ATYPON

Atypon Java and DevOps Bootcamp (Fall 2022)

Capstone Project

Decentralized Cluster-Based NoSQL DB System

Student Name : Diya Momani

Email : diyamomani0@gmail.com

Github : https://github.com/DiyaMomani

Table of Contents

Introduction

What is the Decentralized Cluster-Based NoSQL DB?

In this project I was asked to create Decentralized Cluster-Based NoSQL DB System, A decentralized cluster-based NoSQL database system is a distributed database system that operates on a peer-to-peer network architecture. It allows multiple nodes to share and store data without the need for a centralized authority. The system is designed to scale horizontally, meaning it can add nodes to the network as the data grows.

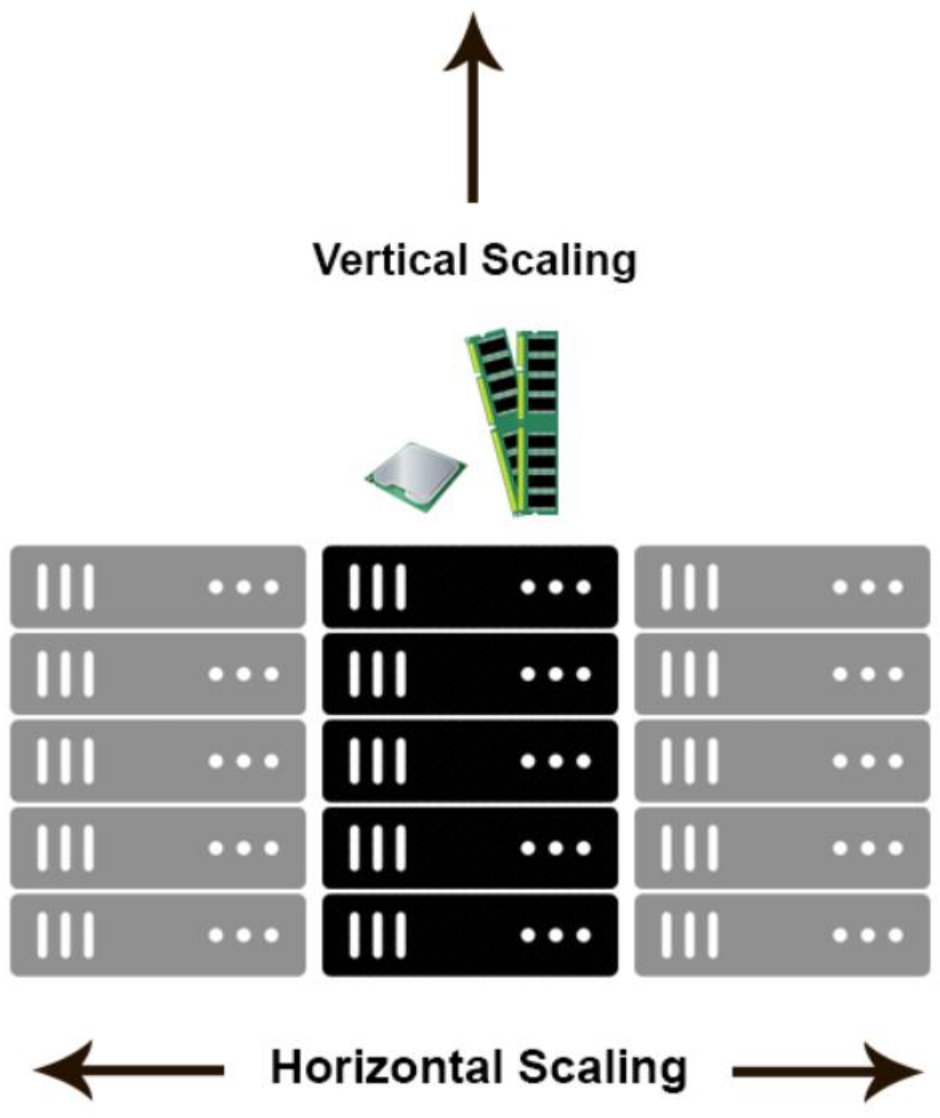


Figure-1 : shows the horizontal scale and how it is more scalable.

NoSQL, which stands for "not only SQL," is a type of database management system that doesn't rely on a traditional relational database structure. Instead, it uses a schemaless data model that allows for more flexible and dynamic data storage. NoSQL databases can store large amounts of unstructured data, making them ideal for big data applications.

NoSQL databases are better suited for applications that require scalability, flexibility, availability, performance, and cost-effectiveness, whereas SQL databases may be a better fit for applications that require strict adherence to a data schema and complex query operations.

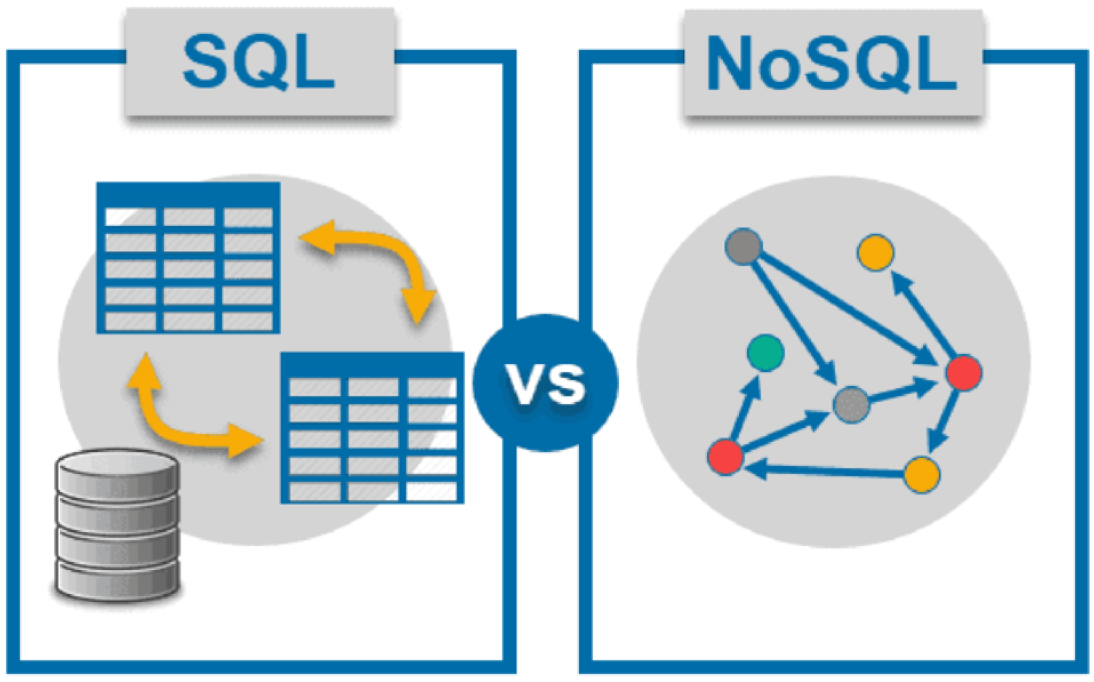


Figure-2 : shows NoSQL vs SQL databases .

In contrast to traditional centralized database systems, where data is stored on a single server, a decentralized cluster-based NoSQL database system allows data to be distributed across multiple nodes. This means that if one node fails, the other nodes can still access the data, ensuring high availability and fault tolerance.

The cluster-based NoSQL database system is also designed to be highly scalable. It can add new nodes to the network as needed to increase storage capacity and processing power. This allows the system to handle increasing amounts of data and growing user demand without any disruption to the service.

There are several types of decentralized cluster-based NoSQL database systems, each with its own unique architecture and features. Some examples include Cassandra, Riak, and Couchbase. And this is an explanation on Casandra database.

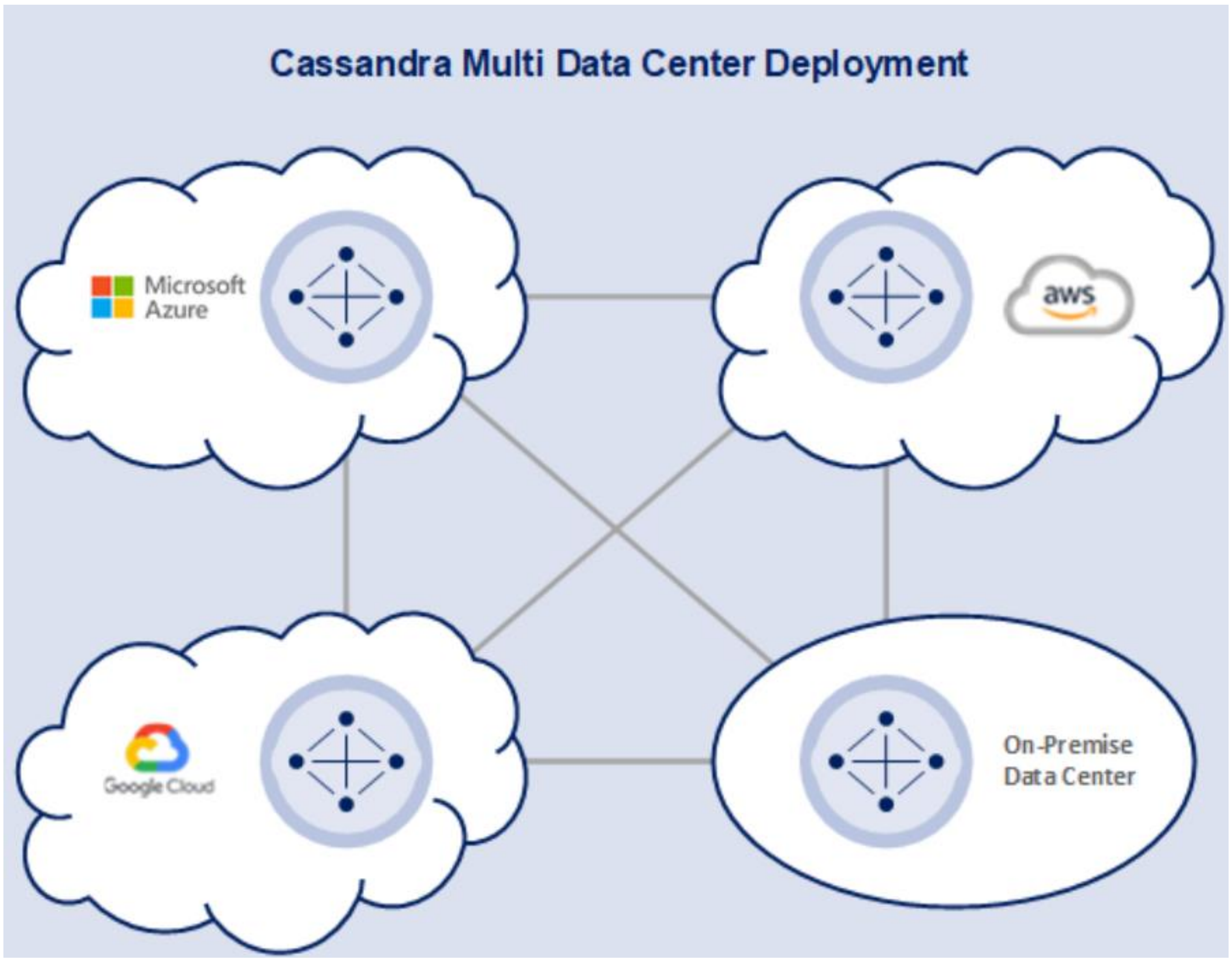


Figure-3: shows the Casandra database.

