait until the bootstrap node is running, it is not a very stable solution since sometimes the worker node finishes the initialization process before the bootstrap node. Also, the demo application depends on the worker node, which means that the demo application service will also depend on the bootstrap node inherently. The worker node is passed the address of the bootstrap node as an environment variable so it can communicate with it.

Code Testing

Some parts of the project are straightforward to test, and some of them are tricky, such as the multi-threading. I have implemented a simple page that displays the stats for the load-balancing results. The page is inside the bootstrap node, and we can access it using the “/stats” endpoint. It communicates with all worker nodes and collects data from them to be displayed on the page. Below we can see how the stats page looks like.



It displays each node as a card view, each card containing the users assigned to it, and the document IDs the belong to it in terms of the affinity.

Another way to test the system, in terms of data consistency and correctness, is to use Docker Desktop application to go through the file system of each node and read the contents of the files.

Finally, I haven't created any testing mechanism for multi-threading, but what I would have done if I had more space of time, is to create a test service that sends multiple requests at the same time to the nodes.

Clean Code

Defending against Uncle Bob

- Naming all variables, classes, and methods are named following the conventions agreed-on.
- Avoided returning null or special codes from any method.
- Limited number of parameters for all methods.

- Avoided using flag arguments, only were utilized once, only to be able to tell if I should execute a block of code or ignore it, but not to branch between two blocks of code.
- Avoided magic numbers or hard-coded strings.
- Implemented exception handling in an effective way, without catching generic exceptions, or implementing empty catch blocks. Also, I have utilized catch with resources.

Defending against Effective Java

- Used static factory methods instead of constructors by utilizing the singleton design pattern.
- Utilized Spring's dependency injection framework instead of hand-wiring objects.
- Utilized try with resources instead of try-catch-finally.
- Overridden the equals method in the conventional way.
- Overridden the hash method whenever the equals method is overridden.
- Tried to override toString method whenever needed.
- Overridden clone method in the B-Tree implementation.

Defending against SOLID Principles

1- Single Responsibility Principle:

   All classes in the system adhere to one purpose, for example, the collection class is only responsible for collections-related operations, such as creating all directories needed for the collection, adding documents to the collection, and more.

2- Open Closed Principle:

   Utilizing the strategy design pattern, I tried to encapsulate what varies of the code as much as possible.

3- Liskuv Substitution Principle:

   There aren't any sub types that implement or extend their parent classes and change the implementation of the parent methods to do something not expected.

4- Interface Segregation Principle:

All methods implementations that override any interface have non-empty bodies. There aren't any concrete types that implement a method they don't need.

5- Dependency Inversion Principle:

Both concrete and abstract types depend on abstraction. Utilizing Spring's dependency injection framework, I have never written a line of code that declares a concrete type.

Design Patterns

The following are the design patterns used in my solution:

- Singleton.

  Used for the data synchronizer class, this class provides a global point where each worker node can broadcast data or forward commands to other nodes easily.

- Strategy.

  Used for implementing the permission of each command, all commands have a private field in their implementation that is of User Permission Strategy, it has a single method that takes a list of roles the user has, then determines the scope of access the user has by deciding whether the user can execute this command or not.

- Factory.

  Used to construct different types of commands, the received command string must be parsed to one of the available command types. The Command Factory class is the one responsible for this.

- MVC.

  Since I have several pages in my system, especially for the demo application, I have utilized the model-view-controller design pattern to achieve the separation of concerns.