A decentralized cluster-based NoSQL database system is a powerful tool for handling large amounts of unstructured data in a distributed environment. By using a peer-to-peer network architecture and a schemaless data model, these systems can provide high availability, fault tolerance, and scalability for modern applications. As the demand for big data solutions continues to grow, we can expect to see more businesses adopting these decentralized NoSQL database systems to meet their data storage and processing needs.

In this project I focused on completing all the requirements completely , I created 3 main Intellj projects : worker, bootstrap and demo. I created database called worker and used data structures to store, organize, and manipulate data efficiently, making it easier to access and process. I used multithreading and locks to improve performance, responsiveness, resource sharing and synchronization, and also I used hashing to efficiently store and retrieve data in a data structure. I focused on security by using tokens. also I made my code follows the Clean Code principles and solid principles .

What is the data model I used?

There is three types of NoSQL database :Key-Value store, Document store and Graph store, I used Document store database because It allows for a flexible data model, as data is stored in documents, which can contain nested and complex data structures, making it easier to store data as it is.

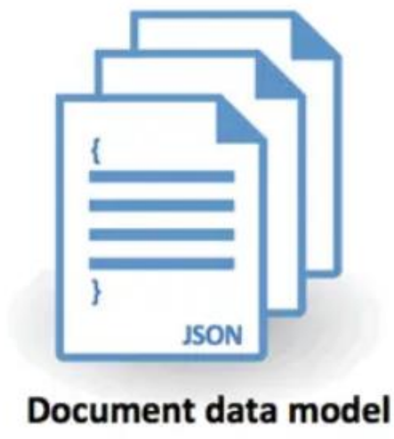


Figure-4: shows the document data model.

Why I used JSON files?

I used JSON files because they are Human-readable that means JSON is easy to read and understand for both humans and machines. Also JSON is a lightweight data format, which means that it requires less storage space compared to other data formats. JSON allows for a flexible data model, which is well-suited for document model NoSQL databases. It can represent complex data structures making it easier to store data as it is, without the need for data normalization. I used them to store the users, admins and schemas.

Schema

a schema is a logical data model that describes the organization and structure of a database. It can be thought of as a blueprint for a database. It provides a formal framework for organizing and storing data, and it specifies the rules that govern the relationships between the various elements of the database.

There is two types of schemas : strict and flexible , A strict schema is a data model that defines a fixed structure for data, with specific rules and constraints that must be followed. A strict schema is a data model that defines a fixed structure for data, with specific rules and constraints that must be followed.

In my project I gave the user the freedom to choose the type he want , he can enter the schema and store records depends on it , but in demo I allow the user to enter strict schema and the only condition of this schema that he should enter "Id" value.

What I used to build database?

I used **Java SE** as main programming language, Maven as build tool and dependency management, **Spring Boot** to create application .

Database Implementation

Over View of system

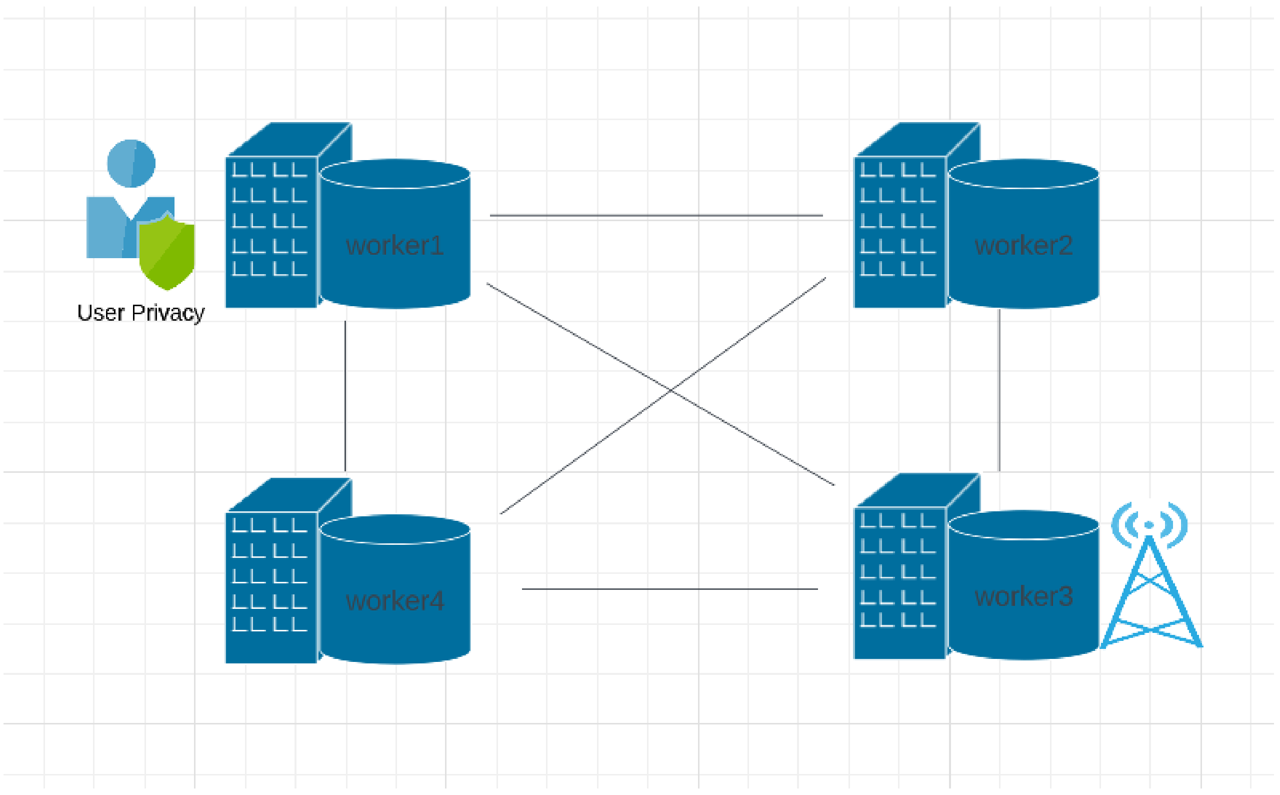


Figure-5: shows the over view of the system

I have four servers : worker1, worker2, worker3 and worker4, each of them has database/s to store records , each worker sends the records it receives to affinity node, and then affinity node will send it to the cluster, and from cluster to all other workers.

The structure of database

The flow of data

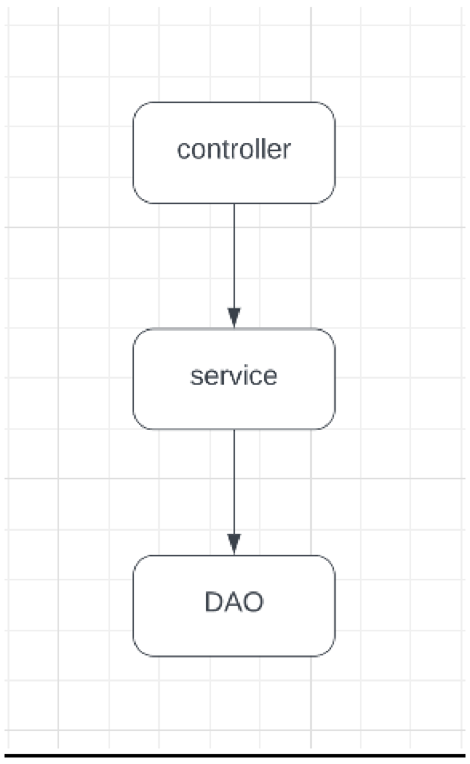


Figure-6: shows the general flow of data

Any request that will be send to server will follows this flow , first it will received by controller and send it to the affinity and check if the user or admin authorized , then send it to the cluster , and from cluster to all workers , and inside workers the controller will invoke the required function in service class, and this function will invoke the required function inside DAO class which will perform the changes required in database.

I will explain how to create database and the structure of it.

Create database

The user can create database by sending a request to the create Database URL, then the database will be created in this structure .

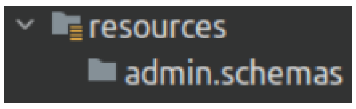


Figure-7: shows the database that created

So, when creating database it will create database folder called with the name specified by the user , and create an empty folder called schemas inside it to store the schema JSON file/s that contain the schema/s.

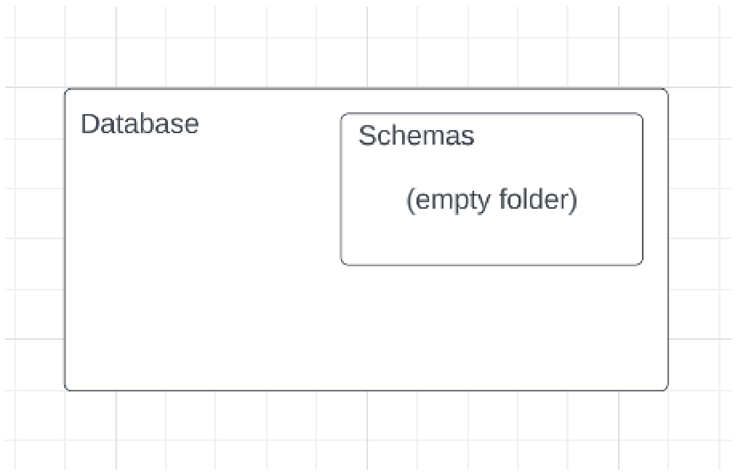


Figure-8: shows the structure of database before adding collection/s.

<u>Create collection</u>

The user can create database by sending a request to the create Collection URL, then the collection will be created in this structure .

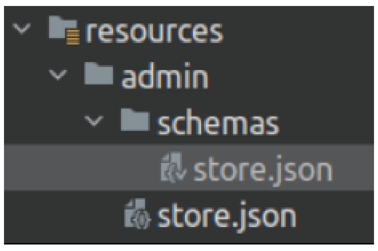


Figure-9: shows the database after add collection

So, when creating database it will create two JSON files called with the name specified by the user , one of them will be inside the schema folder and the another empty one out schema folder but inside database folder, and this is explanation of the structure.

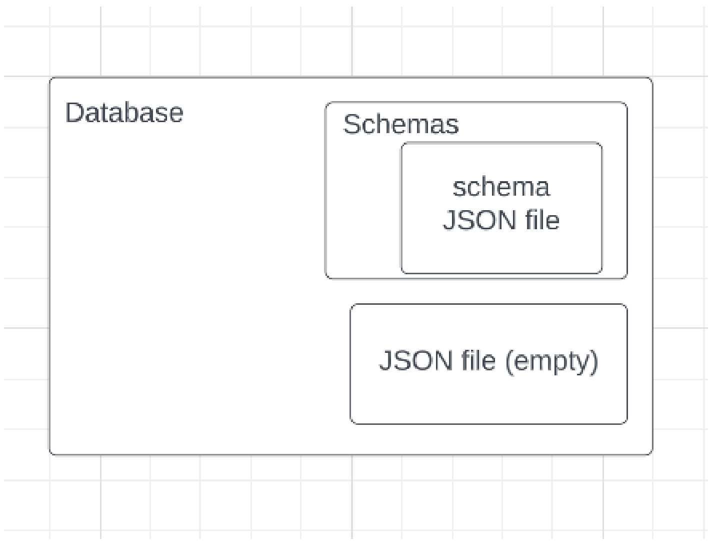
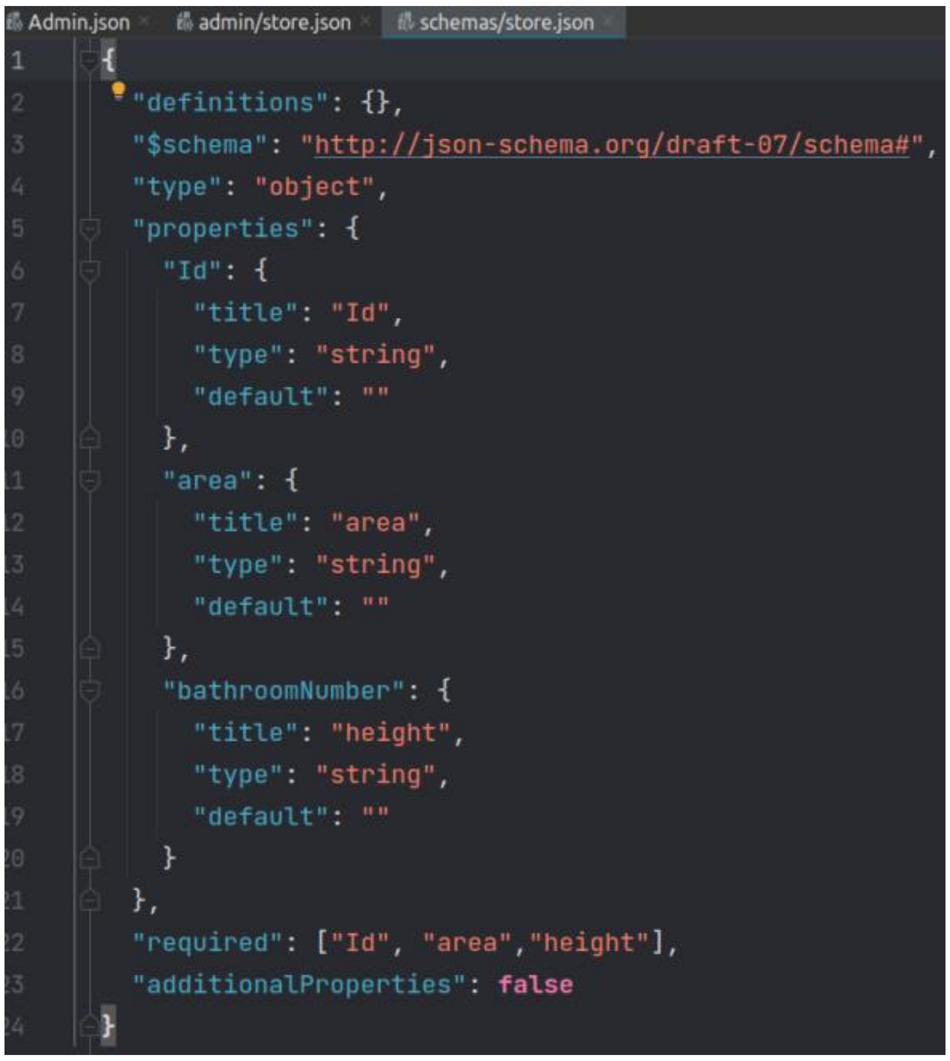


Figure-10: shows the structure of database after adding collection



```
{
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "Id": {
      "title": "Id",
      "type": "string",
      "default": ""
    },
    "area": {
      "title": "area",
      "type": "string",
      "default": ""
    },
    "bathroomNumber": {
      "title": "height",
      "type": "string",
      "default": ""
    }
  },
  "required": ["Id", "area","height"],
  "additionalProperties": false
}
```

Figure-11: shows the schema

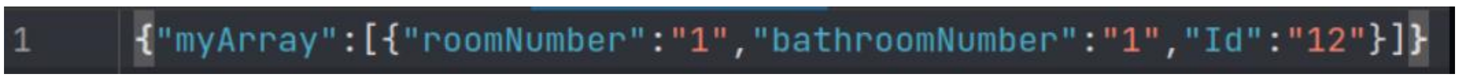
1- Schema JSON file: the schema JSON file includes the schema that entered by user when creating collections.

2- Empty JSON file : It's outside the schemas file and it used to store all records that inserted by user.

There is a class called Schema checker class that check if the record that inserted by user matches the schema rules , and it follows the single responsibility solid principles.

Add Record

The user can add a record to database by sending a request to the add record URL, then the record will be added to the JSON file out the schema folder in this structure .



```
{"myArray":[{"roomNumber":"1","bathroomNumber":"1","Id":"12"}]}
```

Figure-12 : shows the record array in json file after adding record.

When adding a record to the database it will be stored in the JSON file by taking the record as string and use objectMapper to read it as a JSON node, And it read this JSON by using objectMapper .

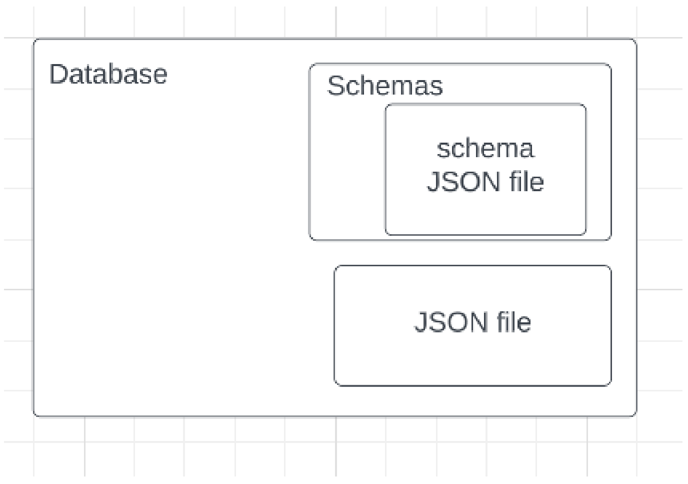


Figure-13: shows the structure of database after adding record

Indexing and the data structure used

I used hashing indexing in this project. So Hashing indexing is a technique used in computer science and database systems to quickly retrieve data based on a key or identifier.

The main advantage of using hashing indexing is its ability to provide constant-time access to data, regardless of the size of the dataset. This is achieved by using a hash function to map each key or identifier to a specific location in memory, where the associated data is stored.

Hashing indexing is commonly used in scenarios where data needs to be quickly accessed and retrieved, such as in database management systems, search engines, and data caches. It can be particularly useful in situations where there are large amounts of data to be searched through, as it allows for quick and efficient retrieval of specific data points.

Another advantage of hashing indexing is its ability to handle collisions, which occur when multiple keys map to the same location in memory. Hashing algorithms typically include methods for handling collisions, such as chaining or open addressing, which ensure that all data can still be accessed correctly even in the presence of collisions.

In my project , I implemented three indexing layers : Database Indexing , collection indexing and property indexing.

Database indexing

Database indexing is a HashMap , the key of it is a string database name , the value is collection indexing as follow :

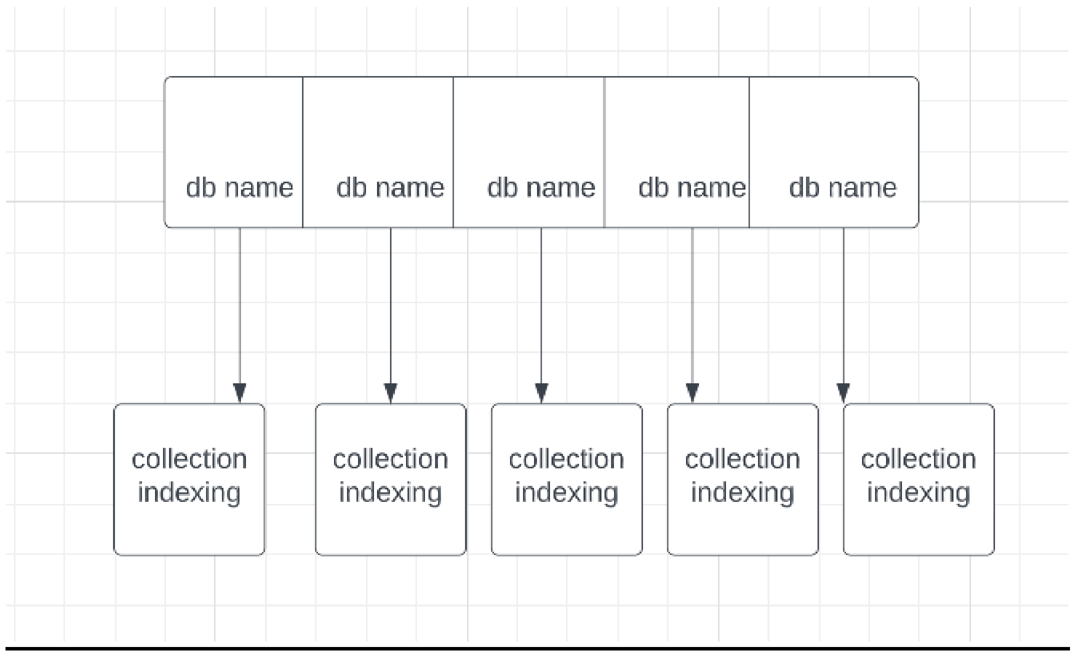


Figure-14: shows the database indexing

Collection indexing

Collection indexing is a HashMap , the key of it is a string collection name , the value is property indexing as follow :

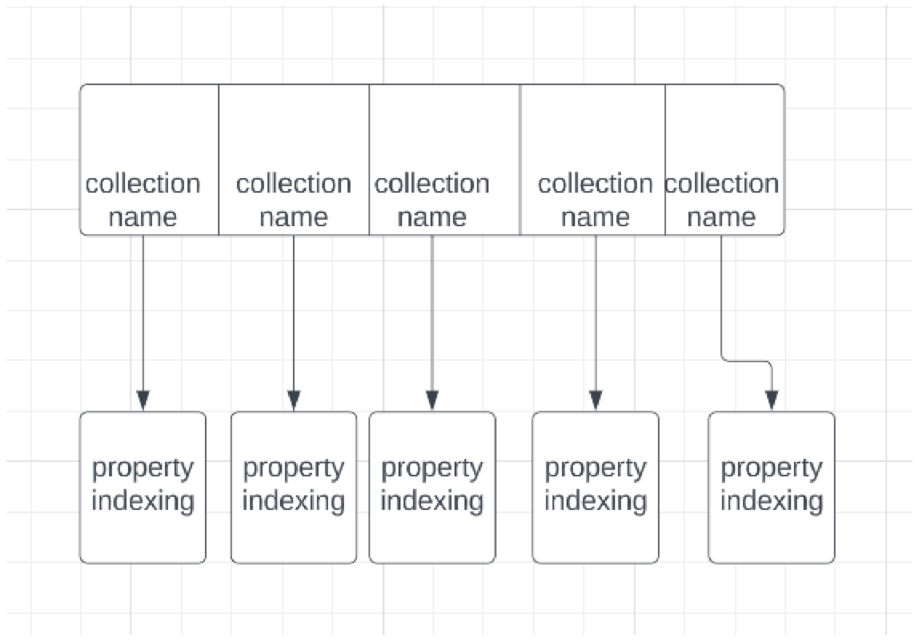


Figure-15: shows the collection indexing

Property indexing

Property indexing is a HashMap , the key of it is a string property name , the value is another HashMap that , the key of it is a string value name and the value is an integer list contains the indexes of JsonNode in the ArrayNode that has that value for the that property as a value as follow :

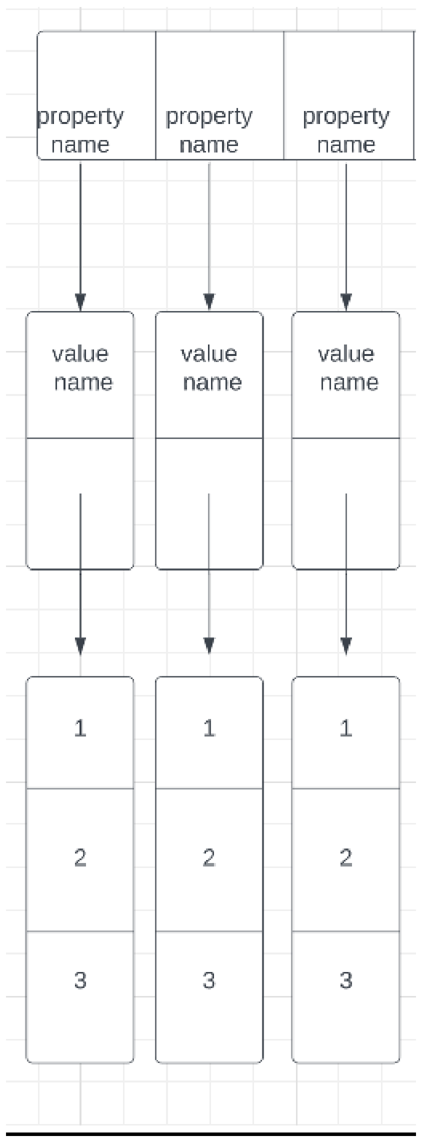


Figure-16: shows the property indexing

Cluster and broadcasting

A cluster refers to a group of connected computers that work together as a single system. Clusters are used to improve the performance and availability of computer applications, as well as to enable high-performance computing for scientific research, financial modeling, and other data-intensive tasks.

Clusters offer several benefits over traditional single-computer systems, including:

1. Scalability: Clusters can be easily scaled up or down depending on the needs of the application. This makes it possible to handle large workloads without requiring expensive upgrades to individual computers.
2. Fault tolerance: Clusters are designed to be fault-tolerant, meaning that they can continue to operate even if one or more nodes fail. This

ensures that the application remains available and that data is not lost.

3. High performance: Clusters can process data in parallel, which can significantly improve performance for data-intensive tasks.
4. Cost-effectiveness: Clusters can be built using commodity hardware, which can significantly reduce the cost of building and maintaining a high-performance computing environment.

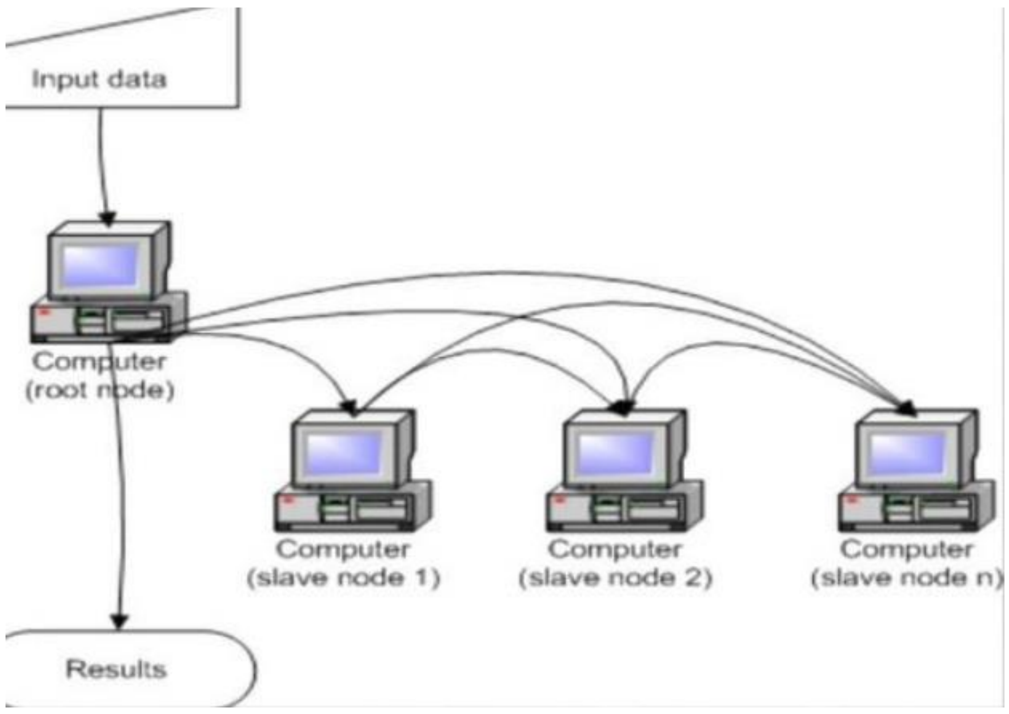


Figure-17 : the cluster explanation

Broadcasting is the process of sending a message or data packet to all devices connected to a network. The purpose of broadcasting is to distribute information or updates to all devices in a network simultaneously.

In the context of a cluster, broadcasting can be used to distribute messages or data between nodes in the cluster. When a message or data packet is broadcast to all nodes in a cluster, it ensures that all nodes receive the same information at the same time, which is essential for maintaining consistency and coordination among the nodes.

For example, in a load-balancing cluster, a central load balancer may broadcast instructions to all the nodes in the cluster, directing them to perform certain tasks or to distribute the workload among the nodes. This ensures that all nodes in the cluster are aware of the current state of the workload and can coordinate their actions accordingly.

Broadcasting can also be used for fault-tolerance in clusters. For instance, in a high-availability cluster, if a node fails, a broadcast message can be sent to all other nodes in the cluster, instructing them to take over the workload of the failed node. This ensures that the workload is distributed evenly across the remaining nodes, preventing any one node from becoming overloaded.

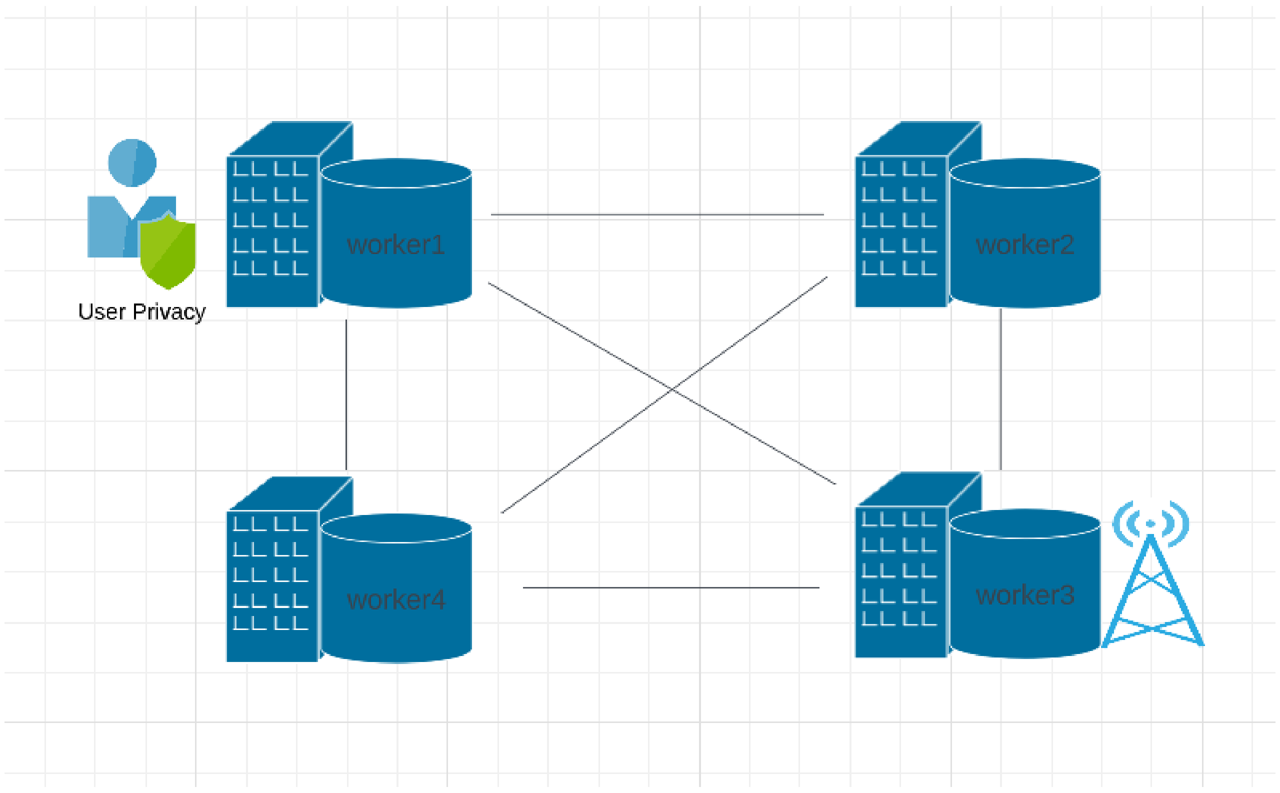


Figure-18: the clusters

In my project I used cluster system , and it is represented with Broadcast class in cluster package . And the flow will be like this :

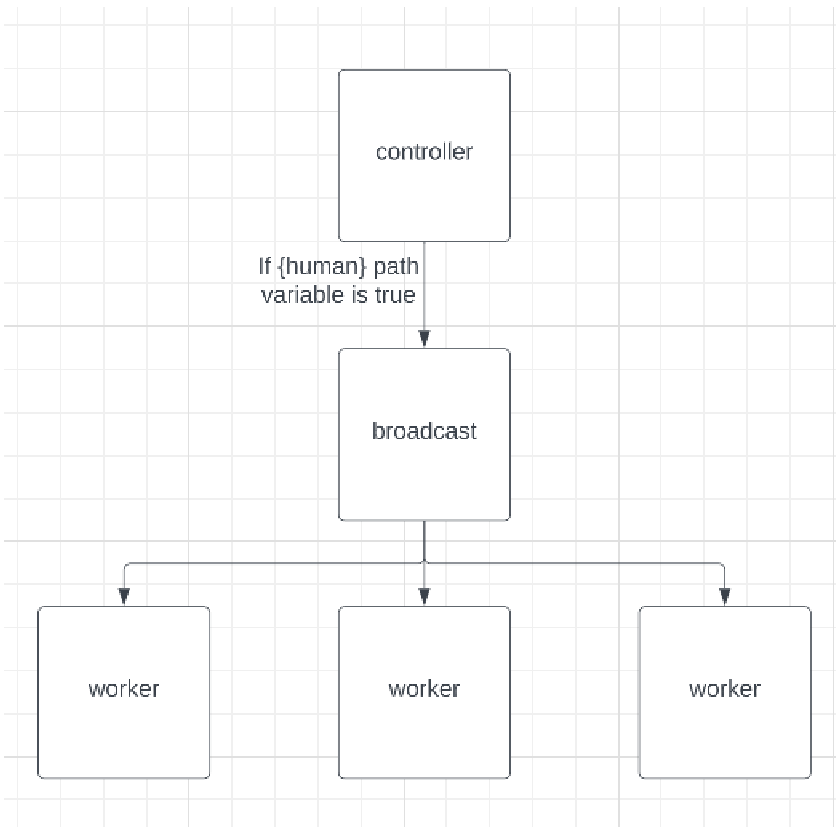


Figure-19: shows the flow of request if it was from human

The controller will receive the request and then check if the path variable {human} in the URL is true (if the URL sent by human not by the cluster), if it is true , then it will invoke the appropriate function in broadcast class . and the function in broadcast class will send the URL to all workers and change the {human} from true to false to indicate that this time the sender is the cluster class not a human.

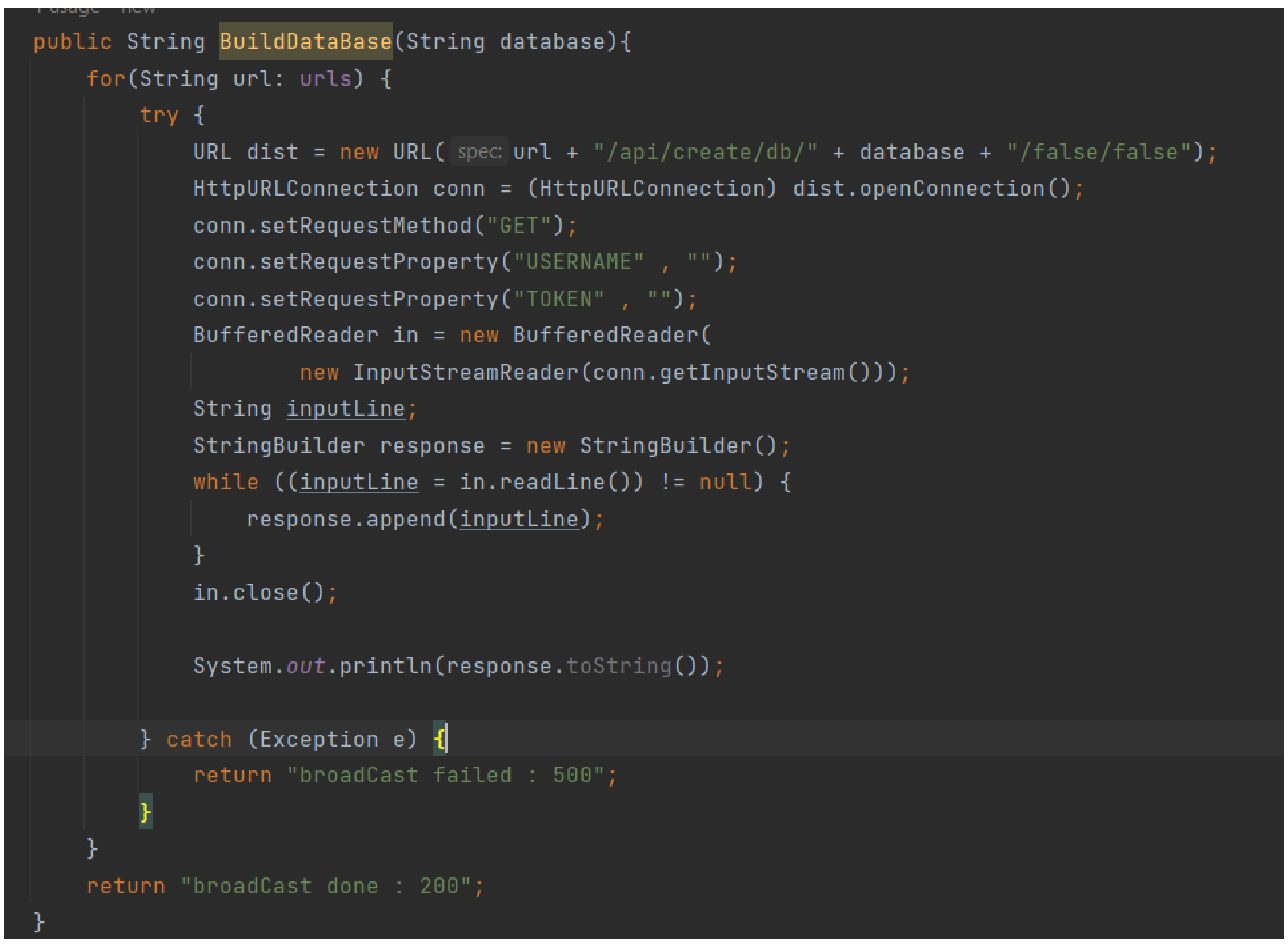
The **Broadcast** class has several methods, each responsible for a different HTTP request. The methods are:

- **BuildDataBase**: Sends a GET request to create a new database.

- **buildCollection**: Sends a POST request to create a new collection in a database.
- **addRecord**: Sends a POST request to add a new record to a collection.
- **deleteRecord**: Sends a GET request to delete a record from a collection.
- **Update**: Sends a PUT request to update a property of a record in a collection.
- **deleteDatabase**: Sends a DELETE request to delete a database.

All of the methods take a list of URLs as input and send the same HTTP request to each URL in the list. The response from each request is logged to the console, and if an error occurs, an error message is returned.

I will explain one of these functions line by line, for example:



```java
public String BuildDataBase(String database){
    for(String url: urls) {
        try {
            URL dist = new URL( spec: url + "/api/create/db/" + database + "/false/false");
            HttpURLConnection conn = (HttpURLConnection) dist.openConnection();
            conn.setRequestMethod("GET");
            conn.setRequestProperty("USERNAME" , "");
            conn.setRequestProperty("TOKEN" , "");
            BufferedReader in = new BufferedReader(
                    new InputStreamReader(conn.getInputStream()));
            String inputLine;
            StringBuilder response = new StringBuilder();
            while ((inputLine = in.readLine()) != null) {
                response.append(inputLine);
            }
            in.close();

            System.out.println(response.toString());

        } catch (Exception e) {
            return "broadCast failed : 500";
        }
    }
    return "broadCast done : 200";
}
```

Figure-20: shows the BuildDataBase function in broadcast class

1. **URL dist = new URL(url + "/api/create/db/" + database + "/false/false");**

This line creates a new **URL** object named **dist** by concatenating the **url** variable with a specific string pattern, including the **database** variable in the URL.

2. **HttpURLConnection conn = (HttpURLConnection) dist.openConnection();**

This line opens a connection to the **URL** object **dist** using the **openConnection** method, which returns a **URLConnection** object. This **URLConnection** object is then cast to an **HttpURLConnection** object and assigned to the variable **conn**.

3. **conn.setRequestMethod("GET");**

   conn.setRequestProperty("USERNAME" , "");

   conn.setRequestProperty("TOKEN" , "");

These lines set the request method to **"GET"** and add two request properties named **"USERNAME"** and **"TOKEN"** to the **HttpURLConnection** object. The values for these properties are empty strings.

4. **BufferedReader in = new BufferedReader(new InputStreamReader(conn.getInputStream()));**

```
 String inputLine;
StringBuilder response = new StringBuilder(); while ((inputLine =
in.readLine()) != null) {
        response.append(inputLine);
}
 in.close();
```

This code creates a new **BufferedReader** object called **in** that reads from the input stream of the **HttpURLConnection** object **conn**. The code then reads each line of the input stream using **in.readLine()** and appends each line to a **StringBuilder** object called **response**. The **BufferedReader** is then closed using the **close()** method.

5. System.out.println(response.toString());

This line prints the contents of the **StringBuilder** object **response** to the console.

Affinity and load balancing

An affinity node is a technique used in computer networking and load balancing to ensure that a user's requests are consistently sent to the same server or node.In a load-balanced environment, requests from a user are distributed across multiple servers to ensure that no single server is overloaded.

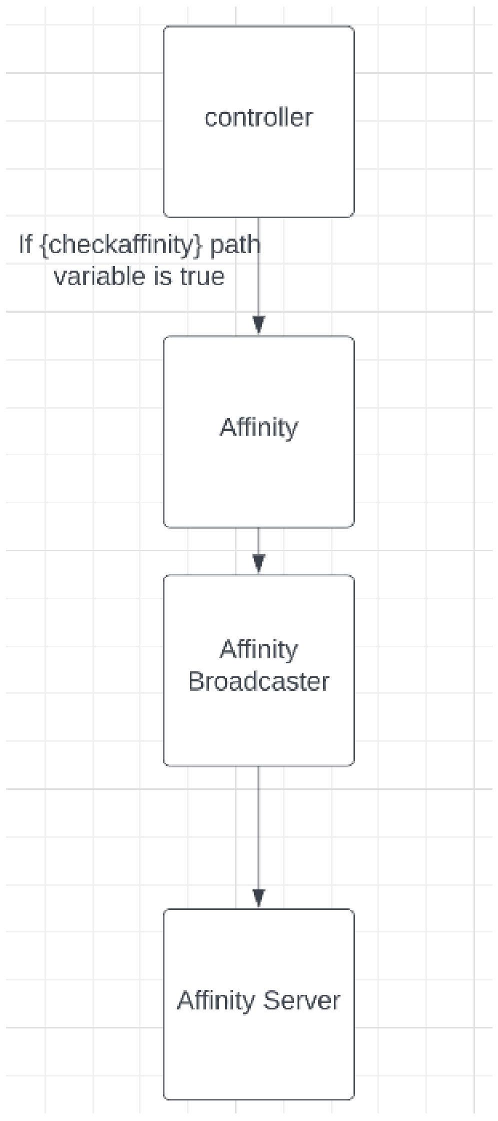


Figure-21: the flow of affinity process in my code

The controller will receive the request and then check if the path variable {checkaffinity} in the URL is true (if the URL sent the first time and required to check the affinity), if it is true , then it will invoke the appropriate function in Affinity class that generate the port of affinity node server ,and then invoke the appropriate function in AffinityBroadcast that receives the port of affinity node and send the request to it, and change the {checkaffinity} from true to false to indicate that this time it is not needed to check the affinity node.

Multithreading

Multithreading is a technique in computer programming where a single program or process can have multiple threads of execution running concurrently, allowing multiple tasks to be performed simultaneously within the same program. A thread is a lightweight sub-process within a program that can be executed independently and shares the same memory space as the main process.

There are several reasons why multithreading is used in computer programming. One of the primary reasons is to improve the overall performance and efficiency of the program. By using multiple threads, a program can take advantage of modern computer architectures with multiple processing cores or CPUs. By dividing a task into smaller subtasks and running them in parallel, a multithreaded program can perform calculations or other tasks much faster than a single-threaded program.

Another reason to use multithreading is to improve the responsiveness and interactivity of a program. By separating long-running tasks from the main thread of execution, a program can continue to respond to user input and events in a timely manner, even while performing complex calculations or I/O operations in the background.

Multithreading is also commonly used in networking and server applications, where multiple clients may connect and request services simultaneously. By using a separate thread to handle each client connection, a server can handle multiple requests concurrently without blocking or slowing down other clients.

multithreading also introduces some challenges, such as the need to manage shared resources and prevent race conditions and deadlocks. Proper synchronization techniques and thread-safe programming practices must be used to ensure the correct behavior and performance of a multithreaded program.

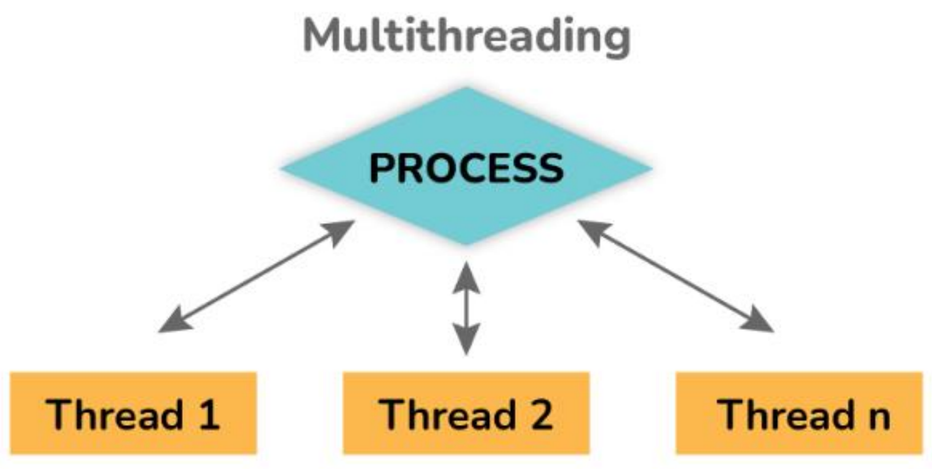


Figure-22: multithreading

I have a package in my code called configuration , that contain one function called AsyncExecuter that configures and returns an Executor bean, which is used for asynchronous method execution in Spring.

I used **@Async** annotation , **@Async** is an annotation used to indicate that a method should be executed asynchronously. When a method is marked with **@Async**, it means that it should be executed in a separate thread from the caller's thread. This allows the caller to continue with its work without waiting for the method to complete, making the application more responsive and efficient.

Docker and Containerization

Figure-23: docker logo

Introduction to Containerization

Containerization is a technology that enables software to be packaged into self-contained, portable environments that can run consistently across different platforms. In the past, software applications were installed directly onto a host operating system, which could lead to compatibility issues when the same application was run on different platforms. Containerization solves this problem by isolating the application and its dependencies from the host operating system, allowing it to run in a consistent environment across different platforms.

What is Docker?

Docker is a platform that enables developers to build, ship, and run applications in containers. Docker provides a set of tools and APIs for creating and managing containers, as well as a registry for storing and sharing container images. With Docker, developers can easily package their applications and dependencies into a single container image, which can then be run on any platform that supports Docker.

How Does Docker Work?

Docker uses a client-server architecture, where the Docker client communicates with the Docker daemon to create, manage, and run containers. The Docker daemon runs on the host operating system and is responsible for managing the containers and their resources. Containers are created from images, which are essentially pre-configured templates that define the application and its dependencies. When a container is created, Docker creates a virtual environment that isolates the container from the host operating system, ensuring that it runs consistently across different platforms.

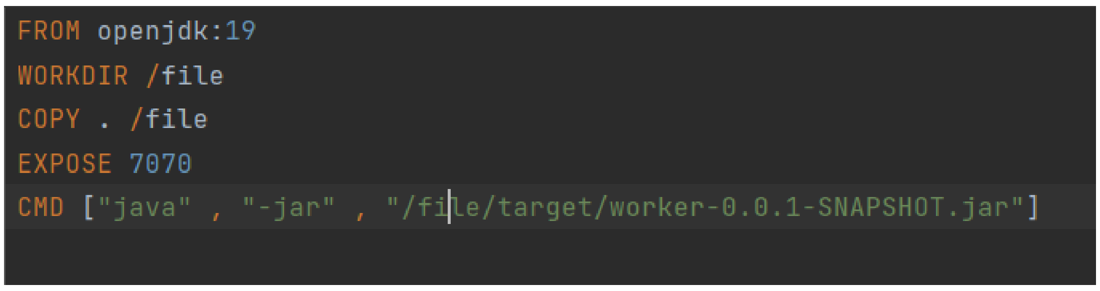
In this project, I have used Docker and containerization to create images for three separate components: Bootstrap, Worker, and demo. This process involved packaging each component into a container, which provides a self-contained, consistent environment that can be easily transported and deployed on different platforms and environments.

By containerizing each component, I was able to simplify the testing and deployment process and ensure that the application runs consistently across different environments. This is because each container includes all the necessary dependencies and libraries, which eliminates compatibility issues and reduces the risk of errors occurring during deployment.

Docker Files

The Dockerfiles used in this project started by specifying a base image, which included the OpenJDK runtime environment required for running Java applications. The working directory for the Docker container was then set, and the contents of the application, including code, dependencies, and other necessary files, were copied to the container.

The EXPOSE command was used to specify the port on which the container will be listening for incoming network connections. Finally, the CMD command specified the command to be executed when the container starts up, which in this case was to run the web application JAR file with the java executable and make it accessible through the exposed port. Overall, containerization simplified the testing and deployment process by providing a consistent and portable environment for the application.

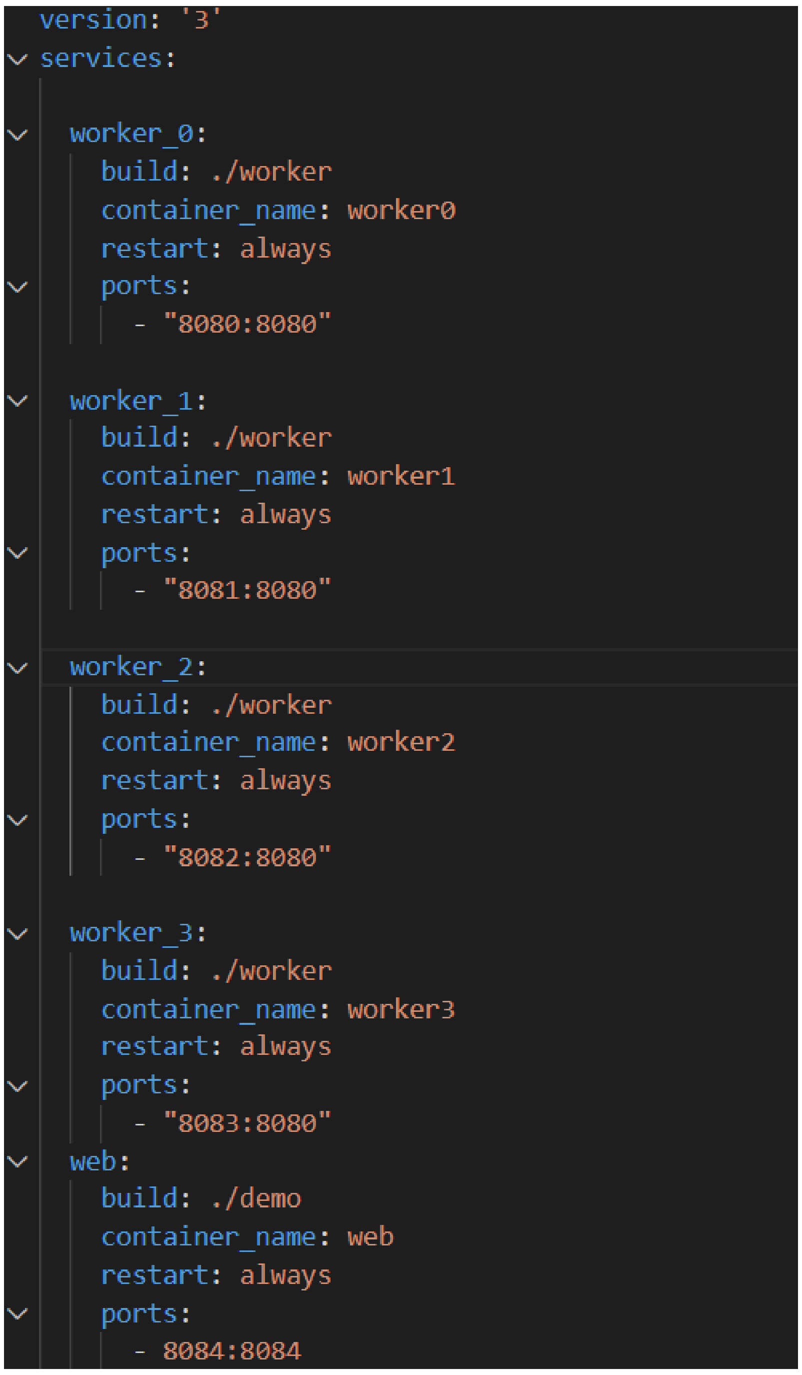


```
FROM openjdk:19
WORKDIR /file
COPY . /file
EXPOSE 7070
CMD ["java" , "-jar" , "/file/target/worker-0.0.1-SNAPSHOT.jar"]
```

Figure-24: Docker file in my project

Docker composed file

A Docker Compose file is a YAML file used to define and configure multiple Docker containers as a single application. It allows you to define the services that make up your application, their dependencies, and how they communicate with each other.



```yaml
version: '3'
services:

  worker_0:
    build: ./worker
    container_name: worker0
    restart: always
    ports:
      - "8080:8080"

  worker_1:
    build: ./worker
    container_name: worker1
    restart: always
    ports:
      - "8081:8080"

  worker_2:
    build: ./worker
    container_name: worker2
    restart: always
    ports:
      - "8082:8080"

  worker_3:
    build: ./worker
    container_name: worker3
    restart: always
    ports:
      - "8083:8080"
  web:
    build: ./demo
    container_name: web
    restart: always
    ports:
      - 8084:8084
```

Figure-25: Docker composed

In this example, there are six services defined:

- Four instances of a worker service, each built from the same Dockerfile located in the "./worker" directory. Each instance has a unique name and is exposed on a different port, allowing for load balancing across the workers.

- A web service, built from a Dockerfile in the "./demo" directory. This service is responsible for hosting the web application and is exposed on port 8084.
- A bootstrap service, built from a Dockerfile in the "./bootstrap" directory. This service is responsible for initializing the worker instances and is exposed on port 6060. It depends on the four worker services to be running before it starts.

The configuration options for each service include the build context, container name, restart policy, exposed ports, and dependencies on other services.

Bootstrap

The bootstrap is a crucial component in the operation of a cluster, serving two fundamental purposes.

The first role of the bootstrap is to initialize the cluster by bringing in user, admin, and affinity JSON files, which contain essential information for the cluster's proper operation. The user JSON file, for example, stores the usernames of all registered users and the workers assigned to them. The admin JSON file, on the other hand, contains the details of all registered administrators in the system. Finally, the affinity JSON file is responsible for tracking the relationship between the workers and the users they serve.

The second critical role of the bootstrap is to check whether a new user or administrator already exists in the system during the sign-up process. If the user or administrator does not exist, the bootstrap generates a new token for them, which grants them access to the cluster's resources. However, if the user or administrator already exists, the bootstrap will not generate a new token but will instead return their existing token. This ensures that there are no duplicate users or administrators in the system, which can cause confusion and potential security issues. If an administrator is added to the system, the bootstrap includes it in all workers, allowing them to access the entire cluster.

In contrast, when a new user is added, the bootstrap assigns them to a specific worker and updates the affinity value in the affinity JSON file. This way, the cluster can optimize its performance by ensuring that each user is assigned to the most suitable worker based on their needs.

In summary, the bootstrap is responsible for initializing the cluster and ensuring that new users and administrators are added correctly, preventing any duplicates or security issues. By assigning users to suitable workers and updating the affinity value, the bootstrap optimizes the performance of the entire system, ensuring that it runs smoothly and efficiently.

Security

In any software system, it is crucial to establish appropriate security measures to protect the system from unauthorized access and to ensure that user data is kept confidential. One important aspect of security is user authentication, which typically involves verifying the identity of a user before granting them access to the system.

When implementing a login system, the first step is to create a user account. This involves collecting information such as the user's name. Once a user account is created, he will be added to the system by AuthenticationController class in server, then the user can sign in to the system. This typically involves entering their username, which is then verified by the system to ensure that it matches the user's account.

However, simply relying on usernames for authentication is not always sufficient, as usernames can be guessed or stolen. Therefore, many systems use additional measures such as two-factor authentication or biometric authentication to further enhance security.

When a user sign up successfully, the system typically generates a unique token by the bootstrap to sign in, and then it will send the user name and token to server, and the server will ensure that the user is authorized in

AuthenticationClass. This token is used to identify the user and to grant them access to restricted areas of the system.

The flow of the process will be like this: First the user will enter his name to the demo , and the request will be sent from demo to bootstrap to generate Token . Then , if sign up as user the bootstrap will add the user to the specific worker . if sign up as admin the bootstrap will add him to all workers , and the worker will check if he is authorized or not and send the response to demo.

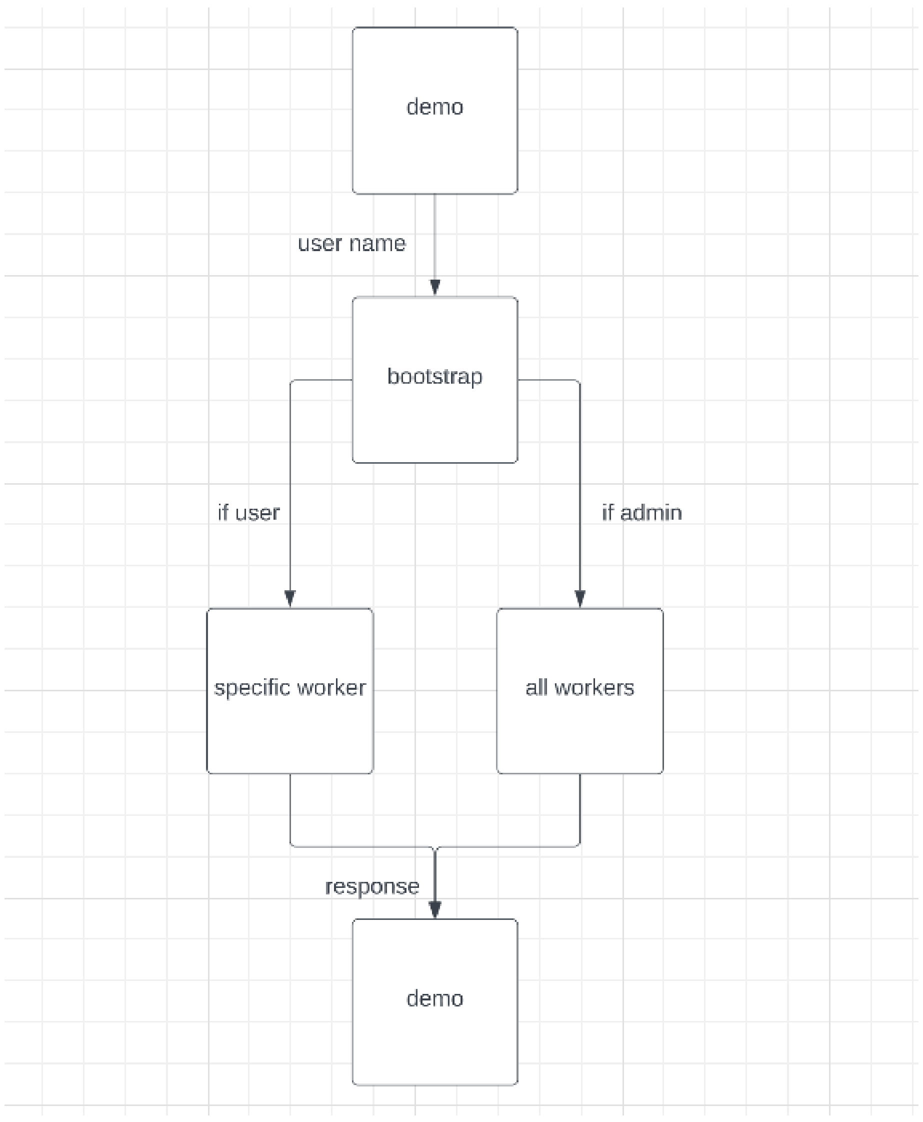
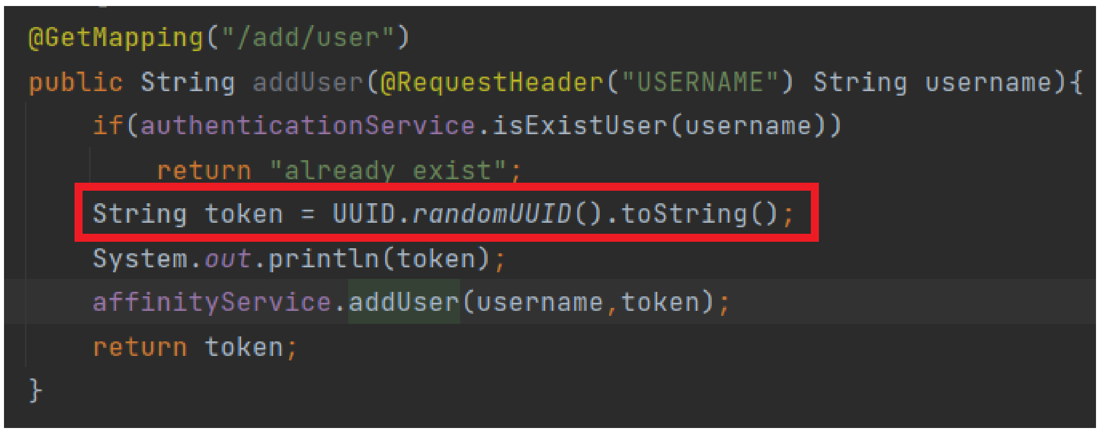


Figure-26: the flow of login process.



```
@GetMapping("/add/user")
public String addUser(@RequestHeader("USERNAME") String username){
    if(authenticationService.isExistUser(username))
        return "already exist";
    String token = UUID.randomUUID().toString();
    System.out.println(token);
    affinityService.addUser(username,token);
    return token;
}
```

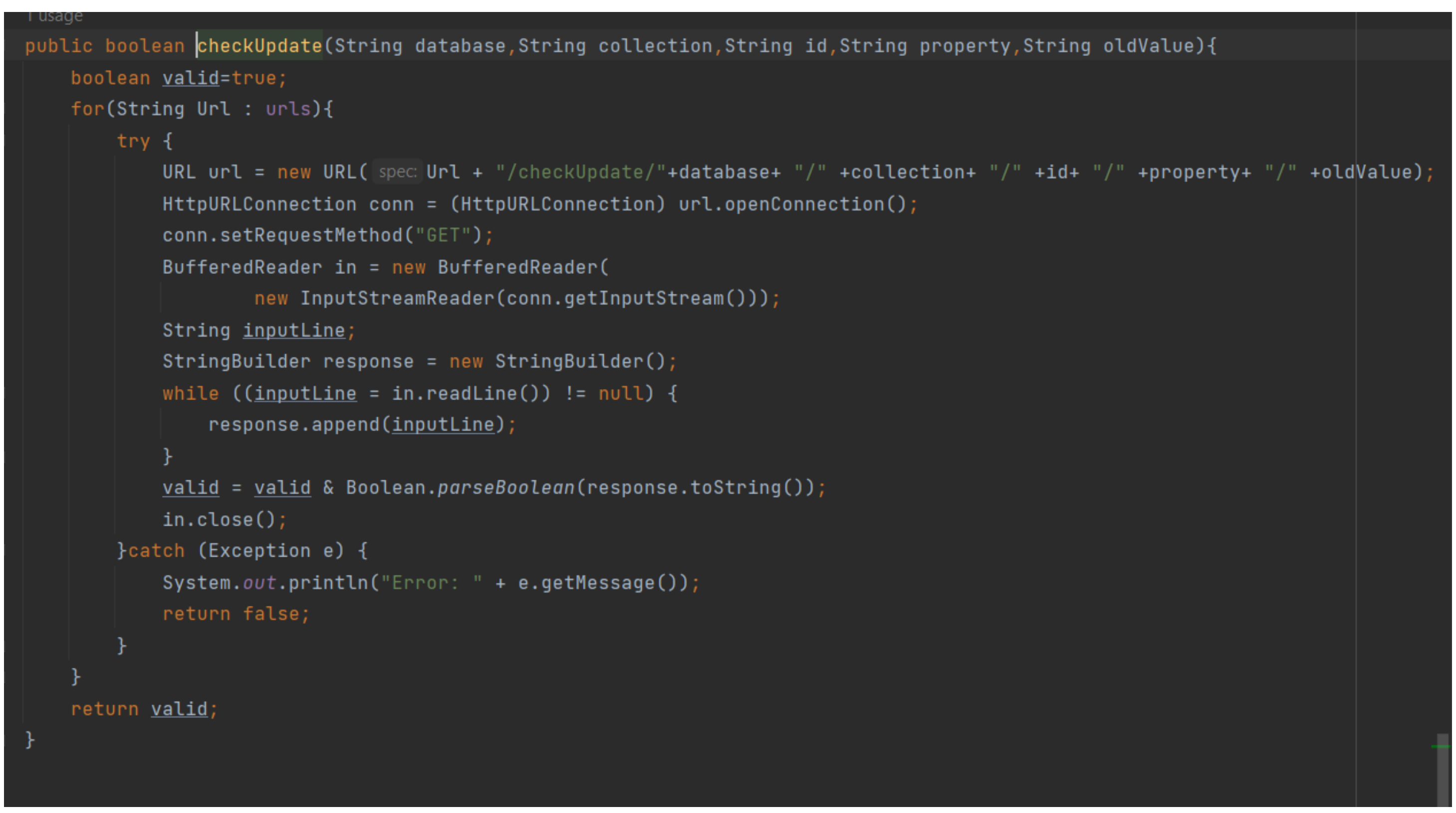
Figure-27: generate token.

Race Condition

I handled the race condition in two levels:

1. Between servers:

I handled it by checking the value of record in all server before and after update . In this way, if any server does not have the same value of record as the other servers it will return false and the update process will stop.

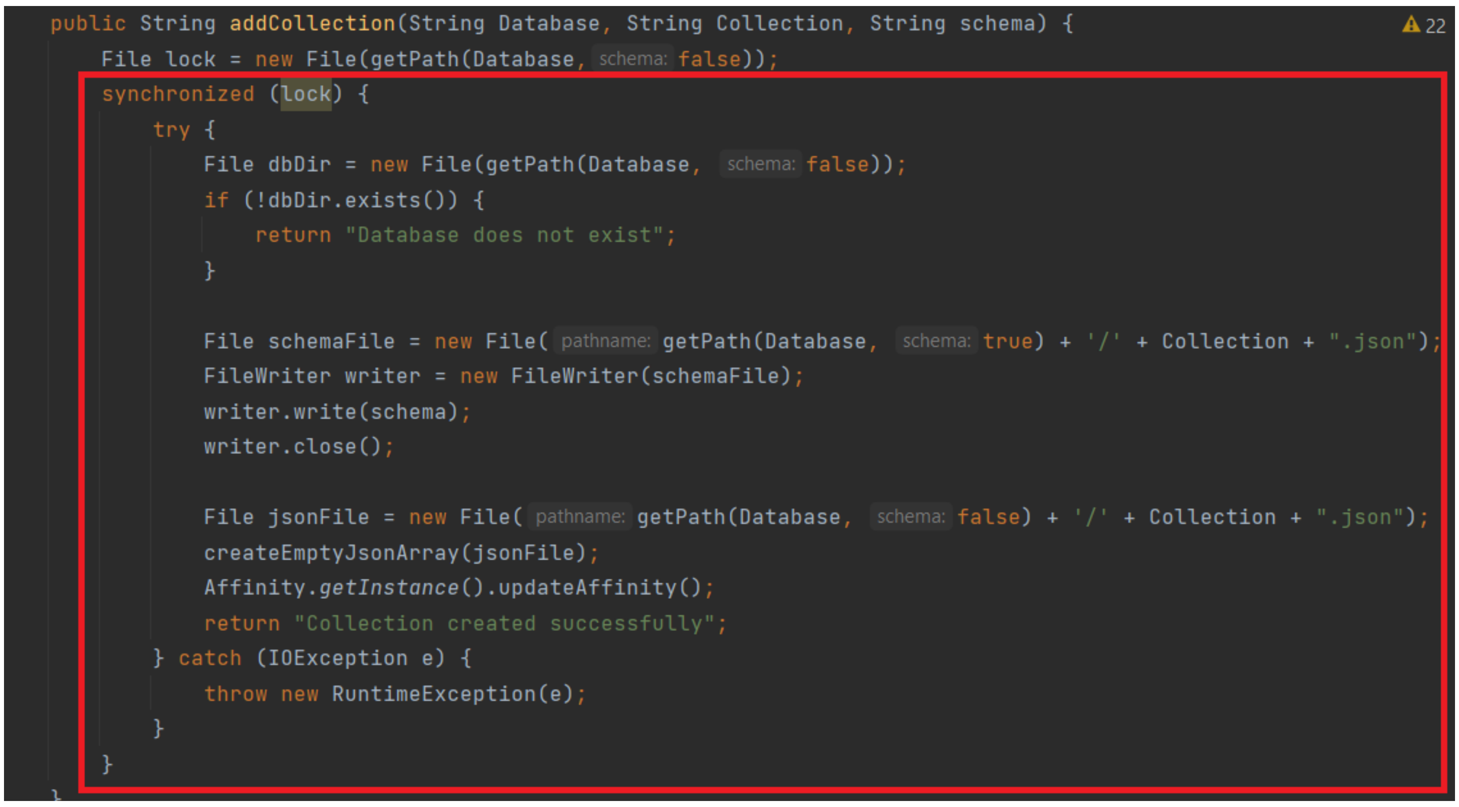


```
1 usage
public boolean checkUpdate(String database,String collection,String id,String property,String oldValue){
    boolean valid=true;
    for(String Url : urls){
        try {
            URL url = new URL( spec: Url + "/checkUpdate/"+database+ "/" +collection+ "/" +id+ "/" +property+ "/" +oldValue);
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            conn.setRequestMethod("GET");
            BufferedReader in = new BufferedReader(
                    new InputStreamReader(conn.getInputStream()));
            String inputLine;
            StringBuilder response = new StringBuilder();
            while ((inputLine = in.readLine()) != null) {
                response.append(inputLine);
            }
            valid = valid & Boolean.parseBoolean(response.toString());
            in.close();
        }catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
            return false;
        }
    }
    return valid;
}
```

Figure-28: Update

2. Inside single server:

I handled it by using synchronization in multithreading . I used synchronization in block level so that every thread can make changes in record without effect another records, and if two thread want to make change in the same record , the second thread will wait until first one finish.



```
public String addCollection(String Database, String Collection, String schema) {
    File lock = new File(getPath(Database, schema: false));
    synchronized (lock) {
        try {
            File dbDir = new File(getPath(Database, schema: false));
            if (!dbDir.exists()) {
                return "Database does not exist";
            }

            File schemaFile = new File( pathname: getPath(Database, schema: true) + '/' + Collection + ".json");
            FileWriter writer = new FileWriter(schemaFile);
            writer.write(schema);
            writer.close();

            File jsonFile = new File( pathname: getPath(Database, schema: false) + '/' + Collection + ".json");
            createEmptyJsonArray(jsonFile);
            Affinity.getInstance().updateAffinity();
            return "Collection created successfully";
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
```

Figure-29: locks

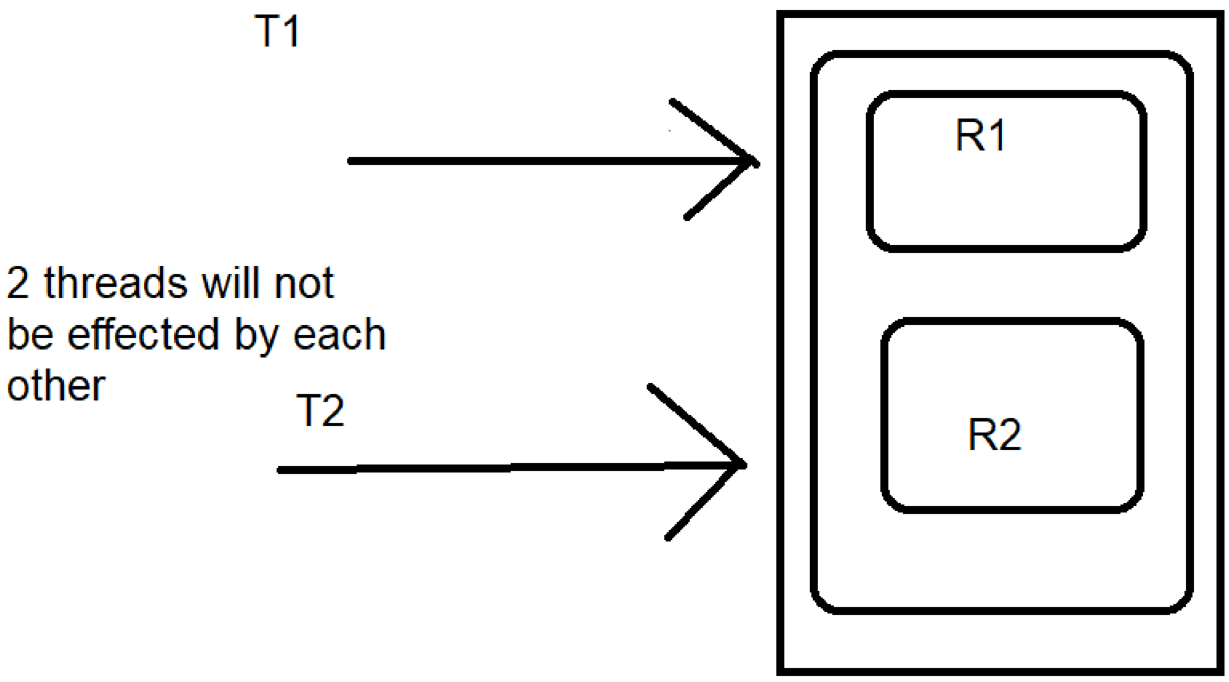


Figure-30: No race condition

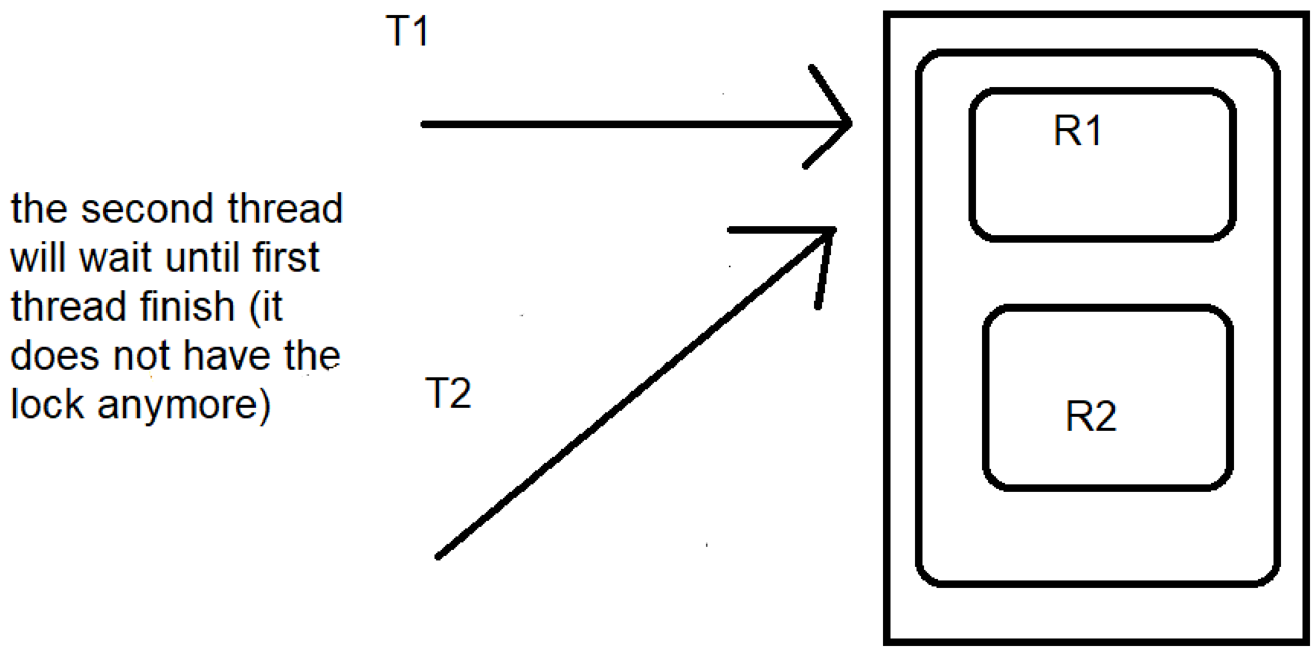

T1
the second thread will wait until first thread finish (it does not have the lock anymore)
T2
R1
R2

Figure-31 : race condition

SOLID principles

1. **The Single Responsibility Principle (SRP)**: is a fundamental principle in software development that states that a class or module should have only one reason to change. In other words, a class or module should have only one responsibility or job to do. This principle helps to ensure that classes and modules are focused and maintainable, and that they are not overly complex or difficult to change. By adhering to the SRP, developers can create software that is easier to test, maintain, and extend over time.

The Single Responsibility Principle (SRP) has been used all over the project such as Affinity, Bootstraping , worker, building schema in demo and validating it in workers.

2. The Open-Closed Principle (OCP): is a fundamental principle in software development that states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. In other words, the behavior of a software entity should be able to be extended without modifying its source code. By adhering to the OCP, developers can create software that is easier to maintain, extend, and test over time.

Most of the classes has been implemented in a way that makes it opened to be extended in the future and closed to modification.

Design patterns

Singleton design pattern

The Singleton design pattern has been implemented for both the DatabaseIndexing class in worker and the Affinity class in worker and bootstrapingm .This means that there can only be one instance of that class created during the lifetime of an application to ensure that there is only one instance of a class that is used throughout the application, such as a database connection or a configuration manager. By ensuring that there is only one instance of the class, we can improve performance and reduce resource usage. This approach can help to simplify the management of these classes and prevent conflicts that may arise from multiple instances.

DAO

I used this structure pattern in worker , it is a structure design pattern that implementing by separating the data access code from the business logic, the DAO design pattern helps to improve the maintainability, scalability, and reusability of an application. It also simplifies the testing process by providing a clear separation of concerns.

DEMO

I created a demonstration utilizing the Spring framework. It contains login system, user options (the operations that can by done by user) and admin options (the operations that can by done by admin).

Figure-32: choose user or admin page

Figure-33: login system

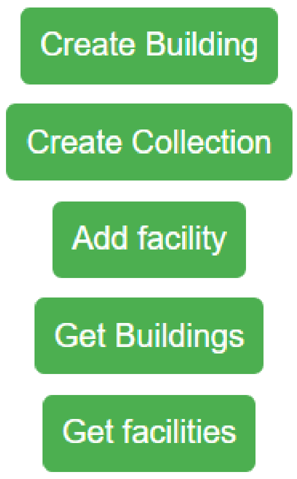


Figure-34: operation that can done by user.

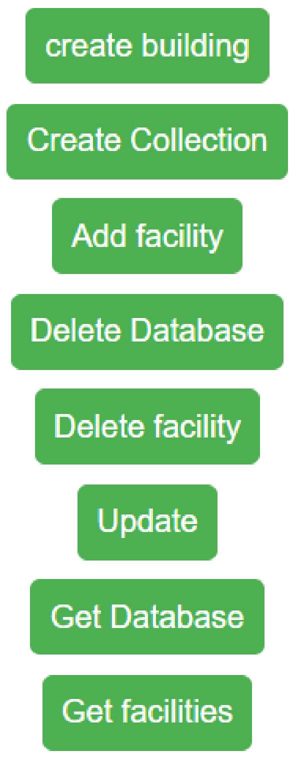


Figure-35: operation that can done by admin